Antonio Restivo
Simona Ronchi Della Rocca
Luca Roversi (Eds.)

LNCS 2202

Theoretical Computer Science

7th Italian Conference, ICTCS 2001
Torino, Italy, October 2001
Proceedings

Springer

Lecture Notes in Computer Science 2202

Edited by G. Goos, J. Hartmanis and J. van Leeuwen

Springer
Berlin
Heidelberg
New York
Barcelona
Hong Kong
London
Milan
Paris
Tokyo

Antonio Restivo Simona Ronchi Della Rocca
Luca Roversi (Eds.)

Theoretical Computer Science

7th Italian Conference, ICTCS 2001
Torino, Italy, October 4-6, 2001
Proceedings

Springer

Volume Editors

Antonio Restivo
Università di Palermo, Dipartimento di Matematica ed Applicazioni
Via Archirafi, n. 34, 90123 Palermo, Italy
E-mail: restivo@dimath.math.unipa.it

Simona Ronchi Della Rocca
Luca Roversi
Università di Torino, Dipartimento di Informatica
C.so Svizzera, n. 185, 10149 Torino, Italy
E-mail:{ronchi,roversi}@di.unito.it

Cataloging-in-Publication Data applied for

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Theoretical computer science : 7th Italian conference ; proceedings /
ICTCS 2001, Torino, Italy, October 4 - 6, 2001. Antonio Restivo ... (ed.). -
Berlin ; Heidelberg ; New York ; Barcelona ; Hong Kong ; London ; Milan ;
Paris ; Tokyo : Springer, 2001
(Lecture notes in computer science ; Vol. 2202)
ISBN 3-540-42672-8

CR Subject Classification (1998): F, E.1, G.1-2

ISSN 0302-9743
ISBN 3-540-42672-8 Springer-Verlag Berlin Heidelberg New York

Springer-Verlag Berlin Heidelberg New York
a member of BertelsmannSpringer Science+Business Media GmbH

http://www.springer.de

Typesetting: Camera-ready by author, data conversion by DaTex Gerd Blumenstein
Printed on acid-free paper SPIN 10840711 06/3142 5 4 3 2 1 0

Preface

The Italian Conference on Theoretical Computer Science (ICTCS) is the conference of the *Italian Chapter of the European Association for Theoretical Computer Science* (IC-EATCS), that takes place every three years. The conference aims at enabling computer scientists, especially young researchers, to enter the EATCS community and to exchange ideas and results, as well as theory based practical experiences and tools in computer science.

This was the seventh Italian Conference on Theoretical Computer Science, and its main topics included analysis of algorithms, automata, computability, computational complexity, cryptography, data types and structures, design of algorithms, formal languages, foundations of functional programming, foundations of logic programming, new computing paradigms, parallel and distributed computation, program specification, program verification, term rewriting, theory of concurrency, theory of data bases, theory of logical design and layout, theory of robotics, theory of knowledge bases, type theory, semantics of programming languages, security, and symbolic and algebraic computation.

ICTCS 2001 was held in Turin, Italy, October 4–6, 2001. Previous conferences took place in Pisa (1972), Mantova (1974 and 1989), L'Aquila (1992), Ravello (1995), and Prato (1998).

The Program Committee selected 25 papers out of 45 submissions, all of them in electronic format. Their authors are from 11 countries, from all over the world. Each submission was sent to three Program Committee members, assisted by their own referees.

The selection meeting took place as an electronic forum. To permit a deeper evaluation of the papers, the Program Committee split them into two subject areas for the preliminary discussion, according to the two tracks of the Journal of Theoretical Computer Science which are "Algorithms, automata, complexity, and games", and "Logic, semantics, and theory of programming", and which reflect the main division in research topics within the community. Then, to preserve the unifying aspects of the research in theoretical computer science, all the papers were evaluated again and all the decisions were taken together.

We would like to warmly thank all the people who submitted their papers to the conference, the Program Committee members, and their referees for their invaluable contribution.

July 2001 Antonio Restivo, Simona Ronchi Della Rocca, and Luca Roversi

Organization

ICTCS 2001 was organized under the auspices of the European Association of Theoretical Computer Science, and the European Educational Forum, with the financial support of the University of Turin, Faculty of Mathematical, Physical & Natural Science, and Computer Science Department

Organizing Committee:	Viviana Bono, University of Turin
	Ferruccio Damiani, University of Turin
	Mariangiola Dezani (Chair), University of Turin
	Luca Roversi, University of Turin
	Stefano Berardi, University of Turin
	Ugo de'Liguoro, University of Turin
	Paola Giannini, University of Alessandria
Technical support:	Bruno Graziano, University of Turin
	Simone Donetti, University of Turin
	Paolo Pasteris, University of Turin
Administrative support:	Daniela Costa, University of Turin
	Claudia Goggioli, University of Turin

Program Committee

Andrea Asperti	Università di Bologna
Bruno Codenotti	University of Pisa
PierPaolo Degano	University of Pisa
Moreno Falaschi	University of Udine
Giuseppe F. Italiano	University of Rom
Linda Pagli	University of Pisa
Simona Ronchi Della Rocca (Co-chair)	University of Turin
Alberto Bertoni	University of Milan
Clelia De Felice	University of Salerno
Rocco De Nicola	University of Florence
Paola Inverardi	Università dell'Aquila
Eugenio Moggi	University of Geneva
Antonio Restivo (Co-chair)	University of Palermo
Ugo Vaccaro	University of Salerno

Referees

Riccardo Solmi
Beatrice Palano
Claudio Gentile
Alessandro Provetti
Brunella Gerla
Giovanni Pighizzini

Silvio Ghilardi
Massimiliano Goldwurm
Massimo Santini
Gianluca De Marco
Giovanni Resta
Adele Rescigno
Margherita Napoli
Domenico Parente
Clemente Galdi
Ivan Visconti
Vincenzo Auletta
Jean-Marc Champarnaud
Jean Berstel
Marcella Anselmo
Flavio Corradini
Giovanni Manzini
Simone Martini
Gianluigi Ferrari
Andrea Masini
Francesca Levi
Vincenzo Manca
Michele Boreale
Rosario Pugliese
Michele Loreti
Donatella Merlini
Lorenzo Bettini
Betti Venneri
Agostino Dovier
Maurizio Gabbrielli
Paolo Coppola
Fabio Alessi
Pietro Di Gianantonio
Marco Comini
Massimo Franceschet
Alberto Policriti
Ivan Scagnetto
Stefania Costantini
Benedetto Intrigila
Eugenio Omodeo
Zena Ariola
Diletta Romana Cacciagrano
Catuscia Palamidessi
Thierry Coquand
Cosimo Laneve
Marco Bozzano
Valentina Ciriani
Fabio Gadducci
Maurizio Bonuccelli
Giuseppe Prencipe
Mireille Reigner
Pierluigi Frisco
Anna bernasconi
Fabrizio Luccio
Filippo Mignosi
Vito Di Ges
Giancarlo Mauri
Rosalba Zizza
Sabrina Mantaci
Marinella Sciortino
Bruno Crispo
Francesco Bergadano
Davide Cavagnino
Ferruccio Damiani
Franco Barbanera
Jeremy Sproston
Luisa Gargano
Annalisa De Bonis
Ferdinando Cicalese
Gaia Nicosia
Michele Flammini
Serafino Cicerone
Antonio Di Nola
Patrizio Cintioli

Table of Contents

Invited Talk 1

Lambda Calculus and Types

Algorithms and Data Structures I

New Computing Paradigms

Formal Languages

Objects and Mobility

Algorithms and Data Structures II

Invited Talk 2

Computational Complexity

Security

Logics and Logic Programming

A LTS Semantics of Ambients via Graph Synchronization with Mobility*

GianLuigi Ferrari, Ugo Montanari, and Emilio Tuosto

Dipartimento di Informatica, Università di Pisa
{giangi,ugo,etuosto}@di.unipi.it

Abstract. We present a simple labelled transition system semantics of Cardelli and Gordon's Ambient calculus. We exploit a general and flexible model based on (hyper)graphs, where graph transformation is obtained via (hyper)edge replacement and local synchronization with mobility. In addition to tree-like ambients, the calculus we define works just as well with graph-like ambients, which are a more realistic model of internetworks.

1 Introduction

Foundational research on global computing aims at describing, modeling and analyzing the complex interactions taking place in internetwork applications encompassing several physical networks, multiple administration domains and a variety of possible users. Several models have been proposed to tackle the new computational phenomena. They usually take the form of distributed process calculi (e.g. Join calculus [7], Ambient calculus [2]), of specialized program logics (e.g. Mobile Unity [13], Mob$_{adtl}$ [6]), and of Linda-like coordination languages (e.g. KLAIM [4], Lime [12]), to mention a few.

Most models mainly focus on the *spatial structure* of global computing. To reflect the idea of administration domains, they exhibit explicit localities, which help modeling distributed computations and the discovery of network resources and services. These features distinguish the models of global computing from the traditional models (and paradigms) for distributed programming (e.g. CORBA), the motto being *network awareness*: localities are under programmer's control.

However network awareness is only one relevant tile of the mosaic of global computing. Another important aspect concerns the *temporal structure* of the applications. The run-time environment typically interleaves computational activities with structuring and managing activities. The temporal structure of applications takes care of describing application rearrangements and security checks. A proper understanding of both spatial and temporal structures is clearly needed to allow formal verification of applications.

* Partially supported by CNR project *Metodi per Sistemi Connessi mediante Reti*; by MURST project *Theory of Concurrency, Higher Order and Types*; by TMR Network GETGRATS; and by Esprit Working Groups *APPLIGRAPH*.

A. Restivo, S. Ronchi Della Rocca, L. Roversi (Eds.): ICTCS 2001, LNCS 2202, pp. 1–16, 2001.

The Ambient calculus is one of the best studied models addressing the needs of global computing, and it has acquired the role of touchstone for the most recent proposals. However the interactive, abstract semantics of ambients is still not fully explored. In fact, as it is the case of most foundational calculi for global computing, reduction semantics for ambients has been found to be simpler than the corresponding labeled transition system (LTS) semantics. However, reduction semantics has the main disadvantage with respect to LTS semantics that it makes harder to define, and reason about, abstract compositional behavior.

A LTS operational semantics for ambients has been defined by Gordon and Cardelli in an unpublished note [1]. It requires the introduction in the calculus of co-actions, abstractions, concretions and outcomes. At the authors' knowledge, the bisimilarity abstract semantics based on this operational semantics has not been compared with the reduction semantics and with the logics developed by Cardelli, Gordon and Caires, which is equipped with specialized modalities to deal with the spatial and the temporal dimensions of global computing. Sewell [14] introduces a technique to develop an LTS-based semantics from a reduction semantics; however the resulting transition semantics exploits arbitrary contexts and, moreover, it is not inductive on process operators.

In this paper we define a LTS semantics of ambients by exploiting a general and flexible model based on (hyper)graphs, where graph transformation is obtained via (hyper)edge replacement and local synchronization with mobility. While getting acquainted with the formal techniques necessary for handling graphs (rather than trees or terms) may require some effort, the actual definition of the Ambient calculus is quite short and intuitive. Moreover, in addition to tree-like ambients, the calculus we define works just as well with graph-like ambients, which are a more realistic model of internetworks.

More generally, we propose our graph-based technique as a tool for modeling internetworking systems. In fact, edges can be used to represent components and nodes to model the network environment of components. Some edges sharing a node means that the corresponding components may interact by exploiting network communication infrastructure. Structured versions of graphs (typed graphs, term graphs, hierarchical graphs) can precisely model complex internetwork configurations and access policies.

Graph synchronization adds to network awareness the ability of dealing with the temporal dimension of computations. Graphs synchronization is purely local and it is obtained by the combination of graph rewriting with constraint solving. The intuitive idea is that local rewritings depends on the outcome of a (possibly global) constraint satisfaction algorithm. Mobility allows to exchange nodes during synchronizations, and thus constraint solving must include unification to allow for node binding.

One may wonder if this approach is too abstract and general and it does not capture the intrinsic limitations of internetworking applications. We feel that on the one side the generality of the approach can be tamed and adapted to the needs of the various layers of applications, more powerful primitives being made available to upper layers, like B2B or CSCW. On the other side, some important network technologies actually require the solution of global constraints, like modifying local router tables according to the routing update information sent by the adjacent routers.

Graph rewriting based on edge replacement and synchronization was introduced in [3,5] and related to distributed constraint satisfaction problems in [11]. The version with mobility, which employs a notation based on logical sequents and inference rules, was introduced recently in [8] and extended in [9] to encode π-calculus. Abstract semantics based on bisimilarity was discussed in [10]. To model Ambient calculus, synchronized hyperedge replacement has been further extended in this paper with fusions. Fusions allow to coalesce in the right member of a production sets of interface nodes which are distinct in the left member. This extension is necessary for representing the effect of the *open* capability, which merges the localities inside and outside the open ambient.

In the paper we handle a limited version of ambients, without restriction (of ambient names) and process communication, and with guarded recursion rather than replication. We relate the operational semantics of ambients based on synchronized edge replacement to the original reduction semantics. We show that there is a bijective correspondence between ambient processes and certain graphs called ambient graphs. We also show that ambient processes and their corresponding graphs have corresponding reductions. Of course the graphs have in addition transitions with observable labels, which can be exploited in the abstract semantics, that however is not studied in the paper.

2 Hypergraphs and Graph Synchronization

We first review (as presented in [9]) the notion of hypergraph and its formalization in terms of well formed syntactic judgements. Then we introduce the notion of graph synchronization.

A *edge*, or simply an edge, is an atomic item with a label (from a ranked alphabet $LE = \{LE_n\}_{n=0,1,\ldots}$) and with as many (ordered) tentacles as the rank of its label. A set of *nodes* together with a set of such edges forms a *hypergraph* (or simply a graph) if each edge is connected, by its tentacles, to its *attachment* nodes. A graph is equipped with a set of external nodes identified by distinct names. External nodes can be seen as the connecting points of a graph with its environment.

Now, we present a definition of graphs as *syntactic judgements*, where nodes correspond to names, external nodes to free names and edges to basic terms of the form $L(x_1, \ldots, x_n)$, where x_i are arbitrary names and $L \in LE$.

Definition 1 (Graphs as Syntactic Judgements). *Let $\mathcal{N}$ be a fixed infinite set of names and LE a ranked alphabet of labels. A* syntactic judgement *(or simply a judgement) is of the form $\Gamma \vdash G$ where,*

1. *$\Gamma \subseteq \mathcal{N}$ is a set of names (the external nodes of the graph).*
2. *G is a term generated by the grammar*
   $G ::= L(\boldsymbol{x}) \mid G|G \mid (\nu y)G \mid nil$ where $\boldsymbol{x}$ is a vector of names, L is an edge label with $rank(L) = |\boldsymbol{x}|$ and y is a name.

Let fn(G) denote the set of all free names of G, i.e. all names in G not bound by a ν operator. We require that $fn(G) \subseteq \Gamma$.

We use the notation Γ, x to denote the set obtained by adding x to Γ, assuming $x \notin \Gamma$. Similarly, we will write Γ_1, Γ_2 to state that the resulting set of names is the disjoint union of Γ_1 and Γ_2.

Definition 2 (Structural Congruence and Well-Formed Judgements).

- *Structural Congruence $\equiv$ on syntactic judgements obeys axioms in Table 1.*
- *The* well-formed judgements *for constructing graphs over LE and $\mathcal{N}$ are those generated by applying the rules in Table 1 up to axioms of structural congruence.*

Table 1. Well-formed judgments

Structural Axioms

$$(AG1)\ (G_1|G_2)|G_3 \equiv G_1|(G_2|G_3) \qquad (AG2)\ G_1|G_2 \equiv G_2|G_1$$

$$(AG3)\ G|nil \equiv G \qquad (AG4)\ \nu x.\nu y.G \equiv \nu y.\nu x.G$$

$$(AG5)\ \nu x.G \equiv G\ if x \notin fn(G) \qquad (AG6)\ \begin{array}{c}\nu x.G \equiv \nu y.G\{y/x\}\\ if\ y \notin fn(G)\end{array}$$

$$(AG7)\ \nu x.(G_1|G_2) \equiv (\nu x.G_1)|G_2\ if\ x \notin fn(G_2)$$

Syntactic Rules

$$(RG1)\ \frac{}{x_1,\ldots,x_n \vdash nil} \qquad (RG2)\ \frac{L \in LE_m \quad y_i \in \{x_j\}}{x_1,\ldots,x_n \vdash L(y_1,\ldots,y_m)}$$

$$(RG3)\ \frac{\Gamma \vdash G_1 \quad \Gamma \vdash G_2}{\Gamma \vdash G_1|G_2} \qquad (RG4)\ \frac{\Gamma, x \vdash G}{\Gamma \vdash \nu x.G}$$

Axioms *(AG1)*, *(AG2)* and *(AG3)* define the associativity, commutativity and identity over *nil* for operation $|$, respectively. Axioms *(AG4)* and *(AG5)* state that the nodes of a graph can be hidden only once and in any order, and axioms *(AG6)* and *(AG7)* define alpha conversion of a graph with respect to its bounded names and the interplay between hiding and the operator for parallel composition, respectively.

Rule *(RG1)* creates a graph with no edges and n nodes and rule *(RG2)* creates a graph with n nodes and one edge labelled by L and with m tentacles (note that there can be repetitions among nodes in $\boldsymbol{y}$, i.e. some tentacles can be attached to the same node). Rule *(RG3)* allows to put together (using $|$) two graphs that share the same set of external nodes. Finally, rule *(RG4)* allows to hide a node from the environment.

If necessary, thanks to axiom *(AG4)*, we will write νX, with $X = \bigcup x_i$, to abbreviate $\nu x_1.\nu x_2 \ldots \nu x_n$. Note that using the axioms, for any judgement we can always have an equivalent normal form $\Gamma \vdash \nu X.G$, with G a subterm containing only composition of edges. It is clear from the above definitions that Γ and X can be made disjoint sets of nodes using the axioms and that $nodes(G) \subseteq (\Gamma \cup X)$.

The correspondence theorem expressing that well-formed syntactic judgements up to structural axioms are isomorphic to graphs up to isomorphism has been proved in [9].

We now introduce the notion of synchronized edge replacement. Synchronized edge replacement is obtained using graph rewriting combined with constraint solving. More specifically, we use *context-free* productions enriched with actions that are used to coordinate the simultaneous application of various productions.

The following definitions introduce synchronized edge replacement systems where actions can declare and refer to names as nodes and where names are bound via unification.

A *context-free edge replacement production* rewrites a single edge into an arbitrary graph. A production $p = (L \rightarrow R)$ can be applied to a graph G yielding H if there is an occurrence of an edge labeled by L in G. Graph H is obtained from G by removing the previously matched edge and by embedding a fresh copy of R in G by coalescing its external nodes with the corresponding attachment nodes of the replaced edge. This notion of edge replacement yields the basic steps in the derivation process of an edge replacement grammar.

To model synchronized rewriting, it is necessary to add some labels to the nodes in productions. Assuming to have a ranked alphabet *Act* of actions, then we associate actions to some of the attachment nodes of the left member of the production. In this way, each rewrite of an edge must synchronize actions with (a number of) its adjacent edges and then all the participants will have to move as well (how many depends on the synchronization policy). It is clear that coordinated rewriting will allow the propagation of synchronization all over the graph where productions are applied. Determining which productions can be synchronized at any given stage corresponds to solve a distributed constraint satisfaction problem [11].

A *synchronized edge replacement grammar*, or simply a grammar, consists of an initial graph and a set of productions. A derivation is obtained by starting with the initial graph and by executing a sequence of transitions, each obtained by synchronizing possibly several productions.

Now, for adding mobility to our model of computation we let a production to declare on each of its connecting nodes new names for the nodes it creates and to share these names and/or other existing names with the rest of the graph using the synchronization process. This is done in a production by adding to the action in a node a tuple of names that one wants to communicate. Therefore, the synchronization of a rewriting rule has to match not only actions, but also has to unify the tuples of names. After the productions are applied, the declared names that were unified are used to obtain the final graph by merging the corresponding nodes.

The expressive power of our model depends on the meaning of the names unified in the synchronization process. If these names correspond only to nodes newly generated in the productions, the expressive power is analogous to the π-I-calculus, where only extruded names can be transmitted. Instead, if also “old” nodes can be communicated, but not unified, we are analogous to the

π-calculus. If all types of nodes can be unified, the corresponding process algebra is the fusion calculus [15]. However we emphasize that in our model it is possible (and easy) to define multiple synchronizations,while the existing calculi are usually limited to binary synchronizations.

In this paper we handle for the first time the general case (with Milner synchronization style). The $\pi - I$-like case was defined in [8,10] while the intermediate case was presented in paper [9], which in fact includes the encoding of the π-calculus.

Below we define the transitions of a grammar as certain logical sequents. We exploit the previously introduced representation of graphs as syntactic judgements. Notice that no distinction is made between nodes and names.

Definition 3 (Transitions (with fusion)). *A transition has the form*

$$\Gamma \vdash G_1 \xrightarrow{\Lambda,\pi} \phi \vdash G_2$$

where:

1. $\Lambda : \Gamma \multimap\!\!\rightarrow (Act \times \mathcal{N}^*)$
2. $\pi : \Gamma \to \Gamma$ *and* $x \in \pi^{-1}(x)$
3. $n(\Lambda) = \{z \mid \exists x.\Lambda(x) = (a, \boldsymbol{y}), z \in Set(\boldsymbol{y})\}$
4. $\Delta = n(\Lambda) - \Gamma$
5. $\phi = \pi(\Gamma) \cup \Delta$

A transition says that G_1 is rewritten into G_2 satisfying a *set of requirements* Λ and a *fusion substitution* π. The free nodes of graph G_2 must include the free nodes of G_1 (after applying π) and those new nodes (Δ) that are used in synchronization. Note that ϕ is determined by the Γ and Λ of the same transition.

The set of requirements $\Lambda \subseteq \Gamma \times Act \times \mathcal{N}^*$ is defined as a partial function in its first argument, i.e. if $(x, a, \boldsymbol{y}) \in \Lambda$ we write $\Lambda(x) = (a, \boldsymbol{y})$ with $rank(a) = |\boldsymbol{y}|$. With $\Lambda(x) \uparrow$ we mean that the function is not defined for x, i.e. that there is no requirement in Λ with x as first argument. Function $set(\boldsymbol{y})$ returns the set of names in vector $\boldsymbol{y}$. The definition of Λ as a function means that all edges in G_1 attached to node x that are participating in a synchronization must satisfy the conditions of the corresponding synchronization algebra. The function is partial since not all nodes need to be loci of synchronization.

Fusion substitution π determines a partition of Γ where all nodes in an equivalence class are mapped to a representative element of the class. We use $x \mapsto y$ to denote the substitution mapping node x to y.

Definition 4 (Productions). *A* synchronized production*, or simply a production, is a special transition of the form,*

$$x_1, \ldots, x_n \vdash L(x_1, \ldots, x_n) \xrightarrow{\Lambda,\pi} \phi \vdash G$$

The context-free character of productions is here made clear by the fact that the graph to be rewritten consists of a single edge with distinct nodes. Productions combine the roles of prefix, sum and recursion in process calculi.

Renaming can be applied to productions in several ways: i) free names $x_1, \ldots, x_n$ can be changed throughout the sequent; ii) names declared in $\Delta = n(\lambda) - \Gamma$ can be α-converted; and iii) the representative names chosen by π can be consistently changed. Also *identity productions* of the form

$$x_1, \ldots, x_n \vdash L(x_1, \ldots, x_n) \xrightarrow{\emptyset, id} x_1, \ldots, x_n \vdash L(x_1, \ldots, x_n)$$

are always considered available.

Definition 5 (Grammars). *Let $\mathcal{N}$ be a fixed infinite set of names, LE a ranked alphabet of labels and Act a ranked set of actions. A* grammar *consists of an initial graph $\Gamma_0 \vdash G_0$ and a set $\mathcal{P}$ of productions on LE and Act.*

A derivation *is a finite or infinite sequence of the form $\Gamma \vdash G \xrightarrow{\Lambda_1, \pi_1} \phi_1 \vdash G_1 \xrightarrow{\Lambda_2, \pi_2} \ldots \xrightarrow{\Lambda_n, \pi_n} \phi_n \vdash G_n \ldots$, where $\Gamma \vdash G \xrightarrow{\Lambda_1, \pi_1} \phi_1 \vdash G_1$ and $\phi_{i-1} \vdash G_{i-1} \xrightarrow{\Lambda_i, \pi_i} \Gamma_i \vdash G_i$, $i = 2, \ldots, n$ are transitions in the set $T(\mathcal{P})$ of transitions generated by $\mathcal{P}$. Transitions $T(\mathcal{P})$ are generated by $\mathcal{P}$ applying the inference rules defined below.*

Definition 6 (Inference rules). *Let $\langle \Gamma \vdash G_0, \mathcal{P} \rangle$be a grammar. The set $T(\mathcal{P})$ of transitions is obtained from the productions P using the inference rules in Table 2 where the side conditions of the rules are:*

$$\psi_1 \overset{\text{def}}{\Longleftrightarrow} \begin{cases} \Delta \cap \sigma(\Gamma) = \emptyset \ \textit{and} \ \forall x \in \Delta . \sigma(x) = x \\ \sigma(x) = \sigma(y) \wedge \Lambda(x) \downarrow \wedge \Lambda(y) \downarrow \wedge x \neq y \Rightarrow \\ \quad (\forall z \notin \{x, y\} . \sigma(z) = \sigma(x) \Rightarrow \Lambda(z) \uparrow) \\ \quad \wedge \ \Lambda(x) = (a, \boldsymbol{v}) \wedge \Lambda(y) = (\overline{a}, \boldsymbol{w}) \wedge a \neq \tau \\ \rho = mgu(\{\sigma(\boldsymbol{v}) = \sigma(\boldsymbol{w}) | \sigma(x) = \sigma(y) \wedge \Lambda(x) = (a, \boldsymbol{v}) \wedge \Lambda(y) = (\overline{a}, \boldsymbol{w})\} \\ \qquad\qquad \cup \{\sigma(x) = \sigma(y) | \pi(x) = \pi(y)\}) \\ \Lambda'(z) = \begin{cases} (\tau, \langle\rangle), & \textit{if } \sigma(x) = \sigma(y) = z \wedge x \neq y \wedge \Lambda(x) \downarrow \wedge \Lambda(y) \downarrow \\ \rho(\sigma(\Lambda))(z), & \textit{otherwise} \end{cases} \\ \pi'(\sigma(x)) = \rho(\sigma(\pi(x))) \\ \boldsymbol{u} = \rho(\sigma(\phi)) - \phi' \end{cases}$$

$$\psi_2 \overset{\text{def}}{\Longleftrightarrow} \begin{cases} (\pi(x) = \pi(y) \wedge x \neq y) \Rightarrow \pi(x) \neq x \\ \Lambda(x) \uparrow \ \textit{or} \ \Lambda(x) = (\tau, \langle\rangle), \\ \Lambda' = \Lambda - (x, \tau, \langle\rangle) \\ \boldsymbol{z} = \phi - \phi' \end{cases}$$

Rule (*par*) simply combines together two disjoint judgements.

Rule (*merge*) is the rule for synchronization. The rule states that in a transition it is possible to merge two nodes x and y that offer complementary non-silent actions (conditions on σ). Here ρ is the most general unifier that fuse the corresponding names of the actions and propagates the previous fusions (determined by π). The label Λ' takes into account all possible synchronizations and leaves unchanged the actions offered on the other nodes up to the necessary fusions (ρ and σ). The new fusion substitution π' acts on $\sigma(\Gamma)$ by applying to it the mgu ρ. Finally, the names in ϕ after the fusion which are not present in

Table 2. Inference rules for graph synchronization

$$(par)\ \frac{\Gamma \vdash G_1 \xrightarrow{\Lambda,\pi} \phi \vdash G_2 \qquad \Gamma' \vdash G'_1 \xrightarrow{\Lambda',\pi'} \phi' \vdash G'_2}{\Gamma,\Gamma' \vdash G_1|G'_1 \xrightarrow{\Lambda\cup\Lambda',\pi\cup\pi'} \phi,\phi' \vdash G_2|G'_2} \qquad \text{where } \Gamma\cap\Gamma'=\emptyset$$

$$(merge)\ \frac{\Gamma \vdash G_1 \xrightarrow{\Lambda,\pi} \phi \vdash G_2}{\sigma\Gamma \vdash \sigma G_1 \xrightarrow{\Lambda',\pi'} \phi' \vdash \nu\ \boldsymbol{u}.\rho(\sigma(G_2))} \qquad \text{where } \psi_1 \text{ holds}$$

$$(res)\ \frac{\Gamma, x \vdash G_1 \xrightarrow{\Lambda,\pi} \phi \vdash G_2}{\Gamma \vdash \nu\ x.G_1 \xrightarrow{\Lambda',\pi|_\Gamma} \phi' \vdash \nu\ \boldsymbol{z}.G_2} \qquad \text{where } \psi_2 \text{ holds}$$

$\phi' = \pi'(\Gamma) \cup (n(\Lambda') - \sigma(\Gamma))$ are restricted; this corresponds to the close rule of the π-calculus.

Rule (res) deals with node restriction. According to the first condition, the restricted node must not be the representative element of its equivalence class induced by π when this class contains nodes different from x. Furthermore, only nodes can be restricted where either no action or only synchronization actions take place. If this conditions hold, Λ' is obtained by hiding the (possible) silent action on x and restricting all the nodes that are not in ϕ'. Notice that ϕ' is defined as usual as $\phi = \pi(\Gamma) \cup \Delta$, with $\Delta = n(\Lambda') - \Gamma$.

3 Ambient Calculus

In this section we apply our graph synchronization framework to the Ambient calculus [2] that is considered one of the most suitable calculi for representing wide area network computations. First we give syntax and semantics of the Ambient calculus and then its representation in terms of graphs is specified.

3.1 The Calculus

The syntax and the reduction semantics of Ambient calculus [2] is given below. The calculus relies on the notion of *ambient* that can be thought of as a bounded environment where processes interact. An ambient has a name, a collection of local agents and a collection of subambients. Ambients can be moved as a whole under the control of agents which are confined to ambients. Processes use *capabilities* for controlling interaction. We do not consider synchronization and restriction, and replication is replaced by (guarded) recursion.

Definition 7 (Syntax). *Let N be an infinite set of names ranged over by $a, b, c, \ldots,\ n, m, p, r\ldots$; let $X, Y, Z, \ldots$ be process variables.*

$$\begin{aligned} M \quad &::= \mathrm{in}\ n \mid \mathrm{out}\ n \mid \mathrm{open}\ n \\ P, Q &::= \mathsf{0} \mid n[P] \mid M.P \mid P|Q \mid rec\, X.\, P \mid X \end{aligned}$$

We assume that X is guarded *by M in $rec\,X.\,P$.*
We denote with Proc the set of the Ambient calculus processes.

Capabilities M are the usual Ambient calculus capabilities: *in* n allows to drive an ambient inside an ambient named n; dually, *out* n allows to exit an ambient n; *open* n dissolves an ambient n.

A process is the void process $\mathtt{0}$, a process $n[P]$ obtained by wrapping P in an ambient n, a *sequential* process $M.P$, the parallel composition of two processes $P|Q$, the recursive process $rec\,X.\,P$ or a process variable X.

Definition 8 (Structural equivalence). *The semantics of the Ambient calculus relies on the structural equivalence defined by the following rules:*

1. **The parallel operator** $\_|\_$ *is associative, commutative and* $\mathtt{0}$ *is its identity;*
2. $\dfrac{P \equiv Q}{M.P \equiv M.Q} \qquad \dfrac{P \equiv Q}{n[P] \equiv n[Q]}$
3. $rec\,X.\,P \equiv rec\,Y.\,P\{Y/X\}$*, if* $Y \notin fv(P)$*;*
4. $rec\,X.\,P \equiv P\{rec\,X.\,P/X\}$*.*

The usual algebraic properties of the parallel composition and the $\mathtt{0}$ process are assumed (rule 1); rule 2 guarantees that structural equivalence is preserved by capabilities and ambient processes; the process variable X is bound in $rec\,X.\,P$ and may be renamed (rule 3); finally, rule 4 is the analogous of the usual structural rule for replication (namely, $!P \equiv P \mid !P$).

Definition 9 (Reduction Semantics). *The reduction relation* $\rightarrow\ \subseteq Proc \times Proc$ *is the relation inductively generated by the axioms and rules in table 3 and closed under the structural equivalence given in Definition 8:*

Table 3. Ambient calculus reduction relation

$$m[n[out\ m.P \mid Q] \mid R] \rightarrow n[P \mid Q] \mid m[R]$$

$$n[in\ m.P \mid Q] \mid m[R] \rightarrow m[n[P \mid Q] \mid R]$$

$$open\ n.P \mid n[Q] \rightarrow P \mid Q$$

$$\frac{P \rightarrow Q}{P \mid R \rightarrow Q \mid R} \qquad \frac{P \rightarrow Q}{n[P] \rightarrow n[Q]}$$

The first two axioms in Table 3 state that an ambient n can be driven by a sequential process inside it to exit the wrapping ambient ($out\ m.P$) or to enter a parallel ambient m ($in\ m.P$). The third axiom is relative to the *open* n capability: an ambient may be dissolved by an external process. Note that all the capabilities are “asynchronous”, in the sense that the only condition under which they can be fired is the presence of a particular ambient.

3.2 Graph Representation of Ambient Calculus

We now show how it is possible to translate the Ambient calculus in our graph synchronization framework maintaining the semantics of processes.

Definition 10 (Translation).

$$\begin{array}{rll} [\![\, \mathtt{0} \,]\!]_x = x \vdash nil & & \\ [\![\, n[P] \,]\!]_x = x \vdash \nu\, y.(G \mid n(y,x)), & & \text{if } \; y \neq x \wedge [\![\, P \,]\!]_y = y \vdash G \\ [\![\, M.P \,]\!]_x = x \vdash L_{M.P}(x) & & \\ [\![\, P_1 | P_2 \,]\!]_x = x \vdash G_1 \mid G_2, & & \text{if } \; [\![\, P_i \,]\!]_x = x \vdash G_i, \text{where } i = 1,2 \\ [\![\, rec\, X.\, P \,]\!]_x = [\![\, P[rec\, X.\, P/X] \,]\!]_x & & \end{array}$$

Definition 10 introduces the mapping function $[\![\, P \,]\!]_x$ that returns a graph whose only free node x corresponds to the root of the ambient process P.

In the above translation, sequential processes $M.P$ are directly represented by edges labelled by $M.P$. While this introduces an infinite number of labels, it is easy to see that only a finite number of them (and of the corresponding activity rules defined below) is needed to derive all computations of any particular ambient.

The graph associated to the $\mathtt{0}$ process is an isolated node. The graph of $n[P]$ with free node x is obtained by constructing the graph of P on node y, attaching it to the graph $n(y,x)$ and restricting y; note that the ambient name n is interpreted as an edge from y to x labelled n. Ambient names N and sequential processes are the only edge labels.

The parallel composition $P_1 \mid P_2$ is obtained by making the graph of P_1 and P_2 to share their root node x; finally, recursive processes are unfolded first[1].

The given translation is injective but not surjective. However, the graphs $[\![\, P \,]\!]_x$ in the image of the translation function can be characterized as follows.

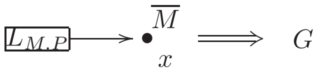
Definition 11 (Ambient graphs). *An* ambient graph *is a graph labeled on* $LE = \{L_{M.P} | M.P \in \mathit{Proc} \text{ is sequential}\} \cup N$ *which*

1. *is acyclic;*
2. *every node has at most one outgoing edge labelled in* N*;*
3. *there is one* root *node with no outgoing edges.*

Theorem 1. $[\![\, \_ \,]\!]_x$ *is a bijection on ambient graphs.*

We now define the productions of our version of the Ambient calculus. There are two kinds of productions: *activity productions*, relative to sequential processes, and *coordination productions* that corresponds to ambients.

Definition 12 (Activity productions). *The activity productions have the following form.*



$$x \vdash L_{M.P}(x) \xrightarrow{\{(x,\overline{M},\langle\rangle)\}} [\![\, P \,]\!]_x$$

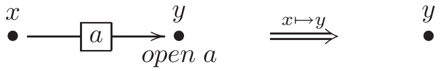
[1] Note that the $[\![\, \_ \,]\!]_x$ is well defined because recursion variables are guarded by capabilities.

Activity productions determine the actions that sequential processes are able to perform. In our approach, sequential processes become edge labels: when an action is performed, an edge labelled by $M.P$ is rewritten as the graph corresponding to P.

The complementary actions to synchronize the activity productions must be offered by ambients; more precisely, ambients must signal their existence emitting the complementary actions on their attaching nodes and, in this manner, performing the correct synchronized steps.

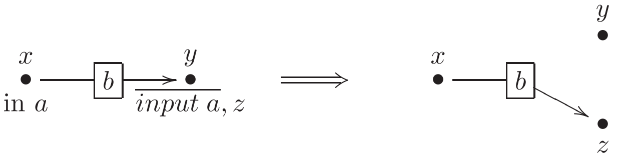
Definition 13 (Coordination productions). *Coordination productions are as follows.*

(open)



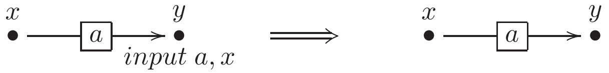
$$x, y \vdash a(x, y) \xrightarrow{\{(y, \mathit{open}\ a, \langle\rangle)\}[x \mapsto y]} y \vdash \mathit{nil}$$

($\mathit{input}1$)



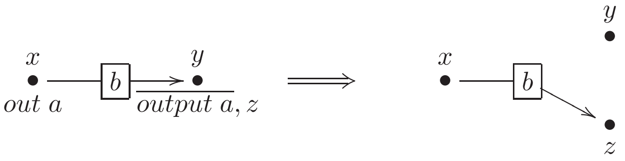
$$x, y \vdash b(x, y) \xrightarrow{\{(x, \mathit{in}\ a, \langle\rangle), (y, \overline{\mathit{input}\ a}, \langle z\rangle)\}} x, y, z \vdash b(x, z)$$

($\mathit{input}2$)



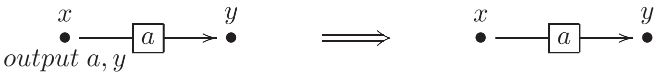
$$x, y \vdash a(x, y) \xrightarrow{\{(y, \mathit{input}\ a, \langle x\rangle)\}} x, y \vdash a(x, y)$$

($\mathit{output}1$)



$$x, y \vdash b(x, y) \xrightarrow{\{(x, \mathit{out}\ a, \langle\rangle), (y, \overline{\mathit{output}\ a}, \langle z\rangle)\}} x, y, z \vdash b(x, z)$$

($\mathit{output}2$)



$$x, y \vdash a(x, y) \xrightarrow{\{(x, \mathit{output}\ a, \langle y\rangle)\}} x, y \vdash a(x, y)$$

For every production, we give both the sequent and its graphical representation. In the latter, left and right members of a production are drawn in the style of Definition 10, but without restricted nodes. When $(x, \mu, \langle y \rangle) \in \Gamma$, node x in the right member is labeled by x, μ. Coordination productions define the complementary actions that ambients must perform in order to synchronize themselves with sequential processes.

The (*open*) production states that if the ambient a has a parallel process that wants to open it, then the edge corresponding to a disappears and x is fused with y.

Production(*input*1) asserts that is a process inside b wants to drive b in an ambient a, then the destination of b will become the new node z. On the other hand, production (*input*2) controls the entrance of an external process in a: this production simply passes the source x of a to the entering process.

Analogously to the input productions, (*output*1) and (*output*2) take care of the output action. We remark that (*output*1) acts quite similarly to (*input*1).

Definition 14 (Basic transition). *A transition* $\Gamma \vdash G \xrightarrow{\Lambda,\pi} \phi \vdash G'$ *is* basic *if:*

- π *is the identity function on* Γ*;*
- *its proofs uses exactly one instance of either* (*open*) *or* (*input*1) *or* (*output*1)*;*
- Λ *is either a singleton* $\{(x, \tau\langle\rangle)\}$ *or it is empty.*

Theorem 2. *For all ambient processes* $P, Q \in Proc$*:*

- *if* $P \to Q$ *then* $[\![P]\!]_x \xrightarrow{\Lambda,\mathrm{id}} [\![Q]\!]_x$ *and either* $\Lambda = \emptyset$ *or* $\Lambda = \{(x, \tau, \langle\rangle)\}$*;*
- *if* $[\![P]\!]_x \xrightarrow{\Lambda,\pi} \phi \vdash G$ *is a basic transition, then* $\phi \vdash G = [\![Q]\!]_x$ *and* $P \to Q$*.*

Proof (sketch): The proof of the theorem is based on the fact that if a basic derivation $[\![P]\!]_x \xrightarrow{\Lambda,\mathrm{id}}$ exists, then it is possible to derive the same transition by

1. applying instances of the (*par*) rule to a suitable set of productions;
2. applying the (*merge*) rule in such a way that the graph relative to P without restrictions is obtained;
3. restricting by means of rule (*res*) all the nodes that are not the root node of P.

It is easy to note that all the reductions that do not take place at the top level of P correspond to basic transitions of the graph whose Λ is empty, while the transitions that involve subprocesses of P at the top level have $\Lambda = \{(x, \tau, \langle\rangle)\}$.

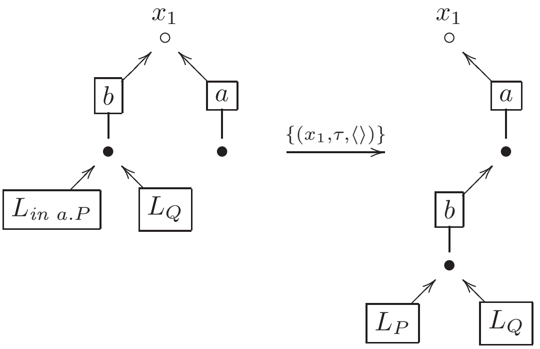
3.3 Example

As an example we show the correspondence between an Ambient calculus reduction and the corresponding graph transition. Let us consider the ambient reduction

$$b[in\ a.P \mid Q] \mid a[0] \to a[b[P \mid Q]]$$

where P and Q are sequential processes.

Following Definition 10 and Theorem 2, we should obtain



(1)

The picture on the left is the graphical representation of $[\![\ b[in\ a.P \mid Q] \mid a[0]\ ]\!]_{x_1}$, while the rightmost picture is $[\![\ a[b[P \mid Q]]\ ]\!]_{x_1}$ (we represent the restricted nodes with • and the free nodes with ◦).

The steps described in the proof sketch of Theorem 2 guide us in applying the productions (activity and coordination) and the inference rules of Table 2 in order to construct a proof for transition (1).

First (step 1) we decompose the graph in its elementary edges and determine the productions that correspond to the elementary components of the transition.

$$x_1, y_1 \vdash b(y_1, x_1) \xrightarrow{\left\{\begin{array}{l}(x_1, \overline{input\ a}, \langle z_1\rangle),\\ (y_1, in\ a, \langle\rangle)\end{array}\right\},\mathrm{id}} x_1, y_1, z_1 \vdash b(y_1, z_1) \tag{2}$$

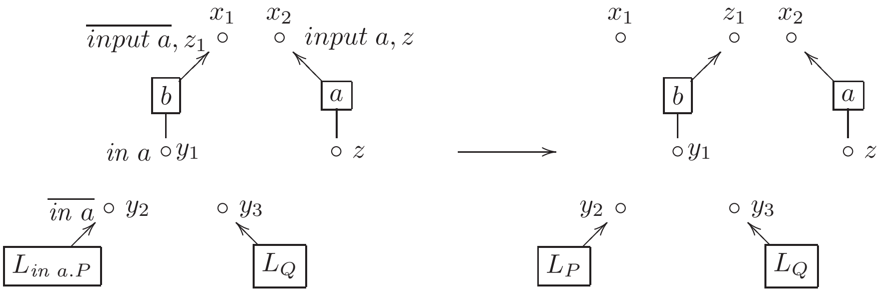
$$y_2 \vdash L_{in\ a.P}(y_2) \xrightarrow{\{(y_2,\overline{in\ a},\langle\rangle)\},\mathrm{id}} y_2 \vdash L_P(y_2) \tag{3}$$

$$x_2, z \vdash a(z, x_2) \xrightarrow{\{(x_2,input\ a,\langle z\rangle)\},\mathrm{id}} x_2, z \vdash a(z, x_2) \tag{4}$$

$$y_3 \vdash L_Q(y_3) \xrightarrow{\emptyset,\mathrm{id}} y_3 \vdash L_Q(y_3) \tag{5}$$

Transitions (2) and (4) are instances of the coordination productions (*input*1) and (*input*2), respectively; transition (3) is the activity production relative to *in a.P* and transition (5) is the identity transition that leaves L_Q idle.

The graphical representation is:



The previous graph represents the transition obtained by applying the (*par*) rule to the productions (2), (3), (4) and (5). Let

$$\begin{array}{ll} G_1 &= b(y_1, x_1) \mid a(z, x_2) \mid L_{in\ a.P}(y_2) \mid L_Q(y_3) \\ G_2 &= b(y_1, z_1) \mid a(z, x_2) \mid L_P(y_2) \mid L_Q(y_3) \\ \Gamma &= \{x_1, x_2, y_1, y_2, y_3, z\} \end{array}$$

then, in terms of sequents we have:

$$\Gamma \vdash G_1 \xrightarrow{\left\{\begin{array}{l}(x_1, \overline{input\ a}, \langle z_1\rangle),\\ (x_2, input\ a, \langle z\rangle)\\ (y_1, in\ a, \langle\rangle)\\ (y_2, \overline{in\ a}, \langle\rangle)\end{array}\right\},\mathrm{id}} \Gamma, z_1 \vdash G_2 \tag{6}$$

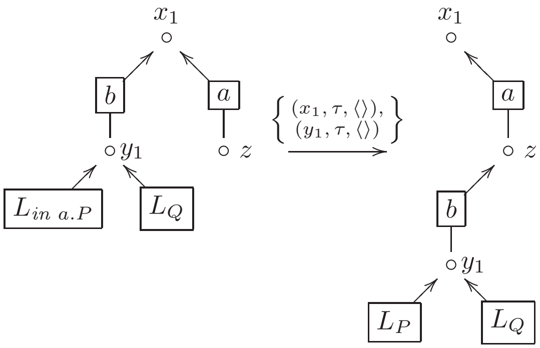
The application of the merge rule (step 2) provides the fusion of the nodes in order to obtain a graph of the same shape of the ambient process but without restricted nodes. Referring to the rule (*merge*), let σ the function that behaves as the identity on all nodes different from x_2, y_2 and y_3 and

$$\sigma : \left\{\begin{array}{l} x_2 \mapsto x_1 \\ y_2 \mapsto y_1 \\ y_3 \mapsto y_1 \end{array}\right.$$

that determines $\Lambda' = \{(x_1, \tau, \langle\rangle), (y_1, \tau, \langle\rangle)\}$ and $\rho : z_1 \mapsto z$. The rule (*merge*) may be applied to transition (6) obtaining the transition

$$x_1, y_1, z \vdash \sigma(G_1) \xrightarrow{\left\{\begin{array}{l}(x_1, \tau, \langle\rangle),\\ (y_1, \tau, \langle\rangle)\end{array}\right\},\mathrm{id}} x_1, y_1, z \vdash \rho(\sigma(G_2))$$

that is graphically represented as


x_1
b
a
y_1
z
$L_{in\ a.P}$
L_Q
$\left\{\begin{array}{l}(x_1, \tau, \langle\rangle),\\ (y_1, \tau, \langle\rangle)\end{array}\right\}$
x_1
a
z
b
y_1
L_P
L_Q

We remark that the above transition requires a synchronization involving three edges and two nodes: the edges relative to *in a.P* and b that synchronize on node y_1, and the edges relative to ambients b and a that synchronize on node x_1. This makes clear that the *in* capability of ambients requires to synchronize three components (the *out* capability is analogous).

Finally, two applications of the (*res*) rule (step 3) are needed in order to restrict nodes z and y_1. This concludes the proof of the transition.

4 Conclusion

In the paper we presented a simple LTS semantics of Cardelli and Gordon's Ambient calculus exploiting a graphical model based on edge replacement and local synchronization with mobility. While the correspondence with the original operational semantics of ambients is shown for a restricted class of graphs (the ambient graphs), it is also conceivable to lift this limitation and to allow all graphs on the same edge labels. Coordination productions should be exactly the same, while activity productions should be allowed to rewrite an edge into any such graph. The resulting calculus should allow programmable ambient mobility on any graphical model of internetworks, providing a more realistic description of real systems.

The work presented here is still at an initial stage. The labeled transition system defined by the logical sequents in the paper automatically provides an abstract semantics of ambients under the usual definition of bisimilarity. Since all nodes in ambient graphs are restricted except for the root, interactions can only be observed there. This should respect the intuition of abstract semantics of ordinary ambients, where barb observation and ambient composition are via the root. However we did not yet study the relation of our abstract semantics with Cardelli and Gordon's, and we do not know if it is a congruence, i.e. if it is respected by our operations of composition and restriction of graphs. Finally, we would like to experiment with network reconfiguration techniques more general than ambients, but still realistic for actual internetworks, taking advantage of our general approach.

References

1. Luca Cardelli and Andrew D. Gordon. A commitment relation for the ambient calculus. Manuscript. 2
2. Luca Cardelli and Andrew D. Gordon. Mobile ambients. *TCS: Theoretical Computer Science*, 240, 2000. 1, 8
3. Ilaria Castellani and Ugo Montanari. Graph Grammars for Distributed Systems. In Hartmut Ehrig, Manfred Nagl, and Grzegorz Rozenberg, editors, *Proc. 2nd Int. Workshop on Graph-Grammars and Their Application to Computer Science*, volume 153 of *Lecture Notes in Computer Science*, pages 20–38. Springer-Verlag, 1983. 3
4. Rocco De Nicola, Gianluigi Ferrari, and Rosario Pugliese. KLAIM: A kernel language for agents interaction and mobility. *IEEE Transactions on Software Engineering*, 24(5):315–330, May 1998. Special Issue: Mobility and Network Aware Computing. 1
5. P. Degano and Ugo Montanari. A model of distributed systems based of graph rewriting. *Journal of the ACM*, 34:411–449, 1987. 3
6. Gianluigi Ferrari, Carlo Montangero, Laura Semini, and Simone Semprini. Mobile agents coordination in Mob$_{adtl}$. In Antonio Porto and Gruia-Catalin Roman, editors, *Coordination Languages and Models*, volume 1906 of *LNCS*. Springer Verlag, 2000. 1

7. Cedric Fournet and George Gonthier. The reflexive CHAM and the join-calculus. In *Conference Record of POPL '96: The* $23^{\rm rd}$ *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 372–385, St. Petersburg Beach, Florida, 21–24 January 1996. 1
8. Dan Hirsch, Paola Inverardi, and Ugo Montanari. Reconfiguration of software architecture styles with name mobility. In Antonio Porto and Gruia-Catalin Roman, editors, *Coordination 2000*, volume 1906 of *LNCS*, pages 148–163. Springer Verlag, 2000. 3, 6
9. Dan Hirsh and Ugo Montanari. Synchronized hyperedge replacement with name mobility. In *To appear in CONCUR01*, 2001. 3, 5, 6
10. Barbara Koenig and Ugo Montanari. Observational equivalence for synchronized graph rewriting. In *Proc. TACS'01*, LNCS. Springer Verlag, 2001. To appear. 3, 6
11. Ugo Montanari and Francesca Rossi. Graph rewriting and constraint solving for modelling distributed systems with synchronization. In P. Ciancarini and C. Hankin, editors, *Proceedings of the First International Conference COORDINATION '96, Cesena, Italy*, volume 1061 of *LNCS*. Springer Verlag, April 1996. 3, 5
12. Gian Pietro Picco, Amy L. Murphy, and Gruia-Catalin Roman. Lime: Linda Meets Mobility. In D. Garlan, editor, *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)*, pages 368–377, Los Angeles, CA, USA, May 1999. ACM Press. Also available as Technical Report WUCS-98-21, July 1998, Washington University in St. Louis, MO, USA. 1
13. Gruia-Catalin Roman, Peter J. McCann, and J. Y. Plunn. Mobile UNITY: Reasoning and specification in mobile computing. *ACM Transactions on Software Engineering and Methodology*, 6(3):250–282, July 1997. 1
14. Peter Sewell. From rewrite rules to bisimulation congruences. *Lecture Notes in Computer Science*, 1466, 1998. 2
15. Bjorn Victor and Joachim Parrow. Concurrent constraints in the fusion calculus. *Lecture Notes in Computer Science*, 1443, 1998. 6

Filter Models and Easy Terms*

Fabio Alessi[1], Mariangiola Dezani-Ciancaglini[2], and Furio Honsell[1]

[1] Dipartimento di Matematica ed Informatica, Università di Udine
Via delle Scienze 208, 33100 Udine, Italy
{alessi,honsell}@dimi.uniud.it

[2] Dipartimento di Informatica, Università di Torino
Corso Svizzera 185, 10149 Torino, Italy
dezani@di.unito.it

Abstract. We illustrate the use of intersection types as a tool for synthesizing λ-models which exhibit special purpose features. We focus on semantical proofs of *easiness*. This allows us to prove that the class of λ-theories induced by *graph models* is strictly included in the class of λ-theories induced by *non-extensional filter models*.

Introduction

Intersection types were introduced in the late 70's by Dezani and Coppo [10,12,6], to overcome the limitations of Curry's type discipline. They are a very expressive type language which allows to describe and capture various properties of λ-terms. For instance, they have been used in [29] to give the first type theoretic characterization of *strongly normalizable* terms and in [13] to capture *persistently normalizing terms* and *normalizing terms*. See [15] for a more complete account of this line of research.

Intersection types have a very significant realizability semantics with respect to applicative structures. This is a generalization of Scott's natural semantics [31] of simple types. According to this interpretation types denote subsets of the applicative structure, an arrow type $A \to B$ denotes the sets of points which map all points belonging to the interpretation of A to points belonging to the interpretation of B, and an intersection type $A \cap B$ denotes the intersections of the interpretation of A and the interpretation of B. Building on this, intersection types have been used in [6] to give a proof of the completeness of the natural semantics of Curry's simple type assignment system in applicative structures, introduced in [31]. See [14] for a more complete treatment of completeness of intersection type assignment systems.

But intersection types have also an alternative semantics based on *duality* which is related to Abramsky's *Domain Theory in Logical Form* [1]. Actually it amounts to the application of that paradigm to the special case of ω-algebraic complete lattice models of pure lambda calculus, [11]. Namely, types correspond

* Partially supported by MURST Cofin '99 TOSCA Project, CNR-GNSAGA and FGV '99.

A. Restivo, S. Ronchi Della Rocca, L. Roversi (Eds.): ICTCS 2001, LNCS 2202, pp. 17–37, 2001.

to *compact elements*: the type Ω denoting the least element, intersections denoting *joins* of compact elements, and arrow types denoting *step functions* of compact elements. A typing judgment then can be interpreted as saying that a given term belongs to a pointed compact open set in a ω-algebraic complete lattice model of λ-calculus. By duality, type theories give rise to *filter λ-models*. Intersection type assignment systems can then be viewed as *finitary logical* definitions of the interpretation of λ-terms in such models, where the meaning of a λ-term is the set of types which are deducible for it.

This duality lies at the heart of the success of intersection types as a powerful tool for the analysis of λ-models, see *e.g.* [2,6,11,13,3,17,21,16,28,19,30].

In this paper we illustrate the use of intersection types as a tool for synthesizing λ-models which exhibit special purpose features. For building our models we will introduce a strengthened version of intersection type theories, namely the *easy* ones. We focus on semantical proofs of *easiness* [23], [5] (Definition 15.3.8) (we recall that a closed term P is *easy* if, for any other closed term M, the theory $\lambda\beta + \{M = P\}$ is consistent). More specifically we will consider the terms $\omega_2\omega_2$ and $\omega_3\omega_3\mathsf{I}$ where $\omega_2 \equiv \lambda x.xx$, $\omega_3 \equiv \lambda x.xxx$ and I is the identity combinator. Let P be $\omega_2\omega_2$ or $\omega_3\omega_3\mathsf{I}$. For any closed term M we will build a non-trivial filter model (that is a non-trivial λ-model built on intersection type theories) where the interpretations of M and P coincide. From this fact it follows that the theory $\lambda\beta + \{P = M\}$ is consistent, hence P is easy. A feature of the present model construction is that with very small changes we can show the consistency of $\lambda\beta\eta + \{P = M\}$ for $P \equiv \omega_2\omega_2$ and $P \equiv \omega_3\omega_3\mathsf{I}$.

The easiness of both $\omega_2\omega_2$ (see [23], [26]) and $\omega_3\omega_3\mathsf{I}$ (see [24], [9]) have been shown by syntactic arguments. These and other easiness results have been mainly obtained either using the "Jacopini technique" [23], [27] (Section 4.4.4), [26], or using Church-Rosser relations which extend $\lambda\beta$ [22], [8], [9]. Actually a semantic proof of the easiness of $\omega_2\omega_2$ has already appeared in the literature [4] with a proof based on non-standard $\mathcal{P}(\omega)$ models. Moreover [20] builds extensional filter models equating $\omega_2\omega_2$ to arbitrary closed terms. See also [33].

One of the by-product of this paper is that it provides a negative answer to an interesting question, namely whether the class of λ-theories induced by *graph models*, (e.g. Scott, Park and Engeler models are instances of graph models), coincide with the class of λ-theories induced by *non-extensional filter models* as defined in [16]. In [25] it was shown that the equation of $\omega_3\omega_3\mathsf{I} = \mathsf{I}$ cannot be proved in any graph model, whilst our paper shows that this is possible with non-extensional filter models. Hence the two mentioned classes differ (actually [16] shows that graph model λ-theories are included in non-extensional filter model λ-theories).

It is an open question of this paper to single out which classes of λ-terms can be proved easy using filter models as semantical tools.

Also a "philosophic" question arises: a part from classical semantics tools, such as Fixed Point Induction, what do we gain when we have an ordered cpo model for a λ-theory? At present, we have not a clear cut answer to such question.

The present paper is organized as follows. In Section 1 we present easy intersection type theories and type assignment systems for them. We prove some meta-theoretic properties including a Generation Theorem. In Section 2 we introduce λ-models based on spaces of filters in easy intersection type theories. Sections 3 and 4 exhibit the mentioned models which allow to prove easiness of $\omega_2\omega_2$ and $\omega_3\omega_3\mathsf{I}$. Section 5 discusses the extensional versions of the above models.

1 Intersection Type Assignment Systems

Intersection types are syntactical objects built inductively by closing a given set $\mathbb{C}$ of *type atoms* (constants) under the *function type* constructor $\to$ and the *intersection* type constructor $\cap$. In this paper we only need to consider intersection types which contain the universal type Ω and possibly the "isolated" type ι in their set of atoms.

Definition 1 (Intersection Type Language).
*Let $\mathbb{C}$ be a countable set of constants such that $\Omega \in \mathbb{C}$. The Ω-*intersection type language *over $\mathbb{C}$, denoted by $\mathbb{T} = \mathbb{T}(\mathbb{C})$ is defined by the following abstract syntax:*

$$\mathbb{T} = \mathbb{C} \mid \mathbb{T} \to \mathbb{T} \mid \mathbb{T} \cap \mathbb{T}. \;\square$$

Notice that the most general form of an intersection type is a finite intersection of arrow types and type constants.

Notation. Upper case Roman letters, i.e. $A, B, \ldots$, will denote arbitrary types. Greek letters will denote constants in $\mathbb{C}$. When writing intersection types we shall use the following convention: the constructor $\cap$ takes precedence over the constructor $\to$ and it associates to the right.

Much of the expressive power of intersection type disciplines comes from the fact that types can be endowed with a *preorder relation* $\leq$, which induces the structure of a meet semi-lattice with respect to $\cap$, the top element being Ω.

The notion we introduce of *easy* intersection type theory is new in the literature. It is tailored in order both to include the type theories of Sections 3, 4, 5 and to have easier proofs of the following Theorems 1, 2, 3. We refer the interested reader to [7] for the general definition of intersection type theory.

$$\begin{array}{rlrl}
(\mathrm{refl}) & A \leq A & (\mathrm{idem}) & A \leq A \cap A \\[1ex]
(\mathrm{incl}_L) & A \cap B \leq A & (\mathrm{incl}_R) & A \cap B \leq B \\[1ex]
(\mathrm{mon}) & \dfrac{A \leq A' \quad B \leq B'}{A \cap B \leq A' \cap B'} & (\mathrm{trans}) & \dfrac{A \leq B \quad B \leq C}{A \leq C} \\[1ex]
(\Omega) & A \leq \Omega & (\Omega\text{-}\eta) & \Omega \leq \Omega \to \Omega \\[1ex]
(\to\text{-}\cap) & (A \to B) \cap (A \to C) \leq A \to B \cap C & (\eta) & \dfrac{A' \leq A \quad B \leq B'}{A \to B \leq A' \to B'}
\end{array}$$

Fig. 1. The set ∇_0 of axioms and rules

Definition 2 (Easy intersection type theories).
Let $\mathbb{T} = \mathbb{T}(\mathbb{C})$ be an Ω-intersection type language. The easy intersection type theory *(*eitt *for short) $\Sigma(\mathbb{C}, \nabla)$ over $\mathbb{T}$ is the set of all judgments $A \leq B$ derivable from ∇, where ∇ is a collection of axioms and rules such that (we write $A \sim B$ for $A \leq B$ & $B \leq A$):*

1. *∇ contains the set ∇_0 of axioms and rules of Figure 1;*
2. *further axioms can be of the following three shapes only:*

$$\psi \leq \psi', \psi \sim (\phi \to A), \psi \sim (\phi \to \zeta) \cap (\xi \to A),$$

   where $\psi, \psi', \phi, \zeta, \xi \in \mathbb{C} \setminus \{\iota\}$, $A \in \mathbb{T}$, and $\psi, \psi' \not\equiv \Omega$;
3. *∇ does not contain further rules;*
4. *for each $\psi \not\equiv \Omega, \iota$ there is exactly one axiom in ∇ of the shape $\psi \sim A$;*
5. *let ∇ contain $\psi \sim A$ and $\psi' \sim A'$ where either $A \equiv \phi \to \zeta$ or $A \equiv (\phi \to \zeta)$ $\cap(\xi \to B)$. Then ∇ contains also $\psi \leq \psi'$ iff $A' \equiv \phi' \to \zeta'$ and ∇ contains both $\phi' \leq \phi$ and $\zeta \leq \zeta'$.*

Notice that:
(a) since $\Omega \sim \Omega \to \Omega \in \Sigma(\mathbb{C}, \nabla)$ by axioms (Ω) and (Ω-η), it follows that all atoms in $\mathbb{C}$ different from ι are equivalent to suitable (intersections of) arrow types;
(b) ∇ cannot contain axioms of the shape $\iota \leq A$ or $A \leq \iota$: this justifies the label "isolated" for ι;
(c) associativity and commutativity of $\cap$ (modulo $\sim$) follow easily from the axioms and rules of ∇_0 as defined in Definition 2.

A consequence of (a) is that an eitt induces an extensional λ-model iff its set of constants does not contain ι: this will be proved in Theorem 3.

Notation.
When we consider an eitt $\Sigma(\mathbb{C}, \nabla)$, we will write $\mathbb{C}^\nabla$ for $\mathbb{C}$, $\mathbb{T}^\nabla$ for $\mathbb{T}(\mathbb{C})$ and Σ^∇ for $\Sigma(\mathbb{C}, \nabla)$. Moreover $A \leq_\nabla B$ will be short for $(A \leq B) \in \Sigma^\nabla$ and $A \sim_\nabla B$ for $A \leq_\nabla B \leq_\nabla A$. We will consider syntactic equivalence "$\equiv$" of types up to associativity and commutativity of $\cap$. We will write $\bigcap_{i \leq n} A_i$ for $A_1 \cap \ldots \cap A_n$. Similarly we will write $\bigcap_{i \in I} A_i$, where I denotes always a finite non-empty set. □

Theorem 1 gives useful properties of eitt's.

Theorem 1.
For all I, and $A_i, B_i, C, D \in \mathbb{T}^\nabla$,

$$\begin{array}{l} \bigcap_{i \in I}(A_i \to B_i) \leq_\nabla C \to D \ \& \ D \not\sim_\nabla \Omega \Rightarrow \\ \exists J \subseteq I.\ C \leq_\nabla \bigcap_{i \in J} A_i \ \& \ \bigcap_{i \in J} B_i \leq_\nabla D. \end{array}$$

Proof. In this proof we assume that ψ, ϕ, ξ denote constants different from ι. Let $A(\cap\iota) \leq_\nabla B(\cap\iota)$ be short for $A \leq_\nabla B$ or $A \cap \iota \leq_\nabla B$ or $A \cap \iota \leq_\nabla B \cap \iota$, where $A, B \not\leq_\nabla \iota$. Notice that we cannot have $A \leq_\nabla B \cap \iota$ and $A \not\leq_\nabla \iota$.

By assumption for each constant ψ there exists in ∇ exactly one judgment of the shape $\psi \sim \bigcap_{l \in L^{(\psi)}}(\xi_l^{(\psi)} \to E_l^{(\psi)})$. We can prove by simultaneous induction on the definition of $\leq_\nabla$ two statements, the first of which implies the thesis.

1. if $(\bigcap_{i\in I}(A_i \to B_i))\cap(\bigcap_{h\in H}\psi_h)(\cap\iota) \leq_{\bigtriangledown} (\bigcap_{j\in J}(C_j \to D_j))\cap(\bigcap_{k\in K}\phi_k)(\cap\iota)$, and $D_j \not\sim_{\bigtriangledown} \Omega$, then for each $j \in J$ there exist $I' \subseteq I$, $H' \subseteq H$ and, for all $h \in H'$, $L^{(\psi_h)'} \subseteq L^{(\psi_h)}$ such that $C_j \leq_{\bigtriangledown} (\bigcap_{i\in I'} A_i) \cap (\bigcap_{h\in H'}(\bigcap_{l\in L^{(\psi_h)'}} \xi_l^{(\psi_h)}))$ and $(\bigcap_{i\in I'} B_i) \cap (\bigcap_{h\in H'}(\bigcap_{l\in L^{(\psi_h)'}} E_l^{(\psi_h)})) \leq_{\bigtriangledown} D_j$;
2. if $(\bigcap_{i\in I}(A_i \to B_i))\cap(\bigcap_{h\in H}\psi_h)(\cap\iota) \leq_{\bigtriangledown} (\bigcap_{j\in J}(C_j \to D_j))\cap(\bigcap_{k\in K}\phi_k)(\cap\iota)$, and $\phi_k \not\sim_{\bigtriangledown} \Omega$, then for each $m \in L^{(\phi_k)}$ there exist $I' \subseteq I$, $H' \subseteq H$ and, for all $h \in H'$, $L^{(\psi_h)'} \subseteq L^{(\psi_h)}$ such that $\xi_m^{(\phi_k)} \leq_{\bigtriangledown} (\bigcap_{i\in I'} A_i) \cap (\bigcap_{h\in H'}(\bigcap_{l\in L^{(\psi_h)'}} \xi_l^{(\psi_h)}))$ and $(\bigcap_{i\in I'} B_i) \cap (\bigcap_{h\in H'}(\bigcap_{l\in L^{(\psi_h)'}} E_l^{(\psi_h)})) \leq_{\bigtriangledown} E_m^{(\phi_k)}$. □

Inside the set of types we single out those which are not ι and not intersections containing ι. Moreover we associate to each type the maximum number of nested arrows in the leftmost path.

Definition 3. *1. A type $A \in \mathbb{T}^{\bigtriangledown}$ is* functional *iff $A \not\equiv \iota$ and there is no $B \in \mathbb{T}^{\bigtriangledown}$ such that $A \equiv \iota \cap B$;*
2. The mapping $\# : \mathbb{T}^{\bigtriangledown} \to \mathbb{N}$ is defined inductively on types as follows:

$$\begin{array}{lll} \#(A) & = 0 & \textit{if } A \in \mathbb{C}^{\bigtriangledown}; \\ \#(A \to B) & = \#(A) + 1; & \\ \#(A \cap B) & = \max\{\#(A), \#(B)\}. \ \square & \end{array}$$

Trivially all types in $\mathbb{T}^{\bigtriangledown}$ are functional when $\iota \notin \mathbb{C}$.

Some properties of functional types are crucial.

Theorem 2. *1. If $A \leq_{\bigtriangledown} B$ and A is functional, then B is functional.*
2. Let $A \in \mathbb{T}^{\bigtriangledown}$ be a functional type. Then, if $\#(A) \geq 1$, there is $B \in \mathbb{T}^{\bigtriangledown}$ such that $A \sim_{\bigtriangledown} B$, $B \equiv \bigcap_{i\in I}(C_i \to D_i)$, and $\#(B) = \#(A)$.

Proof. (1) asy by induction on $\leq_{\bigtriangledown}$.
(2) et $A \equiv (\bigcap_{j\in J}(C'_j \to D'_j)) \cap (\bigcap_{h\in H}\psi_h)$, where $C'_j, D'_j \in \mathbb{T}^{\bigtriangledown}$, $\psi_h \in \mathbb{C}^{\bigtriangledown}$. Being A functional, $\forall h \in H.\ \psi_h \not\equiv \iota$, hence for each $h \in H$ there are I_h, $\xi_{i,h} \in \mathbb{C}^{\bigtriangledown}$, $A_{i,h} \in \mathbb{T}^{\bigtriangledown}$, such that $\psi_h \sim_{\bigtriangledown} \bigcap_{i\in I_h}(\xi_{i,h} \to A_{i,h})$. We can choose $B \equiv (\bigcap_{j\in J}(C'_j \to D'j)) \cap (\bigcap_{h\in H}(\bigcap_{i\in I_h}(\xi_{i,h} \to A_{i,h})))$. □

Before giving the key notion of *intersection-type assignment system*, we introduce bases and some related definitions.

Definition 4 (Bases).

*1. A $\bigtriangledown$-*basis *is a (possibly infinite) set of statements of the shape $x : B$, where $B \in \mathbb{T}^{\bigtriangledown}$, with all variables distinct.*
2. $x \in \Gamma$ is short for $\exists A \in \mathbb{T}^{\bigtriangledown}.\ (x : A) \in \Gamma$ and $\Gamma, x : A$ is short for $\Gamma \cup \{x : A\}$ when $x \notin \Gamma$.
3. Let Γ and Γ' be $\bigtriangledown$-bases. The $\bigtriangledown$-basis $\Gamma \uplus \Gamma'$ is defined as follows:

$$\begin{aligned} \Gamma \uplus \Gamma' = {} & \{x : A \cap B \mid x : A \in \Gamma \textit{ and } x : B \in \Gamma'\} \\ & \cup \{x : A \mid x : A \in \Gamma \textit{ and } x \notin \Gamma'\} \\ & \cup \{x : B \mid x : B \in \Gamma' \textit{ and } x \notin \Gamma\}. \end{aligned}$$

Accordingly we define:

$$\Gamma \mathrel{\underline{\subset\!\!\!\!+\,}} \Gamma' \;\Leftrightarrow\; \exists \Gamma''.\, \Gamma \uplus \Gamma'' = \Gamma'. \;\square$$

Definition 5 (The type assignment system).
The intersection type assignment system *elative to the eitt $\Sigma^{\triangledown}$, notation $\lambda\cap^{\triangledown}$, is a formal system for deriving judgments of the form $\Gamma \vdash^{\triangledown} M : A$, where the* subject *$M$ is an untyped λ-term, the* predicate *A is in $\mathbb{T}^{\triangledown}$, and Γ is a $\triangledown$-basis. Its axioms and rules are the following:*

$$\begin{array}{llll}
(Ax) & \dfrac{(x{:}A) \in \Gamma}{\Gamma \vdash^{\triangledown} x{:}A} & (Ax\text{-}\Omega) & \Gamma \vdash^{\triangledown} M : \Omega \\[2ex]
(\to \mathrm{I}) & \dfrac{\Gamma, x{:}A \vdash^{\triangledown} M : B}{\Gamma \vdash^{\triangledown} \lambda x.M : A \to B} & (\to \mathrm{E}) & \dfrac{\Gamma \vdash^{\triangledown} M : A \to B \quad \Gamma \vdash^{\triangledown} N : A}{\Gamma \vdash^{\triangledown} MN : B} \\[2ex]
(\cap \mathrm{I}) & \dfrac{\Gamma \vdash^{\triangledown} M : A \quad \Gamma \vdash^{\triangledown} M : B}{\Gamma \vdash^{\triangledown} M : A \cap B} & (\leq_{\triangledown}) & \dfrac{\Gamma \vdash^{\triangledown} M : A \quad A \leq_{\triangledown} B}{\Gamma \vdash^{\triangledown} M : B} \;\square
\end{array}$$

As usual we consider λ-terms modulo α-conversion.

Notice that intersection elimination rules

$$(\cap E)\; \frac{\Gamma \vdash^{\triangledown} M : A \cap B}{\Gamma \vdash^{\triangledown} M : A} \quad \frac{\Gamma \vdash^{\triangledown} M : A \cap B}{\Gamma \vdash^{\triangledown} M : B}$$

can be immediately proved to be derivable in all $\lambda\cap^{\triangledown}$.

A first simple proposition, which can be proved straightforwardly by induction on the structure of derivations is the following.

Proposition 1.

1. *If $x \notin FV(M)$ and $\Gamma, x{:}B \vdash^{\triangledown} M : A$, then $\Gamma \vdash^{\triangledown} M : A$;*
2. *If $\Gamma \vdash^{\triangledown} M : A$ and $\Gamma \mathrel{\underline{\subset\!\!\!\!+\,}} \Gamma'$, then $\Gamma' \vdash^{\triangledown} M : A$.*

We end this section by stating a Generation, or Inversion, Theorem for the type assignment systems $\lambda\cap^{\triangledown}$.

Theorem 3 (Generation Theorem).

1. *Assume $A \not\sim_{\triangledown} \Omega$. $\Gamma \vdash^{\triangledown} x : A$ iff $(x{:}B) \in \Gamma$ and $B \leq_{\triangledown} A$ for some $B \in \mathbb{T}^{\triangledown}$.*
2. *$\Gamma \vdash^{\triangledown} MN : A$ iff $\Gamma \vdash^{\triangledown} M : B \to A$, and $\Gamma \vdash^{\triangledown} N : B$ for some $B \in \mathbb{T}^{\triangledown}$.*
3. *$\Gamma \vdash^{\triangledown} \lambda x.M : A$ iff $\Gamma, x : B_i \vdash^{\triangledown} M : C_i$ and $\bigcap_{i \in I}(B_i \to C_i) \leq_{\triangledown} A$, for some I and $B_i, C_i \in \mathbb{T}^{\triangledown}$.*
4. *If $\Gamma \vdash^{\triangledown} \lambda x.M : A$ then A is a functional type.*
5. *$\Gamma \vdash^{\triangledown} \lambda x.M : B \to C$ iff $\Gamma, x{:}B \vdash^{\triangledown} M : C$.*

Proof. The proof of each ($\Leftarrow$) is easy. So we only treat ($\Rightarrow$).

(1) Easy by induction on derivations, since only the axioms (Ax), (Ax-Ω), and the rules ($\cap$I), ($\leq_{\triangledown}$) can be applied. Notice that the condition $A \not\sim_{\triangledown} \Omega$ implies that $\Gamma \vdash^{\triangledown} x : A$ cannot be obtained just using axioms (Ax-Ω).

(2) If $A{\sim_\nabla}\Omega$ we can choose $B{\sim_\nabla}\Omega$. Otherwise the proof is by induction on derivations. The only interesting case is when $A \equiv A_1 \cap A_2$ and the last applied rule is $(\cap\mathrm{I})$:

$$(\cap\mathrm{I})\ \frac{\Gamma \vdash^\nabla MN : A_1 \quad \Gamma \vdash^\nabla MN : A_2}{\Gamma \vdash^\nabla MN : A_1 \cap A_2}.$$

The condition $A{\not\sim_\nabla}\Omega$ implies that we cannot have $A_1{\sim_\nabla}A_2{\sim_\nabla}\Omega$. We do the proof for $A_1{\not\sim_\nabla}\Omega$ and $A_2{\not\sim_\nabla}\Omega$, the other cases can be treated similarly. By induction there are B, C, D, E such that

$$\begin{array}{cc} \Gamma \vdash^\nabla M : B \to C, & \Gamma \vdash^\nabla N : B, \\ \Gamma \vdash^\nabla M : D \to E, & \Gamma \vdash^\nabla N : D, \\ C \leq_\nabla A_1 \quad\ \& & E \leq_\nabla A_2. \end{array}$$

So we are done being $(B \to C) \cap (D \to E) \leq_\nabla B \cap D \to C \cap E \leq B \cap D \to A$ by rules $(\to\text{-}\cap)$ and (η) since $C \cap E \leq_\nabla A$.

(3) The proof is very similar to the proof of (2). It is again by induction on derivations and again the only interesting case is when the last applied rule is $(\cap\mathrm{I})$:

$$(\cap\mathrm{I})\ \frac{\Gamma \vdash^\nabla \lambda x.M : A_1 \quad \Gamma \vdash^\nabla \lambda x.M : A_2}{\Gamma \vdash^\nabla \lambda x.M : A_1 \cap A_2}.$$

By induction there are I, B_i, C_i, J, D_j, E_j such that

$$\begin{array}{l} \forall i \in I.\ G, x{:}B_i \vdash^\nabla M : C_i, \forall j \in J.\ G, x{:}D_j \vdash^\nabla M : E_j, \\ \bigcap_{i\in I}(B_i \to C_i) \leq_\nabla A_1 \quad \& \quad \bigcap_{j\in J}(D_j \to E_j) \leq_\nabla A_2. \end{array}$$

So we are done since $(\bigcap_{i\in I}(B_i \to C_i)) \cap (\bigcap_{j\in J}(D_j \to E_j)) \leq_\nabla A$.

(4) Immediate from (3) and Theorem 2(1).

(5) The case $C{\sim_\nabla}\Omega$ is trivial. Otherwise let I, B_i, C_i be as in (3), where $A \equiv B \to C$. Then $\bigcap_{i\in I}(B_i \to C_i) \leq_\nabla B \to C$ implies by Theorem 1 that there exists a $J \subseteq I$ such that $B \leq_\nabla \bigcap_{i\in J} B_i$ and $\bigcap_{i\in J} C_i \leq_\nabla C$. From $\Gamma, x : B_i \vdash^\nabla M : C_i$ we can derive $\Gamma, x{:}B \vdash^\nabla M : C_i$ by Proposition 1(2), so by $(\cap\mathrm{I})$ we have $\Gamma, x : B \vdash^\nabla M : \bigcap_{i\in J} C_i$. Finally applying rule $(\leq_\nabla)$ we can conclude $\Gamma, x{:}B \vdash^\nabla M : C$. □

2 Filter Models

In this section we discuss how to build λ-models out of type theories. We start with the definition of *filter* for eitt's. Then we show how to turn the space of filters into an applicative structure. Finally we will define a notion of interpretation of λ-terms and show that we get λ-models (*filter models*).

Filter models arise naturally in the context of those generalizations of Stone duality that are used in discussing domain theory in logical form (see [1], [11], [32]). This approach provides a conceptually independent semantics to intersection types, the *lattice semantics*. Types are viewed as *compact elements* of domains. The type Ω denotes the least element, intersections denote joins of compact elements, and arrow types allow to internalize the space of continuous endomorphisms. Following the paradigm of Stone duality, type theories give rise to filter models, where the interpretation of λ-terms can be given through a finitary logical description.

Definition 6.

1. *A $\bigtriangledown$-filter (or a filter over $\mathbb{T}^{\bigtriangledown}$) is a set $X \subseteq \mathbb{T}^{\bigtriangledown}$ such that:*
   - $\Omega \in X$*;*
   - *if $A \leq_{\bigtriangledown} B$ and $A \in X$, then $B \in X$;*
   - *if $A, B \in X$, then $A \cap B \in X$;*
2. *$\mathcal{F}^{\bigtriangledown}$ denotes the set of $\bigtriangledown$-filters over $\mathbb{T}^{\bigtriangledown}$;*
3. *if $X \subseteq \mathbb{T}^{\bigtriangledown}$, $\uparrow X$ denotes the $\bigtriangledown$-filter generated by X;*
4. *a $\bigtriangledown$-filter is* principal *if it is of the shape $\uparrow \{A\}$, for some type A. We shall denote $\uparrow \{A\}$ simply by $\uparrow A$.* □

It is well known that $\mathcal{F}^{\bigtriangledown}$ is a ω-algebraic cpo, whose compact (or finite) elements are the filters of the form $\uparrow A$ for some type A and whose bottom element is $\uparrow \Omega$.

Next we endow the space of filters with the notions of application and of λ-term interpretation. Let $\mathsf{Env}_{\mathcal{F}^{\bigtriangledown}}$ be the set of all mappings from the set of term variables to $\mathcal{F}^{\bigtriangledown}$.

Definition 7.

1. *Application $\cdot : \mathcal{F}^{\bigtriangledown} \times \mathcal{F}^{\bigtriangledown} \to \mathcal{F}^{\bigtriangledown}$ is defined as*

$$X \cdot Y = \{B \mid \exists A \in Y. A \to B \in X\}.$$

2. *The interpretation function: $[\![\ ]\!]^{\bigtriangledown} : \Lambda \times \mathsf{Env}_{\mathcal{F}^{\bigtriangledown}} \to \mathcal{F}^{\bigtriangledown}$ is defined by*

$$[\![M]\!]^{\bigtriangledown}_{\rho} = \{A \in \mathbb{T}^{\bigtriangledown} \mid \exists \Gamma \models \rho.\ \Gamma \vdash^{\bigtriangledown} M : A\},$$

where ρ ranges over $\mathsf{Env}_{\mathcal{F}^{\bigtriangledown}}$ and $\Gamma \models \rho$ if and only $(x : B) \in \Gamma$ implies $B \in \rho(x)$.

3. *The triple $\langle \mathcal{F}^{\bigtriangledown}, \cdot, [\![\ ]\!]^{\bigtriangledown} \rangle$ is called the* filter model *over $\Sigma^{\bigtriangledown}$.* □

Notice that previous definition is sound, since it is easy to verify that $X \cdot Y$ is a $\bigtriangledown$-filter and moreover:

Theorem 4.
The filter model $\langle \mathcal{F}^{\bigtriangledown}, \cdot, [\![\ ]\!]^{\bigtriangledown} \rangle$ is a λ-model, in the sense of Hindley-Longo [18], that is:

1. $[\![x]\!]^{\bigtriangledown}_{\rho} = \rho(x)$*;*
2. $[\![MN]\!]^{\bigtriangledown}_{\rho} = [\![M]\!]^{\bigtriangledown}_{\rho} \cdot [\![N]\!]^{\bigtriangledown}_{\rho}$*;*
3. $[\![\lambda x.M]\!]^{\bigtriangledown}_{\rho} \cdot X = [\![M]\!]^{\bigtriangledown}_{\rho[X/x]}$*;*
4. $(\forall x \in FV(M).\ [\![x]\!]^{\bigtriangledown}_{\rho} = [\![x]\!]^{\bigtriangledown}_{\rho'}) \Rightarrow [\![M]\!]^{\bigtriangledown}_{\rho} = [\![M]\!]^{\bigtriangledown}_{\rho'}$*;*

5. $[\![\lambda x.M]\!]^{\nabla}_{\rho} = [\![\lambda y.M[y/x]]\!]^{\nabla}_{\rho}$*, if* $y \notin FV(M)$*;*
6. $(\forall X \in \mathcal{F}^{\nabla}.[\![M]\!]^{\nabla}_{\rho[X/x]} = [\![N]\!]^{\nabla}_{\rho[X/x]}) \;\Rightarrow\; [\![\lambda x.M]\!]^{\nabla}_{\rho} = [\![\lambda x.N]\!]^{\nabla}_{\rho}$*.*

Moreover it is extensional - that is $[\![\lambda x.Mx]\!]^{\nabla}_{\rho} = [\![M]\!]^{\nabla}_{\rho}$ *when* $x \notin FV(M)$ *- iff* $\iota \notin \mathbb{C}$*.*

Proof. (1) We show $A \in [\![x]\!]^{\nabla}_{\rho}$ iff $A \in \rho(x)$. The case $A \sim_{\nabla} \Omega$ is immediate. If $A \not\sim_{\nabla} \Omega$ and $A \in [\![x]\!]^{\nabla}_{\rho}$, then $\Gamma \vdash^{\nabla} x : A$ for some ∇-basis Γ such that $\Gamma \models \rho$. So there exists a premise $x : A'$ in Γ such that $A' \in \rho(x)$. By Theorem 3(1), $A' \leq_{\nabla} A$, hence $A \in \rho(x)$. The vice versa is trivial.

(2) Let $A \in [\![MN]\!]^{\nabla}_{\rho}$. Then there exists $\Gamma \models \rho$ such that $\Gamma \vdash^{\nabla} MN : A$. By Theorem 3(2), there exists $B \in \mathbb{T}^{\nabla}$ such that $\Gamma \vdash^{\nabla} M : B \to A$ and $\Gamma \vdash^{\nabla} N : B$, hence $B \in [\![N]\!]^{\nabla}_{\rho}$ and $B \to A \in [\![M]\!]^{\nabla}_{\rho}$. By definition of application it follows $A \in [\![M]\!]^{\nabla}_{\rho} \cdot [\![N]\!]^{\nabla}_{\rho}$.
Let now $A \in [\![M]\!]^{\nabla}_{\rho} \cdot [\![N]\!]^{\nabla}_{\rho}$. Then there exists $B \in \mathbb{T}^{\nabla}$ such that $B \to A \in [\![M]\!]^{\nabla}_{\rho}$ and $B \in [\![N]\!]^{\nabla}_{\rho}$, hence there exist two ∇-bases, Γ and Γ', such that $\Gamma \models \rho$, $\Gamma' \models \rho$, and moreover $\Gamma \vdash^{\nabla} M : B \to A$, $\Gamma' \vdash^{\nabla} N : B$. Consider the basis $\Gamma'' = \Gamma \uplus \Gamma'$. We have $\Gamma'' \models \rho$, $\Gamma'' \vdash^{\nabla} M : B \to A$ and $\Gamma'' \vdash^{\nabla} N : B$. From the last two judgments we deduce $\Gamma'' \vdash^{\nabla} MN : A$, which, along with the first judgment, implies $A \in [\![MN]\!]^{\nabla}_{\rho}$.

(3) Let $A \in [\![M]\!]^{\nabla}_{\rho[X/x]}$. Then there exists $\Gamma \models \rho[X/x]$ such that $\Gamma \vdash^{\nabla} M : A$. Let $\Gamma = \Gamma', x{:}B$, then, by rule $(\to \mathrm{I})$, we get $\Gamma' \vdash^{\nabla} \lambda x.M : B \to A$. This implies $B \to A \in [\![\lambda x.M]\!]^{\nabla}_{\rho}$ since from $\Gamma \models \rho[X/x]$ we have $\Gamma' \models \rho$. Being $B \in X$ (because $\Gamma \models \rho[X/x]$), we get $A \in [\![\lambda x.M]\!]^{\nabla}_{\rho} \cdot X$.
Let $A \in [\![\lambda x.M]\!]^{\nabla}_{\rho} \cdot X$. Then there exists $\Gamma \models \rho$ and $B \in X$ such that $\Gamma \vdash^{\nabla} \lambda x.M : B \to A$. Since $x \notin \mathrm{FV}(\lambda x.M)$ by Lemma 1(1) we can assume $x \notin \Gamma$. By Theorem 3(5) it follows $\Gamma, x{:}B \vdash^{\nabla} M : A$. Since $B \in X$ we have $\Gamma, x{:}B \models \rho$, hence $A \in [\![M]\!]^{\nabla}_{\rho}$.

(4) easily proven by induction on the structure of M.

(5) trivial.

(6) Suppose that the premise hold and $A \in [\![\lambda x.M]\!]^{\nabla}_{\rho}$. Then there is $\Gamma \models \rho$ such that $\Gamma \vdash^{\nabla} \lambda x.M : A$. Since $x \notin \mathrm{FV}(\lambda x.M)$ by Lemma 1(1) we can assume $x \notin \Gamma$. By Theorem 3(3) there exist I and $B_i, C_i \in \mathbb{T}^{\nabla}$ such that $\Gamma \vdash^{\nabla} \lambda x.M : B_i \to C_i$ for all $i \in I$ and $\bigcap_{i \in I}(B_i \to C_i) \leq_{\nabla} A$. So we have, for each $i \in I$, by Theorem 3(5) $\Gamma, x : B_i \vdash^{\nabla} M : C_i$. By the premise, we get, for each $i \in I$, $\Gamma, x : B_i \vdash^{\nabla} N : C_i$, which implies $[\![\lambda x.M]\!]^{\nabla}_{\rho} \subseteq [\![\lambda x.N]\!]^{\nabla}_{\rho}$. Similarly one proves $[\![\lambda x.N]\!]^{\nabla}_{\rho} \subseteq [\![\lambda x.M]\!]^{\nabla}_{\rho}$.

We show now that the model is extensional when $\iota \notin \mathbb{C}$. Let $A \in [\![\lambda x.Mx]\!]^{\nabla}_{\rho}$, with $x \notin \mathrm{FV}(M)$. Then there is $\Gamma \models \rho$ such that $\Gamma \vdash^{\nabla} \lambda x.Mx : A$. Reasoning as in the proof of (6), we have that there exist I and $B_i, C_i \in \mathbb{T}^{\nabla}$ such that for each $i \in I$, $\Gamma, x{:}B_i \vdash^{\nabla} Mx : C_i$ and $\bigcap_{i \in I}(B_i \to C_i) \leq_{\nabla} A$. By Theorem 3(2), it follows that there exists, for each $i \in I$, D_i such that $\Gamma, x{:}B_i \vdash^{\nabla} M : D_i \to C_i$, and $\Gamma, x{:}B_i \vdash^{\nabla} x : D_i$. We have $B_i \leq_{\nabla} D_i$ for each $i \in I$ either by Theorem 3(1) if $D_i \not\sim_{\nabla} \Omega$ or by axiom (Ω) and rule (trans) if $D_i \sim_{\nabla} \Omega$. Hence we get, by $(\leq_{\nabla})$,

$\Gamma, x : B_i \vdash^{\triangledown} M : B_i \to C_i$, for each $i \in I$. Since $x \notin \mathrm{FV}(M)$ we can apply Lemma 1(1) and obtain, for each $i \in I$, $\Gamma \vdash^{\triangledown} M : B_i \to C_i$, hence, by rules $(\cap \mathrm{I})$ and $(\leq_{\triangledown})$, $\Gamma \vdash^{\triangledown} M : A$, which implies $A \in [\![M]\!]^{\triangledown}_{\rho}$.
Suppose $A \in [\![M]\!]^{\triangledown}_{\rho}$. Then there exists $\Gamma \models \rho$ such that $\Gamma \vdash^{\triangledown} M : A$. Since $x \notin \mathrm{FV}(M)$ by Lemma 1(1) we can assume $x \notin \Gamma$. By Theorem 2 we have $A \sim_{\triangledown} \bigcap_{i \in I}(B_i \to C_i)$ for suitable I and $B_i, C_i \in \mathbb{T}^{\triangledown}$, so by applying rule $(\leq_{\triangledown})$ we get $\Gamma \vdash^{\triangledown} M : B_i \to C_i$ for each $i \in I$. By Lemma 1(2), we have $\Gamma, x : B_i \vdash^{\triangledown} M : B_i \to C_i$. This judgment, along with $\Gamma, x : B_i \vdash^{\triangledown} x : B_i$ allows to obtain, by rule $(\to \mathrm{E})$, $\Gamma, x : B_i \vdash^{\triangledown} Mx : C_i$, for each $i \in I$. By rule $(\to \mathrm{I})$ we deduce $\Gamma \vdash^{\triangledown} \lambda x.Mx : B_i \to C_i$ for each $i \in I$, hence by rule $(\cap \mathrm{I})$ it follows $\Gamma \vdash^{\triangledown} \lambda x.Mx : \bigcap_{i \in I}(B_i \to C_i)$, which implies, by rule $(\leq_{\triangledown})$, $\Gamma \vdash^{\triangledown} \lambda x.Mx : A$, so we conclude $A \in [\![\lambda x.Mx]\!]^{\triangledown}_{\rho}$.

Finally, if $\iota \in \mathbb{C}$ the model is non-extensional, since taking $\rho(x) = \uparrow \iota$, we get $[\![x]\!]^{\triangledown}_{\rho} = \uparrow \iota$ while $[\![\lambda y.xy]\!]^{\triangledown}_{\rho} = \uparrow \Omega$. □

3 Semantical Proof of the Easiness of $\omega_2\omega_2$

Let ω_2 be the λ-term $\lambda x.xx$. For an arbitrary closed λ-term M we build a non-extensional filter model $\langle \mathcal{F}^{\triangledown'}, \cdot, [\![\ ]\!]^{\triangledown'} \rangle$ such that $[\![M]\!]^{\triangledown'} = [\![\omega_2\omega_2]\!]^{\triangledown'}$.

First we give a lemma which characterizes the types derivable for ω_2 and $\omega_2\omega_2$.

Lemma 1.

1. *$\vdash^{\triangledown} \omega_2 : A \to B$ iff $A \leq_{\triangledown} A \to B$;*
2. *$\vdash^{\triangledown} \omega_2\omega_2 : B$ iff $A \leq_{\triangledown} A \to B$ for some $A \in \mathbb{T}^{\triangledown}$ such that $\vdash^{\triangledown} \omega_2 : A$.*
3. *If $\vdash^{\triangledown} \omega_2\omega_2 : B$ then there exists $A \in \mathbb{T}^{\triangledown}$ such that $\#(A) = 0$, $A \leq_{\triangledown} A \to B$ and $\vdash^{\triangledown} \omega_2 : A$.*

Proof. (1) By a straightforward computation $A \leq_{\triangledown} A \to B$ implies $\vdash^{\triangledown} \omega_2 : A \to B$. Conversely, suppose $\vdash^{\triangledown} \omega_2 : A \to B$. If $B \sim_{\triangledown} \Omega$, then by axioms (Ω), $(\Omega\text{-}\eta)$, and rules (η), (trans), we have $A \leq_{\triangledown} A \to B$. Otherwise, by Theorem 3(5) it follows $x : A \vdash^{\triangledown} xx : B$. By Theorem 3(2) there exists a type $C \in \mathbb{T}^{\triangledown}$ such that $x : A \vdash^{\triangledown} x : C \to B$ and $x : A \vdash^{\triangledown} x : C$. Notice that $B \not\sim_{\triangledown} \Omega$ implies $C \to B \not\sim_{\triangledown} \Omega$, since from $C \to B \sim_{\triangledown} \Omega$ we get $C \to B \sim_{\triangledown} \Omega \to \Omega$ by axiom $(\Omega\text{-}\eta)$ and rule (trans) and this implies $B \sim_{\triangledown} \Omega$ by Theorem 1. So by Theorem 3(1), we get $A \leq_{\triangledown} C \to B$. We have $A \leq_{\triangledown} C$ either by Theorem 3(1) if $C \not\sim_{\triangledown} \Omega$ or by axiom (Ω) and rule (trans) if $C \sim_{\triangledown} \Omega$. From $A \leq_{\triangledown} C \to B$ and $A \leq_{\triangledown} C$ by rule (η) it follows $A \leq_{\triangledown} A \to B$.

(2) The case $B \sim_{\triangledown} \Omega$ is trivial. Otherwise, if $\vdash^{\triangledown} \omega_2\omega_2 : B$, by Theorem 3(2) it follows that there exists $A \in \mathbb{T}^{\triangledown}$ such that $\vdash^{\triangledown} \omega_2 : A$ and $\vdash^{\triangledown} \omega_2 : A \to B$. We conclude by (1).

(3) Let $\vdash^{\triangledown} \omega_2\omega_2 : B$. Then, by Point (2), there exists $A \in \mathbb{T}^{\triangledown}$ such that $\vdash^{\triangledown} \omega_2 : A$ and $A \leq_{\triangledown} A \to B$. We prove the thesis by induction on $\#(A)$. If $\#(A) = 0$ we are done. Suppose now $\#(A) \geq 1$. First by Theorem 3(4) A is functional. By

applying Lemma 2, we obtain a type A' such that $A' \sim_{\triangledown} A$, $A' \equiv \bigcap_{i\in I}(C_i \to D_i)$ and $\#(A') = \#(A)$. From $A \leq_{\triangledown} A \to B$ we have $\bigcap_{i\in I}(C_i \to D_i) \leq_{\triangledown} A \to B$, hence, by Theorem 1, there exists $J \subseteq I$ such that $A \leq_{\triangledown} \bigcap_{i\in J} C_i$ and $\bigcap_{i\in J} D_i \leq_{\triangledown} B$. Since $\vdash^{\triangledown} \omega_2 : A$, by $(\leq_{\triangledown})$ it follows $\vdash^{\triangledown} \omega_2 : C_i \to D_i$ for all $i \in J$ and $\vdash^{\triangledown} \omega_2 : \bigcap_{i\in J} C_i$. By Point (1) it follows $\forall i \in J. C_i \leq_{\triangledown} C_i \to D_i$. By axiom ($\to$-$\cap$) and rule (η) we get $C \leq_{\triangledown} C \to \bigcap_{i\in J} D_i$, and also $C \leq_{\triangledown} C \to B$, where $C \equiv \bigcap_{i\in J} C_i$. We have obtained:

$$\begin{array}{l} \vdash^{\triangledown} \omega_2 : C; \\ C \leq_{\triangledown} C \to B; \\ \#(C) < \#(A') = \#(A). \end{array}$$

The thesis follows by applying the induction properties. □

We build the desired model by taking the union of a suitable countable sequence of eitt's $\Sigma^{\triangledown_n}$ defined in such a way that the final interpretation of M coincides with the interpretation of $\omega_2\omega_2$. In the following $\langle\cdot,\cdot\rangle$ denotes any bijection between $\mathbb{N} \times \mathbb{N}$ and $\mathbb{N}$.

Definition 8.

1. *The eitt's $\Sigma^{\triangledown_n}$ are defined inductively on n as follows:*
   - $\mathbb{C}^{\triangledown_1} = \{\Omega, \iota\}$*;*
   - $\triangledown_1 = \triangledown_0$*;*
   - $\mathbb{C}^{\triangledown_{n+1}} = \mathbb{C}^{\triangledown_n} \cup \{\varsigma_{\langle n,m\rangle} \mid m \in \mathbb{N}\}$*;*
   - $\triangledown_{n+1} = \triangledown_n \cup \{\varsigma_{\langle n,m\rangle} \sim \varsigma_{\langle n,m\rangle} \to W_{\langle n,m\rangle} \mid m \in \mathbb{N}\}$*,*

   where $\langle W_{\langle n,m\rangle}\rangle_{m\in\mathbb{N}}$ is any enumeration of the set $\{A \mid\ \mathit{vdash}^{\triangledown_n} M : A\}$.
2. *We define $\Sigma^{\triangledown'}$ as follows:*

$$\mathbb{C}^{\triangledown'} = \bigcup_{n\in\mathbb{N}} \mathbb{C}^{\triangledown_n};\ \triangledown' = \bigcup_{n\in\mathbb{N}} \triangledown_n. \ \square$$

Since $\Sigma^{\triangledown'}$ is an eitt and $\iota \in \mathbb{C}^{\triangledown'}$, by Theorem 4, it follows that it induces a non-extensional λ-model.

Theorem 5.
The triple $\langle \mathcal{F}^{\triangledown'}, \cdot, [\![\ ]\!]^{\triangledown'} \rangle$ is a non-extensional λ-model.

We need also a negative result on the typing of ω_2.

Lemma 2.

$$\nvdash^{\triangledown'} \omega_2 : \iota \textit{ and } \nvdash^{\triangledown'} \omega_2 : \iota \to \iota.$$

Proof. From Theorem 3(4) we get $\nvdash^{\triangledown'} \omega_2 : \iota$, since the type ι is not functional.

To show $\nvdash^{\triangledown'} \omega_2 : \iota \to \iota$ we define the sets $\mathcal{E}_\Omega$, $\mathcal{G}_\tau$ as the minimal sets such that:

$$\begin{array}{l} \Omega \in \mathcal{E}_\Omega; \quad A \in \mathbb{T}^{\triangledown'}, B \in \mathcal{E}_\Omega \ \Rightarrow\ A \to B \in \mathcal{E}_\Omega; \\ A, B \in \mathcal{E}_\Omega \ \Rightarrow\ A \cap B \in \mathcal{E}_\Omega; \quad W_i \in \mathcal{E}_\Omega \ \Rightarrow\ \varsigma_i \in \mathcal{E}_\Omega; \\ \iota \in \mathcal{G}_\iota; \quad \mathcal{E}_\Omega \subseteq \mathcal{G}_\iota; \quad A, B \in \mathcal{G}_\iota \ \Rightarrow\ A \cap B \in \mathcal{G}_\iota. \end{array}$$

It is easy to check by induction on $\leq_{\nabla'}$ that for all $A \in \mathbb{T}^{\nabla'}$:

$$\begin{array}{l} B \leq_{\nabla'} A \text{ and } B \in \mathcal{E}_{\Omega} \text{ imply } A \in \mathcal{E}_{\Omega}; \\ B \leq_{\nabla'} A \text{ and } B \in \mathcal{G}_{\iota} \text{ imply } A \in \mathcal{G}_{\iota}. \end{array}$$

This implies that $\iota \not\leq_{\nabla'} \iota \to \iota$ since $\iota \in \mathcal{G}_{\iota}$ and $\iota \to \iota \notin \mathcal{G}_{\iota}$.
Suppose by contradiction that $\vdash^{\nabla'} \omega_2 : \iota \to \iota$: we get $\iota \leq_{\nabla'} \iota \to \iota$ by Lemma 1(1). □

We are now able to prove the first main theorem.

Theorem 6.
The filter model $\langle \mathcal{F}^{\nabla'}, \cdot, [\![\ ]\!]^{\nabla'} \rangle$ is not trivial and $[\![M]\!]^{\nabla'} = [\![\omega_2\omega_2]\!]^{\nabla'}$.

Proof. The model is not trivial since it is easy to derive $\vdash^{\nabla'} \mathsf{I} : \iota \to \iota$ while $\not\vdash^{\nabla'} \omega_2 : \iota \to \iota$ by Lemma 2.
The inclusion $[\![M]\!]^{\nabla'} \subseteq [\![\omega_2\omega_2]\!]^{\nabla'}$ is almost immediate by construction. By easy calculation, ω_2 may be given type ς_j for any integer j. In fact, since $\varsigma_j \sim_{\nabla'} \varsigma_j \to W_j$, it follows $x : \varsigma_j \vdash^{\nabla'} xx : W_j$, hence $\vdash^{\nabla'} \lambda x.xx : \varsigma_j \to W_j \sim_{\nabla'} \varsigma_j$. From this last fact, by applying ($\to$E), we obtain $\vdash^{\nabla'} \omega_2\omega_2 : W_j$ for all j. This proves $[\![M]\!]^{\nabla'} \subseteq [\![\omega_2\omega_2]\!]^{\nabla'}$.

On the other hand, let $\vdash^{\nabla'} \omega_2\omega_2 : B$. Then applying Lemma 1(3), it follows that there exists A such that $\#(A) = 0$, $\vdash^{\nabla'} \omega_2 : A$ and $A \leq_{\nabla'} A \to B$. Let $A \equiv \bigcap_{i \in I} \psi_i$. By Lemma 2, for all $i \in I$ we get that ψ_i is either Ω or ς_j for some integer j. This implies either $A \sim_{\nabla'} \Omega \to \Omega$ or $A \sim_{\nabla'} \bigcap_{j \in J}(\varsigma_j \to W_j)$ for some J. Therefore from $A \leq_{\nabla'} A \to B$ either $\Omega \leq_{\nabla'} B$ or $\exists L \subseteq J$ such that $\bigcap_{j \in L} W_j \leq_{\nabla'} B$, by Theorem 1. Since each W_j is in $[\![M]\!]^{\nabla'}$, we have $B \in [\![M]\!]^{\nabla'}$ and we are done. □

4 Semantical Proof of the Easiness of $\omega_3\omega_3\mathsf{I}$

Let ω_3 be the λ-term $\lambda x.xxx$. For an arbitrary closed λ-term M we build a non-extensional filter model $\langle \mathcal{F}^{\nabla''}, \cdot, [\![\ ]\!]^{\nabla''} \rangle$ such that $[\![M]\!]^{\nabla''} = [\![\omega_3\omega_3\mathsf{I}]\!]^{\nabla''}$.

First we give a lemma, which characterizes the types deducible for ω_3 and $\omega_3\omega_3\mathsf{I}$.

Notation.
In the following $\bar{Q}$ will denote types of the shape $\bigcap_{k \in K}(Q_k \to Q_k)$, that is the minimal types which I can receive. □

Lemma 3.

1. $\vdash^{\nabla} \omega_3 : A \to B$ *iff* $A \leq_{\nabla} A \to A \to B$*;*
2. $\vdash^{\nabla} \omega_3\omega_3\mathsf{I} : B$ *iff there exist* $A, \bar{Q}$ *such that* $A \leq_{\nabla} A \to A \to \bar{Q} \to B$ *and* $\vdash^{\nabla} \omega_3 : A$*.*

Proof. Throughout this proof we freely (and heavily!) use Theorem 3. We show just ($\Rightarrow$), since ($\Leftarrow$) are obtained simply by applying the typing rules. The case $B{\sim_\nabla}\Omega$ is trivial.

(1) $\vdash^\nabla \omega_3 : A \to B$ implies $x{:}A \vdash^\nabla xxx : B$, hence there exist C, D such that $x{:}A \vdash^\nabla x : C \to D \to B$, $x{:}A \vdash^\nabla x : C$ and $x{:}A \vdash^\nabla x : D$. The first of these three last judgments implies $A \leq_\nabla C \to D \to B$, while the last two imply $A \leq_\nabla C$ and $A \leq_\nabla D$, hence, by rule (η), we obtain $A \leq_\nabla A \to A \to B$.

(2) If $\vdash^\nabla \omega_3\omega_3\mathsf{I} : B$, then there exists C such that $\vdash^\nabla \omega_3\omega_3 : C \to B$ and $\vdash^\nabla \mathsf{I} : C$. By straightforward calculation it must exist $\bar{Q}$ such that $\bar{Q} \leq_\nabla C$. Moreover this implies $C \to B \leq_\nabla \bar{Q} \to B$, hence by rule $(\leq_\nabla)$,$\vdash^\nabla \omega_3\omega_3 : \bar{Q} \to B$. This last judgment holds only if there exists A such that $\vdash^\nabla \omega_3 : A \to \bar{Q} \to B$ and $\vdash^\nabla \omega_3 : A$. We conclude by using Point (1). □

We define the eitt $\Sigma^{\nabla''}$ as the union of a countable sequence of eitt's Σ^{∇_n} similarly to $\Sigma^{\nabla'}$.

Definition 9.

1. *Σ^{∇_n} are defined inductively on n as the eitt's generated by the following sets $\mathbb{C}^{\nabla_n}$ and ∇_n:*
   - $\mathbb{C}^{\nabla_1} = \{\Omega, \iota\}$;
   - $\nabla_1 = \nabla_0$;
   - $\mathbb{C}^{\nabla_{n+1}} = \mathbb{C}^{\nabla_n} \cup \{\alpha_{\langle n,m\rangle}, \beta_{\langle n,m\rangle}, \gamma_{\langle n,m\rangle} \mid m \in \mathbb{N}\}$;
   - $\nabla_{n+1} = \nabla_n \cup \{\alpha_{\langle n,m\rangle} \leq \beta_{\langle n,m\rangle}, \alpha_{\langle n,m\rangle} \leq \gamma_{\langle n,m\rangle}, \beta_{\langle n,m\rangle} \leq \gamma_{\langle n,m\rangle},$
     $\alpha_{\langle n,m\rangle} \sim (\gamma_{\langle n,m\rangle} \to \alpha_{\langle n,m\rangle}) \cap (\beta_{\langle n,m\rangle} \to \tau \to W_{\langle n,m\rangle}),$
     $\beta_{\langle n,m\rangle} \sim \gamma_{\langle n,m\rangle} \to \alpha_{\langle n,m\rangle},$
     $\gamma_{\langle n,m\rangle} \sim \gamma_{\langle n,m\rangle} \to \beta_{\langle n,m\rangle} \mid m \in \mathbb{N}\},$

   where $\tau \equiv \iota \to \iota$ and $\langle W_{\langle n,m\rangle}\rangle_{m\in\mathbb{N}}$ is any enumeration of the set $\{A \mid\ vdash^{\nabla_n} M : A\}$.
2. *We define $\Sigma^{\nabla''}$ as follows:*

$$\mathbb{C}^{\nabla''} = \bigcup_{n\in\mathbb{N}} \mathbb{C}^{\nabla_n};\ \nabla'' = \bigcup_{n\in\mathbb{N}} \nabla_n. \square$$

From Definition 9 we immediately have that $\Sigma^{\nabla''}$ is an eitt and $\iota \in \mathbb{C}^{\nabla''}$, hence it induces a non-extensional λ-model.

Theorem 7.
The triple $\langle \mathcal{F}^{\nabla''}, F^{\nabla''}, G^{\nabla''}\rangle$ is a non-extensional λ-model. □

Notation. Sometimes we will omit subscript $\langle n,m\rangle$ for α, β, γ, W. □

The proof of $[\![M]\!]^{\nabla''} = [\![\omega_3\omega_3\mathsf{I}]\!]^{\nabla''}$ is done in three steps. First we show that some subtypings do not hold (Lemma 5). The second step is a characterization of the constants which can be deduced for ω_3 (Lemma 6). Lastly we use these results in order to obtain properties for the types which satify particular subtyping relations and which contain proper subtypes deducible for ω_3 (Lemma 7).

For the first step we introduce four subsets of $\mathbb{T}^{\nabla''}$, whose meaning is the following (recall that $\tau \equiv \iota \to \iota$)[1]:

1. $\mathcal{E}_\Omega$ is the set of all types which are equivalent to Ω;
2. $\mathcal{G}_\iota$ is the set of all types which are greater than or equivalent to ι;
3. $\mathcal{L}_\iota$ is the set of all types which are less than or equivalent to ι;
4. $\mathcal{G}_\tau$ is the set of all types which are greater than or equivalent to τ.

Definition 10. *The sets $\mathcal{E}_\Omega$, $\mathcal{G}_\iota$, $\mathcal{L}_\iota$, $\mathcal{G}_\tau$, are defined as the minimal sets such that:*

$$\begin{array}{l}
\Omega \in \mathcal{E}_\Omega; \quad A \in \mathbb{T}^{\nabla''}, B \in \mathcal{E}_\Omega \;\Rightarrow\; A \to B \in \mathcal{E}_\Omega; \quad A, B \in \mathcal{E}_\Omega \;\Rightarrow\; A \cap B \in \mathcal{E}_\Omega;\\
\iota \in \mathcal{G}_\iota; \quad \mathcal{E}_\Omega \subseteq \mathcal{G}_\iota; \quad A, B \in \mathcal{G}_\iota \;\Rightarrow\; A \cap B \in \mathcal{G}_\iota;\\
\iota \in \mathcal{L}_\iota; \quad A \in \mathbb{T}^{\nabla''}, B \in \mathcal{L}_\iota \;\Rightarrow\; A \cap B, B \cap A \in \mathcal{L}_\iota;\\
\mathcal{E}_\Omega \subseteq \mathcal{G}_\tau; \quad A \in \mathcal{L}_\iota, B \in \mathcal{G}_\iota \;\Rightarrow\; A \to B \in \mathcal{G}_\tau; \quad A, B \in \mathcal{G}_\tau \;\Rightarrow\; A \cap B \in \mathcal{G}_\tau. \;\square
\end{array}$$

From this definition we easily get:

Lemma 4.
1. $\mathcal{E}_\Omega = \{A \in \mathbb{T}^{\nabla''} \mid \Omega \leq_{\nabla''} A\}$; *3. $\mathcal{L}_\iota = \{A \in \mathbb{T}^{\nabla''} \mid A \leq_{\nabla''} \iota\}$;*
2. $\mathcal{G}_\iota = \{A \in \mathbb{T}^{\nabla''} \mid \iota \leq_{\nabla''} A\}$; *4. $\mathcal{G}_\tau = \{A \in \mathbb{T}^{\nabla''} \mid \tau \leq_{\nabla''} A\}$.*

Proof. It is easy to check by induction on the definition of $\mathcal{E}_\Omega$ that $A \in \mathcal{E}_\Omega \Rightarrow A \sim_{\nabla''} \Omega$, and similarly for the other sets.
Vice versa one can show by induction on $\leq_{\nabla''}$ that:

$$\begin{array}{llll}
B \leq_{\nabla''} A & \text{and} & B \in \mathcal{E}_\Omega & \text{imply} \;\; A \in \mathcal{E}_\Omega;\\
B \leq_{\nabla''} A & \text{and} & B \in \mathcal{G}_\iota & \text{imply} \;\; A \in \mathcal{G}_\iota;\\
B \leq_{\nabla''} A & \text{and} & A \in \mathcal{L}_\iota & \text{imply} \;\; B \in \mathcal{L}_\iota;\\
B \leq_{\nabla''} A & \text{and} & B \in \mathcal{G}_\tau & \text{imply} \;\; A \in \mathcal{G}_\tau.
\end{array}$$

We consider just the case of application of rule (η) for $B \in \mathcal{G}_\tau$. Let $A \equiv A' \to A''$, $B \equiv B' \to B''$ and let the judgment $B \leq_{\nabla''} A$ be obtained by applying rule (η) from the premises $A' \leq_{\nabla''} B'$ and $B'' \leq_{\nabla''} A''$. Since $B \in \mathcal{G}_\tau$, it must hold $B' \in \mathcal{L}_\iota$ and $B'' \in \mathcal{G}_\iota$. By (3) and (2) we obtain $A' \in \mathcal{L}_\iota$ and $A'' \in \mathcal{G}_\iota$, hence A belongs to $\mathcal{G}_\tau$. □

Next lemma ensures that the types α, β, γ are not equivalent when $W \not\sim_{\nabla''} \Omega$.

Lemma 5. *If $W_i \not\sim_{\nabla''} \Omega$ then $\gamma_i \not\leq_{\nabla''} \beta_i \not\leq_{\nabla''} \alpha_i$ and $\bar{Q} \not\leq_{\nabla''} \gamma_i$.*

Proof. We prove $\gamma_i \not\leq_{\nabla''} \beta_i \not\leq_{\nabla''} \alpha_i$ by contradiction. By Theorem 1 $\gamma_i \leq_{\nabla''} \beta_i$ holds iff $\beta_i \leq_{\nabla''} \alpha_i$, since $\gamma_i \sim_{\nabla''} \gamma_i \to \beta_i$, $\beta_i \sim_{\nabla''} \gamma_i \to \alpha_i$ and $\alpha_i \not\sim_{\nabla''} \Omega$ by Lemma 4(1). So it is sufficient to prove $\beta_i \not\leq_{\nabla''} \alpha_i$. Since $\alpha_i \sim_{\nabla''} \beta_i \cap (\beta_i \to \tau \to W_i)$, we are done if we can prove that it is impossible to have

[1] The symbols $\mathcal{E}_\Omega$, etc. are overloaded, since they were used already in the proof of Lemma 2 and they will be used in Section 5, but no confusion can arise, since it is always clear from the context the eitt we are considering.

$\beta_i \leq_{\nabla''} \beta_i \to \tau \to W_i$, that is $\gamma_i \to \alpha_i \leq_{\nabla''} \beta_i \to \tau \to W_i$. By Theorem 1 being $W_i \not\sim_{\nabla''} \Omega$ this last judgment is equivalent to the pair of judgments $\beta_i \leq_{\nabla''} \gamma_i$ and $\alpha_i \leq_{\nabla''} \tau \to W_i$. Consider this last judgment, that is equivalent to $(\gamma_i \to \alpha_i) \cap (\beta_i \to \tau \to W_i) \leq_{\nabla''} \tau \to W_i$. By Theorem 1, it should hold $\tau \leq_{\nabla''} \gamma_i$, which is impossible by Lemma 4(4), since $\gamma_i \notin \mathcal{G}_\tau$.
Finally we prove $\bar{Q} \not\leq_{\nabla''} \gamma_i$. If by contradiction $\bar{Q} \leq_{\nabla''} \gamma_i$, by Theorem 1 (since $\gamma_i \sim_{\nabla''} \gamma_i \to \beta_i$ and $\beta_i \not\sim_{\nabla''} \Omega$ by Lemma 4(1)), there exists $H \subseteq K$ such that $\gamma_i \leq_{\nabla''} \bigcap_{h \in H} Q_h$ and $\bigcap_{h \in H} Q_h \leq_{\nabla''} \beta_i$, which implies $\gamma_i \leq_{\nabla''} \beta_i$. This is a contradiction by above. □

The definition of $\Sigma^{\nabla''}$ is tailored so that, as far as constants are concerned, ω_3 can receive just α's, and hence β, γ, and Ω, but not ι.

Lemma 6. *Let $\psi \in \mathbb{C}^{\nabla''}$, then $\vdash^{\nabla''} \omega_3 : \psi$ iff $\alpha_i \leq_{\nabla''} \psi$ for some i. Moreover $\nvdash^{\nabla''} \omega_3 : \tau$.*

Proof. We know, from Theorem 3(4), that $\nvdash^{\nabla''} \omega_3 : \iota$. We show:

$$(1) \vdash^{\nabla''} \omega_3 : \alpha_i \text{ for all } i;\ (2) \nvdash^{\nabla''} \omega_3 : \tau.$$

(1) Since $\alpha_i \sim_{\nabla''} (\gamma_i \to \alpha_i) \cap (\beta_i \to \tau \to W_i)$, by Lemma 3(1), it is sufficient to prove that $\gamma_i \leq_{\nabla''} \gamma_i \to \gamma_i \to \alpha_i$ and $\beta_i \leq_{\nabla''} \beta_i \to \beta_i \to \tau \to W_i$. The first judgment is immediate by the equivalences in ∇'' on γ_i and β_i. The second one follows by using rule (η) twice:

$$\begin{aligned} \beta_i &\sim_{\nabla''} \gamma_i \to \alpha_i \\ &\leq_{\nabla''} \gamma_i \to \beta_i \to \tau \to W_i \text{ (since } \alpha_i \leq_{\nabla''} \beta_i \to \tau \to W_i) \\ &\leq_{\nabla''} \beta_i \to \beta_i \to \tau \to W_i \text{ (since } \beta_i \leq_{\nabla''} \gamma_i). \end{aligned}$$

(2) If $\vdash^{\nabla''} \omega_3 : \tau$, then by Lemma 3(1), it should be $\iota \leq_{\nabla''} \iota \to \iota \to \iota$, which is impossible by Lemma 4(2) since $\iota \to \iota \to \iota \notin \mathcal{G}_\iota$. □

Lemma 7.

1. *If $\bigcap_{i \in I} \alpha_i \leq_{\nabla''} A \to B \to C$ and $\vdash^{\nabla''} \omega_3 : B$ then there is $J \subseteq I$ such that $\bigcap_{i \in J} \alpha_i \leq_{\nabla''} B \to C$.*
2. *If $A \leq_{\nabla''} A \to A \to B$ and $\vdash^{\nabla''} \omega_3 : A$ then there are I, $\alpha_i \in \mathbb{C}^{\nabla''}$, and n such that $C \leq_{\nabla''} C \to C \to A^n \to B$, where $C \equiv \bigcap_{i \in I} \alpha_i$.*

Proof. (1) The case $C \sim_{\nabla''} \Omega$ is trivial. Otherwise by Theorem 1 from $\bigcap_{i \in I} \alpha_i \leq_{\nabla''} A \to B \to C$ we get $(\bigcap_{i \in J} \alpha_i) \cap (\bigcap_{i \in H} (\tau \to W_i)) \leq_{\nabla''} B \to C$ for some $J, H \subseteq I$, since $\alpha_i \sim_{\nabla''} (\gamma_i \to \beta_i) \cap (\beta_i \to \tau \to W_i)$. It is impossible to have $B \leq_{\nabla''} \tau$, since this implies $\vdash^{\nabla''} \omega_3 : \tau$, which contradicts Lemma 6. So, by Theorem 1 it must hold $\bigcap_{i \in J} \alpha_i \leq_{\nabla''} B \to C$.

(2) The case $B \sim_{\nabla''} \Omega$ is trivial. Otherwise the proof is by induction on $\#(A)$. Notice that A is functional by Theorem 3(4).
If $\#(A) = 0$, then A is an intersection of constants which cannot contain ι. Thus we have $A \equiv \bigcap_{i \in J} \psi_i$ such that ψ_i is α, β, γ, or Ω, for all $i \in J$ (notice that

we cannot have $\psi_i \equiv \Omega$ for all $i \in J$ by Lemma 4(1) since $A \to A \to B \notin \mathcal{E}_\Omega$ when $B \not\sim_{\nabla''} \Omega$). Consider now the type C obtained from A by replacing every constant β_l and γ_l occurring in A with α_l. By Lemma 6 $\vdash^{\nabla''}\omega_3 : C$. Moreover $C \leq_{\nabla''} A$, hence $C \leq_{\nabla''} C \to C \to B$. The type C has the required shape since by construction $C \equiv \bigcap_{i\in I} \alpha_i$, where $I = \{i \in J \mid \psi_i \not\equiv \Omega\}$, hence the thesis is satisfied (in this case $n = 0$).

Let us suppose now $\#(A) > 0$. Then, by Lemma 2, there exist J, D_j, E_j such that $\bigcap_{j\in J}(D_j \to E_j) \sim_{\nabla''} A$ and moreover $\#(\bigcap_{j\in J}(D_j \to E_j)) = \#(A)$. By Theorem 1, from $\bigcap_{j\in J}(D_j \to E_j) \leq_{\nabla''} A \to A \to B$ there exists $H \subseteq J$ such that $A \leq_{\nabla''} \bigcap_{j\in H} D_j$ and $\bigcap_{j\in H} D_j \leq_{\nabla''} A \to B$. Let $D \equiv \bigcap_{j\in H} D_j$. Since $\vdash^{\nabla''}\omega_3 : A$, it follows by $(\leq_\nabla)$ $\vdash^{\nabla''}\omega_3 : D$. By the same reason, for all $j \in H$, $\vdash^{\nabla''}\omega_3 : D_j \to E_j$. Hence, by Lemma 3(1), it follows, for all $j \in H$, $D_j \leq_{\nabla''} D_j \to D_j \to E_j$, which implies $\bigcap_{j\in H} D_j \leq_{\nabla''} \bigcap_{j\in H}(D_j \to D_j \to E_j)$. Using $(\to\text{-}\cap)$ and (η) we obtain $D \leq_{\nabla''} D \to D \to \bigcap_{j\in H} E_j$, which implies $D \leq_{\nabla''} D \to D \to A \to B$. Since $\#(D) < \#(A)$, we can now apply the inductive hypothesis and deduce that there exists $C \equiv \bigcap_{i\in I} \alpha_i$ and n such that $C \leq_{\nabla''} C \to C \to D^n \to A \to B$, which implies $C \leq_{\nabla''} C \to C \to A^{n+1} \to B$, since $A \leq_{\nabla''} D$. □

Lastly we can prove the second main theorem.

Theorem 8.
The filter model $\langle \mathcal{F}^{\nabla''}, \cdot, [\![\ ]\!]^{\nabla''} \rangle$ *is not trivial and* $[\![M]\!]^{\nabla''} = [\![\omega_3\omega_3\mathsf{I}]\!]^{\nabla''}$.

Proof. The model is not trivial since it is easy to derive $\vdash^{\nabla'} \mathsf{I} : \tau$ while $\not\vdash^{\nabla'} \omega_3 : \tau$ by Lemma 6.

($\subseteq$) By Lemma 6 we have $\vdash^{\nabla''}\omega_3 : \alpha_i$ for all i, hence, by $(\leq_\nabla)$, $\vdash^{\nabla''}\omega_3 : \beta_i$ and $\vdash^{\nabla''}\omega_3 : \beta_i \to \tau \to W_i$. By rule $(\to \mathrm{E})$ it follows $\vdash^{\nabla''}\omega_3\omega_3 : \tau \to W_i$. Since $\vdash^{\nabla''}\mathsf{I} : \tau$, we conclude by rule $(\to \mathrm{E})$ $\vdash^{\nabla''}\omega_3\omega_3\mathsf{I} : W_i$ for all i, which implies $[\![M]\!]^{\nabla''} \subseteq [\![\omega_3\omega_3\mathsf{I}]\!]^{\nabla''}$.

($\supseteq$) Let $B \in [\![\omega_3\omega_3\mathsf{I}]\!]^{\nabla''}$. The only interesting case is $B \not\sim_{\nabla''} \Omega$. We get $\vdash^{\nabla''}\omega_3\omega_3\mathsf{I} : B$ and by Lemma 3(2), there exist $A, \bar{Q}$ such that $A \leq_{\nabla''} A \to A \to \bar{Q} \to B$ and $\vdash^{\nabla} \omega_3 : A$. By Lemma 7(2) there are I, $\alpha_i \in \mathbb{C}^{\nabla''}$, and n such that $C \leq_{\nabla''} C \to C \to A^n \to \bar{Q} \to B$, where $C \equiv \bigcap_{i\in I} \alpha_i$. By applying $n+1$ times Lemma 7(1) we get $\bigcap_{i\in J} \alpha_i \leq_{\nabla''} D \to \bar{Q} \to B$ for some $J \subseteq I$, where $D \equiv C$ if $n = 0$ and $D \equiv A$ otherwise. By Theorem 1 this gives $(\bigcap_{i\in H} \alpha_i) \cap (\bigcap_{i\in K}(\tau \to W_i)) \leq_{\nabla''} \bar{Q} \to B$ for some $H, K \subseteq J$, being $\alpha_i \sim_{\nabla''} (\gamma_i \to \alpha_i) \cap (\beta_i \to \tau \to W_i)$. Let $L = \{i \in H \mid W_i \sim_{\nabla''} \Omega\}$. Since, by Lemma 5 $W_i \not\sim_{\nabla''} \Omega$ implies $\bar{Q} \not\leq_{\nabla''} \gamma_i, \beta_i$, if we apply Theorem 1 to $(\bigcap_{i\in H} \alpha_i) \cap (\bigcap_{i\in K}(\tau \to W_i)) \leq_{\nabla''} \bar{Q} \to B$ we have that there exist $L' \subseteq L$ and $K' \subseteq K$ such that $(\bigcap_{i\in L'} \alpha_i) \cap (\bigcap_{i\in K'} W_i) \leq_{\nabla''} B$. Notice that $W_i \sim_{\nabla''} \Omega$ gives $\alpha_i \sim_{\nabla''} \alpha_i \to \alpha_i$. The proof by structural induction on terms that $\{x{:}A \mid x \in \mathrm{FV}(N)\} \vdash^{\nabla} N : A$ whenever $A \sim_{\nabla''} A \to A$ is easy. This implies $\vdash^{\nabla''} M : \alpha_i$ being M closed, i.e. $\alpha_i \in [\![M]\!]^{\nabla''}$ for all $i \in L'$. Moreover by construction $W_i \in [\![M]\!]^{\nabla''}$ for all $i \in K'$. Since $[\![M]\!]^{\nabla''}$ is a filter, $(\bigcap_{i\in L'} \alpha_i) \cap (\bigcap_{i\in K'} W_i)$ is in it, hence B too. □

5 Extensional Models

In this section we will show how to modify the model construction of previous sections in order to obtain extensional filter models. In this way we will prove the consistency of $\lambda\beta\eta + \{\omega_2\omega_2 = M\}$ and of $\lambda\beta\eta + \{\omega_3\omega_3\mathsf{I} = M\}$.

5.1 Consistency of $\boldsymbol{\lambda\beta\eta + \{\omega_2\omega_2 = M\}}$

If in Definition 8 we put:

- $\mathbb{C}^{\nabla_1} = \{\Omega, \chi\}$;
- $\nabla_1 = \nabla_0 \cup \{\chi \sim \Omega \to \chi\}$;

we obtain an eitt we call $\Sigma^{\nabla'_\eta}$. By Theorem 4, $\langle \mathcal{F}^{\nabla'_\eta}, \cdot, [\![\ ]\!]^{\nabla'_\eta} \rangle$ is an extensional λ-model. It is essentially the model of [20].

In $\Sigma^{\nabla'_\eta}$ the type χ plays the role of ι in $\Sigma^{\nabla'}$. So instead of Lemma 2 we need:

Lemma 8.

$$\nvdash^{\nabla'} \omega_2 : \chi \ \textit{and}\ \nvdash^{\nabla'} \omega_2 : (\chi \to \chi) \to \chi \to \chi.$$

Proof. Define the set $\mathcal{E}_\Omega$ as the minimal set such that:

$$\Omega \in \mathcal{E}_\Omega; \quad A \in \mathbb{T}^{\nabla'_\eta}, B \in \mathcal{E}_\Omega \ \Rightarrow\ A \to B \in \mathcal{E}_\Omega;$$
$$A, B \in \mathcal{E}_\Omega \ \Rightarrow\ A \cap B \in \mathcal{E}_\Omega; \quad W_i \in \mathcal{E}_\Omega \ \Rightarrow\ \varsigma_i \in \mathcal{E}_\Omega.$$

It is easy to check by induction on $\leq_{\nabla'_\eta}$ that for all $A \in \mathbb{T}^{\nabla'_\eta}$:

$$B \leq_{\nabla'_\eta} A \;\&\; B \in \mathcal{E}_\Omega \ \Rightarrow\ A \in \mathcal{E}_\Omega.$$

This implies that $\Omega \not\leq_{\nabla'_\eta} \chi$ since $\Omega \in \mathcal{E}_\Omega$ and $\chi \notin \mathcal{E}_\Omega$.
Suppose by contradiction that $\vdash^{\nabla'_\eta} \omega_2 : \chi$, i.e. $\vdash^{\nabla'_\eta} \omega_2 : \Omega \to \chi$: we get $\Omega \leq_{\nabla'_\eta} \Omega \to \chi \sim_{\nabla'_\eta} \chi$ by Lemma 1(1).
Similarly from $\vdash^{\nabla'_\eta} \omega_2 : (\chi \to \chi) \to \chi \to \chi$ by Lemma 1(1) we get $\chi \to \chi \leq_{\nabla'_\eta} (\chi \to \chi) \to (\chi \to \chi) \to \chi \to \chi$, which implies by Theorem 1 $\chi \to \chi \leq_{\nabla'_\eta} \chi$. Applying again Theorem 1 to this last judgment we get $\Omega \leq_{\nabla'_\eta} \chi$. □

The model $\langle \mathcal{F}^{\nabla'_\eta}, \cdot, [\![\ ]\!]^{\nabla'_\eta} \rangle$ is not trivial since it is easy to derive $\vdash^{\nabla'_\eta} \mathsf{I} : (\chi \to \chi) \to \chi \to \chi$ while $\nvdash^{\nabla'_\eta} \omega_2 : (\chi \to \chi) \to \chi \to \chi$ by Lemma 8. The proof that this model equates $\omega_2\omega_2$ and M is just the proof of Theorem 6 using Lemma 8 instead of Lemma 2.

5.2 Consistency of $\lambda\beta\eta + \{\omega_3\omega_3\mathsf{I} = M\}$

If in Definition 9 we put:

- $\mathbb{C}^{\nabla_1} = \{\Omega, \sigma, \tau\}$;
- $\nabla_1 = \nabla_0 \cup \{\sigma \sim \tau \to \sigma, \tau \sim \sigma \to \sigma\}$;

and we erase the condition $\tau \equiv \iota \to \iota$, we obtain an eitt that we call $\Sigma^{\nabla''_\eta}$.[2] Notice that in this case it is not possible to start from a simpler theory Σ^{∇_1} built on just two constants, say Ω and χ, as in the case of $\Sigma^{\nabla'_\eta}$. If Σ^{∇_1} contains only two constants, we are forced to define $\tau \equiv \chi \to \chi$. Whatever equivalence we choose to make each constant equivalent to a suitable intersection of arrow, we derive type τ for ω_3, which contradicts Lemma 10(2) below.

By Theorem 4 $\langle \mathcal{F}^{\nabla''_\eta}, \cdot, [\![\,]\!]^{\nabla''_\eta} \rangle$ is an extensional λ-model. We prove that this model equates $\omega_3\omega_3\mathsf{I}$ to M just mimicking the same proof for the model $\langle \mathcal{F}^{\nabla''}, \cdot, [\![\,]\!]^{\nabla''} \rangle$.

First we introduce five subsets of $\mathbb{T}^{\nabla''_\eta}$, whose meaning is the following:

1. $\mathcal{E}_\Omega$ is the set of all types which are equivalent to Ω;
2. $\mathcal{G}_\sigma$ is the set of all types which are greater than or equivalent to σ;
3. $\mathcal{L}_\sigma$ is the set of all types which are less than or equivalent to σ;
4. $\mathcal{G}_\tau$ is the set of all types which are greater than or equivalent to τ;
5. $\mathcal{L}_\tau$ is the set of all types which are less than or equivalent to τ.

Definition 11. *The sets $\mathcal{E}_\Omega$, $\mathcal{G}_\sigma$, $\mathcal{L}_\sigma$, $\mathcal{G}_\tau$, $\mathcal{L}_\tau$, are defined as the minimal sets such that:*

$$\begin{array}{l}
\Omega \in \mathcal{E}_\Omega; \quad A \in \mathbb{T}^{\nabla''_\eta}, B \in \mathcal{E}_\Omega \;\Rightarrow\; A \to B \in \mathcal{E}_\Omega; \quad A, B \in \mathcal{E}_\Omega \;\Rightarrow\; A \cap B \in \mathcal{E}_\Omega; \\
\sigma \in \mathcal{G}_\sigma; \quad \mathcal{E}_\Omega \subseteq \mathcal{G}_\sigma; \quad A \in \mathcal{L}_\tau, B \in \mathcal{G}_\sigma \;\Rightarrow\; A \to B \in \mathcal{G}_\sigma; \quad A, B \in \mathcal{G}_\sigma \;\Rightarrow\; A \cap B \in \mathcal{G}_\sigma; \\
\sigma \in \mathcal{L}_\sigma; \quad A \in \mathcal{G}_\tau, B \in \mathcal{L}_\sigma \;\Rightarrow\; A \to B \in \mathcal{L}_\sigma; A \in \mathbb{T}^{\nabla''_\eta}, B \in \mathcal{L}_\sigma \;\Rightarrow\; A \cap B, B \cap A \in \mathcal{L}_\sigma; \\
\tau \in \mathcal{G}_\tau; \quad \mathcal{E}_\Omega \subseteq \mathcal{G}_\tau; \quad A \in \mathcal{L}_\sigma, B \in \mathcal{G}_\sigma \;\Rightarrow\; A \to B \in \mathcal{G}_\tau; \quad A, B \in \mathcal{G}_\tau \;\Rightarrow\; A \cap B \in \mathcal{G}_\tau; \\
\tau \in \mathcal{L}_\tau; \quad A \in \mathcal{G}_\sigma, B \in \mathcal{L}_\sigma \;\Rightarrow\; A \to B \in \mathcal{L}_\tau; \quad A \in \mathbb{T}^{\nabla''_\eta}, B \in \mathcal{L}_\tau \;\Rightarrow\; A \cap B, B \cap A \in \mathcal{L}_\tau. \square
\end{array}$$

Similarly to Lemma 4 we can easily show:

Lemma 9.

1. $\mathcal{E}_\Omega = \{A \in \mathbb{T}^{\nabla''_\eta} \mid \Omega \leq_{\nabla''_\eta} A\}$;

2. $\mathcal{G}_\sigma = \{A \in \mathbb{T}^{\nabla''_\eta} \mid \sigma \leq_{\nabla''_\eta} A\}$; *4.* $\mathcal{G}_\tau = \{A \in \mathbb{T}^{\nabla''_\eta} \mid \tau \leq_{\nabla''_\eta} A\}$;

3. $\mathcal{L}_\sigma = \{A \in \mathbb{T}^{\nabla''_\eta} \mid A \leq_{\nabla''_\eta} \sigma\}$; *5.* $\mathcal{L}_\tau = \{A \in \mathbb{T}^{\nabla''_\eta} \mid A \leq_{\nabla''_\eta} \tau\}$.

From previous lemma we immediately have that σ and τ are incomparable:

Corollary 1. *$\sigma \not\leq_{\nabla''_\eta} \tau$ and $\tau \not\leq_{\nabla''_\eta} \sigma$.*

In correspondence with Lemmas 5, 6, 7 we have:

[2] Remark that $\tau \equiv \iota \to \iota$ in $\Sigma^{\nabla''}$, while τ is an atom in $\Sigma^{\nabla''_\eta}$. We use the same name since this allows us to have the same defininition of $\alpha_{(n,m)}$.

Lemma 10.

1. *If* $W_i \not\sim_{\nabla''_\eta} \Omega$ *then* $\gamma_i \not\leq_{\nabla''_\eta} \beta_i \not\leq_{\nabla''_\eta} \alpha_i$ *and* $\bar{Q} \not\leq_{\nabla''_\eta} \gamma_i$.
2. *Let* $\psi \in \mathbb{C}^{\nabla''}$, *then* $\vdash^{\nabla''_\eta} \omega_3 : \psi$ *iff* $\alpha_i \leq_{\nabla''_\eta} \psi$ *for some* i.
3. *If* $\bigcap_{i \in I} \alpha_i \leq_{\nabla''_\eta} A \to B \to C$ *and* $\vdash^{\nabla''_\eta} \omega_3 : B$ *then there is* $J \subseteq I$ *such that* $\bigcap_{i \in J} \alpha_i \leq_{\nabla''_\eta} B \to C$.
4. *If* $A \leq_{\nabla''_\eta} A \to A \to B$ *and* $\vdash^{\nabla''_\eta} \omega_3 : A$ *then there are* I, $\alpha_i \in \mathbb{C}^{\nabla''_\eta}$, *and* n *such that* $C \leq_{\nabla''_\eta} C \to C \to A^n \to B$, *where* $C \equiv \bigcap_{i \in I} \alpha_i$.

Proof. Almost all proofs are the same as those of the corresponding lemmas in Section 4, by using Lemma 9 instead of Lemma 4. We only need to prove:

$$(a) \not\vdash^{\nabla''_\eta} \omega_3 : \tau;\ (b) \not\vdash^{\nabla''_\eta} \omega_3 : \sigma.$$

(a) If $\vdash^{\nabla''_\eta} \omega_3 : \tau \sim_{\nabla''_\eta} \sigma \to \sigma$, then by Lemma 3(1), it should be $\sigma \leq_{\nabla''_\eta} \sigma \to \sigma \to \sigma$, which implies $\tau \to \sigma \leq_{\nabla''_\eta} \sigma \to \tau$. In particular, by Theorem 1, being $\tau \not\sim_{\nabla''_\eta} \Omega$ by Lemma 9(1), it should be $\sigma \leq_{\nabla''_\eta} \tau$, which is impossible by Corollary 1.
(b) is proven similarly to previous Point. If $\vdash^{\nabla''_\eta} \omega_3 : \sigma$, it should hold $\tau \leq_{\nabla''_\eta} \tau \to \tau \to \sigma$. Since $\tau \sim_{\nabla''_\eta} \sigma \to \sigma$, this should implies $\tau \leq_{\nabla''_\eta} \sigma$, which is impossible by Corollary 1. □

The model $\langle \mathcal{F}^{\nabla''_\eta}, \cdot, [\![\ ]\!]^{\nabla''_\eta} \rangle$is not trivial since it is easy to derive $\vdash^{\nabla''_\eta} \mathsf{I} : \tau$ while $\not\vdash^{\nabla''_\eta} \omega_3 : \tau$ by the proof of Lemma 10(2). The proof that this model equates $\omega_3\omega_3\mathsf{I}$ and M is just the proof of Theorem 8 using Lemma 10 instead of Lemmas 5, 6, 7.

References

1. S. Abramsky. Domain theory in logical form. *Ann. Pure Appl. Logic*, 51(1-2):1–77, 1991. 17, 23
2. S. Abramsky and C.-H. L. Ong. Full abstraction in the lazy lambda calculus. *Inform. and Comput.*, 105(2):159–267, 1993. 18
3. F. Alessi. *Strutture di tipi, teoria dei domini e modelli del lambda calcolo*. PhD thesis, Torino University, 1991. 18
4. J. Baeten and B. Boerboom. ω can be anything it shouldn't be. *Indag.Math.*, 41:111–120, 1979. 18
5. H. Barendregt. *The Lambda Calculus: its Syntax and Semantics*. North-Holland, Amsterdam, revised edition, 1984. 18
6. H. Barendregt, M. Coppo, and M. Dezani-Ciancaglini. A filter lambda model and the completeness of type assignment. *J. Symbolic Logic*, 48(4):931–940, 1983. 17, 18
7. H. Barendregt et. al. *Typed λ-calculus and applications*. North-Holland, 2001. (to appear). 19
8. A. Berarducci and B. Intrigila. Some new results on easy lambda-terms. *Theoret. Comput. Sci.*, 121:71–88, 1993. 18

9. A. Berarducci and B. Intrigila. Church-Rosser λ-theories, infinite λ-calculus and consistency problems. In W. Hodges and M. Hyland et al., editors, *Logic: From Foundations to applications*, pages 33–58. Oxford Sci. Publ., New York, 1996. 18
10. M. Coppo and M. Dezani-Ciancaglini. An extension of the basic functionality theory for the λ-calculus. *Notre Dame J. Formal Logic*, 21(4):685–693, 1980. 17
11. M. Coppo, M. Dezani-Ciancaglini, F. Honsell, and G. Longo. Extended type structures and filter lambda models. In *Logic colloquium '82*, pages 241–262. North-Holland, Amsterdam, 1984. 17, 18, 23
12. M. Coppo, M. Dezani-Ciancaglini, and B. Venneri. Principal type schemes and λ-calculus semantics. In R.Hindley and J.Seldin, editors, *To H. B. Curry: essays on combinatory logic, lambda calculus and formalism*, pages 535–560. Academic Press, London, 1980. 17
13. M. Coppo, M. Dezani-Ciancaglini, and M. Zacchi. Type theories, normal forms, and D_∞-lambda-models. *Inform. and Comput.*, 72(2):85–116, 1987. 17, 18
14. M. Dezani-Ciancaglini, F. Honsell, and F. Alessi. A complete characterization of the complete intersection-type theories. In J. Rolim and A. Broder et al., editors, *ICALP Workshops 2000*, volume 8 of *Proceedings in Informatics*, pages 287–302. Carleton-Scientific, Canada, 2000. 17
15. M. Dezani-Ciancaglini, F. Honsell, and Y. Motohama. Compositional characterization of λ-terms using intersection types. In M. Nielsen and B. Rovan, editors, *Mathematical Foundations of Computer Science 2000*, volume 1893 of *Lecture Notes in Comput. Sci.*, pages 304–313. Springer, 2000. 17
16. P. Di Gianantonio and F. Honsell. An abstract notion of application. In M.Bezem and J.F.Groote, editors, *Typed lambda calculi and applications*, volume 664 of *Lecture Notes in Comput. Sci.*, pages 124–138. Springer, Berlin, 1993. 18
17. L. Egidi, F. Honsell, and S. Ronchi Della Rocca. Operational, denotational and logical descriptions: a case study. *Fund. Inform.*, 16(2):149–169, 1992. 18
18. R. Hindley and G. Longo. Lambda-calculus models and extensionality. *Z. Math. Logik Grundlag. Math.*, 26(4):289–310, 1980. 24
19. F. Honsell and M. Lenisa. Semantical analysis of perpetual strategies in λ-calculus. *Theoret. Comput. Sci.*, 212(1-2):183–209, 1999. 18
20. F. Honsell and S. Ronchi Della Rocca. A filter model for Ω. Technical report, Torino University, 1984. 18, 33
21. F. Honsell and S. Ronchi Della Rocca. An approximation theorem for topological lambda models and the topological incompleteness of lambda calculus. *J. Comput. System Sci.*, 45(1):49–75, 1992. 18
22. B. Intrigila. A problem on easy terms in λ-calculus. *Fund. Inform.*, 15.1:99–106, 1991. 18
23. G. Jacopini. A condition for identifying two elements of whatever model of combinatory logic. In C.Böhm, editor, *λ-calculus and computer science theory*, volume 37 of *Lecture Notes in Comput. Sci.*, pages 213–219. Springer, Berlin, 1975. 18
24. G. Jacopini and M. Venturini Zilli. Easy terms in the lambda calculus. *Fund. Inform.*, 80:225–233, 1985. 18
25. R. Kerth. *Isomorphisme et Équivalence Équationnelle entre Modèles du λ-Calcul.* PhD thesis, Equipe de Logique Mathématique, Université Paris VII, 1995. 18
26. J. Kuper. On the Jacopini technique. *Inform. and Comput.*, 138:101–123, 1997. 18
27. J. Mitchell. *Foundation for Programmimg Languages.* MIT Press, 1996. 18
28. G. D. Plotkin. Set-theoretical and other elementary models of the λ-calculus. *Theoret. Comput. Sci.*, 121(1-2):351–409, 1993. 18

29. G. Pottinger. A type assignment for the strongly normalizable λ-terms. In R.Hindley and J.Seldin, editors, *To H. B. Curry: essays on combinatory logic, lambda calculus and formalism*, pages 561–577. Academic Press, London, 1980. 17

30. A. Pravato, S. Ronchi, and L. Roversi. The call-by-value lambda calculus: a semantic investigation. *Math. Struct. in Comput. Sci.*, 9(5):617–650, 1999. 18

31. D. Scott. Continuous lattices. In F. Lawvere, editor, *Toposes, algebraic geometry and logic*, volume 274 of *Lecture Notes in Math.*, pages 97–136. Springer, Berlin, 1972. 17

32. S. Vickers. *Topology via logic.* Cambridge University Press, Cambridge, 1989. 23

33. C. Zylberajch. *Syntaxe et Semantique de la Facilité en Lambda-calcul.* PhD thesis, Université Paris VII, 1991. 18

Confluence of Untyped Lambda Calculus via Simple Types

Silvia Ghilezan[1,2] and Viktor Kunčak[3]

[1] Faculty of Engineering, University of Novi Sad, Yugoslavia
[2] Computing Science Department, Catholic University
Nijmegen, The Netherlands
silviagh@cs.kun.nl
[3] Laboratory of Computer Science, MIT
Cambridge, USA
vkuncak@mit.edu

Abstract. We present a new proof of confluence of the untyped lambda calculus by reducing the confluence of β-reduction in the untyped lambda calculus to the confluence of β-reduction in the simply typed lambda calculus. This is achieved by embedding typed lambda terms into simply typed lambda terms. Using this embedding, an auxiliary reduction, and β-reduction on simply typed lambda terms we define a new reduction on all lambda terms. The transitive closure of the reduction defined is β-reduction on all lambda terms. This embedding allows us to use the confluence of β-reduction on simply typed lambda terms and thus prove the confluence of the reduction defined. As a consequence we obtain the confluence of β-reduction in the untyped lambda calculus.

1 Introduction

The main axiom of lambda calculus is the axiom of β-reduction. The well-known scheme of one step β-reduction is

$$(\lambda x.M)N \to_\beta M[x := N],$$

whereas β-reduction $\twoheadrightarrow_\beta$ is the reflexive transitive closure of $\to_\beta$.

The *Church-Rosser property* or *confluence* is a fundamental property that holds for β-reduction, $\twoheadrightarrow_\beta$, in the untyped lambda calculus. It states that if

$$M_1 \,{}_\beta\!\twoheadleftarrow M \twoheadrightarrow_\beta M_2$$

for any M, M_1, and M_2, then

$$M_1 \twoheadrightarrow_\beta M_3 \,{}_\beta\!\twoheadleftarrow M_2$$

for some lambda term M_3. There are various approaches and proofs of this property (Barendregt [1], Koletsos and Stavrinos [8], Pfenning [13], Takahashi [16], van Oostrom [17]).

In order to prove the confluence of $\twoheadrightarrow_\beta$ it suffices to prove the confluence of any relation whose transitive closure is $\twoheadrightarrow_\beta$. Unfortunately, one step β-reduction $\to_\beta$ is not confluent. If

$$M_1 \,{}_\beta\!\leftarrow M \to_\beta M_2,$$

then

$$M_1 \twoheadrightarrow_\beta M_3 \,{}_\beta\!\twoheadleftarrow M_2$$

A. Restivo, S. Ronchi Della Rocca, L. Roversi (Eds.): ICTCS 2001, LNCS 2202, pp. 38–49, 2001.

for some M_3, which is referred to as the *weak Church-Rosser property* or *local confluence*. It does not necessarily mean that $M_1 \to_\beta M_3 \,{}_\beta{\leftarrow} M_2$, since the contraction of one redex may duplicate others. Due to the possibility of infinite reduction sequences in the untyped lambda calculus, the confluence cannot be inferred immediately from the local confluence. It is well-known according to Newman's lemma (Newman [12]) that local confluence implies confluence on the set of strongly normalizing lambda terms.

The way to proceed is to find a confluent relation $\to_I$ such that

$$\to_\beta \subseteq \to_I \subseteq \twoheadrightarrow_\beta .$$

Then the transitive closure of $\to_I$ is $\twoheadrightarrow_\beta$, hence the confluence of $\twoheadrightarrow_\beta$ follows immediately.

Although an elementary inductive definition of $\to_I$ is possible, it is not quite clear how to find it. Instead, a deeper understanding of the proof of confluence can be obtained by considering $\to_I$ as an image of a strongly normalizing relation on a different set of lambda terms. This idea is realized in Barendregt [1] by β_0-reduction on the set of marked lambda terms, which is referred to as the finiteness of developments. In Takahashi [16] the notion of parallel reduction is introduced for the same reason. In Koletsos and Stavrinos [8] this idea is achieved by embedding untyped lambda terms into terms typeable with intersection types, which are known to be strongly normalizing.

The central idea of this paper is to reduce the confluence of β-reduction in the untyped lambda calculus to the confluence of β-reduction in the simply typed lambda calculus. For that reason we construct an embedding of untyped lambda terms into simply typed lambda terms. We define the required reduction $\to_I$ on all lambda terms using this embedding, an auxiliary reduction, and β-reduction on simply typed lambda terms. We show that the transitive closure of $\to_I$ is $\twoheadrightarrow_\beta$. The confluence of the auxiliary reduction makes explicit the joining of the sets of redexes to be reduced. This embedding allows us to use the confluence of $\twoheadrightarrow_\beta$ on simply typed lambda terms and thus prove the confluence of $\to_I$. As a consequence we obtain the confluence of $\twoheadrightarrow_\beta$ in the (untyped) lambda calculus.

Section 2 contains the outline of our proof as well as of the proof of confluence presented in Barendregt [1]. In order to keep the work self-contained the notion of simple types and simply typed lambda calculus is presented in Section 3. In Section 4 we define the embedding of (untyped) lambda terms into simply typed ones, the auxiliary reduction on simply typed lambda terms and the required reduction $\to_I$ on lambda terms. In Section 5 the relation between reductions in simply typed lambda calculus is considered. In Section 6 the investigation of the properties of $\to_I$ leads to the confluence of β-reduction in the untyped lambda calculus.

2 Outline of the Proof

The proof of the Church-Rosser theorem for untyped lambda calculus in Barendregt [1] can be represented by diagram in Figure 1. Let Λ denote the set of all (untyped) lambda terms. The set Λ' of marked lambda terms is defined in Barendregt [1].

Definition 2.1. *(i)* $\Lambda = \mathit{Var} | \lambda \mathit{Var} \Lambda | \Lambda\Lambda$*, the set of all lambda terms.*
(ii) $\Lambda' = \mathit{Var} | \lambda \mathit{Var}\Lambda' | \Lambda'\Lambda' | (\lambda_0 \mathit{Var}.\Lambda')\Lambda'$*, the set of marked lambda terms.*

A redex of the form $(\lambda_0 x.M)N$ is called a *marked* redex. Note that a lambda abstraction may be marked only if it is a part of a redex. The notion of β_0-reduction is defined as a process of contracting only marked redexes.

Definition 2.2. $\beta_0 = \{((\lambda_0 x.M)N, M[x := N]) \mid M, N \in \Lambda'\}$.

In Figure 1 the notation $M \rightarrow_- N$ means that the term N is obtained from M by marking a lambda abstraction in one of the redexes in M and $\twoheadrightarrow_-$ is its transitive closure. The relation $\rightarrow_I$ is defined by composition

$$\rightarrow_I = \twoheadrightarrow_- \circ \twoheadrightarrow_{\beta_0} \circ_- \twoheadleftarrow .$$

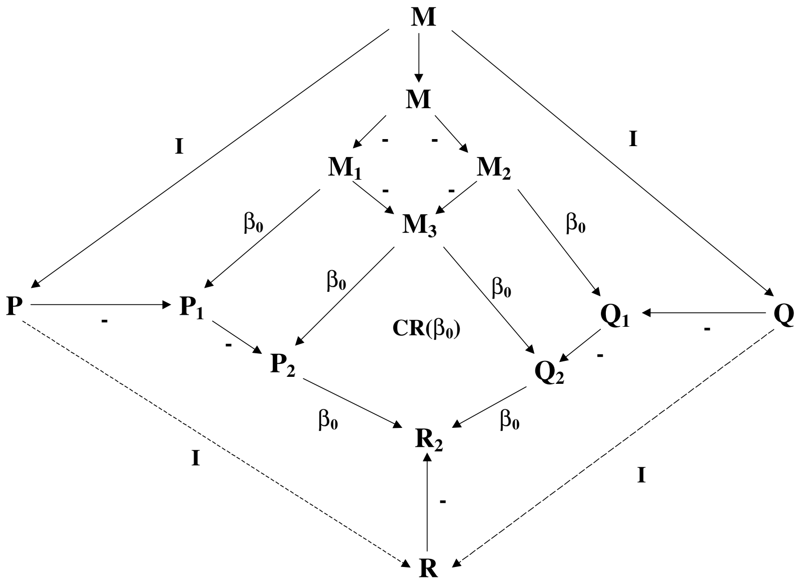


Fig. 1. Proof of the confluence of $\rightarrow_I$ using β_0-reduction (in the interior part of the diagram the arrow $\rightarrow$ denotes $\twoheadrightarrow$ in the standard lambda calculus terminology)

The point of introducing β_0-reduction is that no "new" redexes are created during the reduction since in the substitution $M[x:=N]$ the term N can be an already existing marked redex, but it can never be a marked lambda abstraction, as we noticed previously. A consequence of this is that β_0-reduction is strongly normalizing, i.e. there are no infinite sequences of β_0-reductions. This is proved in Barendregt [1] using an ordering on marked lambda terms. Also, β_0-reduction is proved to be locally confluent and then altogether, by Newman's Lemma (Newman [12]), it follows that β_0-reduction is confluent on marked lambda terms. This result then leads to the confluence of $\rightarrow_I$ and consequently of β-reduction.

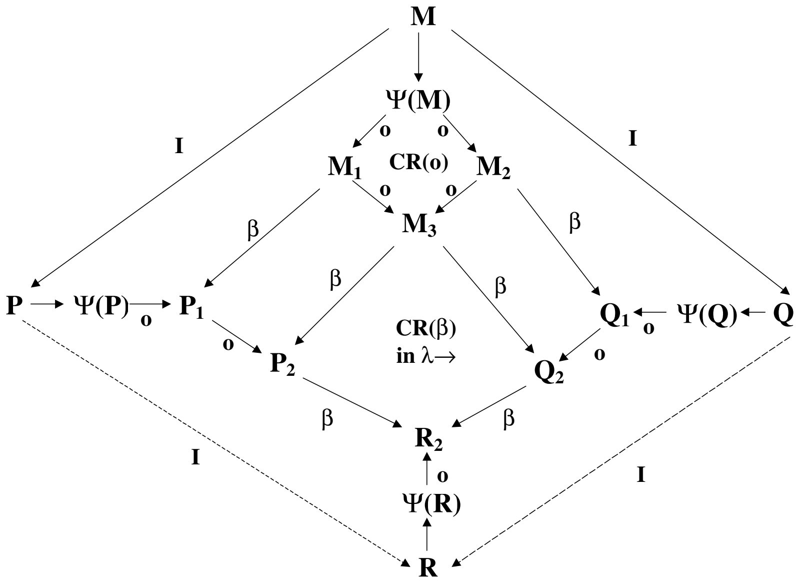


Fig. 2. Proof of the confluence of $\rightarrow_I$ using an embedding in $\lambda\!\rightarrow$ (in the interior part of the diagram the arrow $\rightarrow$ denotes $\twoheadrightarrow$ in the standard lambda calculus terminology)

Our proof has the structure presented by diagram in Figure 2. The embedding Ψ maps untyped terms into terms in the simply typed lambda calculus using constants f and g that can be thought of as a retraction pair used in the interpretation of the simply typed lambda calculus (see Scott [14], Wadsworth [19], and Meyer [10]). From the syntactical point of view Ψ blocks all applications. Therefore Ψ blocks all redexes as well, replacing $(\lambda x.M)N$ by $f(g(\lambda x.M))N$. The notion of o-reduction ($\twoheadrightarrow_o$) is introduced to play an analogous role to the lambda abstraction marking ($\twoheadrightarrow_{\_}$): it replaces a blocked redex $f(g(\lambda x.M))N$ by the unblocked redex $(\lambda x.M)N$ and leaves other applications which are not redexes blocked. In this case ordinary β-reduction in the simply typed lambda calculus will play a similar role to β_0-reduction on marked lambda terms, due to the fact that β-reduction is confluent on simply typed lambda terms. The relation $\rightarrow_I$ is defined to be an inverse image of β-reduction. It is proved that $\rightarrow_I$ has all the required properties according to which β-reduction is proved to be confluent on (untyped) lambda terms.

3 Simply Typed Lambda Calculus

The notion of simple types and the notion of *simply typed lambda calculus* $\lambda\!\!\rightarrow$ are formulated in a suitable way. All types are generated from a basic type 0 in the usual way.

Definition 3.1. *The set* `type` *of types is defined as follows.*

$$\boxed{\texttt{type} = 0 \mid \texttt{type} \rightarrow \texttt{type}}$$

A *type assignment* is an expression of the form $M : \varphi$, where $M \in \Lambda$ and $\varphi \in \texttt{type}$. A *context* Γ is a set $\{x_1 : \sigma_1, \ldots, x_n : \sigma_n\}$ of type assignments with different term variables.

Definition 3.2 (Type assignment system $\lambda\!\!\rightarrow$).
The type assignment $P : \varphi$ is derivable from the context Γ in $\lambda\!\!\rightarrow$, notation $\Gamma \vdash P : \varphi$, if $\Gamma \vdash P : \varphi$ can be generated by the following axiom-scheme and rules.

$$\boxed{\begin{array}{ll}
(ax) & \Gamma, x : \sigma \vdash x : \sigma \\[2em]
(\rightarrow E) & \dfrac{\Gamma \vdash M : \sigma \rightarrow \tau \quad \Gamma \vdash N : \sigma}{\Gamma \vdash MN : \tau} \\[2em]
(\rightarrow I) & \dfrac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash (\lambda x.M) : \sigma \rightarrow \tau}
\end{array}}$$

The crucial point of our proof is the confluence of $\twoheadrightarrow_\beta$ in $\lambda\!\!\rightarrow$. This is proved by Newman's Lemma since β-reduction is locally confluent and the set of simply typed lambda terms is strongly normalizing. There are direct proofs of this property using reducibility arguments and logical relations in Koletsos [7], Statman [15], and Mitchell [11].

Theorem 3.3 (Confluence of $\twoheadrightarrow_\beta$ in $\lambda\!\!\rightarrow$).
The reduction $\twoheadrightarrow_\beta$ is confluent on simply typed lambda terms.

4 Embedding of Lambda Terms into Simply Typed Terms

In order to construct the embedding of lambda terms into simply typed terms we will point out two predefined constants f and g. Hereafter we implicitly assume that untyped lambda terms from Λ do not contain constants f and g. The set of terms typeable by the basic type 0 in the simply typed lambda calculus with the predefined constants f and g will be denoted by Λ_0.

Definition 4.1. *(i)* $\Gamma_0 = \{f : 0 \to (0 \to 0), g : (0 \to 0) \to 0\}$.
(ii) $\Lambda_0 = \{M \in \Lambda \mid (\exists x_1, \ldots, x_n)\, \Gamma_0, x_1 : 0, \ldots, x_n : 0 \vdash M : 0\}$.

The idea is to show that terms in the interior region of diagram 2, i.e. all terms but M, P, Q, and R, are terms typeable by type 0 in the simply typed lambda calculus with predefined constants f and g, namely that they belong to Λ_0.

Note that some of our definitions are stated on more general sets of lambda terms than needed. This makes it simple to immediately check their validity. Later propositions will make the intended domains clear.

The embedding $\Psi : \Lambda \to \Lambda_0$ that allows the representation of arbitrary untyped terms in the simply typed lambda calculus is defined as follows.

Definition 4.2. *(i)* $\Psi(x) = x$.
(ii) $\Psi(MN) = f\Psi(M)\Psi(N)$.
(iii) $\Psi(\lambda x.M) = g(\lambda x.\Psi(M))$.

It is straightforward to verify that $\Psi : \Lambda \to \Lambda_0$.

Proposition 4.3. $(\forall P \in \Lambda)\, \Psi(P) \in \Lambda_0$.

Proof. By induction on the construction of the term P.

Case $P \equiv x$. Clearly, $\Gamma_0, x : 0 \vdash x : 0$.

Case $P \equiv MN$. By the induction hypothesis $\Gamma_0, x_1 : 0, \ldots, x_n : 0 \vdash \Psi(M) : 0$ and $\Gamma_0, y_1 : 0, \ldots, y_m : 0 \vdash \Psi(N) : 0$. Since $f : 0 \to (0 \to 0) \in \Gamma_0$, we have

$$\Gamma_0, z_1 : 0, \ldots, z_k : 0 \vdash f\Psi(M)\Psi(N) : 0,$$

where $\{z_1, \ldots z_k\} = \{x_1, \ldots, x_n, y_1, \ldots, y_m\}$.

Case $P \equiv (\lambda x.M)$. By the induction hypothesis $\Gamma_0, x : 0, x_1 : 0, \ldots, x_n : 0 \vdash \Psi(M) : 0$. Therefore, $\Gamma_0, x_1 : 0, \ldots, x_n : 0 \vdash (\lambda x.\Psi(M)) : 0 \to 0$. Since $g : (0 \to 0) \to 0 \in \Gamma_0$, it follows that $\Gamma_0, x_1 : 0, \ldots, x_n : 0 \vdash g(\lambda x.\Psi(M)) : 0$.

This was the idea behind the definition of Ψ and it will allow us to use the confluence of β-reduction in the simply typed lambda calculus. Obviously, f and g act as a retraction pair for a Scott domain (see Scott [14]).

Next we define the o-relation which induces the auxiliary reduction $\twoheadrightarrow_o$. This corresponds to the marking of lambda abstractions that are parts of already existing redexes.

Definition 4.4. $o = \{(f(g(\lambda x.M))N, (\lambda x.M)N) \mid M, N \in \Lambda\}$.

As usual, $\rightarrow_o$ is the congruent closure of o and $\twoheadrightarrow_o$ is the reflexive transitive closure of $\rightarrow_o$. We use the term *unblocked redex* to denote a β-redex and we refer to a term of the form $f(g(\lambda x.M))N$ as a *blocked redex*. Hence, Ψ blocks all applications (including redexes) and o-reduction turns blocked redexes into unblocked redexes leaving blocked applications which are not redexes.

Proposition 4.5. *(i) Λ_0 is closed under $\twoheadrightarrow_\beta$.*
(ii) Λ_0 is closed under $\twoheadrightarrow_o$.

Proof. (i) Easy, since subject reduction holds in $\lambda\!\rightarrow$.
(ii) Let $P \in \Lambda_0$ and $P \rightarrow_o Q$, where $R \equiv f(g(\lambda x.M))N \rightarrow_o (\lambda x.M)N$ is the contracted o-redex in P. Then $\Gamma_0, \Gamma \vdash P : 0$. According to the types of f and g we obtain that $\Gamma_0, \Gamma \vdash R : 0$. By the typeability of subterms (see Barendregt [2]), it follows that $\Gamma_0, \Gamma \vdash (\lambda x.M) : 0 \rightarrow 0$ and $\Gamma_0, \Gamma \vdash N : 0$. This implies $\Gamma_0, \Gamma \vdash (\lambda x.M)N : 0$. This means that the resulting term Q remains well-typed with the type 0.

Example 4.6. Let $\Omega \equiv (\lambda x.xx)(\lambda x.xx)$. Then

$$\Psi(\Omega) = f(g(\lambda x.fxx))(g(\lambda x.fxx)).$$

Let $N \equiv (\lambda x.fxx)(g(\lambda x.fxx))$. Observe that $\Psi(\Omega)$ has no β-redexes, whereas N has no o-redexes. However, there is an infinite reduction

$$\Psi(\Omega) \rightarrow_o N \rightarrow_\beta \Psi(\Omega) \rightarrow_o N \rightarrow_\beta \Psi(\Omega) \rightarrow_o \ldots,$$

which corresponds to the β-reduction $\Omega \rightarrow_\beta \Omega \rightarrow_\beta \Omega \rightarrow_\beta \ldots$ in the untyped lambda calculus. In this way, the combination of β- and o-reduction enables us to simulate infinite β-reductions of untyped lambda terms in $\lambda\!\rightarrow$.

Proposition 4.7 (Confluence of $\rightarrow_o$).
If $M_1 \,{}_o\!\!\leftarrow M \rightarrow_o M_2$, then there is M_3 such that $M_1 \rightarrow_o M_3 \,{}_o\!\!\leftarrow M_2$.

Proof. If M_1 has an unblocked redex Δ_1 and M_2 has an unblocked redex Δ_2, then M_3 has both redexes Δ_1 and Δ_2 unblocked. (In terms of term-rewriting systems, there are no critical pairs because any two o-redexes are either properly contained one in the other or are in disjoint parts of the term, see Dershowitz and Jounnaud [3].)

The previous proposition is even more obvious in terms of redex marking: if M_1 and M_2 are obtained by marking different redexes in M, then M_3 has marked redexes from both M_1 and M_2.

Corollary 4.8 (Confluence of $\twoheadrightarrow_o$).
If $M_1 \,{}_o\!\!\twoheadleftarrow M \twoheadrightarrow_o M_2$, then there is M_3 such that $M_1 \twoheadrightarrow_o M_3 \,{}_o\!\!\twoheadleftarrow M_2$.

Proof. From Proposition 4.7 by simple diagram chasing.

Following the idea that $\to_I$ has to be an image of a strongly normalizing relation, we are now in the position to define the required relation $\to_I$ on untyped lambda terms. A relation, say τ, between untyped lambda terms and terms typeable in $\lambda{\to}$ can be established in the following way: first, Ψ turns untyped terms from Λ into terms from Λ_0 and then $\twoheadrightarrow_o$ unblocks some redexes (the effects analogous to marking lambda terms are achieved). The relation $\to_I$ is then defined as an inverse image w.r.t. τ of β-reduction in $\lambda{\to}$.

Definition 4.9 (Reduction $\to_I$).

(i) $\tau = \Psi \circ \twoheadrightarrow_o$.
(ii) $\to_I = \tau \circ \twoheadrightarrow_\beta \circ \tau^{-1}$, *and it corresponds to the following diagram.*

$$\begin{tikzcd}
M \arrow[r, "I"] \arrow[d, "\tau"] & N \arrow[d, "\tau"] \\
M_0 \arrow[r, "\beta"] & N_0
\end{tikzcd}$$

Example 4.10. Let $M \equiv (\lambda x.xy)(\lambda z.z)$ and $N \equiv (\lambda z.z)y$. Then

$$\Psi(M) \equiv f(g(\lambda x.fxy))(g(\lambda z.z)) \underset{o}{\twoheadrightarrow} M_0 \equiv (\lambda x.fxy)(g(\lambda z.z)).$$

Therefore, $M\tau M_0$. Let $N_0 \equiv \Psi(N) \equiv f(g(\lambda z.z))y$. Trivially, $\Psi(N) \twoheadrightarrow_o N_0$, so $N\tau N_0$. The fact that $M_0 \twoheadrightarrow_\beta N_0$, together with $M\tau M_0$ and $N\tau N_0$, means that $M \to_I N$.

Note that $M \to_I y$ is *not* true. The reason behind this is that the redex present in N was created during the reduction (it did not exist in M).

5 Relating Reductions

The next two lemmas are the key steps for this proof of the confluence of $\to_I$.

Lemma 5.1. *Let* $M \in \Lambda_0$. *If* $M_0 \; {}_o\!\twoheadleftarrow M \twoheadrightarrow_\beta N$, *then* $M_0 \twoheadrightarrow_\beta N_0 \; {}_o\!\twoheadleftarrow N$ *for some* $N_0 \in \Lambda_0$.

$$\begin{tikzcd}
M \arrow[r, "\beta"] \arrow[d, "o"] & N \arrow[d, "o"] \\
M_0 \arrow[r, "\beta"] & N_0
\end{tikzcd}$$

Proof. Note that all unblocked redexes from M are also unblocked in M_0, so in order to perform the reduction $M_0 \twoheadrightarrow_\beta N_0$ just reduce the unblocked redexes corresponding to those reduced in $M \twoheadrightarrow_\beta N$. For $N \twoheadrightarrow_o N_0$, unblock as many redexes as necessary to obtain N_0. According to Proposition 4.5 $N_0 \in \Lambda_0$.

Example 5.2. According to the previous lemma the reductions which correspond to

$$(\lambda x.yxx)((\lambda z.z)y) \underset{o}{\leftarrow} (\lambda x.yxx)(f(g(\lambda z.z))y) \underset{\beta}{\rightarrow} y(f(g(\lambda z.z))y)(f(g(\lambda z.z))y)$$

are

$$(\lambda x.yxx)((\lambda z.z)y) \underset{\beta}{\rightarrow} y((\lambda z.z)y)((\lambda z.z)y) \underset{o}{\twoheadleftarrow} y(f(g(\lambda z.z))y)(f(g(\lambda z.z))y).$$

Lemma 5.3. *Let $M \in \Lambda$. If $M\,\tau\,M_0 \twoheadrightarrow_\beta N_0$, then $M \twoheadrightarrow_\beta N\,\tau\,N_0$ for some $N \in \Lambda$.*

$$\begin{tikzcd}
M \arrow[r, "\beta"] \arrow[d, "\tau"] & N \arrow[d, "\tau"] \\
M_0 \arrow[r, "\beta"] & N_0
\end{tikzcd}$$

Proof. Let $\Psi(M) \twoheadrightarrow_o M_0$. Suppose $M_0 \twoheadrightarrow_\beta N_0$. This β-reduction reduces redexes from M unblocked by $\twoheadrightarrow_o$, so there is a corresponding reduction $M \twoheadrightarrow_\beta N$ for some $N \in \Lambda$, such that $\Psi(N) \twoheadrightarrow_o N_0$.

Proposition 5.4. $\tau(\Lambda) = \{N \mid (\exists M \in \Lambda)\,(M,N) \in \tau\}$ *is closed under* $\twoheadrightarrow_\beta$.

Proof. Given $M_0 \in \tau(\Lambda)$, if $M_0 \twoheadrightarrow_\beta N_0$, then by Lemma 5.3 there is $N \in \Lambda$ such that $N \tau N_0$. Therefore $N_0 \in \tau(\Lambda)$.

Proposition 5.5. $\tau(\Lambda)$ *is closed under* $\twoheadrightarrow_o$.

Proof. Let $M_0 \in \tau(\Lambda)$ and $M_0 \twoheadrightarrow_o N_0$. Then by the definition of τ (Definition 4.9) $\Psi(M) \twoheadrightarrow_o M_0$ for some $M \in \Lambda$. Also, $\Psi(M) \twoheadrightarrow_o N_0$, hence $N_0 \in \tau(\Lambda)$.

Obviously, Proposition 5.4 and Proposition 5.5 provide stronger results than Proposition 4.5. Previous two propositions show that in Figure 2 all but the terms M, P, Q, and R are in $\tau(\Lambda)$.

Corollary 5.6 (Confluence of $\twoheadrightarrow_\beta$ in $\tau(\Lambda)$).
β-reduction is confluent in $\tau(\Lambda)$.

Proof. The statement follows by the confluence of $\twoheadrightarrow_\beta$ in $\lambda\!\rightarrow$ (Theorem 3.3), since $\tau(\Lambda)$ is closed under $\twoheadrightarrow_\beta$ (Proposition 5.4).

6 Confluence of $\rightarrow_I$ and $\twoheadrightarrow_\beta$

In this section we show that $\rightarrow_I$ is the right relation for our purpose, showing that the transitive closure of $\rightarrow_I$ is exactly $\twoheadrightarrow_\beta$.

Lemma 6.1. $\rightarrow_I \subseteq \twoheadrightarrow_\beta$.

Proof. Let $M \rightarrow_I N$. This means that $M \,\tau\, M_0 \twoheadrightarrow_\beta N_0$ and $N \,\tau\, N_0$. By Lemma 5.3, there is some $N_1 \in \Lambda$ such that $M \twoheadrightarrow_\beta N_1$ and also $N_1 \,\tau\, N_0$. Both $\Psi(N_1) \twoheadrightarrow_o N_0$ and $\Psi(N) \twoheadrightarrow_o N_0$, so it follows that by erasing all constants f and g from N_0 we obtain both N_1 and N. Hence $N \equiv N_1$, so $M \twoheadrightarrow_\beta N$.

Lemma 6.2. $\rightarrow_\beta \subseteq \rightarrow_I$.

Proof. Let $M \overset{\Delta}{\rightarrow}_\beta N$ where Δ is a β-redex in M. Let $\Psi(M) \rightarrow_o M_0$ with only the redex Δ unblocked by $\rightarrow_o$. Let $M_0 \overset{\Delta_0}{\rightarrow}_\beta N_0$ with Δ_0 being the redex in M_0 that corresponds to Δ. It is easy to verify that $\Psi(N) \twoheadrightarrow_o N_0$. Now we have that $\Psi(M) \twoheadrightarrow_o M_0 \twoheadrightarrow_\beta N_0 \;{}_o\!\twoheadleftarrow \Psi(N)$, implying $M \rightarrow_I N$.

Proposition 6.3 (Confluence of $\rightarrow_I$).
The reduction $\rightarrow_I$ is confluent on Λ.

Proof. Let $P \;{}_I\!\leftarrow M \rightarrow_I Q$. We obtain the desired term R with the property $P \rightarrow_I R \;{}_I\!\leftarrow Q$ by constructing the diagram in Figure 2 in several steps.

1. By Definition 4.9 of $\rightarrow_I$ we have that

$$\Psi(P) \underset{o}{\twoheadrightarrow} P_1 \underset{\beta}{\twoheadleftarrow} M_1 \underset{o}{\twoheadleftarrow} \Psi(M) \underset{o}{\twoheadrightarrow} M_2 \underset{\beta}{\twoheadrightarrow} Q_1 \underset{o}{\twoheadleftarrow} \Psi(Q).$$

2. Using the diamond property for $\twoheadrightarrow_o$ (Corollary 4.8), we obtain that $M_1 \twoheadrightarrow_o M_3 \;{}_o\!\twoheadleftarrow M_2$.
3. By Lemma 5.1 we have that $P_1 \twoheadrightarrow_o P_2 \;{}_\beta\!\twoheadleftarrow M_3$ nd $M_3 \twoheadrightarrow_\beta Q_2 \;{}_o\!\twoheadleftarrow Q_1$.
4. Notice that $P_2 \;{}_\beta\!\twoheadleftarrow M_3 \twoheadrightarrow_\beta Q_2$. By Proposition 5.4 and Proposition 5.5 all mentioned terms except M, P, and Q are in $\tau(\Lambda)$, hence by confluence of $\twoheadrightarrow_\beta$ in $\tau(\Lambda)$ (Corollary 5.6) we have that $P_2 \twoheadrightarrow_\beta R_2 \;{}_\beta\!\twoheadleftarrow Q_2$ for some $R_2 \in \tau(\Lambda)$.
5. Finally, by the definition of τ (Definition 4.9(i)) there is $R \in \Lambda$ such that $\Psi(R) \twoheadrightarrow_o R_2$. Therefore $\Psi(P) \twoheadrightarrow_o P_1 \twoheadrightarrow_o P_2 \twoheadrightarrow_\beta R_2 \;{}_o\!\twoheadleftarrow \Psi(R)$, which means that $P \rightarrow_I R$ and also $\Psi(R) \twoheadrightarrow_o R_2 \;{}_\beta\!\twoheadleftarrow Q_2 \twoheadleftarrow_o Q_1 \;{}_o\!\twoheadleftarrow \Psi(Q)$, which means $R \;{}_I\!\leftarrow Q$. Now we have $P \rightarrow_I R \;{}_I\!\leftarrow Q$, which completes the proof.

Theorem 6.4 (Confluence of $\twoheadrightarrow_\beta$).
The reduction $\twoheadrightarrow_\beta$ is confluent on the set Λ.

Proof. By Lemmas 6.1 and 6.2, we have $\rightarrow_\beta \subseteq \rightarrow_I \subseteq \twoheadrightarrow_\beta$, therefore the transitive closure of $\rightarrow_I$ is $\twoheadrightarrow_\beta$. By Proposition 6.3, $\rightarrow_I$ is confluent. Now again a simple diagram chasing argument yields the confluence of $\twoheadrightarrow_\beta$ in Λ.

7 Discussion

Confluence is of great importance in the theory of rewriting as well. Beside Newman's Lemma there are several conditions for confluence of abstract rewrite systems. Detailed studies of this subject can be found in van Oostrom [17], van Oostrom and van Raamsdonk [18], and Klop et al. [6].

The proof of the finiteness of developments presented in Ghilezan [4] and [5] is based on the strong normalization property of the simply typed lambda calculus. For that reason a bijection is established there between β_0-reduction on marked lambda terms and β-reduction on a subset of simply typed lambda terms. This bijection explains the correspondence between the proof of the confluence presented in this paper and the proof in Barendregt [1].

In the proof of the confluence in Koletsos and Stavrinos [8] the confluence of β-reduction in the intersection type system was used. The same system was considered in Krivine [9] in order to prove the finiteness of developments. Despite its simplicity, the system of simply typed lambda calculus turned out, as well, to be adequate for proving the confluence of β-reduction on all (untyped) lambda terms.

Our proof, together with the proofs in Krivine [9], Koletsos and Stavrinos [8], and Ghilezan [4] and [5] adds to the understanding of the relation between the typed and the untyped lambda calculi.

Acknowledgment The authors are grateful to George Stavrinos for helpful remarks.

References

1. Barendregt, H. P.: *The Lambda Calculus-Its Syntax and Semantics*. North-Holland, Amsterdam (1984). 38, 39, 40, 41, 48
2. Barendregt, H. P.: Lambda calculi with types. In: Abramsky, S., Gabbay, D. M. and T. S. E. Maibaum (eds.): *Handbook of Logic in Computer Science*, Vol. 2. Oxford University Press, Oxford (1992) 117–309. 44
3. Dershowitz, N. and J. P. Jounnaud: Rewrite Systems. In: Leeuwen, J. (ed.): *Handbook of Theoretical Computer Science*, Elsevier Science Publishers B. V. (1990). 44
4. Ghilezan, S.: Application of typed lambda calculi in the untyped lambda calculus. In: Nerode, A. and Yu. Matiyasevich (eds.): Logical Foundations of Computer Science '94. *Lecture Notes in Computer Science* **813**, Springer-Verlag, Berlin (1994) 129–139. 48
5. Ghilezan, S.: Generalized finiteness of developments in typed lambda calculi. *Journal of Automata, Languages and Combinatorics* **4** (1996) 247–257. 48
6. Klop, J. W., V. van Oostrom, and R. de Vrijer: A geometric proof of confluence by decreasing diagrams. *Journal of Logic and Computation* **10(3)** (2000) 437–460. 48
7. Koletsos, G.: Church-Rosser theorem for typed functionals. *Journal of Symbolic Logic* **50** (1985) 782–790. 42

8. Koletsos, G. and G. Stavrinos: Church-Rosser theorem for conjunctive type systems. In: Kakas, A. K. and A. Sinachopoulos (eds.): *Proceedings of the First Panhellenic Logic Symposium*. Nicosia (1997) 25–37. 38, 39, 48
9. Krivine, J. L.: *Lambda-calcul types et modèles*. Masson, Paris (1990) 48
10. Meyer, A. M.: What is a model of lambda calculus? *Information and control* **122** (1982) 52–87. 41
11. Mitchell, J.: Type Systems for Programming Languages. In: Leeuwen, J. (ed.): *Handbook of Theoretical Computer Science*, Elsevier Science Publishers B. V. (1990) 365–458. 42
12. Newman, M. H. A.: On theories with a combinatorial definition of 'equivalence'. *Annals of Mathematics* **43** (1942) 223–243. 39, 41
13. Pfenning, F.: A Proof of the Church-Rosser theorem and its representation in a Logical Framework, CMU-CS-92-186, (September 1992), forthcoming in *Journal of Authomated Reasoning*. 38
14. Scott, D.: Relating theories of the lambda calculus. In: Seldin, J. P. and J. R. Hindley: *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, Academic Press, London (1980) 403–450. 41, 43
15. Statman, R.: Logical relations and the simply typed lambda calculus. *Information and Control* **65** (1985) 85–97. 42
16. Takahashi, M.: Parallel Reductions in λ-Calculus. *Journal of Symbolic Computation* **7** (1989) 113–123. 38, 39
17. van Oostrom, V.: Confluence by decreasing diagrams. *Theoretical Computer Science* **126** (1994) 259–280. 38, 48
18. van Oostrom, V. and F. van Raamsdonk: Weak orthogonality implies confluence: the higher order case. In: Nerode, A. and Yu. Matiyasevich (eds.): Logical Foundations of Computer Science '94. *Lecture Notes in Computer Science* **813**, Springer-Verlag, Berlin (1994) 379–392. 48
19. Wadsworth, C. P.: The relation between computational and denotational properties for Scott's D_∞-models of the lambda calculus. *SIAM Journal of Computing* **5(3)** (1976) 488–521. 41

Incremental Inference of Partial Types

Mario Coppo[1] and Daniel Hirschkoff[2]

[1] Dipartimento di Informatica – Università di Torino
coppo@di.unito.it
[2] LIP – École Normale Supérieure de Lyon
Daniel.Hirschkoff@ens-lyon.fr

Abstract. We present a type inference procedure with *partial types* for a λ–calculus equipped with datatypes. Our procedure handles a type language containing greatest and least types (ω and $\perp$ respectively), recursive types, subtyping, and datatypes (yielding constants at the level of terms). The main feature of our algorithm is *incrementality*; this allows us to progressively analyse successive definitions, which is of interest in the setting of a system like the *CuCh machine* (developed at the University of Rome). The methods we describe have led to an implementation; we illustrate its use on a few examples.

1 Introduction

Modern functional programming languages are usually equipped with powerful, polymorphic type systems which preserve most of the great freedom and generality which are typical of the functional paradigm. However a completely type free programming style, allowing one to work with heterogeneous data structures or to define operators (like auto-application) which would not be typable in a standard way, is still appealing and supported by real programming languages ([AS85]).

Even in a type free environment, however, most real functional programs could naturally be typed in an ML-like type system, since there are usually few functions or parts of these which would not get a type although correctly designed for performing their intended task. The definition of these functions often requires a deep understanding of the functional paradigm and a good programming skill. A programmer writing them should then be aware of this and well confident in what he is doing.

One main motivation of this paper is that of studying an inference framework for recursive and polymorphic types, liable to be added to the top of a type-free functional programming language. The system is not designed to reject any program, but rather to give only partial type information for those programs which cannot be typed in the usual sense. Another fundamental feature of our type system is the treatment of subtyping. Subtyping will be motivated by the need to define a partial order structure over the set of types representing different levels of type information, but it will also allow us to properly handle the inclusion properties of user-defined datatypes.

A. Restivo, S. Ronchi Della Rocca, L. Roversi (Eds.): ICTCS 2001, LNCS 2202, pp. 50–73, 2001.

Our system was designed as a typing support for the *CuCh machine*, a system developed at the University of Rome in the team of Corrado Böhm (the design of a type inference procedure for this system actually originated our work). The CuCh machine is a programming language based on the untyped λ-calculus. There are two modes to define objects in CuCh, called `@lam` and `@env`; in `@lam`, the user defines λ-terms using abstraction, application, and some built-in constants including natural numbers, strings, lists and boolean tests. The Cuch system also supports the definition of inductive data types. The `@env` mode is used to define functions on the free algebras generated by user-defined data types by sets of equations, following [BB85]. The introduction of free algebras and of recursive definitions over these algebras is akin to the classic second-order encoding of datatypes; however, in CuCh, the solutions to (possibly recursive) definitions are not defined using a fixpoint operator, but rather following the Böhm-Piperno technique of [BPG94], using self-application. In this setting, more freedom is given in the construction of terms, and "traditional" type systems for functional languages *à la* ML can sometimes be too restrictive.

The basic technical tools for the definition of the system are the introduction of a "greatest" type ω and of a recursion operator over types. The use of type ω has been inspired by the "partial" type system introduced in [Tha94] (following [Gom90]), to describe some terms that are considered as ill-typed in a classical setting. Examples of such terms are auto-applications (e.g. $\lambda x.\,(x\,x)$), or heterogeneous lists (e.g. [true; $\lambda f\,x.\,(f\,x)$]). Using a notion of subtyping among partial types one is able for example to infer judgments like $\lambda x.\,(x\,x)\colon(\omega{\to}\alpha){\to}\alpha$ (where α is a type variable). The type associated to the occurrence of x in argument position can be coerced from $\omega{\to}\alpha$ to ω in order to permit the auto-application, yielding final type α. In [Tha94], however, there are still terms that cannot be typed (like $\lambda x.x(3\,3)$) although their behaviour could be represented by some partial type (like, in the former example, $(\omega{\to}t){\to}t$). To handle these cases (following e.g [BCDC83]), we introduce a rule (ω) postulating that any term has type ω.

Partial types can carry useful information. If we prove, for instance, that a (closed) term M has type $\omega{\to}\omega$ we know that M represents a "function" and not, for instance, an integer. Moreover this also guarantees that M can be head-reduced to a term of the shape $\lambda x.M'$ without going through meaningless applications like those determined for instance by a functional application in which the value in function position is an integer.

Other interesting typings can be derived assuming the existence of recursive types. For instance assuming to have a type c such that $c = c{\to}\alpha$ we can assign to $\lambda x.\,(x\,x)$ type $c{\to}\alpha = (c{\to}\alpha){\to}\alpha$, which turns out to be smaller (and then more informative) than $(\omega{\to}\alpha){\to}\alpha$. The introduction of recursive types amounts to extend the set of types allowed in the system to all regular infinite tree expressions. The type system presented in this paper works indeed on this extended set of types.

The question of partial type inference, as addressed in [Tha94], is shown to be decidable in [WO92], and [KPS94] provides an efficient algorithm to solve the problem. Our study differs from these works by three main aspects. First of all the language we focus on is equipped with user-defined datatypes (as well as with a least type, written $\perp$, that has to be introduced mainly for technical reasons). The introduction of (parametrised) datatypes somehow increases the complexity in the structure of the typing information that has to be dealt with, as will be seen thorough this study.

The second main original aspect of our work is the stress that is put on *incrementality* in defining the type inference method. Indeed, the traditional approach to type inference in presence of subtyping consists in exploring the structure of the term to be typed, and, while doing so, in collecting the corresponding subtyping constraints. Once all these constraints are put together, one can attack the problem of constraints satisfiability using several different approaches ([KPS94], for example, uses an automata-based method).

In this paper, we try on the contrary to preserve the readability of the type information along the exploration of the term. Our approach, inspired by [WO92], consists in representing internally the typing information about a given term on a table, which represents a kind of principal typing of the term itself (see [Jim96]). In doing this, however (and this is were our study differs from [WO92]), we are interested in inferring the consequences of the type constraints as soon as they are generated, and in resolving immediately the possible resulting inconsistencies. This kind of inconsistencies can be eliminated at the typing level, only by the use of the (ω) rule. To take this into account in the inference process (and this is the third new aspect of our approach), we introduce a notion of *guarded* constraint, that allows us to define an incremental and rather flexible type inference procedure. Due to the possibility of incomparable uses of the (ω) rule, however, the number of principal typings (i.e. tables) of a term is in general finite but not unique.

The paper is organized as follows. In Section 2, we introduce our system, defined by the terms of a core subset of CuCh, the (possibly recursive) definitions, the language of types and the two judgments corresponding to the typing and subtyping relations. Section 3 is devoted to the technical definitions we need for our type inference procedure, i.e. tables (to represent the type constraints), properties of tables, and various functions over tables. We define our type inference method in Section 4, as well as an heuristic to recover consistency where an inconsistent table is generated during the type inference process. In Section 5 we introduce the inductive datatypes and show how they can be integrated in the system. We finally conclude. In the appendix, we present the implementation of our system and illustrate its behavior through an example.

2 The System

We introduce a restricted language to develop the basic theory. In this core language we assume to have int, real and bool as basic types, provided with the usual arithmetic and booolean constants. Indeed in the complete language also int and bool are defined as inductive data types. The extension to the full language will be presented in Section 5.

Objects of the Language

Terms The terms we use are defined by the following syntax:

$$M \quad \lambda x.M \mid x \mid M\,N \mid c\,,$$

where constants (denoted by c) are represented by the basic integer, real and boolean functions and terms. In CuCh, recursive functions are not introduced with a fixpoint–like construct, but are instead given by recursive equations (introduced below).

Definitions We consider simple CuCh definitions with the following syntax:

$$I := M$$

where I is an identifier and M an expression, possibly containing occurrences of I, hence we deal in general with recursive equations. We write D to range over a sequence of definitions.

Programs A CuCh program is a list of definitions followed by an expression:

$$P \;:=\; D \quad M\,.$$

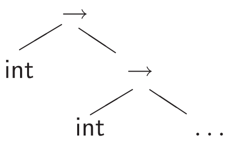
where D is a sequence of definitions and M an expression. An example of Cuch program is the following

```
M := \x y.((y (x 3)) (x x))
N := (M \z.z)
(N (\u v.u))
```

Types and Type Schemes Types are built from a set $\mathsf{B} = \{\mathsf{int}, \mathsf{real}, \mathsf{bool}, \bot, \omega\}$ of *basic* types, where $\bot$ represents the "least type" (which we assume to be included in every other type) and ω the whole domain of values (which then includes all types). We will introduce two notions of types, the *ground types* and the *type schemes*. In order to make our type discipline more liberal we also handle *recursive types*, introduced through the μ operator; recursive types represent infinite (but regular) types. The syntax of type schemes is the following:

$$\mathsf{T} := \mathsf{V} \mid \mathsf{int} \mid \mathsf{real} \mid \mathsf{bool} \mid \omega \mid \bot \mid \mathsf{T} \rightarrow \mathsf{T} \mid \mu t.\mathsf{T}\,,$$

where V is a set of type variables. As usual the operator μ acts as a variable binder. Informally, a type of the shape $\mu t.A$ is intended to represent the infinite regular tree obtained by infinitely unfolding A along t. For example a type $T = \mu t.\mathsf{int} \rightarrow t$ represents the following infinite type:



Ground types are then defined as the subset of closed type schemes, i.e. those containing no free variables. Let G denote the set of ground types.

In the style of the ML type system ([Mil78]) a type scheme intuitively represents all its possible instances with ground types. Note that every ground type is trivially a type scheme.

Subtyping Relation

We define a partial order relation $\leq$ between types (representing inclusion of the corresponding sets of values). The basic inclusion is $\mathsf{int} \leq \mathsf{real}$ (which represents a prototype structural inclusion between datatypes). Types $\bot$ and ω represent respectively the least and greatest elements with respect of this relation, so we have $\bot \leq T \leq \omega$ for all types T.

The inclusion axioms induce a natural partial pre-order relation in T, corresponding to the semantics given in [CC91]. This inclusion can be completely formalized (see e.g. [AC93, AK95, BH98]) and is decidable. We give here only the basic subtyping rules.

Let Σ denote a set of *subtyping assumptions* of the form $t \leq u$, where t and u are type variables. The judgment for inclusion between type schemes is written

$$\Sigma \vdash A \leq B\,.$$

Figure 1 gives the rules defining these judgments, where $A \approx B$ intuitively means that A and B represent the same infinite tree (i.e. have the same infinite unfolding). A complete axiomatization of $\approx$ is given in [AC93]. Note the usual monotonicity-antimonotonicity of →.

We write $A \leq B$ for $\vdash A \leq B$, meaning that $A \leq B$ can be derived from the empty set of assumptions.

$$(S_\bot)\ \Sigma \vdash \bot \leq T \qquad (S_\omega)\ \vdash T \leq \omega \qquad (S_c)\ \Sigma \vdash \mathsf{int} \leq \mathsf{real}$$

$$(S_{id})\ \Sigma \vdash t \leq t \qquad (S_{trans})\ \frac{\Sigma \vdash A \leq B \quad \Sigma \vdash B \leq C}{\Sigma \vdash A \leq C} \qquad (S_{hyp})\ \Sigma,\, t \leq u \vdash t \leq u$$

$$(S_{AC})\ \frac{A \approx B}{\Sigma \vdash A \leq B} \qquad (S_\rightarrow)\ \frac{\Sigma \vdash A_2 \leq A_1 \quad \Sigma \vdash B_1 \leq B_2}{\Sigma \vdash A_1 {\rightarrow} B_1 \leq A_2 {\rightarrow} B_2} \qquad (S_\mu)\ \frac{\Sigma, t \leq u \vdash A \leq B}{\Sigma \vdash \mu t.A \leq \mu u.B}$$

Fig. 1. Subtyping relation

Remark 1. (i) The intuitive type semantics in our approach relies on the notion of types as topologically closed subsets (ideals) of the domain of interpretation of the language [CC90]. This model also supports the notion of recursive type and recursive type equation. In this case the least (undefined) element of the domain belongs to every type. The type ω is then interpreted as the whole domain while $\perp$ is interpreted as the singleton containing only the least element of the domain. This provides a justification of the consistency and semantic correctness of the subtyping assumptions.
(ii) In our core language structural type inclusion is only represented, at the ground level, by the inclusion $\mathsf{int} \leq \mathsf{real}$. More interesting structural subtyping will be introduced by Datatype definitions (see Section 5). That framework shall provide a richer set of schemes and subtyping rules.

Typing Rules

We consider two typing judgments, one for terms and one for definitions. The typing judgement for terms is of the form

$$\varDelta, \varGamma \vdash M : T\,,$$

where M is a term, T is a type, $\varGamma$ is a set of typing assumptions for the free variables of M and $\varDelta$ is the type environment determined by a CuCh definition, associating a type to each defined name. The domains of $\varDelta$ and $\varGamma$ are always disjoint. As usual, $\varGamma$ and $\varDelta$ are seen as sets, modulo permutations. Accordingly, we use $\emptyset$ in the formal system to explicitly denote an empty sequence or an empty environment. The notation $\varGamma.\ x\!:\!A$ denotes a set of assumptions containing $x : A$ (which is assumed not to appear in $\varGamma$). For each constant c we assume a type $\tau(c)$ which captures its functional properties. For instance $\tau(3) = \mathsf{int}$ and $\tau(\mathit{succ}) = \mathsf{int}{\rightarrow}\mathsf{int}$, where *succ* is the successor function on integers. The typing judgments for definitions are of the form

$$\vdash_{def} D \;\Rightarrow\; \varDelta\,,$$

where D is a sequence of CuCh definitions. The meaning is that $\varDelta$ is the type environment determined by the definitions in D. The rules defining typing judgments are given on Figure 2; in rule (env), $\mathit{new}(A)$ is a function that returns a fresh copy of A introducing new type variables.

Notation: We indicate with $\vdash^-$, $\vdash^-_{def}$ derivability in the systems obtained from those of Fig. 2 by eliminating rule (ω).

It is easy to see that type assignment is closed by substitution.

Lemma 1. *Suppose $\varDelta, \varGamma \vdash M : T$. Then for any substitution σ,*

$$\sigma(\varDelta), \sigma(\varGamma) \vdash M : \sigma(T)\,.$$

Typing judgements containing free type variables can thus be seen as typing schemes, representing all their possible instances.

$$(\mathit{var})\quad \varDelta,\varGamma.x\!:\!A \vdash x\!:\!A \qquad\qquad (\mathit{env})\quad \varDelta.I\!:\!A,\varGamma \vdash I\!:\!\mathit{new}(A)$$

$$(\mathit{const})\quad \varDelta,\varGamma \vdash c\!:\!\tau(c) \qquad\qquad (\omega)\quad \varDelta,\varGamma \vdash M\!:\!\omega$$

$$(\rightarrow_I)\ \frac{\varDelta,\varGamma.x\!:\!A \vdash M\!:\!B}{\varDelta,\varGamma \vdash \lambda x.\,M\!:\!A\!\rightarrow\! B} \qquad (\rightarrow_E)\ \frac{\varDelta,\varGamma \vdash M\!:\!A\!\rightarrow\! B \quad \varDelta,\varGamma \vdash N\!:\!C \quad C \leq A}{\varDelta,\varGamma \vdash (M\,N)\!:\!B}$$

$$(\mathit{Def}_{\emptyset})\quad \vdash_{\mathit{def}} \emptyset \;\Rightarrow\; \emptyset \qquad (\mathit{Def}_{\mathit{more}})\ \frac{\vdash_{\mathit{def}} D \;\Rightarrow\; \varDelta \quad \varDelta,\{I\!:\!A\} \vdash M\!:\!B \quad B \leq A}{\vdash_{\mathit{def}}\ D.\ I := M \;\Rightarrow\; \varDelta.\,I\!:\!A}$$

Fig. 2. Typing expressions and definitions

Some Basic Properties

A notion of reduction for expressions of the CuCh machine can be defined by taking into account computation rules introduced by the definitions. Let indeed D be a list of definitions. For each definition

$$I \;:=\; M$$

in D, add a reduction rule

$$I \rightarrow^{D} M\,.$$

Let $\rightarrow^{D\beta}$ be the notion of reduction we obtain by adding this notion of reduction to the usual β. It is routine to prove the subject reduction theorem.

Theorem 1. *Let D be a list of CuCh definitions and let M be an expression such that*

$$\vdash_{\mathit{def}} D \;\Rightarrow\; \varDelta \quad \textit{and} \quad \varDelta,\varGamma \vdash M : T.$$

Then whenever $M\rightarrow^{D\beta}M'$ we have $\varDelta,\varGamma \vdash M' : T$.

Our system has no normalization property, owing to the presence of recursive definitions and of rule (ω). We can however prove a weaker result which is anyway interesting from a programming point of view.

The *top level operator* of a term M is the operator associated to the root of its abstract syntax tree. This can be, in our language, application, abstraction or a constant. In a language with datatypes it could also be a datatype constructor. A *head reduction* is a reduction in which, at every step, only the leftmost outermost redex can be reduced, provided it does not occur inside the scope of some term operator different from application. We say that a term is in *weak head normal form* if no head reduction step can be applied on it. We can prove the following property.

Theorem 2. *Let D be a list of CuCh definitions and let M be an expression such that*

$$\vdash_{def} D \Rightarrow \Delta, \qquad \Delta, \Gamma \vdash M : T \quad \textit{and} \quad T \neq \omega.$$

Then either the head reduction starting from M is infinite or M head reduces to a term M' in weak head normal form whose top level term constructor corresponds (in an obvious sense) to the top level type constructor of T.

By the above theorem, for instance, if $T = \omega \rightarrow \omega$ then M reduces to a term of the shape $\lambda x.M'$ or to a constant function like *succ*. Theorem 2 assures that in the head reduction of a term having a type different from ω, no meaningless applications (like, for instance, (3 3)) can be encountered.

This is not true, in general, for reductions strategies which can reduce non head redexes, like call-by-value. We can prove a stronger result for the system $\vdash^-$ without rule ω. We say that a term is *well formed* if it does not contain meaningless applications in the sense explained above.

Theorem 3. *Let D be a list of CuCh definitions and let M be an expression such that*

$$\vdash_{def} D \Rightarrow \Delta \quad \textit{and} \quad \Delta, \Gamma \vdash^- M : T\,.$$

Then M is well formed.

Form Theorems 1 and 3 we have immediately that terms that can be typed in $\vdash^-$ never produce bad applications independently of the reduction strategy being applied.

3 Systems of Type Constraints

3.1 Type Constraints and Tables

Our inference procedure is based on the representation of relations between types by sets of constraints. In this section we define the procedures to handle these.

A substitution is defined here as a finite mapping σ between type variables and types in T, that is naturally extended to all types. A single substitution is denoted $[t := A]$: it replaces t by A and behaves like the identity on all other variables. Similarly, $[t_1 := A_1, \ldots, t_n := A_n]$ (where t_i does not occur in A_j for all $1 \leq i, j \leq n$) denotes the composition of n single substitutions. If all the type expressions A_i are single variables, we say that $[t_1 := A_1, \ldots, t_n := A_n]$ is *trivial.*

A *ground* substitution γ is instead a mapping from type variables to ground types.

Definition 1 (Constraints). *A* type constraint *is an expression of the form $t \leq u_1 \rightarrow u_2$ or $u_1 \rightarrow u_2 \leq t$ where t, u_1, u_2 are type variables.*

To handle the inference rule (ω) of Fig. 2 we need the notion of guard an some operators on it.

Definition 2. *(i) A* guard *is a list of type variables. Let w range over guards.*

(ii) If w_1, w_2 are guards then $w_1 \rhd w_2$ is the guard obtained by concatenating w_1 and w_2 and by eliminating from w_2 the variables which already occur in w_1.

(iii) A guarded constraint *(g.c. for short) is an expression of the form $w :_G (A \leq B)$ where w is a guard and $A \leq B$ is a type constraint.*

If $S = \{w_i :_G (A_i \leq B_i) \mid 1 \leq i \leq n\}$ is a set of g.c. and w is a guard then $w \rhd S$ denotes the set $\{w \rhd w_i :_G (A_i \leq B_i) \mid 1 \leq i \leq n\}$. A guard hides the constraint associated to it whenever at least one of the variables occurring in it is set to ω.

A *solution* of a set S of g.c. is a ground substitution γ such that for all $w :_G (A \leq B) \in S$ in which $\gamma(t) \neq \omega$ for all variables t in w we have $\gamma(A) \leq \gamma(B)$. A *strong* solution of S is a ground substitution γ such that for all $w :_G (A \leq B) \in S$ we have $\gamma(A) \leq \gamma(B)$ (ignoring guards).

The inference algorithm keeps the information about the types involved in a deduction using the notion of *table*, which has been inspired by [WO92]. A table is simply a structured set of type constraints, which are represented in a slightly different way via the notion of guarded elementary expression.

Definition 3. *(i) A* guarded elementary expression *(g.e. for short) is an expression of the shape $w :_G (v_1 \to v_2)$ where v_1, v_2 are variables.*

(ii) A table *Θ is a set of triples $\langle t, L, U \rangle$ (called the* entries *of the table), where t is a variable and L and U are sets of g.e. which are said, respectively, the* lower *and* upper *sets of t in Θ. If $\langle t, L, U \rangle \in \Theta$ we denote L as $L_\Theta(t)$, or simply $L(t)$ (when Θ is understood) and U as $U_\Theta(t)$, or simply $U(t)$. Moreover define $\mathtt{dom}(\Theta) = \{t \mid \langle t, L, U \rangle \in \Theta\}$.*

A table is just a structured way of representing a set of elementary g.c.s. In fact each $w :_G (A) \in L(t)$ represents a g.c. $w :_G (A \leq t)$, and each $w :_G (A) \in U(t)$ also represents a g.c. $w :_G (t \leq A)$. A solution of a table is a solution of the corresponding set of g.e.s.

A *simplified* table (s-table for short) Ξ is a structure which has the same shape as a table but without guards. So the elements of the upper and lower sets are type expressions (containing only one type constructor) instead of g.e.s. The *kernel* of a table Θ, written $\mathtt{kernel}(\Theta)$ is an s-table Ξ obtained from Θ by erasing all guards.

Definition 4. *A table Θ is* closed *if for all $t \in \mathtt{dom}(\Theta)$ such that both $L(t)$ and $U(t)$ are nonempty and for all $w_1 :_G (u_1 \to u_2) \in L(t)$ and $w_2 :_G (v_1 \to v_2) \in U(t)$ we have:*

- $w_1 \rhd w_2 \rhd L(u_2) \subseteq L(v_2)$.
- $w_1 \rhd w_2 \rhd U(v_2) \subseteq U(u_2)$
- $w_1 \rhd w_2 \rhd L(v_1) \subseteq L(u_1)$
- $w_1 \rhd w_2 \rhd U(u_1) \subseteq U(v_1)$

A table obtained from a set of elementary g.c.s is in general not closed. It is easy to define an algorithm `closure` that takes a table Θ in input and returns its closure $\texttt{closure}(\Theta)$ by adding elements to the sets $L(t)$, $U(t)$ according to the previous definition. Since all the new constraints added to a table by `closure` are simply consequences of the definition of the $\leq$ relation we have immediately the following lemma:

Lemma 2. *A table Θ and its closure* $\texttt{closure}(\Theta)$ *have the same solutions.*

There is a simple condition to decide whether a closed table has a solution.

In the following definition we informally assume the existence of a partial order between type constructors. This coincides with the subtyping relation for constant types. The type constructor → is considered incomparable with all constant types, while $\perp$ (ω) is smaller (greater) than all type constructors.

Definition 5 (Consistent table). *(i) A closed table Θ is* consistent *with respect to a variable $t \in \textit{dom}(\Theta)$ if there is a type constructor c which is a sup for all type constructors occurring in $L_{\Theta}(t)$ and an inf for all type constructors occurring in $U_{\Theta}(t)$*

(ii) A closed table Θ is consistent *(tout court) if it is consistent with respect to every $t \in \textit{dom}(\Theta)$.*

Note that a table Θ is always consistent with respect to a variable t if $U_{\Theta}(t)$ (resp. $L_{\Theta}(t)$) is empty. In such case we can can take $t := \omega$ (resp. $t := \perp$).

3.2 Solving Tables

In this subsection we show that every consistent table admits a strong solution, and we give an algorithm to find it. To obtain it we need to define some more transformations on tables. To keep notations light we consider here only simplified tables. The extension of the transformations to guarded tables is routine.

Let an *elementary substitution* (e.s. for short) $\mathtt{e}$ be an expression of the form $[t := u_1 \rightarrow u_2]$ where u_1, u_2 are variables. A *substitution path* (s.p.) is a list $\langle \mathtt{e}_n, \ldots, \mathtt{e}_1 \rangle$ of e.s. such that no two e.s. in it have the same l.h.s.. A s.p. $\mathtt{s} = \langle \mathtt{e}_n, \ldots, \mathtt{e}_1 \rangle$ naturally determines a substitution $\mathtt{e}_n \circ \ldots \circ \mathtt{e}_1$ (where $\circ$ denotes function composition) which we identify with $\mathtt{s}$ itself. Let $\mathtt{dom}(\mathtt{s})$ denote the set of variables occurring as l.h.s. of the e.s. in $\mathtt{s}$.

We now define a function `solve`, that takes a simplified closed table Ξ, and returns a pair $\langle \Xi', \mathtt{s} \rangle$ where Ξ' is a simplified closed table and $\mathtt{s}$ is a substitution path. We define function `solve` by giving an algorithm to compute it.

Definition 6. *Let Ξ be a closed table. The function* `solve` *is defined by the following steps. The basic operation is to build a sequence of s-tables Ξ_i and substitution paths $\mathtt{s}_i$ ($i \geq 0$). During the construction, we "mark" some entries of the table (to remember that the substitution for the corresponding variables has already been generated).*

1. *Set $i = 0$. Let $\Xi_0 = \Xi$, $\mathtt{s}_0$ be the empty list. All entries of Ξ_0 are unmarked.*

2. *Take any unmarked entry $\langle t, L(t), U(t)\rangle$ of Ξ_i such that both $L(t)$ and $U(t)$ are not empty and mark it. We then distinguish two cases;*
   (a) *if there is a marked entry $\langle u, L(u), U(u)\rangle$ in Ξ_i such that both $L(u) = L(t)$ and $U(u) = U(t)$ then define*
      - *Ξ_{i+1} as the table obtained by removing the entry for t in Ξ_i.*
      - *$\mathtt{s}_{i+1}$ as the s.p. obtained by replacing t with u in $\mathtt{s}_i$.*

   (b) *Otherwise let t_1, t_2 be two new fresh variables. Then:*
      – *Define $\mathtt{s}_{i+1}$ as the s.p. obtained by adding $[t := t_1 \to t_2]$ at the beginning of $\mathtt{s}_i$.*
      – *Define Ξ_{i+1} by adding to Ξ_i two new entries for t_1 and t_2, and set:*

$$\begin{array}{l} L(t_1) = \bigcup\{U_{\Xi_i}(u_1) \mid u_1 \to u_2 \in L(t)\} \\ U(t_1) = \bigcup\{L_{\Xi_i}(u_1) \mid u_1 \to u_2 \in U(t)\} \\ L(t_2) = \bigcup\{L_{\Xi_i}(u_2) \mid u_1 \to u_2 \in L(t)\} \\ U(t_2) = \bigcup\{U_{\Xi_i}(u_2) \mid u_1 \to u_2 \in U(t)\} \end{array}$$

3. *Repeat step 2. until there are no more unmarked entries with both a lower and an upper set nonempty. Let n be the last value of i. Return $\langle \Xi_n, \mathtt{s}_n\rangle$.*

Note that each new upper and lower set built in step 2(b) contains only g.e.s already occurring in Ξ. Then there is only a finite number of possible upper and lower sets that can occur in the tables Ξ_i. Owing to step 2(a) we get immediately the following termination property. .

Lemma 3. *The construction in Def. 6 is always terminating.*

It also easy to verify by induction on i that, in the construction of Definition 6, each Ξ_i is closed and consistent. More generally we have the following property.

Lemma 4. *Let Ξ be a consistent closed table and let $\mathtt{solve}(\Xi) = \langle \Xi', \mathtt{s}\rangle$. Then Ξ' is consistent and closed, and each of its solutions is also a solution of Ξ.*

We now get to the main result of this section.

Proposition 1. *Any consistent closed s-table Ξ has a solution.*

Proof hint. Let $\mathtt{solve}(\Xi) = \langle \Xi', \mathtt{s}\rangle$. By Lemma 4 it is enough to find a solution for Ξ'. Let S denote the subset of $\mathtt{dom}(\Xi')$ containing the variables having an empty upper or lower set. First define a ground substitution γ_0, having domain S, by

$$\begin{cases} \gamma_0(t) = \omega & \text{if } U(t) = \emptyset \\ \gamma_0(t) = \bot & \text{if } L(t) = \emptyset \end{cases}$$

Now let $\mathtt{s} = \langle \mathtt{e}_n, \ldots, \mathtt{e}_1\rangle$ for some $n \geq 0$. Starting from $\sigma_0 = \gamma_0$ define a sequence of ground substitutions σ_i for $0 \leq i \leq n$ in the following way.
Let $\mathtt{e}_{1+i} \equiv [t_{i+1} := u \to v]$ and let $A = \sigma_i(u \to v)$. Then

- If t_{i+1} does not occur in A then take $\sigma_{i+1} = [t_{i+1} := A] \circ \sigma_i$
- otherwise (t_{i+1} occurs in A) take $\sigma_{i+1} = [t_{i+1} := \mu t_{i+1}.A] \circ \sigma_i$

It is easy to see that σ_n is indeed a ground substitution. Moreover σ_n gives a solution of Ξ' and then of Ξ. ♢

Corollary 1. *A consistent table has a strong solution.*

3.3 Finding a Solution Scheme

In Proposition 1, it is proved that a consistent table admits *at least* one ground solution. Actually we are rather interested, in the type inference procedure, to give a readable characterization of a large set of solutions of a table, possibly all of them. Let us define a *solution scheme* for an s-table Ξ as a substitution σ such that $\gamma \circ \sigma$ is a solution of Θ for all ground substitutions γ.

In this subsection we define a simple algorithm that builds a solution scheme for a given table. The scheme that we obtain may fail to capture some solutions; these could be represented only at the cost of introducing more complex subtyping expressions.

In building a solution scheme for an s-table we first define a function `collapse` that "flattens" a table Θ into a simpler one Ξ', preserving most of the solutions (but not all of them). From the flattened table Ξ' we can get in a rather standard way a solution scheme for Θ. In these steps, ω-reductions are not considered, so we can describe our construction for simple tables.

We first give some definitions. An entry for t of a consistent simple table Ξ is *simple* if either both $L(t)$ and $U(t)$ contain only basic types or $L(t) \cup U(t)$ contains only one expression (of the form $u{\rightarrow}v$). A entry is *complex* if it is non simple and $L(t) \cup U(t)$ contain at least two type expressions having $\rightarrow$ as type constructor and no basic types. In all other cases we say that the entry is *easy.* We have to be more precise about complex entries. An open entry is *L-complex* if $U(t) = \emptyset$, *U-complex* if $L(t) = \emptyset$ and *L-U-complex* if both $U(t) \neq \emptyset,\ L(t) \neq \emptyset$.

Definition 7. *Let Ξ be a simplified table. Then* `collapse`(Ξ) *is a pair $\langle \Xi', s\rangle$ where Ξ' is an s-table and ρ is a trivial substitution. The function* `collapse` *is defined by the steps given below. Also in this case the basic operation is to build a succession of s-tables Ξ_i and trivial substitutions ρ_i $(i \geq 0)$. During the construction, we assume that we are able to mark (and unmark) some entries of the considered tables. Take Ξ_0 as Ξ in which all easy entries have been marked and ρ_0 as the empty substitution. Repeat the following steps until there are no more open non marked entries in Ξ_i.*

1. *Take any non marked complex entry $\langle t, L_t, U_t\rangle$ of Ξ_i and let*
   $$L(t) \cup U(t) = \{u_k{\rightarrow}v_k \mid 1 \leq k \leq p\}$$
   for some $p > 1$.
2. *Take two fresh variables u, v and define the substitution* $\rho^* = [u_1 := u, \ldots, u_p := u, v_1 := v, \ldots, v_p := v]$*. If some u_i is t itself then take t instead of u, and similarly for v.*
3. *Add to Ξ_i two entries $\langle u, L(u), U(u)\rangle$, $\langle v, L(v), U(v)\rangle$ and set*
   - $L(u) = \bigcup\{L_{\Xi_i}(u_k) \mid 1 \leq k \leq p\}$
   - $U(u) = \bigcup\{U_{\Xi_i}(u_k) \mid 1 \leq k \leq p\}$
   - $L(v) = \bigcup\{L_{\Xi_i}(v_k) \mid 1 \leq k \leq p\}$
   - $U(v) = \bigcup\{U_{\Xi_i}(v_k) \mid 1 \leq k \leq p\}$
4. *Remove from Ξ_i all entries for the variables u_i, v_i and compute* $\Xi' =$ `closure`$(\rho^*(\Xi_i))$*. Keep in Ξ' the marking of Ξ.*

5. *If Ξ' is consistent then mark the entry for t and set $\Xi_{i+1} = \Xi'$ and $\rho_{i+1} = \rho^* \circ \rho_i$. Otherwise take Ξ_{i+1} as Ξ_i and mark the entry for t*[1]

Let n the final value of i. Then return $\langle \Xi_n, \rho_n \rangle$

Basically, the key step in procedure `collapse` consists in reading the various expressions that occur in the lower and upper sets of a given entry for t "transversally", and map all the variables read this way to a single variable, that can in particular be t itself.

The last step in the construction of the solution scheme for a table is the definition of a substitution. This construction is similar to that given in the proof of Proposition 1.

Definition 8. *Let Ξ be a consistent table and let $\langle \Xi', \rho \rangle =$ `collapse`(Ξ). Let $t_1, \ldots t_n$ be the variables corresponding to the entries of Ξ'. The* canonical *substitution σ_Ξ associated to Ξ is defined by constructing, iteratively on i, a sequence of substitutions σ_i $(0 \leq i \leq n)$ in the following way.*
Let $\sigma_0 = \rho$.
Given σ_i, let $\langle t_{i+1}, L_{i+1}, U_{i+1} \rangle$ $(0 \leq i < n)$ be the entry for t_{i+1} in Ξ'. Distinguish the following cases:

1. *If $U_{i+1} = \emptyset$ and L_{i+1} contains basic types admitting a l.u.b. κ, then define $\sigma_{i+1} = [t_{i+1} := \xi] \circ \sigma_i$, where $\xi = \omega$ if L_{i+1} also contains arrow types, $\xi = \kappa$ otherwise.*
2. *Proceed similarly when $L_{i+1} = \emptyset$, replacing l.u.b. with g.l.b. and ω with $\perp$.*
3. *If $L_{i+1} \cup U_{i+1} = \{u_{i+1} \rightarrow v_{i+1}\}$, let $A = \sigma_i(u_{i+1} \rightarrow v_{i+1})$. Then:*
   - *if t_{i+1} does not occur in A then set $\sigma_{i+1} = [t_{i+1} := A] \circ \sigma_i$;*
   - *otherwise (t_{i+1} occurs in A) set $\sigma_{i+1} = [t_{i+1} := \mu t_{i+1}.A] \circ \sigma_i$.*
4. *Otherwise (because of step 5. in Def. 7) exactly one of L_{i+1}, U_{i+1} is empty, and there are at least two different arrow types in $L_{i+1} \cup U_{i+1}$. Then set $\sigma_{i+1} = [t_{i+1} := \xi] \circ \sigma_i$ where ξ is $\perp$ if $L_{i+1} = \emptyset$ and ω if $U_{i+1} = \emptyset$.*

Note that case 4. occurs when either L_{i+1} or U_{i+1} contains two arrow g.e. which cannot be unified without making the table inconsistent (see case 5. of Def. 7). In general, to get more informative types, we avoid the use of $\perp$ and ω whenever possible.

Lemma 5. *Let Ξ be a consistent table. Then σ_Ξ is a solution scheme for Ξ.*

Remark 2. In some cases the solution scheme we get is the most general one. For instance if a simple table Ξ can be solved without use of subtyping and recursive types (i.e. using only simple types without subtyping) σ_Ξ characterizes all solutions of Ξ.

[1] It can be shown that this is liable to happen only if the entry for t in Ξ_i is L-complex or U-complex; indeed, the closure condition insures that if the entry for t in Ξ_i is L-U-complex then Ξ' is consistent.

3.4 ω Reductions

In order to take into account the non-homogeneous nature of the type assignment system (the ω type has a somewhat particular behaviour), we equip the system with a ω *reduction rule* for tables, which corresponds to assigning type ω to some of its entries and simplifying the table accordingly. By choosing different variables we can reduce the table in different ways, so the ω reduction relation is not functional.

The reduction of tables is denoted by $\Theta \Rightarrow_R \Theta'$, where Θ, Θ' are tables. We first define a function that represents an elementary reduction step.

Definition 9. *If Θ is a table and t a variable occurring in some guards of Θ such that $U_{\Theta}(t)$ is empty (or contains only ω), then $\mathtt{red}_{\omega}(\Theta, t)$ is the table obtained by applying to Θ the following steps.*

1. *Eliminate from Θ all the g.e.s that have an occurrence of t in their guard.*
2. *Set both the upper and lower set of t to $\{\omega\}$.*
3. *Apply the function `closure` to the resulting table.*

The application of `closure` in step 3. is there to propagate ω in the table. The ω reduction relation for tables is defined by the following rules:

$$(ax1) \;\; \Theta \Rightarrow_R \Theta \qquad\qquad (\omega - red) \;\; \frac{\Theta \Rightarrow_R \Theta' \quad t \in \mathtt{dom}(\Theta')}{\Theta \Rightarrow_R \mathtt{red}_{\omega}(\Theta', t)}$$

Note in particular that the reduction step can be applied with any variable in $\mathtt{dom}(\Theta')$. The following is easily proved:

Lemma 6. *Let $\Theta \Rightarrow_R \Theta'$. Then any strong solution of Θ' is a solution of Θ.*

4 Type Inference

4.1 Operators on Tables

We will need in the following a couple of operators to handle tables.

Definition 10. *(i) If Θ_1 and Θ_2 are two (closed) tables, then $\Theta_1 \uplus \Theta_2$ is the table defined by merging them and applying `closure`.*
(ii) If g is an elementary g.c. and Θ is a table, then $\mathtt{addtable}(\Theta, g)$ is the table obtained from Θ by adding the constraints in g and applying `closure` to the resulting table.

Definition 11. *If Θ is a table and w a guard, then $w \triangleright \Theta$ is the table obtained by replacing the guard w' of each g.e. occurring in the L and U sets of Θ by $w \triangleright w'$.*

Let Θ be a table and $\mathcal{V}$ a set of type variables. The function `simplify` extracts from a table Θ the subpart of it which is relevant for finding the solution relative to the variables in $\mathcal{V}$. In particular

$$\mathtt{simplify}(\Theta, \mathcal{V})$$

is the table Θ' obtained from the empty table through the following steps:

1. Put in Θ' all the entries for variables in $\mathcal{V}$.
2. Add to Θ' all the entries of variables which occur in the upper or lower sets of variables already in Θ'.
3. Repeat step 2. until no other entry can be added to Θ'.

It is easy to see that if Θ is a closed table, so is the case for $\mathtt{simplify}(\Theta, \mathcal{V})$ as well. Moreover, the basic property of $\mathtt{simplify}(\_, \_)$ is the following.

Lemma 7. *Let $\Theta' = \mathtt{simplify}(\Theta, \mathcal{V})$. Then any solution of Θ' can be extended to a solution of Θ.*

4.2 From Terms to Tables

The inference procedure we define in this section will yield, for each term M defined inside a set D of CuCh definitions, a characterization of all possible typings of M in D with respect to the system of Fig. 2. In our approach this characterization will be given by a set of consistent tables, whose solutions characterize in a complete way all possible typings of M.

Our type inference method is defined by a set of rules in Natural Semantics through a judgment of the form

$$\mathbb{\Delta} \vdash_{TI} M \Rightarrow \Gamma \mid t \mid \Theta$$

where M is a term, t is a type variable representing the type of M, Γ a typing context and Θ a *consistent* table. The context $\mathbb{\Delta}$ is a finite function mapping identifiers x (used in CuCh definitions) to pairs of the shape $\langle t, \Theta \rangle$ where t is a variable and Θ a table representing the constraints that characterize the type of x. The intuitive meaning of this judgment is that typing M starting from $\mathbb{\Delta}$ we get a context Γ' and a table Θ which characterizes the typings of M, whose type is associated to the variable t. The definition of judgment $\vdash_{TI}$ involves the application of the reduction relation $\Rightarrow_R$ in a nondeterministic way. This is essential to have a complete inference procedure. We will define later a heuristic to avoid nondeterminism and produce a more practical typing procedure. Indeed the cases in which the use of nondeterministic reduction is needed seem to occur rarely.

Type inference for definitions is based on a judgement of the shape

$$\vdash_{DI} D \ \Rightarrow \ \mathbb{\Delta} ,$$

where D is a sequence of CuCh definitions and $\mathbb{\Delta}$ characterizes the generated type environment. Informally, tables will be brought along in the computation and progressively updated as we get new information about the term, thus providing the incrementality of our approach.

In the inference rules Γ ranges over sets of statements of the shape $x : t$ where x is a term variable and t a type variable. We define an auxiliary function on contexts, called $\mathtt{merge}$, to merge the assignment contexts. In particular

$$\mathtt{merge}(\Gamma_1, \Gamma_2) = < \Gamma, \rho > ,$$

where ρ is the trivial substitution (only a variable renaming) that identifies all (and only) the type variables which are predicates of the same term variable in Γ_1 and Γ_2 and $\Gamma = \rho(\Gamma_1) \cup \rho(\Gamma_2)$. We shall use `merge` (which is associative) also with more than two arguments.

Canonical tables For each constant c of the language, we shall assume that we have a (closed and consistent) *canonical* table $\Theta^{can}(c)$, representing the type constraints that characterize the typings of c. The *root variable* of $\Theta^{can}(c)$ is the variable representing the type of c. For example $\Theta^{can}(n)$, where n is any integer is given by[2]:

$$t \;\big|\; t : \mathsf{int} \;\big|$$

whose root variable is t while $\Theta^{can}(succ)$ where $succ$ is the integer function is given by:

$$\begin{array}{r|c|l} t & u{\rightarrow}v & \\ u & & t : \mathsf{int} \\ v & t : \mathsf{int} & \end{array}$$

The description of canonical tables we give does not tell how such tables are built—although we hope that an intuition can be found in the examples above. For the moment, we rely on an assumption that such tables "have the right shape"; the treatment of datatypes described in Sec. 5 will provide precise definitions. The type inference procedure for terms and definitions is formalized on Figure 3 as a set of natural semantics rules.

Rule (T_{app}) is the only case in which we can reduce the size of the table by applying `simplify`. The type inference relation is non-deterministic owing to the possibility of ω-reducing Θ in rule (T_{app}). This corresponds to the possibility of applying a (ω) rule to different subterms of a given term. The different tables produced by a ω-reduction of Θ can have uncomparable sets of solutions. So if one sees the table produced by the type inference procedure as a kind of "principal" type we could say that a term has in general a finite set of principal types.

Let now $\vdash^{-}_{TI}$ and $\vdash^{-}_{DI}$ denote derivability in the system obtained from the one in Fig. 3 by eliminating in rule (T_{app}) the possibility of using (ω)-reduction to get a consistent table. Note that in this system type inference is deterministic and the table resulting from the analysis of a program is unique.

The following theorem states (in a somewhat simplified form to make it more readeable) the soundness and completeness of the inference procedure with respect to the rules of Fig. 2.

Theorem 4. *(i) Let D be a list of CuCh definitions and M a term and let*

$$\vdash_{DI} D \;\Rightarrow\; \mathbb{\Delta} \quad \textit{and} \quad \mathbb{\Delta} \vdash_{TI} M \Rightarrow \Gamma \,|\, t \,|\, \Theta$$

for some $\mathbb{\Delta}$, Γ and Θ. Then there is a type environment Δ such that for all ground substitutions γ solving Θ we have

[2] A table is represented by a column of entries, where an entry for a variable t is represented by $t \mid L(t) \mid U(t)$.

$$(T_{var1})\quad \boldsymbol{\Delta} \vdash_{TI} x \Rightarrow \{x\!:\!t\} \mid t \mid \emptyset \qquad \text{(if } x \notin \mathtt{dom}(\boldsymbol{\Delta})\text{)}\ \text{where } t \text{ is a fresh variable}$$

$$(T_{var2})\quad \boldsymbol{\Delta} \vdash_{TI} x \Rightarrow \emptyset \mid t \mid \Theta \qquad \text{(if } x\!:\!\langle t_0, \Theta_0\rangle \in \boldsymbol{\Delta}\text{)}\ \text{where } \langle t, \Theta\rangle = \mathit{new}(\boldsymbol{\Delta}(x))$$

$$(T_{const})\quad \boldsymbol{\Delta} \vdash_{TI} c \Rightarrow \emptyset \mid t \mid \mathit{new}(\Theta^{can}(c)) \qquad \text{where } t \text{ is the root variable of } \mathit{new}(\Theta^{can}(c))$$

$$(T_{\lambda})\ \frac{\boldsymbol{\Delta} \vdash_{TI} M \Rightarrow \Gamma, x\!:\!u \mid v \mid \Theta}{\boldsymbol{\Delta} \vdash_{TI} \lambda x.\, M \Rightarrow \Gamma \mid t \mid \mathtt{addtable}(t \triangleright \Theta,\ \{t\!:_G (u \rightarrow v \leq t)\})} \qquad \text{where } u \text{ is a fresh variable}$$

$$(T_{app})\ \frac{\boldsymbol{\Delta} \vdash_{TI} M \Rightarrow \Gamma_1 \mid u \mid \Theta_1 \quad \boldsymbol{\Delta} \vdash_{TI} N \Rightarrow \Gamma_2 \mid v \mid \Theta_2}{\boldsymbol{\Delta} \vdash_{TI} M\,N \Rightarrow \rho(\Gamma) \mid t \mid \mathtt{simplify}(\Theta', t)}$$

where t is a fresh variable, $< \Gamma, \rho >= \mathtt{merge}(\Gamma_1, \Gamma_2)$
$\Theta = \mathtt{addtable}(\rho(\Theta_1) \uplus \rho(\Theta_2), \{t\!:_G (\rho(u) \leq \rho(v) \rightarrow t)\})$,
$t \triangleright \Theta \Rightarrow_R \Theta'$, and Θ' is consistent

$$(D_{\emptyset})\quad \vdash_{DI} \emptyset \ \Rightarrow\ \emptyset \qquad (D_{more})\ \frac{\vdash_{DI} D \ \Rightarrow\ \boldsymbol{\Delta} \qquad \boldsymbol{\Delta}.I\!:\!\langle t, \emptyset\rangle \vdash_{TI} M \Rightarrow \emptyset \mid u \mid \Theta}{\vdash_{DI} D, I := M \ \Rightarrow\ \boldsymbol{\Delta}.\ I\!:\!\langle t, \mathtt{closure}([u := t]\Theta')\rangle}$$

Fig. 3. Type inference procedure

$$\vdash_{def} D \ \Rightarrow\ \Delta \quad \mathit{and} \quad \Delta, \gamma(\Gamma) \vdash M : \gamma(T)\,.$$

Conversely, for all ground Γ' and T' such that for some type environment Δ

$$\vdash_{def} D \ \Rightarrow\ \Delta \quad \mathit{and} \quad \Delta, \Gamma' \vdash M : T'\,,$$

there is a ground substitution γ' solving Θ and such that $\Gamma' = \gamma'(\Gamma)$ and $T' = \gamma'(t)$.
(ii) The same property holds by replacing $\vdash^-$ with $\vdash$, i.e. type inference without ω-reduction corresponds to deductions not using the (ω) rule.

The typechecking procedure defined in Fig. 3 keeps the whole table as an internal representation of the typing of a term. The table is indeed the only way of representing the "most general" typing. Taking the canonical substitution for a table defined in Sec. 3.3 we can give, however, a readeable representation of the typing of a term. As remarked in Sec. 3.3 this implies a possible loss of information, but only at the level of interface with the user.

Lemma 8. *Let $\vdash_{DI} D \ \Rightarrow\ \boldsymbol{\Delta}$ and $\boldsymbol{\Delta} \vdash_{TI} M \Rightarrow \Gamma \,|\, t \,|\, \Theta$. Then there is a typing context Δ such that $\vdash_{def} D \ \Rightarrow\ \Delta$ and $\Delta, \sigma_{\Theta}(\Gamma) \vdash M : \sigma_{\Theta}(T)$*

Remark 3. There are interesting cases in which the typing scheme determined by a canonical substitution is complete. For example, by Remark 2, this happens when a term has a type in the standard ML type system (without subtyping). In this case we get the very type scheme produced by the ML typechecker.

4.3 A Heuristic to Handle ω Reductions

The type checking relation defined in subsection 4.2 is not deterministic, due to the presence of ω-reductions, but we are interested in turning it into a deterministic process, in order to get a reasonably efficient implementation of type inference. Of course, we do this at the cost of losing the completeness of the inference procedure.

We present here the basic intuitions behind a heuristic to transform a (closed) table that is not consistent into a consistent one. We have then to apply the reduction relation $\Rightarrow_R$ to eliminate the constraints on variables with respect to which the table is not consistent. This actually means simulating an application of rule (ω) to the subterms for which we are not able to find a meaningful type.

Since we want to preserve as much information as possible, our strategy is to try to apply rule (ω) starting from the inner subterms. To do this, we exploit the notion of guard. We rely on the assuption that guards are kept topologically sorted w.r.t. the inclusion of the corresponding subterms when building the table, so that we can easily have access to an innermost guard in the sense of the subterm relation. The heuristic anihilates those entries in U and L sets that cause inconsistency by triggering their corresponding innermost guard. Moreover, when doing this, we choose if possible to perform ω-reduction on variables having an empty U set. Indeed, putting a type variable v to ω has the effect of "pushing" to ω every type possibly occurring in the U set of v, and we want to keep the effect of ω-reductions as local as possible in order to keep a meaningful typing information.

The precise design of our heuristic involves some choices at several steps, in particular when selecting the constraint we eliminate, and the guard we trigger (when several type variables may apply). We have been experimenting with our tool in order to understand these tuning issues, but we do not have enough insight to explicitely choose a deterministic way to perform ω-reductions. For this reason, we have decided to keep the explanations about our heuristics informal, and just sketch here the main ideas.

5 Adding User-Defined Datatypes

In this section, we show how our framework for type inference can be adapted to a richer language allowing the user to define his own datatypes.

Introducing datatypes The syntax we adopt for datatype definitions is as follows

$$\texttt{DType}\ D[X_1, \ldots, X_k]\ \texttt{is}\ c_1^D : arg[T_1^1; \ldots; T_{m_1}^1],\ \ldots,\ c_n^D : arg[T_1^n; \ldots; T_{m_n}^n]\,,$$

where the X_is are the parameters of the datatype and each T^i_j is either a parameter X_j or another datatype (possibly D itself) having the shape $T_{i,j}[X_1, \ldots, X_k]$.

The definition above reads *"D is a datatype that has n constructors and k parameters $X_1, \ldots, X_k$; each constructor c^D_i, for $1 \leq i \leq n$ has type*

$$T^i_1 \rightarrow \ldots \rightarrow T^i_{m_i} \rightarrow D[X_1, \ldots, X_k]$$

where the T_is are either parameters or datatypes". Note that nested arrow types are not allowed in the definition of constructors.

Example: In this framework, the declaration of the dataype *List* would be:

$$\texttt{DType}\ \ \mathit{List}[X]\ \ \texttt{is}\ \ \mathit{Nil} : \mathit{arg}[],\ \ \mathit{Cons} : \mathit{arg}[X;\, \mathit{List}[X]]$$

CuCh Definitions Having introduced datatypes, we can enrich the shape of definitions and take into account declarations of the following form:

$$\mathtt{f}\,(c^D_i\, x_1 \ldots x_{m_i})\ =\ e\,.$$

e is an expression possibly containing occurrences of the x_is and $\mathtt{f}$. Such definitions are used as an alternative to the case construct (a case–like definition can easily be translated into a set of recursive equations). Taking into account this kind of definition in the typing and type inference rules then imposes to enrich locally the typing context with hypotheses for the x_is. This extension is quite natural but would require some more work on the technical details of the system, which are left out of this presentation.

Structural subtyping In the extended framework of this section, we obtain a richer notion of structural subtyping on datatypes. This relation, written $D \sqsubseteq D'$, means that datatype D is "structurally smaller" than D'. Fig. 4 gives the corresponding rules. Let us make a few comments about the definition of $\sqsubseteq$. It

$$(D_{\sqsubseteq})\quad \frac{\begin{array}{l} D[X_1, \ldots, X_k]\ \texttt{is}\ (c_1 : \mathit{arg}[T^1_1; \ldots; T^1_{m_1}] \ldots c_n : \mathit{arg}[T^n_1; \ldots; T^n_{m_n}]) \\ D'[X_1, \ldots, X_k]\ \texttt{is}\ (c_1 : \mathit{arg}[T'^1_1; \ldots; T'^1_{m_1}] \ldots c_n : \mathit{arg}[T'^n_1; \ldots; T'^n_{m_n}] \ldots \\ \qquad\qquad\qquad \ldots c_{n+l} : \mathit{arg}[T'^{n+l}_1; \ldots; T'^{n+l}_{m_{n+l}}]) \\ T^i_l \leq T'^i_l \quad (1 \leq l \leq n,\, 1 \leq i \leq m_l) \end{array}}{D \sqsubseteq D'}$$

$$(S_D)\quad \frac{D \sqsubseteq D' \qquad \forall i,\, (1 \leq i \leq k)\,.\, \Sigma \vdash A_i \leq A'_i}{\Sigma \vdash D[A_1; \ldots; A_k] \leq D'[A'_1; \ldots; A'_k]}$$

Fig. 4. Structural subtyping relation

has first to be noted that one can always suppose that both datatypes have the same number of parameters, some of them possibly being unused in the smaller type. Moreover, as the context of typing assumptions is empty in $T_l^i \leq {T'}_l^i$ (rule $(D_\sqsubseteq)$), this condition means that either T_l^i and ${T'}_l^i$ are comparable datatypes, or they represent *the same* type variable. Rule (S_D) makes the link with the rules of Fig. 1, by allowing one to inject $\sqsubseteq$ into $\leq$.
Examples: let us illustrate the meaning of relation $\sqsubseteq$ on two examples.
$(i)-$ consider the datatypes of booleans and tri–valued tags, defined as follows:

$$\texttt{DType}\ \ Bool\ \ \texttt{is}\ \ \texttt{true} : arg[],\ \texttt{false} : arg[]\,;$$
$$\texttt{DType}\ \ Bool'\ \ \texttt{is}\ \ \texttt{true} : arg[],\ \texttt{false} : arg[],\ \texttt{unknown} : arg[]\,.$$

It holds that $Bool \sqsubseteq Bool'$, because $Bool'$ has two constructors in common with $Bool$, and one extra constructor (no parameter is involved here).
$(ii)-$ suppose now we want to tag a term (of any type) with an element of $Bool$ or of $Bool'$; this would lead to the following definitions:

$$\texttt{DType}\ \ Tagged[X]\ \ \texttt{is}\ \ c : arg[X, Bool]\,;$$
$$\texttt{DType}\ \ Tagged'[X]\ \ \texttt{is}\ \ c : arg[X, Bool']\,.$$

We can derive $Tagged \sqsubseteq Tagged'$: indeed, they have the same number of parameters and we can derive both subtyping judgments $X \leq X$ and $Bool \leq Bool'$ for their first and second argument respectively.

Remark 4 (Real numbers). The framework we have introduced does not make it possible to introduce a datatype real for real numbers, as presented in Sec. 2. However, there is a priori no difficulty in mixing the approach we have adopted until this section with the introduction of user-defined datatypes, and keep an axiomatical presentation of real numbers, together with the base rule int $\leq$ real.

Canonical tables We now explain how to construct *canonical tables*, as used in subsection 4.2, for datatype constructors.

Definition 12 (Canonical table for datatype constructors).
Consider a datatype D, with its parameters $X_1, \ldots, X_k$. Recall that a datatype constructor c_i^D (we shall abbreviate it simply to c) is defined by arg $[L]$, where $L = T_1^i, \ldots, T_{m_i}^i$. We now define a function returning a pair $\langle \Theta^{can}(c), t\rangle$ where $\Theta^{can}(c)$ is the canonical table *associated to constructor c and t is the corresponding* root (type) variable.

Let $u_1, \ldots, u_k$ be fresh variables, we let $\sigma = \{X_1 := u_1, \ldots, X_k : u_k\}$, and define Θ^{param} as the table consisting in the k rows of the form $u_j \,|\, \emptyset \,|\, \emptyset$, $1 \leq j \leq k$. The result is then defined by recursion over the list L of "arguments" of c:

- *if $L = []$, take a fresh variable v, and return $\langle \texttt{addtable}(\Theta^{param}, \{v :_G (D[u_1, \ldots, u_k] \leq v)\}), v\rangle$;*
- *if $L = T_1^i, \ldots, T_{m_i}^i$, compute the canonical table for $T_2^i, \ldots, T_{m_i}^i$, yielding $\langle \Theta, t\rangle$. Then distinguish two cases, according to the shape of T_1^i:*
  - *if T_1^i is X_p for some p, then let v be a fresh variable; return $\langle \texttt{addtable}(v \triangleright \Theta, \{v :_G (u_p \to t \leq v)\}), v\rangle$;*

- *otherwise, let* $T_1^i = D'[X_1,\ldots,X_k]$ *and let* v *and* v' *be two fresh variables; return* $\langle \texttt{addtable}(v' \triangleright \Theta, \{v' :_G (v \to t \leq v'), [v'] :_G (v \leq D'[X_1,\ldots,X_k])\}), v' \rangle$.

Note that by definition, $\Theta^{can}(c)$ is already closed and consistent (consistency is insured by the fact that there is no application).

To evidenciate the dependency towards type variables, we can adopt the notation $\Theta^{can}(c)[u_1,\ldots,u_k,v_1,\ \ldots,v_m,t]$, t being the root variable of $\Theta^{can}(c)$ and $v_1,\ldots,v_m$ the type variables introduced during the analysis of L described above.

Example: consider the list constructor *cons*, of type $X \to list[X] \to list[X]$; its associated canonical table is

$$\begin{array}{r|c|l}
 & u_1 & \\
t : (u_1 \to v_1) & t & \\
t, v_1 : (t_2 \to v_2) & v_1 & \\
 & t_2 & t, v_1, t_2 : list[u_1] \\
t, v_1, v_2 : list[u_1] & v_2 &
\end{array}$$

Note that variables t_1 and u_1 collapsed, and we only keep u_1. Here are the scopes of the type variables:

$$cons \quad underbrace \underbrace{X}_{u_1} \to \overbrace{\underbrace{list[X]}_{t_2} \to \overbrace{list[X]}^{v_2}}^{v_1}{}_{t} \qquad D = list[X],\, u_1 = X\,.$$

The canonical tables we build can then be inserted in the type inference framework according to rule (T_{const}) of Fig. 3.

References

[AC93] R. Amadio and L. Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, 1993. 54

[AK95] Z. M. Ariola and J. W. Klop. Equational term graph rewriting. Technical report, University of Oregon, 1995. 54

[AS85] H. Abelson and G. J. Susman. *Structure and Interpretation of Computer Programs*. MIT Press, 1985. 50

[BB85] C. Böhm and A. Berarducci. Automatic Synthesis of Typed λ–programs on Term Algebras. *Theoretical Computer Science*, 39:135–154, 1985. 51

[BCDC83] H. P. Barendregt, M. Coppo, and M. Dezani-Ciancaglini. A filter lambda model and the completeness of type assignment. *Journal of Symbolic Logic*, 48:931–940, 1983. 51

[BH98] M. Brandt and F. Henglein. Coinductive axiomatization of recursive type equality and subtyping. *Fundamenta Informaticæ*, 33:309–338, 1998. 54

[BPG94] C. Böhm, A. Piperno, and S. Guerrini. Lambda-definition of Function(al)s by Normal Forms. In *Proc. of ESOP'94*, volume 788 of *LNCS*, pages 135–154. Springer Verlag, 1994. 51

[CC90] F. Cardone and M. Coppo. Two Extensions of Curry's Inference System. In P. Odifreddi, editor, *Logic and Computer Science*, pages 19–75. Accademic Press, 1990. 55

[CC91] F. Cardone and M. Coppo. Type inference with recursive types. Syntax and Semantics. *Information and Computation*, 92(1):48–80, 1991. 54

[Gom90] C. Gomard. Partial Type Inference for Untyped Functional Programs. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, pages 282–287, 1990. 51

[Jim96] T. Jim. What are principal typings and what are they good for? In *POPL'96*, pages 42–53. ACM, 1996. 52

[KPS94] D. Kozen, J. Palsberg, and M. I. Schwartzbach. Efficient inference of partial types. *Journal of Computer and System Sciences*, 49(2):306–324, 1994. also in Proceedings of FOCS'92, pages 363–371. 51, 52

[Mil78] R. Milner. A Theory of Type Polimorphism in Programming. *Journal of Computer and System Sciences*, 17:348–375, 1978. 54

[Tha94] S. Thatte. Type inference with partial types. *TCS*, 124:127–148, 1994. also in Proceedings of ICALP '88, pages 615–629. 51

[WO92] Mitchell Wand and Patrick M. O'Keefe. Type inference for partial types is decidable. In B. Krieg-Brückner, editor, *European Symposium on Programming '92*, volume 582 of *LNCS*, pages 408–417. Springer Verlag, 1992. 51, 52, 58

A A Detailed Example

We illustrate the way our procedure works on an example, that shows the building of the table and the treatment of what would be considered as typing errors in a classical setting. This example is studied with an implementation of the algorithms we have described—the source code is available at

`http://www.ens-lyon.fr/~hirschko/typetables`.

The user of our system introduces definitions of terms, and the system answers by showing the corresponding table being constructed, and its evolution (as the closure function is applied). The initial environment contains several constants, such as elements of types bool, int, real, lists, and a few constant functions of types int→bool, int→int, etc.

For the moment, we only have the implementation of the type inference procedure (table construction), and of the closure function. The generation of solutions and the heuristic to resolve inconsistencies are in beta version, and we leave their discussion to a later presentation of this work.

We shall start with the definition

```
M := \x y.((y (x 3)) (x x))
```

By using x both in an auto-application and as function on integers, we force a typing conflict in the table that is generated (not leading to an inconsistency, though). The type inference procedure yields the following table:

```
< i | {[i]. b->h} | {} >
< h | {[i;h]. a->g} | {} >
< g | {} | {} >
< f |{} | {} >
< e | {} | {[i;h;g]. f->g} >
< d | {} | {} >
< c | {[i;h;g;e;d]. INT} | {} >
< b | {} | {[i;h;g;f]. b->f;[i;h;g;e;d]. c->d} >
< a | {} | {[i;h;g;e]. d->e} >
```

Each line corresponds to a row in the table, and for each row, we successively give the corresponding type variable, and its sets L and U. For example, we can see that type variable a has an empty L set, and has the guarded expression $[i;h;g;e]:_G d\rightarrow e$ in its U set. The system also gives some extra information:

```
Current context is: M:i
To help debugging, here are the "old" variable assumptions
  x:b /  y:a
```

This means that during type inference term variables `x` and `y` have been associated to type variables `b` and `a` respectively. Using this information, we can reconstruct the structure of the term, by establishing a correspondence between the type variables and every subterm of the term M, as follows:

$$\overbrace{\lambda x^b.\,\overbrace{\lambda y^a.\,\underbrace{\underbrace{y\ \underbrace{(x\,3^c)}_{d}}_{e}\ \underbrace{(x\,x)}_{f}}_{g}}^{h}}^{i}$$

To compute this decoration, we reason like this: $y\ x\,3)$ being of type e, we read in the row corresponding to e that e is less than $f\rightarrow g$ (omitting the guards); as $(x\,x)$ is of type f, we obtain that $y\ x\,3)\ x\,x)$ is of type g. We can also read the following type constraints:

$$b \leq b\rightarrow f \qquad\qquad b \leq c\rightarrow d \qquad\qquad c \geq \mathtt{INT}\,.$$

The cyclic constraint on b comes from the auto-application of x, and the application of x to 3 generates the two other inequalities. The system then applies the closure rules; the previous table being already closed, it remains unchanged in this case. The conflicts coming from the "non–standard" use of x in M are not well visible on the table above. The type scheme for `M` produced by applying the canonical substitution defined in Sec. 3.3 would be $\bot\rightarrow(d\rightarrow f\rightarrow g)\rightarrow g$. This looks certainly strange but is obtained without using any (ω) reduction. To meet real inconsistencies we need to go further in our session.

Let us now define

```
N := (M \z.z)
```

After the application of the `closure` operation we get for `N` the following:

```
The closed table is:
  < l | {[l;i;h]. a->g} | {} >
  < k | {[l;k]. j->j} | {[l;i;h;g;f]. b->f;[l;i;h;g;e;d].c->d}>
```

```
* < j | {[l;i;h;g;f;k]. j->j;[l;i;h;g;e;d;k]. INT} | {} >
  < i | {[l;i]. b->h} | {[l]. k->l} >
  < h | {[l;i;h]. a->g} | {} >
  < g | {} | {} >
* < f |  {[l;i;k;h;g;f]. j->j;[l;i;k;h;g;f;e;d]. INT} | {} >
  < e | {} | {[l;i;h;g]. f->g} >
* < d | {[l;i;k;h;g;e;d;f]. j->j;[l;k;i;h;g;e;d]. INT} | {} >
  < c | {[l;i;h;g;e;d]. INT} | {} >
  < b | {[l;i;k]. j->j} | {[l;i;h;g;f].b->f;[l;i;h;g;e;d].c->d}>
  < a | {} | {[l;i;h;g;e]. d->e} >
```

A star "*" indicates the rows where a conflict is apparent (between an arrow type and datatype `INT`): this is the case for type variables `d`, `f` and `j` (that intuitively correspond to the points where the two different typings for x "meet"). However, the resulting table is still consistent, since the upper set is empty for these entries (which means that we can still use rule (S_ω)). The type scheme for `N` that we get from the canonical substitution is now $(\omega\to\omega\to g)\to g$. Note how this type, although meaningful, does not fully represent all the informations contained in the tables.

Let us now define:

```
P := (N (\u v.v))
```

we obtain a table which, after closure, indicates a root type variable `q` for the whole term whose corresponding entry is

```
<q | {[r;q;l;p;g;e;f]. j->j;[r;q;l;p;g;e;f;d]. INT} | {}>
```

the table is still consistent but to solve it we are forced to take $q = \omega$. This is meaningful since this type is obtained without using the ω reduction (i.e. in the system $\vdash^{-}$), and this guarantees that the term can be reduced without encountering bad applications.

Indeed the right type for `P` would be int. But if we try now to typecheck the expression:

```
(S P)
```

where `S` is the constant for the successor function of type int→int we get an inconsistent table containig entries like this:

```
< t | {[v;u;t;o;s;j;h;i]. m->m;[v;u;t;o;s;j;h;i;g]. INT}|
      {[v;u;c;a]. INT}>
```

However with an ω reduction we are still able to infer the correct type int for `(S P)`.

Call-by-Value Separability and Computability

Luca Paolini

[1] DISI - Università di Genova
Via Dodecaneso, 35, 16146 Genova - Italia
[2] IML - Université de la Méditerranée, Campus de Luminy, Case 907
13288 MARSEILLE Cedex 9 - France
paolini@{disi.unige.it,iml.univ-mrs.fr}

Abstract The aim of this paper is to study the notion of separability in the call-by-value setting.
Separability is the key notion used in the Böhm Theorem, proving that syntactically different $\beta\eta$-normal forms are separable in the classical λ-calculus endowed with β-reduction, i.e. in the call-by-name setting.
In the case of call-by-value λ-calculus endowed with β_v-reduction and η_v-reduction (see Plotkin [7]), it turns out that two syntactically different $\beta\eta$-normal forms are separable too, while the notion of β_v-normal form and η_v-normal form is semantically meaningful.
An explicit representation of Kleene's recursive functions is presented. The separability result guarantees that the representation makes sense in every consistent theory of call-by-value, i.e. theories in which not all terms are equals.

1 Introduction

The call-by-value λ-calculus ($\lambda\beta_v$-calculus) and the operational machine for its evaluation has been introduced by Plotkin [7] inspired by the seminal work of Landin [4] on the language ISWIM and the SECD machine.

The $\lambda\beta_v$-calculus is a paradigmatic language able to capture two features present in many real functional programming languages: call-by-value parameter passing and lazy evaluation. The parameters are passed in a call-by-value way, when they are evaluated before being passed and a function is evaluated in a lazy way when its body is evaluated only when parameters are supplied.

In this paper we are dealing with pure (i.e. without constants) version of $\lambda\beta_v$-calculus. Plotkin has endowed this calculus by two rules, namely β_v and η_v, which are obtained by restriction from respectively β-rule and η-rule of classical (call-by-name) λ-calculus. This restriction is based on the notion of *value*. Values are either variables or abstractions.

Formally, the β_v-rule is: $(\lambda x.M)N \to M[N/x]$, if and only if N is a value. Let $=_{\beta_v}$ be the congruence relation induced by the β_v-reduction. A term $M \in \Lambda$ is said *valuable* if and only if $M =_{\beta_v} P$, for some value P.

A. Restivo, S. Ronchi Della Rocca, L. Roversi (Eds.): ICTCS 2001, LNCS 2202, pp. 74–89, 2001.

Plotkin has proved that the $\lambda\beta_v$-calculus enjoys some basic properties we expected from a calculus, namely Church-Rosser and standardisation property.

However in standard λ-calculus there is another fundamental theorem: Böhm Theorem [2].
The standard notion of separability is: "two terms M, N are *separable* if and only if there exists a context $C[.]$, such that $C[M] =_\beta x$ and $C[N] =_\beta y$, where x, y are different variables" (see [1,8]). The Böhm Theorem says that two different $\beta\eta$-normal forms are separable.

The importance of Böhm Theorem has been pointed out by Wadsworth, which in [9] says: "The Church-Rosser Theorem shows that distinct normal forms cannot be proved equals by the conversion rules; the Böhm Theorem shows that if one were ever to postulate, as an extra axiom, the equality of two distinct normal forms, the resulting system would be inconsistent".

In particular, the Böhm Theorem allows the coding of computable functions in λ-calculus, since by representing different natural numbers by different $\beta\eta$-normal forms, assures that they are different in every consistent λ-theory.

It is natural, to state that two terms M, N are *v-separable* if and only if there exists a context $C[.]$, such that $C[M] =_{\beta_v} x$ and $C[N] =_{\beta_v} y$, where x, y are different variables.

The naïve adaptation of Böhm-Theorem to call-by-value λ-calculus would be:

"two different $\beta_v\eta_v$-normal forms are v-separable".

It is immediate to check that two syntactically different $\beta_v\eta_v$-normal forms are not always separable, for example consider the following terms: $\lambda x.xxx$ and $\lambda x.(\lambda z.xxx)(xx)$. Thus, $\beta_v\eta_v$-normal forms are not semantically meaningful.

Actually, there is a subtler problem with $\beta_v\eta_v$-normal forms. Let $I \equiv \lambda x.x$, $\Delta \equiv \lambda x.xx$ and $M \equiv (\lambda x.\Delta)(xI)\Delta$. Clearly M is a $\beta_v\eta_v$-normal form (since xI is not a value), but you can check that $C[M] \rightarrow_{\beta_v} x$ implies $C[N] \rightarrow_{\beta_v} x$, for all $N \in \Lambda$. Terms as M are discovered and studied in [5], [6]: a term is said *potentially valuable* if and only if there exists a substitution that make it valuable. Clearly, M is not potentially valuable.

Plotkin in [7] gives simulations of call-by-value λ-calculus in the call-by-name and vice versa, by using continuation techniques. Thus, he has implicitly showed that the computational power of the two calculus is the same.

In this paper, an explicit representation of Kleene's recursive functions is presented, based on a coding of natural numbers using $\beta\eta$-normal forms, as in the classical λ-calculus case, but using the β_v-reduction as computational rule. Let $n, n \in \mathbb{N}$ be different; if $\bar{m}$ and $\bar{n}$ are their λ-representation then $\bar{m}$ and $\bar{n}$ must be different in our theory. This is true in the call-by-value setting, since:

"two different $\beta\eta$-normal forms are v-separable"

whose proof is the most important result of this paper.

This separation result is based on the fact that every subterm of a $\beta\eta$-normal form is a potentially valuable term.

The main difficulty in carrying out the proof of Böhm-Theorem, basically consists in handling open subterms that are neither values nor valuables (because they are in normal form). For instance, let $M \equiv x(xP_0)Q$ and $N \equiv x(xP_1)Q$ be $\beta\eta$-normal forms. A context $C[.]$ v-separating M and N need to handle subterms as xP_0, xP_1 and Q by using the β_v-reduction. Thus, $C[.]$ needs being able to transform xP_0, xP_1 and Q in values, by a "uniform substitution" preserving the structural difference. Our main goal is to show as it is possible to build such a substitution.

In the algorithm, some β-reduction is taken in order to normalise terms after substitutions. Thus, an additional problem is to show that these β-reductions can be "reconciled", in some sense, with β_v-reduction. In general, from $=_\beta \not\subseteq =_{\beta_v}$ follows that separation results using β-reduction as computation rule do not imply the v-separation results.

The semantical consequence of the separability result, is that two different $\beta\eta$-normal forms cannot be equated in models of call-by-value λ-calculus.

A theory of call-by-value λ-calculus is a congruence relation, containing the relation $=_{\beta_v}$.
Let $=_\mathcal{T}$ be a such theory; if M and N are v-separable terms, such that $M =_\mathcal{T} N$ then $=_\mathcal{T}$ is inconsistent, i.e. all terms are equals. In fact, if $C[.]$ is the context such that $C[M] =_{\beta_v} x$ and $C[N] =_{\beta_v} y$ then $P =_{\beta_v} (\lambda xy.C[M])PQ =_{\beta_v} (\lambda xy.C[N])PQ =_{\beta_v} Q$, for every $P, Q \in \Lambda$.

The paper is organised in the following way. In section 2 basic definitions and notions are recalled. In section 3 the notion of v-separability and similarity are introduced, together with the Separability Algorithm; furthermore, its termination and correctness are proved. In section 4 a representation of Kleene's recursive function for the call-by-value λ-calculus is presented and proved correct.

2 $\lambda\beta$-calculus and $\lambda\beta_v$-calculus

The pure language of λ-calculus is defined as usual (see [1,8]).

Definition 1. *Let Var be a denumerable set of variables, ranged over $x, y, z, ...$.*
Let Λ be the set of λ-terms M built by the following grammar:

$$M ::= x | MM | \lambda x.M$$

Let $M, N, P, Q, ...$ to denote terms. A term of shape (MN) is said application*, while a term of shape $(\lambda x.M)$ is said* abstraction*.*

Free and bound variables are defined as usual, $FV(M)$ denotes the set of free variables of a term M and $\Lambda^0 \subset \Lambda$ denotes the set of closed terms. Terms are considered modulo α-conversion, that is up to renaming of bound variables. $M[N/x]$ denotes the substitution of N for every free occurrence of x in M, eventually by renaming bound variables of M in order to avoid a wrong binding of free variables of N. $\equiv$ denotes syntactical identity on terms, up to α-conversion. $\lambda x_1 \ldots x_n.M$ is an abbreviation for $\lambda x_1.(\lambda x_2.(\ldots\ (\lambda x_n.M)))$ and $M_1 \ldots M_m$ is an abbreviation for $((\ldots((M_1 M_2)M_3)\ldots)M_m)$. A context is a term containing some hole [.] .

Definition 2.

- *The β-rule is:* $(\lambda x.M)N \to M[N/x]$.
- *The η-rule is:* $\lambda x.Mx \to M$ *if and only if* $x \notin FV(M)$.
- *The β_v-rule is:* $(\lambda x.M)N \to M[N/x]$ *if and only if* $N \in Val$;
  where $Val = Var \cup \{\lambda x.M \mid x \in Var \text{ and } M \in \Lambda\}$ *is the set of* values.
- *The η_v-rule is:* $\lambda x.Mx \to M$ *if and only if* $x \notin FV(M)$ *and* $M \in Val$.

Let $\diamond \in \{\beta, \eta, \beta_v, \eta_v, \beta\eta, \beta_v\eta_v\}$ then $\to_\diamond$, $\to_\diamond^*$ and $=_\diamond$ denote respectively the contextual closure of $\diamond$-rule(s), the reflexive and transitive closure of $\to_\diamond$ and the reflexive, symmetric and transitive closure of $\to_\diamond$.

A term M is in $\diamond$-*normal form* (noted $M \in \diamond$-nf) if and only if, in M there are no occurrences of $\diamond$-redexes, i.e. there are no subterms that can be $\diamond$-reduced.

It is well-known that the β-normal forms have the shape: $\lambda x_1 \ldots x_n.x M_1 \ldots M_m$ where $n, m \geq 0$ and all M_i are in β-normal forms. While, the shape of a β_v-normal form is: $\lambda x_1 \ldots x_n.\xi M_1 \ldots M_m$ where $n, m \geq 0$, all M_i are in β_v-normal forms and $\xi \in Var$ or $\xi \equiv (\lambda x.P)Q$, with $P, Q \in \beta_v$-nf and $Q \notin$ Val .

Definition 3.

– $M \in \Lambda$ *is* valuable *if and only if* $M \to_{\beta_v}^* N \in Val$, *for some* $N \in \Lambda$.
– *A term is* potentially valuable *if and only if there is a substitution* s *of values for free variables such that* $\mathsf{s}(M)$ *is valuable.*

The set of potentially valuable terms, noted $\mathcal{P}_v$ has been completely characterised in [5,6].
Let $I \equiv \lambda x.x$ and $\Delta \equiv \lambda x.xx$. Note that $x(I\Delta)$ and $(\lambda y.\Delta)(xI)\Delta$ are two not valuable β_v-normal forms; moreover, only the first term is potentially valuable.

Let $M \equiv zM_0M_1$, $M_1 \in \text{Val} \cap \Lambda^0$ and $z \notin \text{FV}(M_0)$. It is possible to build a context $C[.]$, such that $C[M] \to_{\beta_v} M_1$, only under the necessary and sufficient condition that M_0 is a potentially valuable term. Thus a term is potentially valuable if and only if it can be erased, or simply handled by β_v-reduction, after some substitution.

It is easy to check that every β-normal form M is a potentially valuable term; furthermore, recursively a subterm N of M is a β-normal form too, and so a potentially valuable term.

Definition 4. *Let $M \in \beta$-nf; step$[M]$ is the natural number given by:*

- $step[xM_1 \ldots M_m] = 1 + step[M_1] + ... + step[M_m]$
- $step[\lambda y.N] = 1$

step is a structural measure on term, considering recursively the number of arguments of head variables not under a λ-abstraction. See [5,6] for more details. Clearly, $\forall M \in \beta$-nf $\exists n \in \mathbb{N}$ such that $step[M] = n$.

Lemma 1. *Let $M \in \beta$-nf, $FV(M) \subseteq \{x_1, \ldots, x_n\}$ and $r \geq step[M]$. If $Q_j^r \equiv \lambda x_1 \ldots x_{r-1}.Q_j$ and $Q_j \in Val$ $(1 \leq j \leq n)$ then $M[Q_1^r/x_1, ..., Q_n^r/x_n] \rightarrow^*_{\beta_v} \bar{M}$, for some $\bar{M} \in Val$.*

Proof. By induction on *step*. Let $step[M] = 1$, thus $M \equiv \lambda y.N$ or $M \equiv x$.
If $step[M] > 1$ then $M \equiv xM_1 \ldots M_m$, so there exists $k \leq n$ such that $M' \equiv M[Q_1^r/x_1, ..., Q_n^r/x_n] \equiv Q_k^r M_1' \ldots M_m'$, where $M_i' \equiv M_i[Q_1^r/x_1, ..., Q_n^r/x_n]$ $(1 \leq i \leq m)$.
$step[M] = 1 + \sum_{i=1}^m step[M_i]$ and $step[M_i] \geq 1 (1 \leq i \leq m)$ imply $r \geq step[M] > m$; moreover, by induction $M_i' \rightarrow^*_{\beta_v} \bar{M}_i \in$ Val $(1 \leq i \leq m)$. Thus $M' \rightarrow^*_{\beta_v} \lambda x_{m+1} ... x_{r-1}.Q_k \in$ Val. □

In order to extend this property to every subterm N of a β-normal form M, let Vis be a structural measure on a β-normal form, considering recursively the number of arguments of head variables.

Definition 5. *Let $M \in \beta$-nf; $Vis[M]$ is the natural number given by:*

- $Vis[xM_1 \ldots M_m] = 1 + Vis[M_1] + ... + Vis[M_m]$
- $Vis[\lambda y.N] = 1 + Vis[N]$

It is easily seen that $\forall M \in \beta$-nf $\exists n \in \mathbb{N}$ such that $Vis[M] = n$.

Lemma 2. *If N is a subterm of $M \in \beta$-nf then $N \in \beta$-nf and $step[N] \leq Vis[M]$.*

Proof. Trivial. □

Let $\rightarrow_s$ to denote the strategy that reduce, at every step, the leftmost β_v-redex, not under the scope of a λ-abstraction. This strategy is normalising, i.e. if the terms $M \in \Lambda^0$ is valuable then $\exists \bar{M} \in$ Val such that $M \rightarrow_s \bar{M}$. The operational evaluation of call-by-value, showed in [7] by Plotkin, can be obtained by this reduction. Thus, in the section 4 we use the s-reduction.

3 v-Separability

Let us recall the formal definition of v-separability.

Definition 6. *Two terms $M, N \in \Lambda$ are v-separable if and only if $\exists C[.]$ such that $C[M] =_{\beta_v} \tilde{x}$ and $C[N] =_{\beta_v} \tilde{y}$, where $\tilde{x}, \tilde{y}$ are different variables.*

In order to design the v-separability algorithm, we introduce the notion of similarity between β-normal-forms.

Definition 7. *Let $M, N \in \beta\text{-}nf$, $M =_\eta \lambda x_1 \ldots x_p.xM_1 \ldots M_n$, $N =_\eta \lambda x_1 \ldots x_p.y$ $N_1 \ldots N_n$ with p, $n \geq 0$. We say that they are similar, noted $\simeq$, if and only if $x \equiv y$ and $\forall i.\ M_i \simeq N_i$.*

The relation $\simeq$ is introduced in order to make explicit the interesting structure of terms, for the separability goal (see [3]).

Let $M, N \in \beta\text{-}nf$; it is easy to check that $M =_\eta N$ if and only if $M \simeq N$.

Definition 8. *Let σ be a sequence of natural numbers (ϵ is the empty sequence) and $M, N \in \beta\text{-}nf$.*
$M \not\cong_\sigma N$ if and only if one of following cases arises

1. *if $x \not\equiv y$ then $M =_\eta \lambda x_1 \ldots x_p.xM_1 \ldots M_m \not\cong_\sigma \lambda x_1 \ldots x_q.yN_1 \ldots N_n =_\eta N$ and $\sigma \equiv \epsilon$;*
2. *if $|p-m| \neq |q-n|$ then $M =_\eta \lambda x_1 \ldots x_p.xM_1 \ldots M_m \not\cong_\sigma \lambda x_1 \ldots x_q.xN_1 \ldots N_n =_\eta N$ and $\sigma \equiv \epsilon$;*
3. *if $M_i \not\cong_{\sigma'} N_i$ then $M =_\eta \lambda x_1 \ldots x_p.xM_1 \ldots M_n \not\cong_\sigma \lambda x_1 \ldots x_p.xN_1 \ldots N_n =_\eta N$ and $\sigma \equiv i, \sigma'$.*

The $\not\simeq$ relation is formalised by the following lemma.

Lemma 3. *Let $M, N \in \beta\text{-}nf$. $M \not\simeq N$ if and only if $M \not\cong_\sigma N$, for some sequence σ.*

Proof. Trivial. □

We will prove that two not similar β-normal forms M, N are v-separable. More precisely, let $\mathrm{FV}(M) \cup \mathrm{FV}(N) \subseteq \{z_1, \ldots, z_h\}$, $\breve{M} \equiv \lambda z_1 \ldots z_h.M$ and $\breve{N} \equiv \lambda z_1 \ldots z_h.N$; we will design an algorithm which, builds a context $\breve{C}[.]$ on $\breve{M}, \breve{N}$ such that:

- if $\breve{M}, \breve{N}$ are closed then $\breve{C}[\breve{M}] =_{\beta_v} \tilde{x}$ and $\breve{C}[\breve{N}] =_{\beta_v} \tilde{y}$, where $\tilde{x} \not\equiv \tilde{y}$.

It is easily seen that $\breve{M} \not\simeq \breve{N}$ and $C[\lambda z_1 \ldots z_h.[.]]$ is v-separating M, N.

If $M, N \in \beta\eta\text{-}nf$ and $M \not\equiv N$ then $M \neq_{\beta\eta} N$ and $M \not\simeq N$; thus, it would be clear that two different $\beta\eta$-normal forms will be v-separable.

For sake of simplicity, in the algorithm description, we assume that all bound variables are denoted with a variable symbol different from each other (free or bound) variable symbol, in the same term.

Definition 9 (Separability Algorithm).
Let $M, N \in \beta\text{-}nf$, $M \not\simeq N$, $\tilde{x} \not\equiv \tilde{y}$ and $r = \max\{Vis[M], Vis[N]\}$.

The separability algorithm is a set of logical rules for proving statements of shape $M, N \Rightarrow C[.]$, where $C[.]$ is a context.

Let $I \equiv \lambda x.x$, $O^n \equiv \lambda x_1 \ldots x_n.I$, $U_n^i \equiv \lambda x_1 \ldots x_n.x_i$ *and* $\pi^n \equiv \lambda x_1 \ldots x_n z.z\, x_1 \ldots x_n$.

Furthermore, if $S \subseteq Var$ *then* $X^n_{x_i,S} \equiv \begin{cases} \lambda z_1 \ldots z_{n-1}.\ x_i z_1 \ldots z_{n-1} & \textit{if } x_i \notin S; \\ x_i & \textit{otherwise.} \end{cases}$

$$\dfrac{\begin{array}{c} \forall i.\quad M_i[X^r_{x_1,\{x,y\}}/x_1, ..., X^r_{x_p,\{x,y\}}/x_p] \to^*_\beta \bar{M}_i \in \beta\text{-}nf \\ \forall i.\quad N_i[X^r_{x_1,\{x,y\}}/x_1, ..., X^r_{x_q,\{x,y\}}/x_q] \to^*_\beta \bar{N}_i \in \beta\text{-}nf \\ p \leq q \quad x\bar{M}_1 \ldots \bar{M}_m X^r_{x_{p+1},\{x,y\}} ... X^r_{x_q,\{x,y\}}, y\bar{N}_1 \ldots \bar{N}_n \Rightarrow C[.] \end{array}}{\lambda x_1 \ldots x_p.xM_1 \ldots M_m, \lambda x_1 \ldots x_q.yN_1 \ldots N_n \Rightarrow C[[.]X^r_{x_1,\{x,y\}} ... X^r_{x_q,\{x,y\}}]}\ (1)$$

$$\dfrac{\begin{array}{c} \forall i.\quad M_i[X^r_{x_1,\{x,y\}}/x_1, ..., X^r_{x_p,\{x,y\}}/x_p] \to^*_\beta \bar{M}_i \in \beta\text{-}nf \\ \forall i.\quad N_i[X^r_{x_1,\{x,y\}}/x_1, ..., X^r_{x_q,\{x,y\}}/x_q] \to^*_\beta \bar{N}_i \in \beta\text{-}nf \\ p > q \quad x\bar{M}_1 \ldots \bar{M}_m, y\bar{N}_1 \ldots \bar{N}_n X^r_{x_{q+1},\{x,y\}} ... X^r_{x_p,\{x,y\}} \Rightarrow C[.] \end{array}}{\lambda x_1 \ldots x_p.xM_1 \ldots M_m, \lambda x_1 \ldots x_q.yN_1 \ldots N_n \Rightarrow C[[.]X^r_{x_1,\{x,y\}} ... X^r_{x_p,\{x,y\}}]}\ (2)$$

$$\dfrac{x \not\equiv y \qquad s = \max\{r, m, n\}}{xM_1 \ldots M_m, yN_1 \ldots N_n \Rightarrow (\lambda xy.[.])(\lambda x_1 \ldots x_{s+m}.\tilde{x})(\lambda x_1 \ldots x_{s+n}.\tilde{y})\underbrace{I.....I}_{s}}\ (3)$$

$$\dfrac{m > n \qquad s = \max\{r, m, n\}}{xM_1 \ldots M_m, xN_1 \ldots N_n \Rightarrow (\lambda x.[.])O^{s+n}\underbrace{I.....I}_{s+n-m}(\lambda x_1 \ldots x_{m-n}.\tilde{x})\underbrace{\tilde{y}.....\tilde{y}}_{m-n}}\ (4)$$

$$\dfrac{m < n \qquad s = \max\{r, m, n\}}{xM_1 \ldots M_m, xN_1 \ldots N_n \Rightarrow (\lambda x.[.])O^{s+m}\underbrace{I.....I}_{s+m-n}(\lambda x_1 \ldots x_{n-m}.\tilde{y})\underbrace{\tilde{x}.....\tilde{x}}_{n-m}}\ (5)$$

$$\dfrac{M_k \not\simeq N_k \qquad s = \max\{r, m, n\} \qquad x \notin FV(M_k) \cup FV(N_k) \quad M_k, N_k \Rightarrow C[.]}{xM_1 \ldots M_m, xN_1 \ldots N_m \Rightarrow C[(\lambda x.[.])U^s_k\underbrace{I.....I}_{s-m}]}\ (6)$$

$$\frac{\begin{array}{c} M_k \not\approx N_k \quad s = \max\{r, m, n\} \quad x \in FV(M_k) \cup FV(N_k) \quad C'[.] \equiv (\lambda x.[.])\pi^s \\ C'[M_k] \to^*_\beta \tilde{M}_k \in \beta\eta\text{-}nf \quad C'[N_k] \to^*_\beta \tilde{N}_k \in \beta\eta\text{-}nf \quad \tilde{M}_k, \tilde{N}_k \Rightarrow C[.] \end{array}}{xM_1 \ldots M_m, xN_1 \ldots N_m \Rightarrow C[(\lambda x.[.])\pi^s \underbrace{I.....I}_{s-m} U^s_k]} \ (7)$$

In order to prove both correctness and termination of algorithm, we need some preliminary lemmas.

Lemma 4. *Let $M, N \in \beta$-nf, $r \geq \max\{Vis[M], Vis[N]\}$ and $C'[.] \equiv (\lambda x.[.])\pi^r$.*

1. *$\exists \bar{M} \in \beta$-nf such that $C'[M] \to^*_\beta \bar{M}$ and $Vis[M] \leq Vis[\bar{M}]$.*
2. *If $P \equiv xP_1 \ldots P_p$ is a subterm of $\bar{M}$ then $step[P_j] \leq r$ $(1 \leq j \leq p)$.*
3. *If $M \not\cong_\sigma N$ and $C'[N] \to^*_\beta \bar{N} \in \beta$-nf then $\bar{M} \not\cong_\sigma \bar{N}$.*

Proof. $1, 2$. By induction on M. 3. By induction on σ. □

Lemma 5. *Let $M, N \in \beta$-nf, $r \geq \max\{Vis[M], Vis[N]\}$ and $C''_p[.] \equiv [.] X^r_{x_1,\{x,y\}} \ldots X^r_{x_p,\{x,y\}}$.*

1. *$\exists \bar{M} \in \beta$-nf such that $C''_p[M] \to^*_\beta \bar{M}$ and $Vis[M] \leq Vis[\bar{M}]$.*
2. *If $P \equiv xP_1 \ldots P_p$ is a subterm of $\bar{M}$ then $step[P_j] \leq r$ $(1 \leq j \leq p)$.*
3. *If $M \not\cong_\sigma N$ and $C''_p[N] \to^*_\beta \bar{N} \in \beta$-nf then $\bar{M} \not\cong_\sigma \bar{N}$.*

Proof. $1, 2$. By induction on M. 3. By induction on σ. □

Be careful to understand the statement of Lemmas 4 and 5, since for some subterm M' of $\bar{M}$, *step*$[M'] \leq r$ but it is possible that $Vis[\bar{M}] > r$.

Lemma 6 (Termination).
If $M, N \in \beta$-nf and $M \not\approx N$ then $M, N \Rightarrow C[.]$.

Proof. $M \not\approx N$ implies $M \not\cong_\sigma N$, for some sequence of numbers σ. By induction on σ. Observe that the rules (3), (4), (5) are axioms and the rules (6), (7) follow by induction. Rules (1) and (2) must be followed, by a rule between (3), (4), (5), (6) and (7). □

Thus $M, N \in \beta$-nf and $M \not\approx N$ implies that there is a finite derivation such that $M, N \Rightarrow C[.]$.

Theorem 1 (Correctness). *Let $M, N \in \beta\text{-}nf$, $M \not\approx N$ and $FV(M) \cup FV(N) \subseteq \{z_1, \ldots, z_h\}$.*
Let $\breve{M} \equiv \lambda z_1 \ldots z_h.M$ and $\breve{N} \equiv \lambda z_1 \ldots z_h.N$.
If $\breve{M}, \breve{N} \Rightarrow \breve{C}[.]$ then $C^*[.] \equiv \breve{C}[\lambda z_1 \ldots z_h.[.]]$ is a context v-separating M and N, namely $C^*[M] =_{\beta_v} \tilde{x}$ and $C^*[N] =_{\beta_v} \tilde{y}$, where $\tilde{x} \not\equiv \tilde{y}$.

Proof. By next proposition. □

The use of β-reduction in the rules (1), (2) and (7) of our algorithm cause some technical difficulty in the development of correctness proof, since the β_v-reduction cannot to execute, in general, the same redexes. In order to fill this gap, we will prove something more the statement of Correctness Theorem, namely the Proposition 1.

Some observation is needed before to build the statement of Proposition 1. Let $P \in \beta$-nf and $S, R \subseteq Var$. If $x_1 \in S \cap R$ then $X^r_{x_1,S} = X^r_{x_1,R} = x_1$ and $P[X^r_{x_1,S}/x_1][X^r_{x_1,R}/x_1] = P$, otherwise $P[X^r_{x_1,S}/x_1][X^r_{x_1,R}/x_1] = P[\lambda z_1 \ldots z_r.x_1 z_1 \ldots z_r/x_1]$.

Thus $P[X^r_{x_1,S}/x_1][X^r_{x_1,R}/x_1] = P[X^r_{x_1,R}/x_1][X^r_{x_1,S}/x_1] = P[X^r_{x_1,R \cap S}/x_1]$.

Furthermore, $P[X^r_{x_1,S}/x_1][\pi^r/x_1] = P[\pi^r/x_1][X^r_{x_1,S}/x_1] = P[\pi^r/x_1]$.

Proposition 1. *Let $P, Q \in \beta$-nf, such that $P \not\simeq Q$.*
Let $\rho \geq \max\{Vis[P], Vis[Q]\}$ and $FV(P) \cup FV(Q) \subseteq \{u_1, ..., u_t\}$.

Let $\forall i$. U_i be values of shape π^r or $X^r_{z_i,S_i}$ ($S_i \subseteq Var$ and $r \geq \rho$), such that U_i has shape $X^r_{z_i,S_i} \equiv z_i$ (in case $z_i \in S_i$) if and only if $M \equiv z_i M_1 ... M_m$ or $N \equiv z_i N_1 ... N_n$.
Let $P' \equiv (\lambda u_1 \ldots u_t.P)U_1 ... U_T =_\beta M \in \beta$-nf, $Q' \equiv (\lambda u_1 \ldots u_t.Q)U_1 ... U_T =_\beta N \in \beta$-nf and $t \leq T$.

If $M, N \Rightarrow C^*[.]$ then $C^*[P'] =_{\beta_v} \tilde{x}$ and $C^*[Q'] =_{\beta_v} \tilde{y}$, where $\tilde{x} \not\equiv \tilde{y}$.

Proof. By induction on the derivation proving $M, N \Rightarrow C^*[.]$.

Note that $M \not\simeq N$, $Vis[P] \leq Vis[M]$, $Vis[Q] \leq Vis[N]$ and for some subterm R of M or N, $step[R] \leq \rho$, by Lemmas 4 and 5.

Let $M \equiv \lambda x_1 \ldots x_p.x M_1 \ldots M_m$ and $N \equiv \lambda y_1 \ldots y_q.y N_1 \ldots N_n$.

(1) $x\bar{M}_1 \ldots \bar{M}_m X^r_{x_{p+1},\{x,y\}} \cdots X^r_{x_q,\{x,y\}}, \ y\bar{N}_1 \ldots \bar{N}_n \ \Rightarrow \ C[.]$ and $C^*[.] \ \equiv \ C[[.] \ X^r_{x_1,\{x,y\}} \cdots X^r_{x_q,\{x,y\}}]$.
Let $P'' \equiv P' X^r_{x_1,\{x,y\}} \cdots X^r_{x_q,\{x,y\}}$, thus
$P'' \to^*_\beta x\bar{M}_1 ... \bar{M}_m X^r_{x_{p+1},\{x,y\}} \cdots X^r_{x_q,\{x,y\}} \in \beta$-nf.
By induction $\tilde{x} =_{\beta_v} C[P'']$, but

$$C[P''] \equiv C[P' X^r_{x_1,\{x,y\}} ... X^r_{x_q,\{x,y\}}] \equiv C^*[P']$$

$C^*[Q'] =_{\beta_v} \tilde{y}$ is similar.

(2) Similar to case (1).

(3) In such a case $x \not\equiv y$ and $p = q = 0$. Let $P \equiv \lambda w_1 \ldots w_{n_p}.w P_1 \ldots P_{m_p} \in \beta$-nf ($m_p \leq m$) and $P' \equiv (\lambda u_1 \ldots u_t.P)U_1 \ldots U_T \to^*_\beta x M_1 \ldots M_m \equiv M$. Note that $m = m_p + T - (t + n_p)$.

Since $\forall i.\ U_i \in \text{Val}$, then $P' =_{\beta_v} (\lambda w_1\ldots w_{n_p}.wP'_1\ldots P'_{m_p})U_{t+1}\ldots U_T$ where $P'_i \equiv P_i[U_1/u_1\ldots U_t/u_t]$ $(1 \le i \le m_p)$.
Moreover $P' =_{\beta_v} xP''_1\ldots P''_{m_p}U_{t+n_p+1}\ldots U_T \equiv P''$ where $P''_i \equiv P'_i[U_{t+1}/w_1\ldots U_{t+n_p}/w_{n_p}] =_\beta M_i$ $(1 \le i \le m_p)$ and $U_j \equiv M_j$ $(t+n_p+1 \le j \le T)$.

$$\begin{aligned}
&C^*[P'] =_{\beta_v} C^*[P'']\\
&\equiv (\lambda xy.x\,\underbrace{P''_1\ldots P''_{m_p}U_{t+n_p+1}...U_T}_{m})(\lambda x_1\ldots x_{s+m}.\tilde{x})(\lambda x_1\ldots x_{s+n}.\tilde{y})\,\underbrace{I...I}_{s}\\
&=_{\beta_v} (\lambda x_1\ldots x_{s+m}.\tilde{x})\,\underbrace{P'''_1\ldots P'''_{m_p}U'_{t+n_p+1}...U'_T}_{m}\,\underbrace{I.....I}_{s}
\end{aligned}$$

where $\begin{cases} P'''_i \equiv P''_i[(\lambda x_1\ldots x_{s+m}.\tilde{x})/x][(\lambda x_1\ldots x_{s+n}.\tilde{y})/y];\\ U'_j \equiv U_j[(\lambda x_1\ldots x_{s+m}.\tilde{x})/x][(\lambda x_1\ldots x_{s+n}.\tilde{y})/y].\end{cases}$

Since $FV(P_i) \subseteq \{u_1, ..., u_t\}$

$$P''_i[(\lambda x_1\ldots x_{r+m}.\tilde{x})/x][(\lambda x_1\ldots x_{r+n}.\tilde{y})/y] \to^*_{\beta_v} \bar{P}_i \in Val$$

by Lemma 1, so the proof is immediate.

In the same manner $C^*[Q'] =_{\beta_v} \tilde{y}$.

(4) In such a case $m > n$, $p = q = 0$ and $s = \max\{r, m, n\}$. Let $P \equiv \lambda w_1\ldots w_{n_p}.wP_1\ldots P_{m_p} \in \beta$-nf $(m_p \le m)$ and $P' \equiv (\lambda u_1\ldots u_t.P)U_1...U_T \to^*_\beta xM_1\ldots M_m \equiv M$. Note that $m = m_p + T - (t + n_p)$.
Since $\forall i.\ U_i \in \text{Val}$, then $P' =_{\beta_v} xP''_1\ldots P''_{m_p}U_{t+n_p+1}...U_T \equiv P''$ where

$$P''_i \equiv P_i[U_1/u_1...U_t/u_t][U_{t+1}/w_1...U_{t+n_p}/w_{n_p}] =_\beta M_i$$

$(1 \le i \le m_p)$ and $U_j \equiv M_j$ $(t + n_p + 1 \le j \le T)$.

$$\begin{aligned}
&C^*[P'] =_{\beta_v} C^*[P'']\\
&\equiv (\lambda x.x\,\underbrace{P''_1\ldots P''_{m_p}U_{t+n_p+1}...U_T}_{m})O^{s+n}\ \underbrace{I...I}_{s+n-m}\ (\lambda x_1\ldots x_{m-n}.\tilde{x})\,\underbrace{\tilde{y}....\tilde{y}}_{m-n}\\
&=_{\beta_v} O^{s+n}\,\underbrace{P'''_1\ldots P'''_{m_p}U'_{t+n_p+1}...U'_T}_{m}\,\underbrace{I.....I}_{s+n-m}(\lambda x_1\ldots x_{m-n}.\tilde{x})\,\underbrace{\tilde{y}.....\tilde{y}}_{m-n} \equiv A
\end{aligned}$$

where $P'''_i \equiv P''_i[O^{s+n}/x]$ and $U'_j \equiv U_j[O^{s+n}/x]$. Since $FV(P_i) \subseteq \{u_1, ..., u_t\}$ by Lemma 1 $P'''_i =_{\beta_v} \bar{P}_i \in Val$. Thus $A =_{\beta_v} O^0(\lambda x_1\ldots x_{m-n}.\tilde{x})\,\underbrace{\tilde{y}.....\tilde{y}}_{m-n} =_{\beta_v} \tilde{x}$.

On the other hand,

$$C^*[Q'] =_{\beta_v} O^{s+n}N'_1\ldots N'_n\ \underbrace{I.....I}_{s+n-m}(\lambda x_1\ldots x_{m-n}.\tilde{x})\,\underbrace{\tilde{y}.....\tilde{y}}_{m-n} =_{\beta_v} \tilde{y}$$

for some valuable terms N'_i and $s + n + 1 = n + (s + n - m) + 1 + (m - n)$ is the number of terms postponed to O^{s+n}.

(5) Similar to previous case.

(6) In such a case $M_k, N_k \in \beta$-nf, $M_k, N_k \Rightarrow C[.]$ and $C^*[.] \equiv C[(\lambda x.[.])U_k^s \underbrace{I.....I}_{s-m}]$.

Let $P \equiv \lambda w_1 \ldots w_{n_p}.wP_1 \ldots P_{m_p} \in \beta$-nf $(m_p \leq m)$ and $P' \equiv (\lambda u_1 \ldots u_t.P)$ $U_1 \ldots U_T \to_\beta^* xM_1 \ldots M_m \equiv M$. Note that $m = m_p + T - (t + n_p)$.

Since $\forall i.\ U_i \in$ Val, then $P' =_{\beta_v} (\lambda w_1 \ldots w_{n_p}.wP'_1 \ldots P'_{m_p})U_{t+1} ... U_T$ where $P'_i \equiv P_i[U_1/u_1 ... U_t/u_t]$.

Moreover, $P' =_{\beta_v} xP''_1 \ldots P''_m \equiv P''$ where $P''_i \equiv P'_i[U_{t+1}/w_1 ... U_{t+n_p}/w_{n_p}] =_\beta M_i$ $(1 \leq i \leq m_p)$ and $P''_j \equiv U_j \equiv M_j$ $(t + n_p + 1 \leq j \leq T)$.

$P''_k =_{\beta_v} (\lambda u_1 \ldots u_t w_1 \ldots w_{n_p}.P_k)\ U_1 \ldots U_T =_{\beta_v} (\lambda x u_1 \ldots u_t w_1 \ldots w_{n_p}.P_k)$ $U_k^s U_1 ... U_T =_\beta M_k$ since $x \notin \mathrm{FV}(M_k)$, so by induction hypothesis $C[P''_k] =_{\beta_v} \tilde{x}$.

$$\begin{aligned}
C^*[P'] &=_{\beta_v} C[(\lambda x.((\lambda u_1 \ldots u_t.P)U_1 ... U_T))U_k^s \underbrace{I.....I}_{s-m}] \\
&=_{\beta_v} C[(\lambda x.xP''_1 \ldots P''_m)U_k^s \underbrace{I.....I}_{s-m}] \\
&=_{\beta_v} C[(\lambda x_1 \ldots x_s.x_k)P'''_1 \ldots P'''_m \underbrace{I.....I}_{s-m}] \\
&=_{\beta_v} C^*[P'''_k] =_{\beta_v} C[P''_k] =_{\beta_v} \tilde{x}.
\end{aligned}$$

where $P'''_i \equiv P''_i[U_k^s/x] \to_\beta^* \bar{P}_i \in$ Val by Lemma 1, since $FV(P_i) \subseteq \{u_1, ..., u_t\}$.

Case $C^*[Q']$ is very similar to the previous.

(7) $C^*[.] \equiv C[(\lambda x.[.])\pi^s \underbrace{I.....I}_{s-m} U_k^s]$, $M_k \not\simeq N_k$, $x \in \mathrm{FV}(M_k)$, $C'[.] \equiv (\lambda x.[.])\pi^s$ and $\tilde{M}_k, \tilde{N}_k \Rightarrow C[.]$.

Let $P \equiv \lambda w_1 \ldots w_{n_p}.wP_1 \ldots P_{m_p} \in \beta$-nf $(m_p \leq m)$ and $P' \equiv (\lambda u_1 \ldots u_t.P)$ $U_1 ... U_T \to_\beta^* xM_1 \ldots M_m \equiv M$. Note that $m = m_p + T - (t + n_p)$.

$P' =_{\beta_v} xP''_1 \ldots P''_m \equiv P''$ where $P''_i \equiv P_i[U_1/u_1 ... U_t/u_t][U_{t+1}/w_1 ... U_{t+n_p}/w_{n_p}]$ $=_\beta M_i$ $(1 \leq i \leq m_p)$ and $P''_j \equiv U_j \equiv M_j$ $(t + n_p + 1 \leq j \leq T)$.

Let $\bar{P}'''_k \equiv (\lambda x u_1 \ldots u_t w_1 \ldots w_{n_p}.P_k)\pi^s U_1 ... U_T =_\beta \tilde{M}_k$, so by induction hypothesis $C[\bar{P}'''_k] =_{\beta_v} \tilde{x}$.

$$\begin{aligned}
& C^*[P'] \equiv C[(\lambda x.((\lambda u_1 \ldots u_t.P)U_1 ... U_T))\pi^s \underbrace{I.....I}_{s-m} U_k^s] \\
& =_{\beta_v} C[(\lambda x.xP''_1 \ldots P''_m)\pi^s \underbrace{I.....I}_{s-m} U_k^s] \\
& =_{\beta_v} C[(\lambda x_1 \ldots x_s z.zx_1 \ldots x_s)P'''_1 \ldots P'''_m \underbrace{I.....I}_{s-m} U_k^s] \\
& =_{\beta_v} C[U_k^s P'''_1 \ldots P'''_m \underbrace{I.....I}_{s-m}] =_{\beta_v} C[P'''_k] =_{\beta_v} C[\bar{P}'''_k] =_{\beta_v} \tilde{x}
\end{aligned}$$

where $P'''_i \equiv P''_i[\pi^s/x] \to_\beta^* \bar{P}_i \in$ Val by Lemma 1, since $FV(P_i) \subseteq \{u_1, \ldots, u_t\}$.

Case $C^*[Q'] =_{\beta_v} \tilde{y}$ is similar. □

4 Computability

In order to show a representation of recursive functions, we need a numerical system; we will use the Barendregt system ([1]). Naïvely, $T \equiv \lambda xy.x$ denote *True* and $F \equiv \lambda xy.y$ denote *False*.

The pair of terms M and N is represented by $[M, N] \equiv \lambda z.zMN$.

Definition 10 (Barendregt Numerical System).
- **zero** $\bar{0} \equiv \lambda x.x$
- **successor** $\bar{s} \equiv \lambda x.[F, x] \equiv \lambda xz.zFx$
- **predecessor** $\bar{p} \equiv \lambda x.xF$
- **test for zero** $\delta \equiv \lambda x.xT$

Conventionally, by an overlined mathematical entity we will denote the corresponding λ-term; for instance, $\bar{1}$ denotes $\lambda z.zF(\lambda x.x)$

The Separability Theorem guarantees that different natural numbers are denoted with different λ-terms. Further, we use the reduction $\rightarrow_s$ defined in section 2, in order to check that the representation can be used for computational goal, using the operational machine presented by Plotkin in [7].

Definition 11. *A partial function $f : \mathbb{N}^n \rightarrow \mathbb{N}$ is v-definable if and only if there is a term $\bar{f} \in \Lambda^0$ such that:*

$$\bar{f}\bar{x}_1 \ldots \bar{x}_n \rightarrow_s^* \overline{f(x_1, \ldots, x_n)}$$

for all n-tuple of numerals $\bar{x}_1, \ldots, \bar{x}_n$ for which the function is defined, otherwise $\bar{f}\bar{x}_1 \ldots \bar{x}_n$ must be not valuable.

Our goal is to show that every recursive function is v-definable.

Let Ξ be the following recursion operator $(\lambda xf.f(\lambda z.xxfz))(\lambda xf.f(\lambda z.xxfz))$; note that this term is different from the original operator introduced by Plotkin in [7].

Theorem 2 (Recursion). *If $M \in V$ then $\Xi M \rightarrow_s^* M(\lambda z.\Xi Mz)$, where $z \notin FV(M)$.*

Proof. Let $A \equiv (\lambda xf.f(\lambda z.xxfz))$. Clearly

$$\Xi M \equiv (AA)M \rightarrow_s^* M(\lambda z.AAMz) \equiv M(\lambda z.\Xi Mz).$$

□

Note that Ξ is not a fixed point operator, in fact $\Xi M \rightarrow_s^* M(\lambda z.\Xi Mz)$, where $z \notin FV(M)$.
If Θ is a fixed point operator then $\Theta M \rightarrow_s^* M(\Theta M)$; but ΘM is neither a value nor valuable, thus it cannot be used in a β_v-reduction in order to obtain the recursion.

We start with primitive recursive functions, that are all total functions.

Lemma 7.

Zero *is v-definable:* $\bar z \equiv \lambda x.\bar 0$.
Successor is v-definable: $\bar s \equiv \lambda x.[F, x] \equiv \lambda xz.zFx$.
Projections are v-definable: $U^i_n \equiv \lambda x_1 \ldots x_n.x_i$.

Proof. Trivial. □

Lemma 8.
Let $h : \mathbb{N}^n \to \mathbb{N}$ *be a v-definable primitive recursive function and let* $g_1,\ldots,g_n : \mathbb{N}^m \to \mathbb{N}$ *be v-definable primitive recursive functions. The following function* f *is v-definable*

$$f(x_1,\ldots,x_m) = h(g_1(x_1,\ldots,x_m), \ldots.., g_n(x_1,\ldots,x_m)).$$

Proof. By hypothesis there exists the terms $\bar h$ and $\bar g_1,\ldots,\bar g_n$.
Let $\bar f \equiv \lambda x_1 \ldots x_m.\bar h(\bar{g_0} x_1 \ldots x_m)\ldots..(\bar{g_n} x_1 \ldots x_m)$. Thus, for all $\bar n_1,\ldots,\bar n_m$

$$\bar f \bar n_1 \ldots \bar n_m \to^*_s \overline{h(g_1(n_1,\ldots,n_m), \ldots.., g_n(n_1,\ldots,n_m))}.$$

□

Lemma 9. *Let* $h : \mathbb{N}^{m+2} \to \mathbb{N}$ *be a v-definable primitive recursive function and let* $g : \mathbb{N}^m \to \mathbb{N}$ *be v-definable primitive recursive functions. The following function* f *is v-definable*

$$f(k, x_1,\ldots,x_m) = \begin{cases} g(x_1,\ldots,x_m) & \textit{if } k = 0 \\ h(f(k-1, x_1,\ldots,x_m), k-1, x_1,\ldots,x_m) & \textit{otherwise.} \end{cases}$$

Proof. By induction on k. There exists terms $\bar h$ and $\bar g$, by hypothesis.
Let $M \equiv \lambda tzx_1 \ldots x_m.((\delta y)(\lambda y.\bar g x_1 \ldots x_m)(\lambda y.\bar h(t(\bar p y)x_1 \ldots x_m)(\bar p y)x_1 \ldots x_m))z$.

$k = 0$.

$$\begin{aligned}
&\Xi M \bar 0 \bar n_1 \ldots \bar n_m \to^*_s M(\lambda z.\Xi M z)\bar 0 \bar n_1 \ldots \bar n_m \to^*_s \\
&(\delta \bar 0)(\lambda y.\bar g \bar n_1 \ldots \bar n_m)(\lambda y.\bar h((\lambda z.\Xi M z)(\bar p y)\bar n_1 \ldots \bar n_m)(\bar p y)\bar n_1 \ldots \bar n_m))\bar 0 \to^*_s \\
&T(\lambda y.\bar g \bar n_1 \ldots \bar n_m)(\lambda y.\bar h((\lambda z.\Xi M z)(\bar p y)\bar n_1 \ldots \bar n_m)(\bar p y)\bar n_1 \ldots \bar n_m))\bar 0 \to^*_s \\
&(\lambda y.\bar g \bar n_1 \ldots \bar n_m)\bar 0 \to_s \bar g \bar n_1 \ldots \bar n_m \to^*_s \overline{f(0, n_1,\ldots,n_m)}.
\end{aligned}$$

$k > 0$.

$$\begin{aligned}
&\Xi M \bar k \bar n_1 \ldots \bar n_m \to^*_s M(\lambda z.\Xi M z)\bar k \bar n_1 \ldots \bar n_m \to^*_s \\
&(\delta \bar k)(\lambda y.\bar g \bar n_1 \ldots \bar n_m)(\lambda y.\bar h((\lambda z.\Xi M z)(\bar p y)\bar n_1 \ldots \bar n_m)(\bar p y)\bar n_1 \ldots \bar n_m)\bar k \to^*_s \\
&F(\lambda y.\bar g \bar n_1 \ldots \bar n_m)(\lambda y.\bar h((\lambda z.\Xi M z)(\bar p y)\bar n_1 \ldots \bar n_m)(\bar p y)\bar n_1 \ldots \bar n_m)\bar k \to^*_s \\
&\bar h((\lambda z.\Xi M z)(\bar p \bar k)\bar n_1 \ldots \bar n_m)(\bar p \bar k)\bar n_1 \ldots \bar n_m) \to^*_s \\
&\bar h(\Xi M(\bar p \bar k)\bar n_1 \ldots \bar n_m)(\bar p \bar k)\bar n_1 \ldots \bar n_m) \to^*_s \\
&\bar h \overline{f(k-1, n_1,\ldots,n_m)}\ \overline{k-1}\ \bar n_1,\ldots,\bar n_m \to^*_s \\
&\overline{h(f(k-1, n_1,\ldots,n_m), k-1, n_1,\ldots,n_m)}
\end{aligned}$$

since by induction

$$(\Xi M(\bar{p}\bar{k})\bar{n}_1 \ldots \bar{n}_m) \to_s^* \overline{f(k-1, n_1, \ldots, n_m)}.$$

□

Now we prove that every recursive function is v-definable. As usual $f(x) \uparrow$ and $f(x) = \perp$ mean that f is undefined in x, while $f(x) \downarrow$ and $f(x) \neq \perp$ mean that f is defined in x. If f is a m-ary recursive function, we must to check that $\bar{f}\bar{x}_1 \ldots \bar{x}_m \to_s \overline{f(x_1, \ldots, x_m)} \in Val$ if and only if $f(x_1, \ldots, x_m)$ is defined. Actually, $\bar{f}\bar{x}_1 \ldots \bar{x}_m$ is a v-unsolvable term (see [5,6]) if and only if $f(x_1, \ldots, x_m)$ is not defined.

Lemma 10. *Let $h : \mathbb{N}^n \to \mathbb{N}$ be a v-definable recursive function and let $g_1, \ldots, g_n : \mathbb{N}^m \to \mathbb{N}$ be v-definable recursive functions. The following function is v-definable*

$$f(x_1, \ldots, x_m) = h(g_1(x_1, \ldots, x_m), \ldots\ldots, g_n(x_1, \ldots, x_m)).$$

Proof. f is undefined if and only if there is a function between $h, g_1, ..., g_n$ undefined on its arguments, thus the proof follows by hypothesis that $h, g_1, ..., g_n$ are v-definable recursive functions. □

Lemma 11. *Let $h : \mathbb{N}^{m+2} \to \mathbb{N}$ and $g : \mathbb{N}^m \to \mathbb{N}$ be a v-definable recursive functions.*
The following function f is v-definable

$$f(k, x_1, \ldots, x_m) = \begin{cases} g(x_1, \ldots, x_m) & \textit{if } k = 0 \\ h(f(k-1, x_1, \ldots, x_m), k-1, x_1, \ldots, x_m) & \textit{otherwise.} \end{cases}$$

Proof. $f(k, x_1, \ldots, x_m) \downarrow$ if and only if $\bar{f}(\bar{k}, \bar{x}_1, \ldots, \bar{x}_m) \to_s^* P \in Val$. In fact $f(k, x_1, \ldots, x_m) \uparrow$ if and only if, in the computation $\bar{g}$ or $\bar{h}$ are not valuable on some argument. □

Finally, we check the v-definability of minimalisation function.

Let h be a binary recursive and total function and let $x \in \mathbb{N}$. The minimalisation function $f : \mathbb{N} \to \mathbb{N}$ associated to $h : \mathbb{N}^2 \to \mathbb{N}$ is defined as

$$f(x) = \mu y[h(x, y)] = \begin{cases} min\{k \in \mathbb{N} \mid h(x, k) = 0\} & \text{if a such } k \in \mathbb{N} \text{ exists} \\ \perp & \text{otherwise.} \end{cases}$$

Let h be a total recursive function v-defined by $\bar{h}$, $M \equiv \lambda t.\lambda hxy.((\bar{\delta}(hxy))I$ $(\lambda y.\bar{t}(\bar{h}x(\bar{s}y))y$ and $n \in \mathbb{N}$. We want to check that:

1. If $f(n) \downarrow$ then $(\Xi M)\bar{h}\bar{n}\bar{0} \to_s^* \overline{f(x)}$.
2. If $f(n) \uparrow$ then $(\Xi M)\bar{h}\bar{n}\bar{0} \to_s^* P\bar{k}$, for all numerals $\bar{k}$, for some $P \in \Lambda$.

Lemma 12.

1. *If $h(n, k) = 0$ then $(\Xi M)\bar{h}\bar{n}\bar{k} \to_s^* \bar{k}$.*

2. If $h(n,k) \neq 0$ *then* $(\Xi M)\bar{h}\bar{n}\bar{k} \to_s^* (\Xi M)\bar{h}\bar{n}(\bar{s}\bar{k})$.

Proof. By induction on k. □

Lemma 13. *Let* $h : \mathbb{N}^2 \to \mathbb{N}$ *be a v-definable total recursive function. If* $f(x) = \mu y[h(x,y) = 0]$ *is defined for* $x = n$ *then* $(\Xi M)\bar{h}\bar{n}\bar{0} \to_s^* \overline{f(x)}$ *where*

$$M \equiv \lambda thxy.(\bar{\delta}(hxy))I(\lambda y.\bar{t}(\bar{h}x(\bar{s}y))y)$$

Proof. Let $f(n) = k$, thus k the minimum number such that $h(n,k) = 0$. By Lemma 12.2 we have

$$(\Xi M)\bar{h}\bar{n}\bar{0} \to_s^* (\Xi M)\bar{h}\bar{n}\bar{k}$$

and by Lemma 12.1 we conclude. □

Lemma 14. *Let* $h : \mathbb{N}^2 \to \mathbb{N}$ *be a v-definable total recursive function. If* $f(x) = \mu y[h(x,y) = 0]$ *for* $x = n$ *is always different from zero then* $\forall \bar{k} \quad (\Xi M)\bar{h}\bar{n}\bar{0} \to_s^* P\bar{k}$*, for some* $P \in \Lambda$.

Proof. By Lemma 12.2 . □

Lemma 15.
Let $h : \mathbb{N}^2 \to \mathbb{N}$ *be a v-definable total recursive function. The following function is v-definable:*

$$f(x) = \mu y[h(x,y) = 0].$$

Proof. Trivial, by using previous lemmas. □

5 Acknowledgements

The author wishes to express his gratitude to Simona Ronchi Della Rocca for her helpful suggestions, comments and stimulating conversations, during the preparation of this paper.

References

1. H. Barendregt, *The Lambda Calculus: its syntax and semantics, North Holland, 1984.* 75, 76, 85
2. C. Böhm, *Alcune proprietà delle forme* $\beta\eta$*-normali* λK*-calcolo, Pubblicazioni dell' istituto per le applicazioni del calcolo, n.696, 1968.* 75
3. M. Hyland, *A syntactic characterization of the equality in some models of the Lambda calculus, Journal of London Mathematical Society, 1976, pp. 361-370.* 79
4. P. J. Landin, *The mechanical evaluation of expressions, Computer Journal, 1964.* 74
5. L. Paolini, S. Ronchi della Rocca, *Call-by-value solvability, Theoretical Informatics and Applications, n.33, 1999, pp. 507-534.* 75, 77, 78, 87
6. L. Paolini, *La chiamata per valore e la valutazione pigra nel lambda calcolo, Tesi di Laurea, Università degli Studi di Torino, Dip. Informatica, 1998.* 75, 77, 78, 87

7. G. Plotkin, *Call-by-Value, Call-by-Name and the λ-calculus*, *Theoretical Computer Science, 1975, pp. 125-159.* 74, 75, 78, 85
8. S. Ronchi Della Rocca, *Notes for the Summer School "Proof and Types"*, *EDP Sciences, 1993.* 75, 76
9. C. P. Wadsworth, *The relation between computational and denotational properties for Scott D_∞-models of the lambda-calculus*, *SIAM Journal of computing, vol.5, 1976, pp. 488-522.* 75

Job Shop Scheduling with Unit Length Tasks: Bounds and Algorithms

Juraj Hromkovič[1], Kathleen Steinhöfel[2], and Peter Widmayer[3]

[1] Lehrstuhl für Informatik I, RWTH Aachen
Ahornstraße 55, 52074 Aachen, Germany
[2] GMD – Forschungszentrum Informationstechnik GmbH
Kekuléstraße 7, 12489 Berlin, Germany
[3] Institut für Theoretische Informatik, ETH Zürich
Clausiusstraße 49, 8092 Zürich, Switzerland

Abstract. We consider the job shop scheduling problem *unit*–J_m, where each job is processed once on each of m given machines. The execution of any task on its corresponding machine takes exactly one time unit. The objective is to minimize the overall completion time, called makespan. The contribution of this paper are the following results: (i) For any input instance of *unit*–$J_m$with d jobs, the makespan of an optimum schedule is at most $m+o(m)$, for $d = o(m^{1/2})$. For $d = o(m^{1/2})$, this improves on the upper bound $O(m+d)$ given in [LMR99] with a constant equal to two as shown in [S98]. For $d = 2$ the upper bound is improved to $m + \lceil \sqrt{m} \rceil$. (ii) There exist input instances of *unit*$-J_m$with $d = 2$ such that the makespan of an optimum schedule is at least $m + \lceil \sqrt{m} \rceil$, i.e., the result (i) cannot be improved for $d = 2$. (iii) We present a randomized on-line approximation algorithm for *unit*–$J_m$with the best known approximation ratio for $d = o(m^{1/2})$. (iv) A deterministic approximation algorithm for *unit*–$J_m$is described that works in quadratic time for constant d and has an approximation ratio of $1 + 2^d/\lfloor \sqrt{m} \rfloor$ for $d \leq 2 \log_2 m$.

1 Introduction

Minimizing the makespan for general job shop scheduling is one of the fundamental optimization problems. It is NP-hard, and Williamson et al. [WHH97] proved that the minimum makespan is not even approximable in polynomial time within $5/4 - \varepsilon$ for any ε. Moreover, no constant approximation algorithm is known, see Goldberg et al. [GPSS97] and Shmoys et al. [SSW94].

Many job shop scheduling models have been identified as having a number of practical applications. But even severely restricted models remain strongly NP-hard. In this paper, we consider a problem setting that relates to finding optimum schedules for routing packets through a network, see [LMR99]. It is a well-studied version of job shop scheduling with m different machines and unit length tasks, denoted by *unit*–J_m. There are d jobs $J_1, J_2, \ldots, J_d$ for some integer $d \geq 2$. Each job consists of a sequence of m tasks, such that each machine processes exactly one task of the job. Therefore, for each job the order of the tasks

A. Restivo, S. Ronchi Della Rocca, L. Roversi (Eds.): ICTCS 2001, LNCS 2202, pp. 90–106, 2001.

$\sigma_1, \sigma_2, \ldots, \sigma_m$ determines a permutation of the m machines, where σ_i requires processing on the i-th machine. As in the general job shop, each machine can process only one task at a time and each job must be executed on the machines in the order given by its permutation. A feasible schedule is an assignment of starting times to tasks that satisfies all stated restrictions. The makespan of a schedule is the maximum over the completion times of all jobs. The objective is to minimize the makespan over all feasible schedules. The problem *unit*–$J_m$is NP-hard for $m \geq 3$, see Lenstra and Rinnooy Kan [LR79].

The algorithm of Goldberg et al. [GPSS97] improved a result of Shmoys et al. [SSW94] and provides an approximation ratio $O((\log_2 m)/(\log_2 \log_2 m)^2)$ for *unit*–J_m. Instances with two jobs have been shown by Brucker [B88] to be solvable in linear time. Later, we shall see that a straightforward extension of this algorithm leads to an $O(m^d)$ time algorithm for any input instance of *unit*–$J_m$with d jobs. Leighton et al. [LMR99] proved that there exists always a schedule with makespan $O(m + d)$. This provides a randomized constant approximation algorithm for this problem. The constant is equal to two and was determined by Scheideler [S98]. Feige and Scheideler [FS98] proved that the bound does not extend to the case of arbitrary task lengths.

In this paper, we analyze the hardest input instances of *unit*–J_m. As already mentioned, finding the optimal makespan of job shop instances with two jobs is solveable in linear time. Therefore, in this paper, the term hard instance is used in the sense of makespan length only. Our observations lead to the design of a randomized on-line algorithm that solves *unit*–$J_m$with d jobs in linear time with expected approximation ratio that tends to 1 for $d = o(m^{1/2})$. The contributions of this paper can be formulated as follows.

1. The makespan of an optimum schedule is at most
$$m + 2d\sqrt{m};$$
this amounts to $m + o(m)$ for every problem instance of *unit*–$J_m$with $d = o(m^{1/2})$ jobs, and thus for this case improves on the upper bound $O(m + d)$ derived by Leighton et al. [LMR99]. For $d = 2$ we prove the stronger upper bound $m + \lceil\sqrt{m}\rceil$.
2. There exist input instances of *unit*–$J_m$with two jobs such that every schedule has a makespan of at least
$$m + \sqrt{m}.$$
Hence, the result (i) cannot be improved for $d = 2$.
3. For every positive integer m, there is a randomized on-line approximation algorithm that solves *unit*–J_m in linear time with an expected approximation ratio of
$$1 + \frac{2d}{\sqrt{m}};$$
this amounts to $1 + o(1)$ for $d = o(m^{1/2})$. These results demonstrate an extreme power of randomness for *unit*–J_m for several reasons. First of all our randomized on-line algorithm is competitive with respect to the makespan

of an optimum solution. For $d = o(m^{1/2})$ the algorithm is the best approximation algorithm for *unit–*J_m. We do not know any off-line polynomial-time approximation algorithm with an approximation ratio that would tend to 1 for $d = o(m^{1/2})$ with growing m. Moreover, no deterministic on-line algorithm can achieve a makespan better than $d \cdot (m-1)/\log_2 d$ [LMR99].

4. We present a deterministic approximation algorithm that is efficient at least for small d's in comparison with m. Its run-time is $O(d^2m^2)$, and it has an approximation ratio of at most
$$1 + \frac{2^d}{\lfloor\sqrt{m}\rfloor}$$
which tends to 1 with growing m for $d = o(\log_2 m)$.

The paper is organized as follows. Section 2 presents a geometrical representation of the input instances of *unit–*J_m that is essential for a transparent analysis of *unit–*J_m. In Section 3 we present some hard input instances with two jobs only. Section 4 shows the existence of efficient schedules for all input instances of *unit–*J_m. In Section 5 the randomized algorithm with the properties as described in (iii) is given. Our deterministic approximation algorithm is presented in Section 6.

2 A Geometrical Representation of Instances

We start with the representation of input instances with two jobs that was employed in [B88] to design a linear time algorithm for this special case of *unit–*J_m.

Let $(i_1, \ldots, i_m)$ and $(j_1, \ldots, j_m)$ be two permutations of $(1, 2, \ldots, m)$ that represent the input instance $(\sigma_{i_1}, \sigma_{i_2}, \ldots, \sigma_{i_m})$, $(\sigma_{j_1}, \sigma_{j_2}, \ldots, \sigma_{j_m})$ of *unit–*J_m. We consider a grid G_m of size $m \times m$, where for all $k, l \in \{1, \ldots, m\}$ the k-th row of G_m is labeled by j_k and the l-th column of G_m by i_l. A pair (k, l), i.e., the intersection of the k-th row and the l-th column, is called an *obstacle*, if and only if $i_l = j_k$. The corresponding square is depicted by a black box.

Fig. 1a illustrates the G_9 of the input instance with two jobs that are given by the two permutations $(1, 2, 3, 4, 5, 6, 7, 8, 9)$ and $(1, 3, 2, 6, 5, 4, 8, 7, 9)$. The term obstacle is motivated by the following observation. Assume that the first job has executed its first $l-1$ tasks and the second job its first $k-1$ tasks. If $i_l = j_k$, then both tasks σ_{i_l} and σ_{j_k} require the same machine and therefore, only one of the two jobs can continue its execution in the next time unit and the other one is *delayed*. Otherwise, both jobs can proceed simultaneously.

We assign to the grid G_m the Graph(G_m)=(V, E), where V consists of all vertices of the grid and the set E includes all orthogonal edges of the grid. Additionally, E contains diagonal edges that connect the upper-left corner with the lower-right corner of a grid square that is not an obstacle. Fig. 1b shows the corresponding Graph(G_9) of G_9 given in Fig. 1a. Any feasible schedule is represented by a path from the upper-left corner of G_9 to the lower-right corner of G_9. The path consists of edges of Graph(G_9), where each edge represents one unit of time. A vertical grid edge indicates that in this time unit, a task of the first job is *delayed*; a horizontal grid edge indicates a *delay* of a task of the second job; a diagonal edge tells that both jobs are processed at the same time with no delay.

An optimum schedule corresponds to a shortest path from the upper-left corner a to the lower-right corner b. The bold polygonal line in Fig. 1 represents an optimum schedule of our example. In the schedule, there are 6 delays that are equally distributed between the two jobs. Therefore, the makespan of the illustrated schedule is $m + 6/2 = 9 + 3 = 12$.

Let S be a schedule of an instance with two jobs. The number of vertical edges of the path representing S is called the *delay of the first job according to* S, and the number of horizontal edges of S is called the *delay of the second job according to* S. The *delay of* $\boldsymbol{S}$ is the maximum over these two delays. Obviously, the makespan of S is exactly the sum of m and the delay of S. For later use, we denote by *sum-delay(*$\boldsymbol{S}$*)* the sum of the delays of jobs according to S.

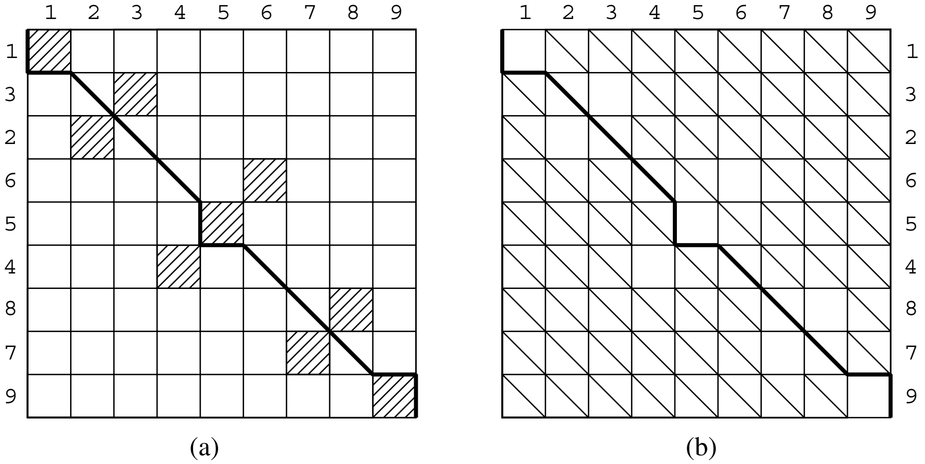


Fig. 1. An hard input instance of *unit–*$J_m$with two jobs and nine machines

We outline the extension of this representation for an arbitrary number d of jobs. In this case we have a d-dimensional grid G_m^d that contains m^d d-dimensional grid cells. Again, the unit intervals of each axis are labeled by the tasks according to the sequence of machines of the corresponding job. A label i of some axis determines a $(d-1)$-dimensional subgrid of G_m^d.

The intersection of two such different subgrids with labels i is a $(d-2)$-dimensional subgrid of m^{d-2} grid hypercubes that are obstacles in the following sense. If d' and d'' are the dimensions in which the common label i defining the obstacles occurs, then any diagonal of a grid square Q in the intersection is forbidden whose projection on dimensions d' and d'' is a diagonal (and all others are allowed w.r.t. to this intersection). In particular, the main diagonal of such a square Q (that corresponds to the execution of all tasks determined by the coordinates of this grid square Q) is forbidden, and so are the diagonals of the surface of Q that correspond to the intersection. For instance, if Q is part of the intersection of q $(d-1)$-dimensional subgrids determined by the same task σ_i and q different axes, then to go from the "lowest" corner of Q to the opposite corner of Q requires at least q time units: Since in this case q tasks request the same machine, this congestion can be resolved by q subsequent steps only. Fig. 2 gives an exsample of such an obstacle in the 3-dimensional case.

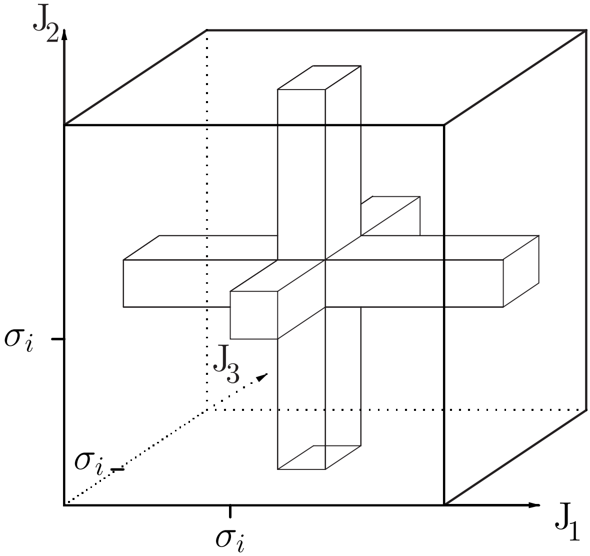


Fig. 2. An obstacle in the 3-dimensional case

Again, any optimum schedule corresponds to a shortest path between the two extreme corners of the grid. Therefore, for any constant d we get a polynomial-time algorithm for input instances with d jobs. The notions *delay of S* and *sum-delay(S)* can be extended for $d > 0$ jobs in a straightforward way.

3 Some Hard Instances

The aim of this section is to construct some of the hardest problem instances with two jobs, i.e., instances where the optimum schedule has a maximum number of delays. Let *makespan(I)* denote the length of an optimum schedule for the problem instance I in what follows.

Lemma 1. *For every $m = \binom{k+1}{2}, k$ a positive integer, there exists an input instance I_R of two job unit–J_m such that*

$$makespan(I_R) \geq m + \sqrt{\frac{m}{2}} - \frac{1}{2}.$$

Proof: Let $I_R^m = (J_1, J_2)$, where

$$J_1 = w_1, w_2, \ \ldots \ , w_k \ , \text{ and } J_2 = w_1^R, w_2^R, \ \ldots \ , w_k^R,$$

with w_i denoting a subsequence of tasks (represented by integers for the respective machine numbers), and w_i^R denoting the reverse of w_i.
The subsequences w_i and w_i^R, with $i = 1, \ \ldots \ , k$, are defined as

$$w_i = [\binom{i}{2} + 1, \binom{i}{2} + 2, \ \ldots \ , \binom{i}{2} + i - 1, \binom{i}{2} + i] \ , \text{ and hence}$$

$$w_i^R = [\binom{i}{2} + i, \binom{i}{2} + i - 1, \ \ldots \ , \binom{i}{2} + 2, \binom{i}{2} + 1].$$

Observe that w_i is a sequence of i integers, for $i = 1, \ \ldots \ , k$, and that $J_1 = 1, 2, \ \ldots \ , m$. An example for $m = 10 = \binom{5}{2}$ is

$$J_1 = [1], [2, 3], [4, 5, 6], [7, 8, 9, 10], \text{ and } \qquad J_2 = [1], [3, 2], [6, 5, 4], [10, 9, 8, 7].$$

In the full proof (in the Appendix for the convenience of the reader) we show that every schedule on I_R^m contains at least k delays, i.e., every shortest path contains at least k orthogonal grid edges. Note that this is sufficient to prove our result because it implies that at least one of the jobs J_1 and J_2 is delayed by at least $k/2 \geq \sqrt{m}/\sqrt{2} - 1/2$ time units and therefore, the makespan must be at least $m + k/2$. □

Consider an input instance $I_{\overline{R}}^m = (\pi_1, \pi_2)$, for $m = k^2, k$ a positive integer, where

$$\pi_1 = w_1, w_2, \ldots, w_k, u_{k-1}, u_{k-2}, \ldots, u_1,$$
$$\pi_2 = w_1^R, w_2^R, \ldots, w_k^R, u_{k-1}^R, u_{k-2}^R, \ldots, u_1^R,$$

where the w_i have the same meaning as before, and u_l is a sequence of l tasks for $l = 1, \ \ldots \ , k - 1$, with u_l^R denoting the reverse of u_l . The example of $I_{\overline{R}}^9$ for $\pi_1 = 1, 2, \ldots, m$ is given in Fig. 1. An extension of the analysis presented in Lemma 1 leads to the following result.

Lemma 2. *For every $m = k^2$, k a positive integer,*

$$makespan(I_{\frac{m}{R}}) \geq m + \sqrt{m} = m + k.$$

Proof: To prove the Lemma we show (in the Appendix, for the convenience of the reader) that every shortest path between the two opposite corners of the grid contains at least $2 \cdot k$ orthogonal grid edges; this implies that at least one of the two jobs is delayed by at least $k = \sqrt{m}$ time units and therefore, the makespan must be at least $m + \sqrt{m}$. □

4 Upper Bounds on the Number of Delays

In this section, we show that any input instance of *unit–J_m* can be scheduled with $2 \cdot m^{1-\varepsilon}$ delays for $d \leq m^{1/2-\varepsilon}$, as compared with the lower bound on the makespan. This improves on the upper bound $O(m + d)$ [LMR99] for $d = o(m)$.

First, we give the upper bound for two jobs. Note that this upper bound meets the lower bound of Lemma 2.

Lemma 3. *For every positive integer m, any two job problem instance I of unit–J_m satisfies*

$$makespan(I) \leq m + \lceil \sqrt{m} \rceil.$$

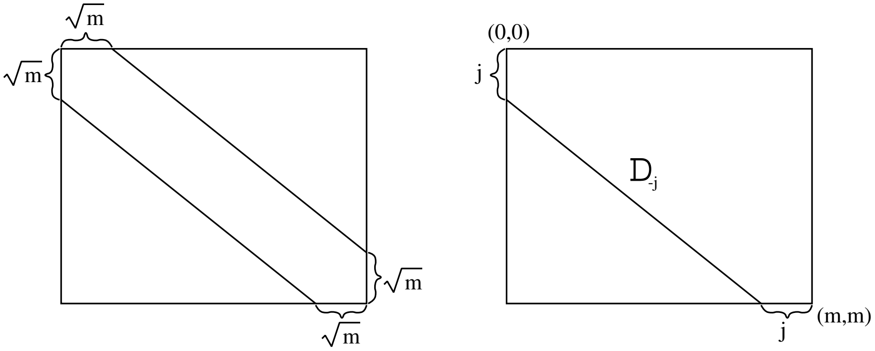


Fig. 3. The considered diagonals of G_m

Proof: For simplicity we present the proof for the case $m = k^2$ only. To do this we use the geometric representation. In what follows for $i = 0, 1, \ldots, \sqrt{m}$, the diagonal D_i of the grid G_m is the diagonal going from the position $(0, i)$ to the position $(m-i, m)$; similarly, diagonal D_{-i} goes from $(i, 0)$ to $(m, m-i)$, see Fig. 3.

For each $i \in \{-\sqrt{m}, \ \ldots \ , 0, \ \ldots \ , \sqrt{m}\}$, we associate a schedule $S(D_i)$ to diagonal D_i. The schedule $S(D_i)$ uses first i orthogonal grid edges to reach the beginning of the diagonal D_i, then it runs via this diagonal and avoids each obstacle on this diagonal by one horizontal move and one vertical move. Finally, it uses i grid edges on the border of G_m in order to reach (m, m). Observe that the makespan of this schedule is exactly

$$m + i + \text{ the number of obstacles at } D_i$$

because the length of D_i is $m - i$ and the schedule uses $2 \cdot i$ steps to reach and to leave this diagonal. Therefore, the delay of the schedule $S(D_i)$ is $i+$ the number of obstacles at D_i. The sum of all delays over all $2\sqrt{m}+1$ considered schedules D_i is at most

$$m + \sum_{i=-\sqrt{m}}^{\sqrt{m}} |i| = m + 2 \cdot \sum_{i=1}^{\sqrt{m}} i = m + \sqrt{m} \cdot (\sqrt{m} + 1)$$

because the number of all obstacles in the whole G_m is exactly m, the number of machines[1]. Since the average delay over all $2 \cdot \sqrt{m} + 1$ considered schedules is

$$\frac{m + \sqrt{m} \cdot (\sqrt{m} + 1)}{2 \cdot \sqrt{m} + 1} \leq \sqrt{m} + \frac{1}{2},$$

there must exist a schedule that has delay at most $\sqrt{m}$. □

Now, we extend Lemma 3 to all input instances, i.e., any number of jobs.

Theorem 1. *For every positive integer m, and every instance I of unit–J_m with $d = o(m^{1/2})$ jobs, the length of any optimum schedule can be bounded from above by*

$$makespan(I) \leq m + 2d\sqrt{m} = m + o(m).$$

Proof: The idea of the proof is to generalize the case with $d = 2$ to any dimension. We can view the d-dimensional $m \times m \times \ldots \times m$ grid $G_{m,d}(I)$ as a subgrid of an infinite d-dimensional grid. We consider the following set $\mathcal{D}$ of diagonals that are parallel to the main diagonal of $G_{m,d}(I)$ that starts in the point $(0, 0, \ldots, 0)$ and ends in $(m, m, \ldots, m)$: We take every diagonal with a starting point $(i_1, i_2, \ldots, i_d)$, where there is exactly one $j \in \{1, \ldots, d\}$ such that $i_j = 0$, and $0 > i_b \geq -r$, for $b \in \{1, \ldots, d\} - \{j\}$ and some $m \geq r > 0$. Let $D(i_1, i_2, \ldots, i_d)$ denote the diagonal starting in $(i_1, i_2, \ldots, i_d)$ that ends in the point $(i_1{+}m{+}a, i_2{+}m{+}a, \ldots, i_d{+}m{+}a)$, where $a{=}\max\{|i_c|\,|\,c{\in}\{1, \ldots, d\}\}{\leq}r$. Every diagonal $D(i_1, i_2, \ldots, i_d)$ corresponds to a job schedule where the $j-th$

[1] Therefore, in the worst case, all obstacles of G_m lie on the $2k+1$ diagonals, see Fig. 3.

job is postponed by i_j time units with respect to jobs starting with the delay 0. If this schedule reaches the final point $(i_1+m+a, i_2+m+a, \dots, i_d+m+a)$ then all jobs were completely executed because $i_j+m+a \geq m$ for all $j \in \{1, \dots, d\}$.

Obviously, the number of all such diagonals is exactly

$$d \cdot r^{d-1}. \tag{1}$$

Note, that one could consider also diagonals with starting points containing several 0 elements, but this makes the calculation more complex and the achieved gain is negligible.

Similarly, as in the 2-dimensional case we calculate an upper bound on the total delay of all $d \cdot r^{d-1}$ schedules. This bound can be obtained as the sum of an upper bound on the sum of the lengths of all diagonals and of an upper bound on the number of all delays occurring on these diagonals.

Because the starting points of all diagonals in $\mathcal{D}$ lie on the bounding surface of the grid, translated by m diagonally, and because at the end at most r extra diagonal steps are added, the length of each described diagonal is bounded from above by $m+r$. Because of (1) the sum of the lengths of all diagonals is at most

$$d \cdot r^{d-1} \cdot (m+2r). \tag{2}$$

Now, we count the number of possible delays. The d axes of the subgrid $G_{m,d}(I)$ are labeled by the d jobs. A label σ_i on an axis determines a $(d-1)$-dimensional subgrid of $G_{m,d}(I)$ of m^{d-1} d-dimensional unit grid cubes. An intersection of two such subgrids determined by the same label σ_i on two different axes is a $(d-2)$-dimensional subgrid of m^{d-2} d-dimensional unit grid cubes. Observe that the inner diagonal of any unit grid cube induced by this intersection subgrid as well as the corresponding diagonal on the surface of this unit grid cube are forbidden, for any schedule. Therefore, any of our diagonal schedules containing such a unit grid cube will get a delay. Obviously, if q $(d-1)$-dimensional subgrids labeled by σ_i meet in one unit grid cube, the diagonal schedule containing such a grid cube must use $q-1$ additional steps to avoid this obstacle.

We calculate the total number of delays as the sum of the number of delays caused by pairs of $(d-1)$-dimensional subgrids with the same label. We start with the following technical fact (whose proof is in the Appendix for the convenience of the reader):

Fact 1 *The intersection of every pair of $(d-1)$-dimensional subgrids determined by the same task σ affects at most*

$$(d-1) \cdot r^{d-2}$$

diagonals of $\mathcal{D}$, each of them in exactly one unit grid cube.

Since we have m tasks in each of the d jobs and $\binom{d}{2}$ pairs of axes (jobs), the number of schedule delays on all $d \cdot r^{d-1}$ diagonals is at most

$$m \cdot \binom{d}{2} \cdot (d-1) \cdot r^{d-2}. \tag{3}$$

Therefore, the average number of delays per diagonal is at most

$$\frac{m \cdot \frac{d \cdot (d-1)}{2} \cdot (d-1) \cdot r^{d-2}}{d \cdot r^{d-1}} \leq \frac{m \cdot (d-1)^2}{2 \cdot r}.$$

Since the length of every diagonal is bounded by $m+2r$, the average makespan over all diagonal strategies in $\mathcal{D}$ is bounded by

$$m + 2r + \frac{m(d-1)^2}{2r} \leq m + 2r + \frac{md^2}{2r}. \tag{4}$$

Choosing $r = d\sqrt{m}/2$ we obtain an average makespan over our dr^{d-1} diagonal strategies of at most

$$m + 2d\sqrt{m}.$$

Thus, there must exist at least one diagonal strategy with a makespan of at most $m + 2d\sqrt{m} = m + o(m)$ for $d = o(m^{1/2})$.

□

Corollary 1 *For every positive integer m and every instance I of unit–J_m with $d \leq m^{1/2-\varepsilon}$ jobs, with $0 < \varepsilon \leq 1/2$, the makespan of any optimal schedule can be bounded from above by*

$$makespan(I) \leq m + 2m^{1-\varepsilon}.$$

Proof: We choose

$$r = \lfloor \frac{1}{2} m^{1-\varepsilon} \rfloor,$$

and insert it into (4). Then we have

$$m + 2\lfloor \frac{1}{2} m^{1-\varepsilon} \rfloor + \frac{m(d-1)^2}{2\lfloor \frac{1}{2} m^{1-\varepsilon} \rfloor} \leq m + 2m^{1-\varepsilon}.$$

□

Since the best known upper bound on the makespan is $2(m+d) \geq 2m$, our upper bound is an improvement for $d = o(m)$.

5 A Randomized On-Line Approximation Algorithm

We propose the following randomized on-line algorithm for *unit–J_m*.

Algorithm $\boldsymbol{OLR_m}$

Input: The number of jobs d and the number of machines m are known initially and $d = o(m^{1/2})$. The tasks of the jobs are presented one by one, within each job in the order of their occurrence, and in arbitrary order across the jobs.

Step 1: Choose uniformly a diagonal D at random from $\mathcal{D}$, i.e., generate the start coordinates of a diagonal from $\mathcal{D}$ at random by following Theorem 1.

Step 2: Apply the schedule determined by Step 1 by avoiding the obstacles as they appear.

Theorem 2. *The randomized on-line algorithm OLR_m for unit–J_m*

1. *has an expected competitive ratio of at most $1 + 2d/\sqrt{m}$, that is, $1 + o(1)$ if $d = o(m^{1/2})$, and*
2. *runs in linear time.*

Proof: First we prove (ii). We have an input of length $m \cdot d$. A number $d \cdot \lceil \log_2 (d\sqrt{m}/2) \rceil$ of random bits is sufficient to determine a diagonal and therefore, Step 1 can be executed in linear time. It is straightforward to follow a given path for actual jobs (using diagonals whenever possible) in linear time.

Now, we prove (i). Since the average makespan over all schedules determined by the diagonals from $\mathcal{D}$ is at most $m + 2d\sqrt{m}$, and the optimum makespan is at least m, the expected approximation ratio of OLR is at most

$$\frac{m + 2d\sqrt{m}}{m} = 1 + \frac{2d}{\sqrt{m}}$$

□

Therefore, OLR is $(1 + 2d/\sqrt{m})$ competitive w.r.t. optimum schedules. Note that no (randomized) polynomial-time algorithm with an approximation ratio tending to 1 for $d = o(m^{1/2})$ with growing m has been known before. For $d \leq m^{1/2-\varepsilon}$ our algorithm is better than the 2-approximation algorithm of Leighton et al. [LMR99]. Moreover, OLR_m shows nicely the power of randomization, because every deterministic on-line algorithm for *unit–J_m* has its competitive ratio at least $\Omega\big(d \,/\, \log_2 d\big)$ [LMR99].

6 A Deterministic Approximation Algorithm

As we already observed our grid representation provides an $O(m^d)$ algorithm for input instances with m machines and d jobs. The complexity of this algorithm is too large even for constant d's and it is not polynomial for d growing with m. The aim of this section is to present an efficient approximation algorithm at least for small d in comparison with m.

The idea is again to find a diagonal strategy, but in a deterministic way by looking on the $\binom{d}{2}$ 2-dimensional surfaces of $G_{m,d}(I)$ only. Remember that fixing a diagonal strategy is nothing else than fixing the relative delays between all pairs of jobs.

*Algorithm SURFACE(**I**)*

Input: $I = (J_1, J_2, \ldots, J_d)$, where J_i is the i^{th} job, i.e., a permutation of $(1, 2, \ldots, m)$, and $d \leq 1/2 \quad \log_2 m$.

Step 1: If $d = 2$ take the best diagonal strategy from the $2\sqrt{m} + 1$ diagonal strategies with the relative delay between J_1 and J_2 bounded by $\sqrt{m}$. If $d > 2$, then apply SURFACE$(J_1, J_2, \ldots, J_{d-1})$ in order to find a diagonal strategy D for $(J_1, J_2, \ldots, J_{d-1})$, that contains at most $2^{d-1}\sqrt{m}$ delays and for every $j \in 2, \ldots, d-1$ the relative delay between J_1 and J_j is at most $\sqrt{m}$. (Observe, that D fixes the delay between any two of the first $d-1$ jobs.)

Step 2: Fix consecutively the relative delays between J_d and the jobs J_1, J_2, $J_3, \ldots, J_{d-1}$ in the following way:

(2.1) Set S_1 as the set of the best[1] $\lfloor\sqrt{m}\rfloor$ diagonal strategies from the $2 \cdot \lfloor\sqrt{m}\rfloor + 1$ diagonal strategies for the input instance (J_1, J_d). (S_1 can be viewed as a set of relative delays from $\{\lfloor-\sqrt{m}\rfloor, \ldots, \lfloor\sqrt{m}\rfloor\}$ between J_1 and J_d and together with D it determines $\lfloor\sqrt{m}\rfloor$ diagonal strategies for $(J_1, J_2, \ldots, J_d)$).

(2.2) Set S_2 as the set of the best $\lfloor\sqrt{m}\rfloor/2$ diagonal strategies from the diagonal strategies of S_1 according to the input instance (J_2, J_d).

$\vdots$

(2.i) Set S_i as the set of the best $\lfloor\sqrt{m}\rfloor/2^{i-1}$ diagonal strategies from the diagonal strategies of S_{i-1} according to the input instance (J_i, J_d).

(2.d-1) Choose the best diagonal strategy $\overline{D}$ from S_{i-1} according to (J_{d-1}, J_d).

Output: The diagonal strategy determined by D and $\overline{D}$.

Theorem 3. *For every input instance $I = (J_1, J_2, \ldots, J_d)$ of unit$-J_m$ with $d \leq 2\log_2 m$, the algorithm SURFACE(I)*

(i) *runs in time $O(d^2m^2)$, and*

(ii) *has an approximation ratio of at most $1 + \frac{2^d}{\sqrt{m}}$.*

Proof: SURFACE(I) does nothing else than looking on all $\binom{d}{2}$ 2-dimensional surfaces of $G_{m,d}(I)$ in order to choose a set of convenient delays with respect to every pair of jobs. The size of each surface is m^2 and the choice of a group of the best diagonals from a given set of diagonals can be done in $O(m^2)$ time. Thus, the overall time is in $O(d^2m^2)$.

To prove (ii) we first prove

(ii)′ The diagonal strategy computed by the algorithm SURFACE(I) contains at most $2^d\lceil\sqrt{m}\,\rceil$ delays.

We prove (ii)′ by induction on d. For $d = 2$ Lemma 3 guarantees at most $\lceil\sqrt{m}\,\rceil$ delays. Let (ii)′ be true for $d-1$, i.e., the strategy D computed for $(J_1, J_2, \ldots, J_{d-1})$ in the first step of SURFACE(I) contains at most $2^{d-1} \cdot \lceil\sqrt{m}\,\rceil$ delays between the first $d-1$ jobs. In Step (2.1) we look on the surface determined by (J_1, J_d). Following Lemma 3 the average number per diagonal of obstacles on the main $2 \cdot \lfloor\sqrt{m}\,\rfloor + 1$ diagonals of this surface is at most

$$\frac{m}{2 \cdot \lfloor\sqrt{m}\,\rfloor + 1} \leq \frac{\lceil\sqrt{m}\,\rceil}{2}.$$

[1] with respect to the number of obstacles

So, there must exist a set S_1 of $\lfloor\sqrt{m}\,\rfloor$ diagonals such that every diagonal of S_1 has at most $\lceil\sqrt{m}\,\rceil$ obstacles, i.e., at most twice the average. Observe, that each of these diagonals from S together with D determines a diagonal strategy for the whole instance $I = (J_1, J_2, \ldots, J_d)$, where J_1 and J_d have at most $\lceil\sqrt{m}\,\rceil$ delays. Thus, we have $\mid S_1 \mid= \lfloor\sqrt{m}\,\rfloor$ candidates for the output. In Step (2.2) we choose the best $\lfloor\sqrt{m}\,\rfloor/2$ from these candidates with respect to the obstacles for J_2 and J_d. Since these $\lfloor\sqrt{m}\,\rfloor$ candidates can contain together at most m obstacles, the average number of obstacles is $\lceil\sqrt{m}\,\rceil$, and so there exist $\lfloor\sqrt{m}\,\rfloor/2$ diagonals each with at most $2 \cdot \lceil\sqrt{m}\,\rceil$ obstacles. In general, in Step (2.i) for $2 \leq i \leq d-2$ we choose from the remaining $\lfloor\sqrt{m}\,\rfloor/2^{i-2}$ candidates the best $\lfloor\sqrt{m}\,\rfloor/2^{i-1}$ candidates with respect to the number of obstacles on the surface determined by J_i and J_d. Each of the candidates of S_i has at most $2^{i-1} \cdot \lceil\sqrt{m}\,\rceil$ obstacles between J_i and J_d. The last Step (2.d-1) corresponds to the choice of the best diagonal D' (with respect to the relation between J_{d-1} and J_d) from $\lfloor\sqrt{m}\,\rfloor/2^{d-3}$ candidates. The number of obstacles between J_{d-1} and J_d on D' is bounded by the average

$$\frac{m}{\lfloor\sqrt{m}\,\rfloor/2^{d-3}} = 2^{d-3} \cdot \lceil\sqrt{m}\,\rceil.$$

Let $\overline{D}$ be the resulting strategy for I. Thus, the overall number of obstacles between J_d and all other jobs in $\overline{D}$ is at most

$$\sum_{i=1}^{d-2} 2^{i-1} \cdot \lceil\sqrt{m}\,\rceil = (2^{d-2} - 1) \cdot \lceil\sqrt{m}\,\rceil < (2^{d-1} - 2) \cdot \lceil\sqrt{m}\,\rceil.$$

By the induction hypothesis the number of obstacles between the first $d-1$ jobs is at most $2^{d-1} \cdot \lceil\sqrt{m}\,\rceil$, and therefore, the overall number of obstacles on all $\binom{d}{2}$ 2-dimensional surfaces is at most

$$(2^d - 2) \cdot \lceil\sqrt{m}\,\rceil.$$

Obviously, these obstacles together cause at most $(2^d - 2)\lceil\sqrt{m}\,\rceil$ delays when following the diagonal strategy $\overline{D}$. The length of $\overline{D}$ is at most $m + 2 \cdot \lfloor\sqrt{m}\,\rfloor$ because $\overline{D}$ was constructed in such a way that no relative delay between J_1 and any other job would be greater than $\lfloor\sqrt{m}\,\rfloor$ (i.e., the relative delay between any pair of jobs is at most $2 \cdot \lfloor\sqrt{m}\,\rfloor$). Thus, the schedule that follows $\overline{D}$ has a makespan of at most

$$m + 2 \cdot \lfloor\sqrt{m}\,\rfloor + (2^d - 2) \cdot \lceil\sqrt{m}\,\rceil \leq m + 2^d \cdot \lceil\sqrt{m}\,\rceil.$$

Since the optimum makespan is at least m, the approximation ratio is at most

$$1 + \frac{2^d}{\lfloor\sqrt{m}\,\rfloor}.$$

□

The main point is that SURFACE works in quadratic time for constant d and can provide a good approximation ratio in that case. Observe, that the approximation ratio of SURFACE(I) tends to 1 with growing m for $d = o(\log_2 m)$.

7 Conclusions

For the job shop schedule problem *unit*–$J_m$we derived an upper bound on the makespan of optimum schedules that improves on the result given in [LMR99] for $d = o(m^{1/2})$. We presented a competitive w.r.t. the makespan of an optimum solution, randomized on-line approximation algorithm that solves *unit*–J_m in linear time with an expected approximation ratio of $1+2d/\sqrt{m}$ which amounts to $1+o(1)$ for $d = o(m^{1/2})$. For $d = o(m^{1/2})$ the algorithm is the best approximation algorithm for *unit*–J_m. Our deterministic approximation algorithm is efficient at least for small d's in comparison with m. Its run-time is $O(d^2m^2)$, and it has an approximation ratio of at most $1 + \frac{2^d}{\lfloor\sqrt{m}\,\rfloor}$ which tends to 1 with growing m for $d = o(\log_2 m)$. For the special case of *unit*–$J_m$with two jobs, which is solvable in linear time, we have shown that there exist input instances such that every schedule has a makespan of at least $m + \sqrt{m}$. Therewith, we proved that our upper bound on the makespan for $m + \lceil\sqrt{m}\rceil$, for $d = 2$ cannot be improved.

References

[B88] Brucker, P.: An Efficient Algorithm for the Job Shop Problem with Two Jobs. *Computing*, 40:353–359, 1988. 91, 92

[FS98] Feige U., Scheideler, C.: Improved Bounds for Acyclic Job Shop Scheduling. *Proc. 28th ACM Symposium on Theory of Computing*, pp. 624–233, 1998. 91

[GPSS97] Goldberg, L. A., Paterson, M., Srinivasan, A., Sweedyk, E.: Better Approximation Guarantees for Job-shop Scheduling. *Proc. 8th ACM-SIAM Symposium on Discrete Algorithms*, pp. 599–608, 1997. 90, 91

[LMR94] Leighton, F. T., Maggs, B. M., Rao, S. B.: Packet Routing and Job-Shop Scheduling in $O(Congestion+Dilation)$ steps. *Combinatorica*, 14:167–186, 1994.

[LMR99] Leighton, F. T., Maggs, B. M., Richa, A. W.: Fast Algorithms for Finding $O(Congestion+Dilation)$ Packet Routing Schedules. *Combinatorica*, 19:375–401, 1999. 90, 91, 92, 96, 100, 103

[LR79] Lenstra, J. K., Rinnooy Kan A. H. G.: Computational Complexity of Discrete Optimization Problems. *Annals of Discrete Mathematics*, 4:121–140, 1979.

[S98] Scheideler, C.: *Universal Routing Strategies for Interconnection Networks*. Lecture Notes in Computer Science 1390, Springer Verlag, 1998. 90, 91

[SSW94] Shmoys, D. B., Stein, C., Wein, J.: Improved Approximation Algorithms for Shop Scheduling Problems. *SIAM J. on Computing*, 23:617–632, 1994. 90, 91

[WHH97] Williamson, D. P., Hall, L. A., Hoogeveen, J. A., Hurkens, C. A. J., Lenstra, J. K., Sevast'janov, S. V., Shmoys, D. B.: Short Shop Schedules. *Operations Research*, 45:288–294, 1997. 90

8 Appendix: Proofs

8.1 Proof of Lemma 1: The Remaining Part

Remaining part of the proof of Lemma 1: We prove by induction on $i+1$ that any schedule for the jobs $(w_1, w_2, \ \ldots \ , w_i)$ and $(w_1^R, w_2^R, \ \ldots \ , w_i^R)$ causes at least i delays. To do so, we use the following induction hypothesis:

Any schedule for $\overline{I}_R^{\binom{i+1}{2}}$, where one job is completed and for the other job a prefix of length $\binom{i+1}{2} - r$, for $r \leq i$, is already processed (r is called the relative delay*), uses at least i orthogonal grid edges (sum-delay is at least i), and it uses at least $i+1$ orthogonal grid edges if the parities of r and i differ (i.e., r is odd and i is even, or r is even and i is odd).*

Obviously, this is true for $i = 1$. Let the hypothesis be true for $i' = i - 1$.

Now, consider a prefix of a schedule S for $\overline{I}_R^{\binom{i+1}{2}}, i > 1$, and i is odd. The case that i is even is left to the reader. Let us consider the last time unit t before the first task of w_i or of w_i^R will be executed. We distinguish between two possibilities according to the relative delay r of the executions of the prefixes up to t of J_1 to J_2 (i.e., the distance to the diagonal) in Graph($G_{\binom{i+1}{2}}$).

(ii)′ Let the relative delay be at least i', i.e., the distance to the main diagonal is $r \geq i'$. If $r \geq i' + 1 = i$, we are done. If $r = i'$, then one can use the diagonal edges only to execute w_i or w_i^R, but because of the same parity of r and i', the induction hypothesis is satisfied. Since any change of the relative delay during the work on w_i or w_i^R causes a new delay, the hypothesis is true after processing w_i or w_i^R, too.

(ii)′ Let the relative delay r be at most i'. Then, following the induction hypothesis, the schedule contains in this moment at least i' delays if r is even, and at least $i'+1$ delays if r is odd. If r is even, then it is sufficient to observe that it is impossible to reach the border of the grid $G_{\binom{i+1}{2}}$ by using diagonal edges only. This is because $w_i = \binom{i}{2} + 1, \ \ldots \ , \binom{i}{2} + i$, $w_i^R = \binom{i}{2} + i, \ \ldots \ , \binom{i}{2} + 1$. Therefore, the execution of the task $\sigma_{\binom{i}{2}+j}$ is an obstacle for the following sequence of diagonal edges running parallel to the main diagonal in the distance $i-2j+1$ (corresponding to relative delay $i-2j+1$) for $j = 1, \ \ldots \ , \lfloor i/2 \rfloor$. Hence, at least 1 additional delay is necessary, and two additional delays are necessary if the schedule finishes in the same distance r from the diagonal. If r is odd, and the schedule S executes w_i or w_i^R by using diagonal edges only, we have i "old" delays (induction hypothesis) and we are done. Obviously, if the distance to the diagonal changes, at least one additional delay occurs.

□

8.2 Proof of Lemma 2: The Remaining Part

Remaining part of the proof of Lemma 2: We use the induction of the proof of Lemma 1 in the following way. The prefixes $\pi_1' = w_1, w_2, \ldots, w_{k-1}$ and

$\pi_2' = w_1^R, w_2^R, \ldots, w_{k-1}^R$ of the instance $I_R^{k^2}$ define an instance $I'^{\binom{k'+1}{2}}_R$ considered in Lemma 1, with $k' = k-1$. The suffixes $\pi_1'' = u_{k-1}, u_{k-2}, \ldots, u_1$ and $\pi_2'' = u_{k-1}^R, u_{k-2}^R, \ldots, u_1^R$ define the same instance in a symmetric way. We distinguish two cases.

(ii)′ The relative delay r caused by the prefix $I'^{\binom{k'+1}{2}}_R$ is $r \leq k'$ and the parities of k' and r are the same. Then we know from Lemma 1 that any schedule of this prefix uses k' orthogonal grid edges. However, in the case that the parities of k' and r are the same it is impossible to reach the border of the grid $G_{\binom{k+1}{2}}$ by using diagonal edges only. This is because of w_k and w_k^R and hence, at least 1 additional delay is necessary, and two additional delays are necessary if the schedule finishes in the same distance r from the diagonal. After executing the tasks of w_k and w_k^R the schedule uses either $k'+1 = k$ orthogonal grid edges and changes the parity of r or it uses $k'+2 = k+1$ orthogonal grid edges and does not change the parity of r.

(ii)′ The relative delay r caused by the prefix $I'^{\binom{k'+1}{2}}_R$ is $r \leq k'$ and the parities of k' and r differ. Then we know from Lemma 1 that any schedule of this prefix uses $k'+1$ orthogonal grid edges. In this case, the schedule can execute the tasks of w_k and w_k^R by using diagonal grid edges only and therefore, does not need to change the parity of r.

Now, if the parities of r and k' are the same and the schedule uses two additional delays to execute w_k and w_k^R then we have k' delays for the prefix and k' delays for the suffix, i.e., the sum-delay equals $2(k-1)+2 = 2k$. If the schedule uses only one additional delay to execute w_k and w_k^R then the parities of r and k' for the suffix differ. Hence, we have k' delays for the prefix and $k'+1$ delays for the suffix, i.e., the sum-delay equals $k+k-1+1 = 2k$. The case that the parities of r and k' differ for the prefix are symmetrical. □

8.3 Proof of Fact 1

Proof of Fact 1: It is obvious that every $(d-1)$-dimensional subgrid determined by a task σ intersects each of the diagonals of $\mathcal{D}$ in exactly one unit grid cube[2]. Thus, it remains to bound the number of diagonals intersecting the $(d-2)$-dimensional subgrid considered.

The intersection G_σ of two subgrids labeled by the same task σ corresponds to a fixed relative delay between the execution of two jobs. If the task σ is at the i^{th} position on the a^{th}-axis and at the j^{th} position on the b^{th} axis, $j \leq i$, then the relative delay between the execution of the b^{th} job and the a^{th} job is $j-i$ for all diagonals intersecting G_σ.

Thus, we count the number of diagonals from $\mathcal{D}$ with the relative delay $j-i$ between the b^{th} and the a^{th} job. Since $\mathcal{D}$ is the union of all $\mathcal{D}_p$'s, where $\mathcal{D}_p$

[2] This is the cube that corresponds to the execution of the task σ in the job determined by the considered axis.

contains all diagonals with the p^{th} element equal to 0 and $\mathcal{D}_u \cap \mathcal{D}_v = \emptyset$ for $u \neq v$, $u, v \in \{1, 2, \ldots, d\}$, we count the number of such diagonals in $\mathcal{D}_p$ for every p separately.

Let $p \in \{1, 2, \ldots, d\} - \{a, b\}$. The intersection of $\mathcal{D}_p$ with G_σ meets all the diagonals with $D(c_1, c_2, \ldots, c_d)$, where $c_p = 0$ and $c_b = c_a + j - i$. One has r possible choices for every position from the $d - 3$ positions of $\{1, 2, \ldots, d\} - \{p, a, b\}$, and at most $r - (j - i) \leq r$ choices for the a^{th} axis. The b^{th} axis is unambiguously determined by the a^{th} position. So, we have at most r^{d-2} grid cubes in the intersection of G_σ and $\mathcal{D}_p$ for $p \in \{1, 2, \ldots, d\} - \{a, b\}$. G_σ meets exactly the diagonals $D(t_1, t_2, \ldots, t_d)$ of $\mathcal{D}_b$, that has $t_b = 0$ and $t_a = i - j$. The number of such diagonals[3] is exactly r^{d-2}. G_σ does not intersect any diagonal from $\mathcal{D}_a$ because the diagonals $D(s_1, s_2, \ldots, s_d)$ in $\mathcal{D}_a$ have $s_a \geq s_u$ for every $u \in \{1, 2, \ldots, d\}$, i.e., the a^{th} job is executed as the first one and so it cannot be delayed with respect to any other job (including the b^{th} job). Thus, all together G_σ intersects at most

$$(d - 1) \cdot r^{d-2}$$

diagonals. □

[3] with 2 fixed positions

Job Shop Scheduling Problems with Controllable Processing Times

Klaus Jansen[1]*, Monaldo Mastrolilli[2]**, and Roberto Solis-Oba[3]***

[1] Institut für Informatik und Praktische Mathematik, Universität zu Kiel
Germany
kj@informatik.uni-kiel.de
[2] IDSIA, Switzerland
monaldo@idsia.ch
[3] Department of Computer Science, The University of Western Ontario
Canada
solis@csd.uwo.ca

Abstract. Most scheduling models assume that the jobs have *fixed* processing times. However, in real-life applications the processing time of a job often depends on the amount of resources such as facilities, manpower, funds, etc. allocated to it, and so its processing time can be reduced when additional resources are assigned to the job. A scheduling problem in which the processing times of the jobs can be reduced at some expense is called a scheduling problem with *controllable* processing times. In this paper we study the job shop scheduling problem under the assumption that the jobs have controllable processing times. We consider two models of controllable processing times: continuous and discrete. For both models we present polynomial time approximation schemes when the number of machines and the number of operations per job are fixed.

1 Introduction

Scheduling is one of the fundamental areas of combinatorial optimization. Many scheduling problems are known to be hard to solve optimally, thus, much of the research on these problems focuses on giving efficient approximation algorithms that produce solutions close to the optimum. Ideally, one hopes to obtain an algorithm that for any given $\varepsilon > 0$ it produces in polynomial time a solution of value within a factor of $(1 + \varepsilon)$ of the optimum. Such an algorithm is called a polynomial time approximation scheme (PTAS).

* Partially supported by EU project APPOL, "Approximation and Online Algorithms", IST-1999-14084.

** Supported by Swiss National Science Foundation project 20-63733.00/1, "Resource Allocation and Scheduling in Flexible Manufacturing Systems", and by the "Metaheuristics Network", grant HPRN-CT-1999-00106.

*** Partially supported by Natural Sciences and Engineering Research Council of Canada grant R3050A01.

A. Restivo, S. Ronchi Della Rocca, L. Roversi (Eds.): ICTCS 2001, LNCS 2202, pp. 107–122, 2001.

Most classical scheduling models assume that in a scheduling problem the jobs to be scheduled have *fixed* processing times. However, in real-life applications the processing time of a job often depends on the amount of resources such as facilities, manpower, funds, etc. allocated to it, and so its processing time can be reduced when additional resources are assigned to the job. This accelerated processing of a job comes at a certain cost, though. A scheduling problem in which the processing times of the jobs can be reduced at some expense is called a scheduling problem with *controllable* processing times.

Scheduling problems with controllable processing times have gained importance in scheduling research since the early works of Vickson [14,15]. For a survey of this area until 1990, the reader is referred to [8]. More recent results include [2,3,10]. Two interesting related results are due to Trick [13] and Shmoys & Tardos [12]. They studied the scheduling of jobs with controllable processing times on unrelated machines. Trick [13] gave a 2.618-approximation algorithm for problem P3 (see below for definition of problem P3) on unrelated machines. This was improved by Shmoys and Tardos [12] who designed a 2-approximation algorithm. Furthermore, they also found a 2-approximation algorithm for problem P1 (see below) on unrelated machines.

1.1 Job Shop Scheduling with Controllable Processing Times

The job shop scheduling problem is a fundamental problem in Operations Research. In this problem there is a set $\mathcal{J} = \{J_1, ..., J_n\}$ of jobs that must be processed by a set of m machines. Every job J_j consists of an ordered sequence of μ operations $O_{1j}, O_{2j}, \ldots, O_{\mu j}$. Every operation O_{ij} must be processed without interruption by machine m_{ij} for p_{ij} units of time. A machine can process only one operation at a time, and for any job at most one of its operations can be processed at any moment. The problem is to schedule the jobs so that the maximum completion time $T_{\max}$ is minimized. Time $T_{\max}$ is called the *length* or *makespan* of the schedule.

The job shop scheduling problem is considered to be one of the most difficult to solve problems in combinatorial optimization, both, from the theoretical and the practical points of view. The problem is strongly NP-hard even if each job has at most three operations and there are only two machines [7]. Williamson et al. [16] show that the optimum solution of the problem cannot be approximated in polynomial time within a factor smaller than 5/4 unless P=NP. However, when m and μ are fixed, Jansen et al. [5] designed a polynomial time approximation scheme for the problem.

Solving a scheduling problem with controllable processing times amounts to specifying a schedule σ that indicates the starting times for the operations and a vector δ that gives their processing times and costs. We denote by $T(\sigma, \delta)$ the makespan of schedule σ with processing times according to δ, and we denote by $C(\delta)$ the total cost of δ. The problem of scheduling jobs with controllable processing times is a bicriteria optimization problem for which we can define the following three natural optimization problems.

P1. Minimize $T(\sigma, \delta)$, subject to $C(\delta) \leq \kappa$, for some given value $\kappa \geq 0$.
P2. Minimize $C(\delta)$, while ensuring that $T(\sigma, \delta) \leq \tau$, for some given value $\tau \geq 0$.
P3. Minimize $T(\sigma, \delta) + \alpha C(\delta)$, for some given value $\alpha > 0$.

In this paper we consider two variants for each one of the three above problems. The first variant allows *continuous* changes to the processing times of the operations. The second assumes only *discrete* changes. In the case of continuously controllable processing times, we assume that the cost of reducing the time needed to process an operation is an affine function of the processing time. This is a common assumption made when studying problems with controllable processing times [12,13]. In the case of discretely controllable processing times, there is a finite set of possible processing times and costs for every operation O_{ij}. We observe that, since for problem P2 deciding whether there is a solution of length $T(\sigma, \delta) \leq \tau$ is already NP-complete, the best result that we might expect to obtain (unless P=NP) is a solution with cost at most the optimal cost and makespan not greater than $\tau(1 + \varepsilon)$, $\varepsilon > 0$.

The problems addressed in this paper are all generalizations of the job shop scheduling problem, and therefore, they are strongly NP-hard. Nowicki and Zdrzalka [9] show that the version of problem P3 for the less general flow shop problem with continuously controllable processing times is NP-hard even when there are only two machines.

1.2 Our Contribution

We present the first known polynomial time approximation schemes for problems P1, P2, and P3, when the number m of machines and the number μ of operations per job are fixed. Our algorithms can handle both, continuously and discretely controllable processing times, and they can be extended to the case of convex piecewise linear processing times and cost functions. These results improve the 4/3-approximation algorithm for problem P3 described in Nowicki [10]. Moreover, the linear time complexity of our PTAS for problem P3 is the best possible with respect to the number of jobs.

Our algorithms are based on a paradigm that has been successfully applied to solve other scheduling problems. First partition the set of jobs into “large”, and “small” jobs. The set of large jobs has a constant number of jobs in it. Compute all possible schedules for the large jobs and, for each one of them, schedule the remaining jobs inside the empty gaps that the large jobs leave by first using a flinear program to assign jobs to gaps, and then computing a feasible schedule for the jobs assigned to each gap.

A major difficulty with using this approach for our problems is that the processing times and costs of the operations are not fixed, so we must determine their values before we can use the above approach. One possibility is to use a linear program to assign jobs to gaps and to determine the processing times and costs of the operations. But, we must be careful, since, for example, a natural extension of the linear program described in [5] defines a polytope with an exponential number of extreme points, and it does not seem to be possible to solve such linear program in polynomial time. We show how to construct a small polytope with only a polynomial number of extreme points that contains all the optimum solutions of the above linear program. This polytope is defined by a linear program that can be solved exactly in polynomial time and approximately, to within any pre-specified precision, in strongly polynomial time.

Our approach is robust enough so that it can be used to design polynomial time approximation schemes for both the discrete and the continuous versions of problems P1-P3. Due to space limitation we focus our attention on the continuous case of P1 and the discrete case of P3. The remaining results will be given in the long version of this paper.

Throughout this paper we present a series of transformations that simplify any instance of the above problems. Some transformations may potentially increase the value of the objective function by a factor of $1 + O(\varepsilon)$, $\varepsilon > 0$, so we can perform a constant number of them while still staying within $1 + O(\varepsilon)$ of the optimum. We say that this kind of transformations produce $1 + O(\varepsilon)$ *loss*. A transformation that does not modify the value of the optimum solution is said to produce *no loss*.

The rest of the paper is organized in the following way. In Section 2, we present a polynomial time approximation scheme for problem P1 with continuous processing times. In Section 3 we study problem P3 with discrete processing times, and show how to design a linear time PTAS for it.

2 Minimizing the Makespan Subject to Budget Constraints: Continuous Processing Times

In the case of continuously controllable processing times, we assume that for each job O_{ij} there is an interval $[\ell_{ij}, u_{ij}]$, $0 \leq \ell_{ij} \leq u_{ij}$, specifying its possible processing times. The cost for processing operation O_{ij} in time ℓ_{ij} is denoted as $c_{ij}^{\ell} \geq 0$ and for processing it in time u_{ij} the cost is $c_{ij}^{u} \geq 0$. For any value $\delta_{ij} \in [0, 1]$ the cost for processing operation O_{ij} in time $p_{ij}^{\delta_{ij}} = \delta_{ij}\ell_{ij} + (1 - \delta_{ij})u_{ij}$ is $c_{ij}^{\delta_{ij}} = \delta_{ij}c_{ij}^{\ell} + (1 - \delta_{ij})c_{ij}^{u}$. We assume that ℓ_{ij}, u_{ij}, c_{ij}^{ℓ}, c_{ij}^{u} and δ_{ij} are rational numbers. Moreover, without loss of generality, we assume that for every operation O_{ij}, $c_{ij}^{u} \leq c_{ij}^{\ell}$, and if $c_{ij}^{u} = c_{ij}^{\ell}$ then $u_{ij} = \ell_{ij}$.

Let us consider an instance of problem P1. Divide all c_{ij}^{u} and c_{ij}^{ℓ} values by κ to get an equivalent instance in which the bound on the total cost of the solution is one, i.e. $C(\delta) \leq 1$. If $c_{ij}^{\ell} > 1$ for some operation O_{ij}, we set $c_{ij}^{\ell} = 1$ and make $\ell_{ij} = (u_{ij}(c_{ij}^{\ell} - 1) - \ell_{ij}(c_{ij}^{u} - 1))/(c_{ij}^{\ell} - c_{ij}^{u})$ to get an equivalent instance of the problem in which $c_{ij}^{\ell} \leq 1$ for all operations O_{ij}.

2.1 A Simple m-Approximation Algorithm

We begin by showing that it is possible to compute in linear time an m-approximate solution for problem P1. This allows us to compute upper and lower bounds for the minimum makespan that we use in the following sections.

Let $U = \sum_{j=1}^{n} \sum_{i=1}^{\mu} u_{ij}$; note that U is a constant value. Consider an optimal solution (σ^*, δ^*) of problem P1 and let us use OPT to denote the optimum makespan. Let $P^* = \sum_{j=1}^{n} \sum_{i=1}^{\mu} p_{ij}^{\delta_{ij}^*} = \sum_{j=1}^{n} \sum_{i=1}^{\mu} (\ell_{ij} - u_{ij})\delta_{ij}^* + U$ be the sum of the processing times of all jobs in this optimum solution. Define $c_{ij} = c_{ij}^{\ell} - c_{ij}^{u}$, so $C(\delta^*) = \sum_{j=1}^{n} \sum_{i=1}^{\mu} (c_{ij}\delta_{ij}^* + c_{ij}^{u}) \leq 1$. Let $\tilde{x}$ be an optimal solution for the following linear program and $P(\tilde{x})$ the corresponding value.

$$\begin{array}{ll} \min\ P = \sum_{j=1}^{n} \sum_{i=1}^{\mu} (\ell_{ij} - u_{ij}) x_{ij} + U & \\ \text{s.t.}\ \ \sum_{j=1}^{n} \sum_{i=1}^{\mu} x_{ij} c_{ij} \leq 1 - \sum_{j=1}^{n} \sum_{i=1}^{\mu} c_{ij}^{u}. & \\ \qquad 0 \leq x_{ij} \leq 1 & j = 1, ..., n \text{ and } i = 1, ..., \mu. \end{array}$$

Observe that $P(\tilde{x}) \leq P^*$ and $C(\tilde{x}) \leq 1$. Note that this linear program is equivalent to the knapsack problem when the integrality constraints have been relaxed, and thus it can be solved in $O(n)$ time [6]. If we schedule all the jobs one after another with processing times as defined by $\tilde{x}$, we obtain a feasible schedule for the jobs $\mathcal{J}$ with makespan at most $P(\tilde{x})$. Since $OPT \geq P^*/m$ (this is the makespan of a schedule which leaves no idle times in the machines), the obtained schedule has cost at most 1 and makespan $P(\tilde{x}) \leq m \cdot OPT$. Therefore, $OPT \in [P(\tilde{x})/m, P(\tilde{x})]$, and by dividing all ℓ_{ij} and u_{ij} values by $P(\tilde{x})$, we get the following bounds for the optimum makespan:

$$1/m \leq OPT \leq 1. \tag{1}$$

2.2 Large, Medium, and Small Jobs

We partition the set of jobs in three groups in the following way. Let $P_j^* = \sum_{i=1}^{\mu} p_{ij}^{\delta_{ij}^*}$ be the sum of the processing times of the operations of job J_j according to an optimum solution (σ^*, δ^*). Let k and q be two constants, to be defined later, that depend on m, μ and ε. We define the set $\mathcal{L}$ of large jobs consisting of the k longest jobs according to δ^*. The next $(\frac{q\mu m^2}{\varepsilon} - 1)k$ longest jobs define the set $\mathcal{M}$ of medium jobs. The remaining jobs form the set $\mathcal{S}$ of small jobs. Even when we do not know an optimum solution (σ^*, δ^*) for the problem, we show in Section 2.4 how to select the set of long and medium jobs in polynomial time. Hence, we assume that we know sets $\mathcal{L}$, $\mathcal{M}$ and $\mathcal{S}$. An operation that belongs to a large, medium, or small job is called a large, medium, or small operation, respectively, regardless of its processing time.

In the following we simplify the input by creating a well-structured set of possible processing times. We begin by showing that it is possible to bound from above the possible processing times of small operations by $\frac{\varepsilon}{qk\mu m}$.

Lemma 1. *With no loss, for each small operation O_{ij} we can set $u_{ij} \leftarrow \bar{u}_{ij}$ and $c_{ij}^{u} \leftarrow \frac{c_{ij}^{\ell} - c_{ij}^{u}}{u_{ij} - \ell_{ij}}(u_{ij} - \bar{u}_{ij}) + c_{ij}^{u}$, where $\bar{u}_{ij} = \min\{u_{ij}, \frac{\varepsilon}{qk\mu m}\}$.*

Proof. By Inequality (1), the optimal makespan is at most 1, and therefore the sum of all processing times cannot be larger than m. Let p the length of the longest small job according to δ^*. By definition of $\mathcal{L}$ and $\mathcal{M}$, $|\mathcal{M} \cup \mathcal{L}| \cdot p = \frac{qk\mu m^2}{\varepsilon} p \leq \sum_{J_j \in \mathcal{M} \cup \mathcal{L}} P_j^* \leq m$, and so $p \leq \frac{\varepsilon}{qk\mu m}$. Therefore, the length of the schedule is not increased if the largest processing time u_{ij} of any small operation O_{ij} is set to be $\bar{u}_{ij} = \min\{u_{ij}, \frac{\varepsilon}{qk\mu m}\}$. It is easy to check that, in order to get an equivalent instance it is necessary to set $c_{ij}^u \leftarrow \frac{c_{ij}^\ell - c_{ij}^u}{u_{ij} - \ell_{ij}}(u_{ij} - \bar{u}_{ij}) + c_{ij}^u$ (this is the cost to process operation O_{ij} in time $\bar{u}_{ij}$). □

In order to compute a $1 + O(\varepsilon)$-approximate solution for P1 we show that it is sufficient to take into consideration only a constant number of different processing times and costs for medium jobs.

Lemma 2. *There exists a* $(1 + 2\varepsilon)$*-optimal schedule where each medium operation has processing time equal to* $\frac{\varepsilon}{m|\mathcal{M}|\mu}(1 + \varepsilon)^i$, *for* $i \in \mathbb{N}$.

Proof. Let us use (σ^*, δ^*) to denote an optimal solution. Let A be the set of medium operations for which $p_{ij}^{\delta_{ij}^*} \leq \frac{\varepsilon}{m|\mathcal{M}|\mu}$. Since $c_{ij}^u \leq c_{ij}^\ell$, we observe that by increasing the processing time of any operation then the corresponding cost cannot increase. Therefore, if we increase the processing times for the operations in A up to $\frac{\varepsilon}{m|\mathcal{M}|\mu}$, the makespan may potentially increase by at most $|A| \frac{\varepsilon}{m|\mathcal{M}|\mu} \leq \varepsilon/m$, and by Inequality (1) the schedule length may be increased by a factor of $1 + \varepsilon$. Now, consider the remaining medium operations and round every processing time $p_{ij}^{\delta_{ij}^*}$ up to the nearest value of $\frac{\varepsilon}{m|\mathcal{M}|\mu}(1 + \varepsilon)^i$, for some $i \in \mathbb{N}$. Since each processing time is increased by a factor $1 + \varepsilon$, the value of an optimum solution potentially increases by the same factor $1 + \varepsilon$. □

Recall that by Inequality (1), every processing time cannot be greater than 1. Then, by the previous lemma, the number of different processing times for medium operations can be bounded by $O(\log(m|\mathcal{M}|\mu)/\varepsilon)$ (clearly, the same bound applies to the number of different costs, since processing times and costs are closely related). Since there is a constant number of medium operations, there is also a constant number of choices for the values of their processing times and costs. We consider all these choices (thus, we also consider the case where the processing times $\bar{p}_{ij}$ and costs $\bar{c}_{ij}$ for the medium operations are chosen as in the $(1 + 2\varepsilon)$-optimal schedule of the previous lemma). In the following we show that when the medium operations are processed according to these $(\bar{p}_{ij}, \bar{c}_{ij})$-values, it is possible to compute a $1 + O(\varepsilon)$-approximate solution for P1 in polynomial time. Clearly, a $1 + O(\varepsilon)$-approximate algorithm is obtained by considering all possible choices for processing times for the medium operations, and by returning the solution that is of minimum length. From now on, we assume, without loss of generality, that we know these $(\bar{p}_{ij}, \bar{c}_{ij})$-values for medium operations. In order to simplify the following discussion, for each medium operation O_{ij} we set $\ell_{ij} = u_{ij} = \bar{p}_{ij}$ and $c_{ij}^\ell = c_{ij}^u = \bar{c}_{ij}$ (this settings fix the processing time and cost of O_{ij} to be $\bar{p}_{ij}$ and $\bar{c}_{ij}$, respectively).

2.3 Computing Processing Times and Assigning Operations to Snapshots

A *relative schedule* for the large operations is an ordering of the starting and ending times of the operations. A feasible schedule S for the large operations *respects* a relative schedule R if the starting and ending times of the operations as defined by S are ordered as indicated in R (breaking ties in an appropriate way).

Fix a relative schedule R for the large operations. The starting and finishing times of the operations define a set of intervals that we call *snapshots*. Let $M(1), M(2), \ldots, M(g)$, be the snapshots defined by R. Snapshots $M(1)$ and $M(g)$ are empty. The number of snapshots g can be bounded by $g \leq 2k\mu + 1$.

Lemma 3. *The number of different relative schedules for the large jobs is at most* $(2ek)^{2k\mu}$.

Proof. The number of possible starting times for the operations of a large job J_j is at most the number of subsets of size μ that can be chosen from a set of $(2k\mu - 1)$ positions (there are $2\mu k - 1$ choices for the starting times of each operations of J_j). Since each large operation can end in the same snapshot in which it starts, the number of ways of choosing the starting and ending times of the operations of a large job is at most the number of subsets of size 2μ that can be chosen from a set of $2(2k\mu - 1)$ positions (we consider two positions associated to each snapshot, one to start and one to end an operation, but both positions denote the same snapshot). For each large job J_j there are at most $\binom{4\mu k-2}{2\mu}$ different choices of snapshots where operations of J_j can start and end. Since $\binom{4\mu k-2}{2\mu} = \frac{(4\mu k-2)(4\mu k-3)\ldots(4\mu k-2\mu-1)}{(2\mu)!} \leq \frac{(4\mu k)^{2\mu}}{(2\mu/e)^{2\mu}} = (2ek)^{2\mu}$ and the number of large jobs is k, then there are at most $(2ek)^{2k\mu}$ different relative schedules. □

By the previous lemma the number of different relative schedules is bounded by a constant. Our algorithm considers all relative schedules for the large jobs. Therefore, one of them must be equal to the relative schedule R^* defined by some optimum solution. In the following we will show that when R^* is taken into consideration we are able to provide in polynomial time a $1 + O(\varepsilon)$-approximate solution.

Given a relative schedule R^* as described above, to obtain a solution for problem P1 that respects R^* we must select the processing times for the operations and schedule the medium and small operations within the snapshots defined by R^*. We use a linear program to compute the processing times and for deciding the snapshots where the small and medium operations must be placed. Then we find a feasible schedule for the operations in every snapshot.

To design the linear program, we create a variable x_{ij}^{ℓ} for each large operation O_{ij}; this variable defines the processing time and cost of operation O_{ij}. For convenience we define another variable x_{ij}^{u} with value $1 - x_{ij}^{\ell}$. The processing time of operation O_{ij} is then $x_{ij}^{\ell}\ell_{ij} + x_{ij}^{u}u_{ij}$, and its cost is $x_{ij}^{\ell}c_{ij}^{\ell} + x_{ij}^{u}c_{ij}^{u}$. Let α_{ij} be the snapshot where the large operation O_{ij} starts processing in the relative schedule R^* and let β_{ij} be the snapshot where it finishes processing.

Let $Free(R^*) = \{(\ell, h) \mid \ell = 1, ..., g,\ h = 1, ..., m$ and no long operation is scheduled by R^* in snapshot $M(\ell)$ on machine $h\}$ be the set of snapshots and machines not used by the large jobs in relative schedule R^*. We represent the processing times and costs for the medium and small operations as follows. For every job $J_j \in \mathcal{S} \cup \mathcal{M}$, let Λ_j be the set of tuples of the form $\Lambda_j = \{(s_1, s_2, \ldots, s_\mu) \mid 1 \le s_1 \le s_2 \le \ldots, \le s_\mu \le g$, and $(s_i, m_{ij}) \in Free(R^*)$, for all $i = 1, \ldots, \mu\}$. Set Λ_j defines the set of μ-snapshots where it is possible to put the operations of job J_j.

Let $\Delta = \{(\delta_1, \delta_2, \ldots, \delta_\mu) \mid \delta_k \in \{0, 1\}$ for all $k = 1, \ldots, \mu\}$. For each job $J_j \in \mathcal{S} \cup \mathcal{M}$ we define a set of at most $(2g)^\mu$ variables $x_{j,(s,\delta)}$, where $s \in \Lambda_j$ and $\delta \in \Delta$. To explain the meaning of these variables, let us define $x_{ij}(w, 1) = \sum_{(s,\delta) \in \Lambda_j \times \Delta \mid s_i = w,\ \delta_i = 1} x_{j,(s,\delta)}$ and $x_{ij}(w, 0) = \sum_{(s,\delta) \in \Lambda_j \times \Delta \mid s_i = w,\ \delta_i = 0} x_{j,(s,\delta)}$, for each operation i, job J_j, and snapshot w. Given a set of values for the variables $x_{j,(s,\delta)}$, they define the processing times and an assignment of medium and small operations to snapshots in which the amount of time that operation O_{ij} is processed within snapshot $M(w)$ is $x_{ij}(w, 1) \cdot \ell_{ij} + x_{ij}(w, 0) \cdot u_{ij}$, and the fraction of O_{ij} that is assigned to this snapshot is $x_{ij}(w, 0) + x_{ij}(w, 1)$.

For each snapshot $M(\ell)$ we use a variable t_ℓ to denote its length. For any $(\ell, h) \in Free(R^*)$, we define the load $L_{\ell,h}$ on machine h in snapshot $M(\ell)$ as the total processing time of the operations from small and medium jobs that get assigned to h during $M(\ell)$, i.e.,

$$L_{\ell,h} = \sum_{J_j \in \mathcal{S} \cup \mathcal{M}} \ \sum_{(s,\delta) \in \Lambda_j \times \Delta} \left(\sum_{\substack{i=1 \\ s_i = \ell, m_{ij} = h \\ \delta_i = 1}}^{\mu} x_{j,(s,\delta)} \ell_{ij} + \sum_{\substack{i=1 \\ s_i = \ell, m_{ij} = h \\ \delta_i = 0}}^{\mu} x_{j,(s,\delta)} u_{ij} \right).$$

Let the cost be

$$C = \sum_{J_j \in \mathcal{S} \cup \mathcal{M}} \ \sum_{(s,\delta) \in \Lambda_j \times \Delta} \left(\sum_{\substack{i=1 \\ \delta_i = 1}}^{\mu} x_{j,(s,\delta)} c_{ij}^{\ell} + \sum_{\substack{i=1 \\ \delta_i = 0}}^{\mu} x_{j,(s,\delta)} c_{ij}^{u} \right) + \sum_{J_j \in \mathcal{L}} \sum_{i=1}^{\mu} (x_{ij}^{\ell} c_{ij}^{\ell} + x_{ij}^{u} c_{ij}^{u}).$$

We use the following linear program $LP(R^*)$ to determine processing times and to allocate operations to snapshots.

$$
\begin{array}{lll}
\min T = \sum_{\ell=1}^{g} t_\ell & & \\
\text{s.t. } C \le 1, & & (1)\\
\quad \sum_{\ell=\alpha_{ij}}^{\beta_{ij}} t_\ell = x_{ij}^{\ell}\ell_{ij} + x_{ij}^{u}u_{ij}, & J_j \in \mathcal{L},\, i = 1,\dots,\mu, & (2)\\
\quad x_{ij}^{\ell} + x_{ij}^{u} = 1, & J_j \in \mathcal{L},\, i = 1,\dots,\mu, & (3)\\
\quad \sum_{(s,\delta)\in\Lambda_j\times\Delta} x_{j,(s,\delta)} = 1, & J_j \in \mathcal{S}\cup\mathcal{M}, & (4)\\
\quad L_{\ell,h} \le t_\ell, & (\ell,h) \in Free(R^*), & (5)\\
\quad x_{ij}^{\ell}, x_{ij}^{u} \ge 0, & J_j \in \mathcal{L},\, i = 1,\dots,\mu, & (6)\\
\quad x_{j,(s,\delta)} \ge 0, & J_j \in \mathcal{S}\cup\mathcal{M},\, (s,\delta)\in\Lambda_j\times\Delta, & (7)\\
\quad t_\ell \ge 0, & \ell = 1,\dots,g. & (8)
\end{array}
$$

In this linear program the value of the objective function T is the length of the schedule, which we want to minimize. Constraint (1) ensures that the total cost of the solution is at most one. Condition (2) requires that the total length of the snapshots where a long operation is scheduled is exactly equal to the length of the operation. Constraint (4) assures that every small and medium operation is completely assigned to snapshots, while constraint (5) checks that the total load of a machine h during some snapshot ℓ does not exceed the length of the snapshot.

Let (σ^*, δ^*) denote an optimal schedule when the processing times and costs of medium jobs are fixed as described in the previous section.

Lemma 4. *The optimal solution of $LP(R^*)$ has value no larger than the makespan of (σ^*, δ^*).*

Proof. We only need to show that for any job $J_j \in \mathcal{S}\cup\mathcal{M}$ there is a feasible solution of $LP(R^*)$ that schedules all operations O_{ij} in the same snapshots and with the same processing times and costs as the optimum schedule (σ^*, δ^*).

For any operation O_{ij}, let $p_{ij}^{\delta_{ij}^*}(w)$ be the amount of time that O_{ij} is assigned to snapshot $M(w)$ by the optimum schedule (σ^*, δ^*). Let $x_{ij}^*(w,1) = \delta_{ij}^* p_{ij}^{\delta_{ij}^*}(w)/p_{ij}^{\delta_{ij}^*}$ and $x_{ij}^*(w,0) = (1-\delta_{ij}^*)p_{ij}^{\delta_{ij}^*}(w)/p_{ij}^{\delta_{ij}^*}$. The processing time $p_{ij}^{\delta_{ij}^*}(w)$ and cost $c_{ij}^{\delta_{ij}^*}(w)$ of O_{ij} can be written as $x_{ij}^*(w,1)\cdot\ell_{ij}+x_{ij}^*(w,0)\cdot u_{ij}$ and $x_{ij}^*(w,1)\cdot c_{ij}^{\ell} + x_{ij}^*(w,0)\cdot c_{ij}^{u}$, respectively.

Now we show that there is a feasible solution of $LP(R^*)$ such that
(i) $x_{ij}^*(w,1) = \sum_{(s,\delta)\in\Lambda_j\times\Delta \,|\, s_i=w,\ \delta_i=1} x_{j,(s,\delta)}$ and
(ii) $x_{ij}^*(w,0) = \sum_{(s,\delta)\in\Lambda_j\times\Delta \,|\, s_i=w,\ \delta_i=0} x_{j,(s,\delta)}$.

Therefore, for this solution $p_{ij}^{\delta_{ij}^*}(w)$ and $c_{ij}^{\delta_{ij}^*}(w)$ are linear combinations of the variables $x_{j,(s,\delta)}$. We assign values to the variables $x_{j,(s,\delta)}$ as follows.

```
1. For each job $J_j \in \mathcal{S}\cup\mathcal{M}$ do
2. (a) Compute $S_j = \{(i,\ell,d) \mid x_{ij}^*(\ell,d) > 0, i = 1,\dots,\mu,\ \ell = 1,\dots,g,\ d = 0,1\}$.
3. (a) If $S_j = \emptyset$ then exit.
```

4. (a) Let $f = \min \{x^*_{ij}(\ell, d) \mid (i, \ell, d) \in S_j\}$ and let i', ℓ', and d' be such that $x^*_{i'j}(\ell', d') = f$.
5. (a) Let $s = (s_1, s_2, \ldots, s_\mu) \in \Lambda_j$ and $\delta = (d_1, d_2, \ldots, d_\mu) \in \Delta$ be such that $s_{i'} = \ell'$, $d_{i'} = d'$ and $x^*_{ij}(s_i, d_i) > 0$, for all $i = 1, \ldots, \mu$.
6. (a) $x_{j,(s,\delta)} \leftarrow f$
7. (a) $x^*_{ij}(s_i, d_i) \leftarrow x^*_{ij}(s_i, d_i) - f$ for all $i = 1, 2, \ldots, \mu$.
8. (a) Go back to step 2.

With this assignment of values to the variables $x_{j,(s,\delta)}$, equations (i) and (ii) hold for all jobs $J_j \in \mathcal{S} \cup \mathcal{M}$, all operations i, and all snapshots w. Therefore, this solution of $LP(R^*)$ schedules the jobs in the same positions and with the same processing times as the optimum schedule. □

2.4 Finding a Feasible Schedule

Linear program $LP(R^*)$ has at most $1 + 2\mu k + n - k + mg - \mu k$ constraints. By condition (3) every large operation O_{ij} must have at least one of its variables x^ℓ_{ij} or x^u_{ij} set to a non-zero value. By condition (4) every small and medium job J_j must have at least one of its variables $x_{j,(s,\delta)}$ set to a positive value. Furthermore, there is at least one snapshot ℓ for which $t_\ell > 0$. Since in any basic feasible solution of $LP(R^*)$ the number of variables that receive positive values is at most equal to the number of rows of the constraint matrix, then in a basic feasible solution there are at most mg variables with fractional values. This means that in the schedule defined by a basic feasible solution of $LP(R^*)$ at most mg medium and small jobs receive fractional assignments, and therefore, there are at most that many jobs from $\mathcal{M} \cup \mathcal{S}$ for which at least one operation is assigned to at least two different snapshots. Let $\mathcal{F}$ be the set of jobs that received fractional assignments. For the time being let us forget about those jobs. We show later how to schedule them.

Note that this schedule is still not feasible because there might be ordering conflicts among the small or medium operations assigned to a snapshot. To eliminate these conflicts, we first remove the set $\mathcal{V}$ of jobs from $\mathcal{M} \cup \mathcal{S}$ which have at least one operation with processing time larger than $\frac{\varepsilon}{\mu^3 m^2 g}$ according to the solution of the linear program. Since the sum of the processing times computed by the linear program is at most m, then $|\mathcal{V}| \leq \mu^3 m^3 g/\varepsilon$, so we remove only a constant number of jobs.

Let $\mathcal{O}(\ell)$ be the set of operations from $\mathcal{M} \cup \mathcal{S}$ that remain in snapshot $M(\ell)$. Let $p_{max}(\ell)$ be the maximum processing time among the operations in $\mathcal{O}(\ell)$. Every snapshot $M(\ell)$ defines an instance of the classical job shop problem, since the solution of $LP(R^*)$ determined the processing time of every operation. Hence, we can use Sevastianov's algorithm [11] to find in $O(n^2\mu^2m^2)$ time a feasible schedule for the operations in $\mathcal{O}(\ell)$; this schedule has length at most $\bar{t}_\ell = t_\ell + \mu^3 m p_{max}(\ell)$. We must increase the length of every snapshot $M(\ell)$ to $\bar{t}_\ell$ to accommodate the schedule produced by Sevastianov's algorithm. Summing up all these snapshot enlargements, we get a solution of length at most the value of the solution of $LP(R^*)$ plus ε/m.

It remains to show how to schedule the jobs $\mathcal{V} \cup \mathcal{F}$. First we choose the value for parameter q:

$$q = 6\mu^4 m^3/\epsilon. \tag{2}$$

Then the number of jobs in $\mathcal{V} \cup \mathcal{F}$ is

$$|\mathcal{V} \cup \mathcal{F}| \leq \mu^3 m^3 g/\epsilon + mg \leq qk. \tag{3}$$

Lemma 5. *Consider solution (σ^*, δ^*). Let $P_j^* = \sum_{i=1}^{\mu} p_{ij}^{\delta_{ij}^*}$ denote the length of job J_j according to δ^*. There exists a positive constant k such that if the set of large jobs contains the k jobs J_j with the largest P_j^* value, then $\sum_{J_j \in \mathcal{V} \cup \mathcal{F}} P_j^* \leq \varepsilon/m$.*

Proof. Let us sort the jobs J_j non-increasingly by P_j^* values, and assume for convenience that $P_1^* \geq P_2^* \geq ... \geq P_n^*$. Partition the jobs into groups $G_1, G_2, ..., G_d$ as follows $G_i = \{J_{(1+qb)^{i-1}+1}, ..., J_{(1+qb)^i}\}$, and let $P(G_j) = \sum_{J_i \in G_j} P_j^*$. Let $G_{\rho+1}$ be the first group for which $P(G_{\rho+1}) \leq \varepsilon/m$. Since $\sum_{J_j \in \mathcal{J}} P_j^* \leq m$ and $\sum_{i=1}^{\rho} P(G_i) > \rho\varepsilon/m$ then $\rho < \frac{m^2}{\varepsilon}$. We choose $\mathcal{L}$ to contain all jobs in groups G_1 to G_ρ, and so $k = (1+q)^\rho$. Since $|\mathcal{V} \cup \mathcal{F}| \leq qk$, then with this choice of $\mathcal{L}$, $|G_{\rho+1}| = qk \geq |\mathcal{V} \cup \mathcal{F}|$ and therefore $\sum_{J_j \in \mathcal{V} \cup \mathcal{F}} P_j^* \leq \sum_{J_j \in G_\rho} P_j^* \leq \varepsilon/m$. □

We choose the set of large jobs by considering all subsets of k jobs, for all integer values $1 \leq k \leq \frac{m^2}{\varepsilon}$. For each choice of k the set of medium jobs is obtained by considering all possible subsets of $\left(q\mu m^2/\varepsilon - 1\right) k$ jobs. Since there is only a polynomial number of choices for large and medium jobs, we require only a polynomial number of attempts before detecting the set of large and medium jobs as defined by an optimum solution.

The processing time of every small operation O_{ij} in $\mathcal{V} \cup \mathcal{F}$, is chosen as $p_{ij} = u_{ij}$ and so its cost is $c_{ij} = c_{ij}^u$. Furthermore, recall that we are assuming that each medium operation O_{ij} is processed in time $p_{ij} = \bar{p}_{ij}$ and cost $c_{ij} = \bar{c}_{ij}$ (see section 2.2). Let $P_j = \sum_{i=1}^{\mu} p_{ij}$. Then $\sum_{J_j \in \mathcal{V} \cup \mathcal{F}} P_j = \sum_{J_j \in \mathcal{M} \cap (\mathcal{V} \cup \mathcal{F})} P_j + \sum_{J_j \in \mathcal{S} \cap (\mathcal{V} \cup \mathcal{F})} P_j$. Note that by Lemma 1 and inequality (3), $\sum_{J_j \in \mathcal{S} \cap (\mathcal{V} \cup \mathcal{F})} P_j \leq qk\mu\varepsilon/(qk\mu m) = \varepsilon/m$. By the arguments of section 2.2, $\bar{p}_{ij} \leq \max\{p_{ij}^{\delta_{ij}^*}(1+\varepsilon), \varepsilon/(m|\mathcal{M}|\mu)\}$ and, therefore, $\sum_{J_j \in \mathcal{M} \cap (\mathcal{V} \cup \mathcal{F})} P_j \leq \sum_{J_j \in \mathcal{M} \cap (\mathcal{V} \cup \mathcal{F})} P_j^*(1+\varepsilon) + \varepsilon/m \leq \varepsilon(2+\varepsilon)/m$, by Lemma 5. Therefore, we can schedule the jobs from $\mathcal{V} \cup \mathcal{F}$ one after the other at the end of the schedule without increasing too much the length of the schedule.

Theorem 1. *For any fixed m and μ, there exists a polynomial-time approximation scheme for P1 that computes a feasible schedule with makespan at most $(1+\epsilon)$ times the optimal makespan and cost not greater than κ, for any fixed $\epsilon > 0$.*

3 Problem P3 with Discrete Processing Times

In the case of discretely controllable processing times, the possible processing times and costs of an operation O_{ij} are specified by a set of values $\Delta_{ij} = \{\delta_1, \delta_2, \ldots, \delta_{w(i,j)}\}$, $0 \leq \delta_k \leq 1$ for all $k = 1, 2, \ldots, w(i,j)$. When the processing time for operation O_{ij} is $p_{ij}^{\delta_k} = \delta_k \ell_{ij} + (1-\delta_k)u_{ij}$, the cost is equal to $c_{ij}^{\delta_k} = \delta_k c_{ij}^{\ell} + (1-\delta_k)c_{ij}^{u}$.

For each operation O_{ij}, let $d_{ij} = \min_{\delta_{ij} \in \Delta_{ij}} \{p_{ij}^{\delta_{ij}} + c_{ij}^{\delta_{ij}}\}$. For every job J_j let $d_j = \sum_{i=1}^{\mu} d_{ij}$, and let $D = \sum_j d_j$. We partition the set of jobs into large $\mathcal{L}$ and small $\mathcal{S}$ jobs, where the set $\mathcal{L}$ includes the k jobs with the largest d_j values, and k is a constant computed as in Lemma 5 so that the set T containing the qk jobs with the next d_j values has $\sum_{J_j \in T} d_j \leq \varepsilon/m$. The set of large jobs can be computed in $O(n\mu|\Delta_{\max}|)$ time, where $|\Delta_{\max}| = \max_{ij} |\Delta_{ij}|$.

By dividing all $c_{ij}^{\delta_{ij}}$ values by parameter α, we can assume without loss of generality that the objective function for problem P3 is: $\min T(\sigma,\delta) + C(\delta)$. Let $p_{ij}^{\delta_{ij}^*}$ and $c_{ij}^{\delta_{ij}^*}$ be the processing time and cost of operation O_{ij} in an optimal solution. Let F^* be the value of an optimal solution for P3. It is easy to see that $F^* \leq D$ and $F^* \geq \frac{1}{m}\sum_{ij} p_{ij}^{\delta_{ij}^*} + \sum_{ij} c_{ij}^{\delta_{ij}^*} \geq \frac{D}{m}$. By dividing all execution times and costs by D, we may assume that $D = 1$ and

$$\frac{1}{m} \leq F^* \leq 1. \tag{4}$$

The following lemma shows that with $1 + 2\varepsilon$ loss we can reduce the number of different costs and processing times for each operation (proof in appendix).

Lemma 6. *With $1 + 2\varepsilon$ loss, we assume that $|\Delta_{ij}| = O(\log n)$ for every O_{ij}.*

Proof. To prove this claim, divide the interval $[0,1]$ into b subintervals as follows, $I_1 = [0, \frac{\varepsilon}{\mu nm}], I_2 = (\frac{\varepsilon}{\mu nm}, \frac{\varepsilon}{\mu nm}(1+\varepsilon)], \ldots, I_b = (\frac{\varepsilon}{\mu nm}(1+\varepsilon)^{b-1}, 1]$, where b is the largest integer such that $\frac{\varepsilon}{\mu nm}(1+\varepsilon)^{b-1} < 1$. Clearly $b = O(\log n)$. We say that d is a *choice* for operation O_{ij} if $d \in \Delta_{ij}$. For each operation O_{ij}, partition the set of choices Δ_{ij} into b groups $g_1, g_2, \ldots, g_b$, such that $d \in \Delta_{ij}$ belongs to group g_h iff c_{ij}^{d} falls in interval I_h, $h \in \{1, \ldots, b\}$.

For each group take the choice (if any) with the lowest processing time and delete the others. The new set of choices has at most $O(\min\{|\Delta_{ij}|, \log n\})$ elements and by using arguments similar to those used in the proof of Lemma 2 we can prove that with this transformation the cost of an optimum solution can be up to $1 + 2\varepsilon$ times the optimum value for the original problem. The transformed instance can be computed in $O(n\mu|\Delta_{\max}|)$ time. □

By using arguments similar to those used to prove Lemma 6 we can obtain, with $1+2\varepsilon$ loss, a new instance with $O(\log k)$ different costs and processing times for each large operation. Since there is a constant number of large operations, there is only a constant number of possible assignments of costs and processing times for them.

By trying all possible assignments of cost and processing times, we can find for each large operation O_{ij} a processing time $\bar{p}_{ij}$ and cost $\bar{c}_{ij}$ such that $\bar{p}_{ij} \leq \max\{\, p_{ij}^{\delta_{ij}^*}(1+\varepsilon), \varepsilon/(mk\mu)\,\}$ and $\bar{c}_{ij} \leq c_{ij}^{\delta_{ij}^*}$. Let us use the same definition of relative schedule given for the continuous case. Let R denote a relative schedule that respects the ordering of the large operations in some optimal schedule. For each small job J_j we define a set of at most $O((g \log n)^\mu)$ variables $x_{j,(s,\delta)}$, where $s \in \Lambda_j$ and $\delta \in \Delta_j = \{(\delta_{1j}, \delta_{2j}, \ldots, \delta_{\mu j}) \mid \delta_{ij} \in \Delta_{ij}$ for all $i = 1, \ldots, \mu\}$.

As in the continuous case we define a linear program $LP'(R)$ to compute the processing times and snapshots for the small jobs. $LP'(R)$ is obtained from $LP(R)$ by deleting constraints (1), (3), and (6), and considering the following changes. Variable $x_{j,(s,\delta)}$ takes value $0 \leq f \leq 1$ to indicate that a fraction f of operation O_{ij}, $i = 1, \ldots, \mu$ is scheduled in snapshot s_i with processing time $p_{ij}^{\delta_i}$. Let C be the cost function, i.e., $C = \sum_{J_j \in \mathcal{S}} \sum_{(s,\delta) \in \Lambda_j \times \Delta} \sum_{i=1}^{\mu} \ x_{j,(s,\delta)} c_{ij}^{\delta_k} + \sum_{J_j \in \mathcal{L}} \sum_{i=1}^{\mu} \bar{c}_{ij}$. The objective function is now to minimize $\sum_{\ell=1}^{g} t_\ell \ + C$. Constraint (2) is replaced with

$$\sum_{\ell=\alpha_{ij}}^{\beta_{ij}} t_\ell = \bar{p}_{ij}, \quad \text{for all } J_j \in \mathcal{L}, i = 1, \ldots, \mu.$$

As in Lemma 4, we can prove that any optimum solution of problem P3 is a feasible solution for $LP'(R)$. The rest of the algorithm is as that described in the previous subsection 2.4.

By using interior point methods to solve the linear program, we get a total running time for the above algorithm that is polynomial in the input size [1]. It is easy to check that similar results can be obtained if, instead of finding the optimum solution of the linear program, we solve it with a given accuracy $\varepsilon > 0$. In the following section we show that we can solve approximately the linear program in $O(n|\Delta_{\max}|)$ time. Therefore, for every fixed m, μ and ϵ, all computations (including Sevastianov's algorithm [5]) can be carried out in time $O(n|\Delta_{\max}| + n \min\{\log n, |\Delta_{\max}|\} \cdot f(\varepsilon, \mu, m))$, where $f(\varepsilon, \mu, m)$ is a function that depends on ε, μ and m. This running time is linear in the size of the input.

Theorem 2. *For any fixed m and μ, there exists a linear time approximation scheme for P3 with discretely controllable processing times.*

3.1 Approximate Solution of the Linear Program

In this section we show how to find efficiently a solution for $LP'(R)$ of value no more than $1 + O(\epsilon)$ times the value of the optimum solution for problem P3. To find an approximate solution for the linear program we rewrite it as a convex block-angular resource-sharing problem, and then use the algorithm of [4] to solve it with a given accuracy. A *convex block-angular resource sharing problem* has the form:
$$\lambda^* = \min\left\{\lambda \,\middle|\, \sum_{j=1}^{K} f_i^j(x^j) \leq \lambda, \text{for all } i = 1, \ldots, N, \text{and } x^j \in \mathcal{B}^j, j = 1, \ldots, K\right\},$$

where $f_i^j : \mathcal{B}^j \rightarrow \mathbb{R}^+$ are N non-negative continuous convex functions, and $\mathcal{B}^j$ are disjoint convex compact nonempty sets called *blocks*. The algorithm in [4] finds a $(1+\rho)$-approximate solution for this problem for any $\rho > 0$ in $O(N(\rho^{-2} \ln \rho^{-1} + \ln N)(N \ln\ln(N/\rho) + KF))$ time, where F is the time needed to find a ρ-approximate solution to the problem: $\min \left\{ \sum_{i=1}^{N} p_i f_i^j(x^j) \,\middle|\, x^j \in \mathcal{B}^j \right\}$, for some vector $(p_1, \ldots, p_N) \in \mathbb{R}^N$.

We can write $LP'(R)$ as a convex block-angular resource sharing problem as follows. First we guess the value V of an optimum solution of problem P3, and add the constraint: $\sum_{\ell=1}^{g} t_\ell + C + 1 - V \leq \lambda$, to the linear program, where λ is a non-negative value. Since $1/m \leq V \leq 1$, we can use binary search on the interval $[1/m, 1]$ to guess V with a given accuracy $\varepsilon > 0$. This search can be completed in $O(\log(\frac{1}{\varepsilon} \log m))$ iterations by doing the binary search on the values: $\frac{1}{m}(1+\varepsilon), \frac{1}{m}(1+\varepsilon)^2, ..., 1$. We replace constraint (5) of $LP'(R)$ by

$$\textbf{(5')} \qquad L_{\ell,h} + 1 - t_\ell \leq \lambda, \quad \text{for all } (\ell, h) \in \mathit{Free}(R)$$

where

$$L_{\ell,h} = \sum_{J_j \in \mathcal{S}} \sum_{(s,\delta) \in \Sigma_j \times \Delta} \sum_{\substack{q=1 \\ s_q = \ell, m_{qj} = h}}^{\mu} x_{j,(s,\delta)} p_{qj}^{\delta_q}.$$

This new linear program, that we denote as $LP(R, V, \lambda)$, has the above block-angular structure. The blocks $\mathcal{B}^j = \{x_{j,(s,\delta)} \mid (s,\delta) \in \Sigma_j \times \Delta$, constraints (4) and (8) hold$\}$, for $J_j \in \mathcal{S}$ are $(g|\Delta_{\max}|)^\mu$-dimensional *simplicies*. The block $\mathcal{B}^0 = \{< t_1, t_2, .., t_g > |$constraints (2) and (9) hold$\}$ has constant dimension. Let $f_{\ell,h} = L_{\ell,h} + 1 - t_\ell$. Since $t_\ell \leq V \leq 1$, these functions are non-negative.

For every small job J_j, let $f_0^j(x^j) = \sum_{(s,\delta) \in \Lambda_j \times \Delta} \sum_{i=1}^{\mu} x_{j,(s,\delta)} c_{ij}^{\delta_i}$. For every $(\ell, h) \in \mathit{Free}(R)$, let $f_{\ell h}^j(x^j) = \sum_{(s,\delta) \in \Lambda_j \times \Delta} \sum^{\mu}_{\substack{q=1 \\ s_q = \ell, m_{qj} = h}} x_{j,(s,\delta)} p_{qj}^{\delta_q}$. For every $x^0 \in \mathcal{B}^0$ let $f_0^0(x^0) = \sum_{J_j \in \mathcal{L}} \sum_{i=1}^{\mu} c_{ij} + \sum_{\ell=1}^{g} t_\ell + 1 - V$, and for every $(\ell, h) \in \mathit{Free}(R)$, let $f_{\ell h}^0(x^0) = 1 - t_\ell$. All these functions are nonnegative. Now $LP(R, V, \lambda)$ is to find the smallest value λ such that

$$\begin{aligned} &\textstyle\sum_{J_j \in \mathcal{S}} f_0^j(x^j) + f_0^0(x^0) \leq \lambda, && \text{for all } x^k \in \mathcal{B}^k, \\ &\textstyle\sum_{J_j \in \mathcal{S}} f_{\ell h}^j(x^j) + f_{\ell h}^0(x^0) \leq \lambda, && \text{for all } (\ell, h) \in \mathit{Free}(R) \text{ and } x^k \in \mathcal{B}^k. \end{aligned}$$

Using the algorithm in [4], a $1+\rho$, $\rho > 0$ approximation for the problem can be obtained by solving on each block $\mathcal{B}^j$ a constant number of block optimization problems of the form: $\min\{p^T f^j(x) \mid x \in \mathcal{B}^j\}$, where p is a $(g|\Delta_{\max}| + 1)$-dimensional positive price vector, and f^j is a $(g|\Delta_{\max}| + 1)$-dimensional vector whose components are the functions $f_0^j, f_{\ell h}^j$.

Note that $\mathcal{B}^0$ has constant dimension, and the corresponding block optimization problem can be solved in constant time. But, the blocks $\mathcal{B}^j$ for $J_j \in \mathcal{S}$ do not have a constant dimension. To solve the block optimization problem on $\mathcal{B}^j$ we must find a snapshot where to place each operation of J_j and determine its processing time, so that the total cost plus processing time of all operations times the price vector is minimized. To choose the snapshots we select for each operation the snapshot in which the corresponding component of the price vector is minimum. Then we select for each O_{ij} the value δ_{ij} that minimizes its cost plus processing time. This can be done in $O(|\Delta_{\max}|)$ time for each block, so the algorithm of [4] finds a feasible solution for $LP(R, V, 1+\rho)$ in $O(nw)$ time. Linear program $LP(R, V, 1+\rho)$ increases the length of each snapshot by ρ, and therefore the total length of the solution is $V + g\rho \leq (1+2\varepsilon)V^*$, for $\rho = \frac{\epsilon}{mg}$, where V^* is the optimal solution value.

There is a problem with this method: we cannot guarantee that the solution found by the algorithm is basic feasible. Hence it might have a large number of fractional assignments. In the following we show that the number of fractional assignments is $O(n)$. Since the number of fractional assignments is $O(n)$, using the rounding technique described in [5], we can obtain in linear time a new feasible solution with only a constant number of fractional assignments.

The algorithm in [4] works by choosing a starting solution $x_0 \in \mathcal{B}^j$ and then it repeats the following three steps for at most $O(mg\log(mg))$ times: (step 1) use a deterministic or randomized procedure to compute a price vector p; (step 2) use a block solver to compute an optimal solution of each block problem, (step 3) replace the current solution by a convex combination of the previous solutions. By starting from a solution x_0 in which every vector $x_0^j \in \mathcal{B}^j$ is an integer vector, for $j \neq 0$, we get at the end at most $O(n \cdot mg\log(mg))$ fractional assignments. To achieve the promised running time we additionally need that $\lambda(x_0) \leq c\lambda^*$ [4], where c is a constant and $\lambda(x_0)$ is the value of λ corresponding to x_0. This is accomplished as follows.

For convenience, let us rename jobs such that $J_1, ..., J_{\bar{n}}$ are the small jobs, where $\bar{n} = n - k$. Choose the processing time $p_{ij}^{\delta_{ij}}$ and cost $c_{ij}^{\delta_{ij}}$ for every small operation O_{ij} so that $d_{ij} = p_{ij}^{\delta_{ij}} + c_{ij}^{\delta_{ij}}$. Put the small jobs one after the other in the last snapshot. Set $t_g = \sum_{J_j \in \mathcal{S}} \sum_{i=1}^{\mu} p_{ij}^{\delta_{ij}}$. The large operations are scheduled as early as possible, according to the optimal relative schedule R. Set each $t_\ell \in \{t_1, t_2, ..., t_{g-1}\}$ equal to the maximum load of snapshot ℓ according to the described schedule. By inequality (4), we know that $\sum_{J_j \in \mathcal{S}} \sum_{i=1}^{\mu} d_{ij} \leq 1$. Furthermore, we have $\sum_{\ell=1}^{g-1} t_\ell + \sum_{J_j \in \mathcal{L}} \sum_{i=1}^{\mu} c_{ij} \leq V$, since by construction $\sum_{\ell=1}^{g-1} t_\ell$ cannot be greater than the optimal length and the costs of large operations are chosen according to the optimal solution. Hence, $\sum_{\ell=1}^{g} t_\ell + C \leq 1 + V$, and $\sum_{\ell=1}^{g} t_\ell + C + 1 - V \leq 2$, $L_{\ell,h} + 1 - t_\ell \leq 1$; so $\lambda(x_0) \leq 2$. Since $\lambda^* = 1$, it follows that $\lambda(x_0) \leq 2\lambda^*$.

References

1. K. M. Anstreicher, *Linear programming in $O(\frac{n^3}{\ln n}L)$ operations*, SIAM Journal on Optimization, Vol. 9, No. 4, (1999), pp. 803–812. 119
2. Z. L. Chen, Q. Lu, G. Tang, *Single machine scheduling with discretely controllable processing times*, Operations Research Letters, 21 (1997), pp. 69–76. 108

3. T. C. E. Cheng, N. Shakhlevich, *Proportionate flow shop with controllable processing times*, Journal of Scheduling, Volume 2, Number 6, 1999, pp. 253-265. 108
4. M. Grigoriadis and L. Khachiyan, *Coordination complexity of parallel price-directive decomposition*, Mathematics of Operations Research, 21 (2), 1996, pp. 321–340. 119, 120, 121
5. K. Jansen, R. Solis-Oba and M. I. Sviridenko, *A linear time approximation scheme for the job shop scheduling problem*, Proceedings of the Second International Workshop on Approximation Algorithms (APPROX 99), 177-188. 108, 109, 119, 121
6. E. L. Lawler, (1977) "Fast Approximation Algorithms for Knapsack Problems", 18th Annual Symposium on Foundations of Computer Science, 206–213. 111
7. E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, and D. B. Shmoys, *Sequencing and scheduling: Algorithms and complexity*, in: Handbook in Operations Research and Management Science, Vol. 4, North-Holland, 1993, 445-522. 108
8. E. Nowicki and S. Zdrzalka, *A survey of results for sequencing problems with controllable processing times*, Discrete Applied Mathematics, 26, 1990, pp. 271-287. 108
9. E. Nowicki and S. Zdrzalka, *A two-machine flow shop scheduling problem with controllable job processing times*, European Journal of Operational Research, 34, 1988, pp. 208-220. 109
10. E. Nowicki, *An approximation algorithm for the m-machine permutation flow shop scheduling problem with controllable processing time*, European Journal of Operational Research, 70 (1993), pp. 342–349. 108, 109
11. S. V. Sevastianov, *Bounding algorithms for the routing problem with arbitrary paths and alternative servers*, Cybernetics (in Russian), 22, 1986, pp. 773–780. 116
12. D. B. Shmoys and E. Tardos, *An approximation algorithm for the generalized assignment problem*, Mathematical Programming , 62, 1993, pp. 461–470. 108, 109
13. M. Trick, *Scheduling multiple variable-speed machines*, Operations research, 42, 1994, pp.234–248. 108, 109
14. R. G. Vickson, *Two single machine sequencing problems involving controllable job processing times*, AIIE Transactions, 12, 1980, pp. 258-262. 108
15. R. G. Vickson, *Choosing the job sequence and processing times to minimize total processing plus flow cost on a single machine*, Operations Research, 28, 1980, pp. 1155-1167. 108
16. D. Williamson, L. Hall, J. Hoogeveen, C.. Hurkens, J.. Lenstra, S. Sevastianov, and D. Shmoys, *Short shop schedules*, *Operations Research* 45 (1997), 288-294. 108

Upper Bounds on the Size of One-Way Quantum Finite Automata

Carlo Mereghetti[1] and Beatrice Palano[2]

[1] Dipartimento di Informatica, Sistemistica e Comunicazione, Università degli Studi di Milano – Bicocca
via Bicocca degli Arcimboldi 8, 20126 Milano, Italy
mereghetti@disco.unimib.it

[2] Dipartimento di Informatica, Università degli Studi di Torino
c.so Svizzera 185, 10149 Torino, Italy
beatrice@di.unito.it

Abstract. We show that, for any stochastic event p of period n, there exists a *measure-once one-way quantum finite automaton (1qfa)* with at most $2\sqrt{6n} + 25$ states inducing the event $ap + b$, for constants $a > 0$, $b \geq 0$, satisfying $a + b \leq 1$. This fact is proved by designing an algorithm which constructs the desired 1qfa in polynomial time. As a consequence, we get that any periodic language of period n can be accepted with isolated cut point by a 1qfa with no more than $2\sqrt{6n}+26$ states. Our results give added evidence of the strength of measure-once 1qfa's with respect to classical automata.

Keywords: quantum finite automata; periodic events and languages

1 Introduction

One of the main investigations in the field of quantum computing certainly deals with the study of the computational power of quantum devices with respect to their classical counterparts. In this sense, the results obtained by, e.g., Shor [18,19] and Grover [7] give evidences that the quantum paradigm might lead to faster algorithms. Nevertheless, it is reasonable to think that the first implementations of quantum machines will not be fully quantum mechanical. Instead, we can expect that they will consists of "expensive" quantum components embedded in classical devices (see, e.g., [3]). This motivates the study of the computational power of "small" quantum devices such as *quantum finite automata (qfa's)*.

The simplest version of qfa's are the *one-way qfa's (1qfa's)* which are basically defined by imposing the quantum paradigm — unitary evolution plus observation — to the classical model of one-way deterministic or probabilistic automata (1dfa's and 1pfa's, resp.). Two variants are considered: In the first one, called *measure-once* [5,17], the probability of accepting strings is evaluated by "observing" 1qfa's just once, at the end of input scanning. In the *measure-many* model [4,5,12], instead, observation is performed after each move. In this

A. Restivo, S. Ronchi Della Rocca, L. Roversi (Eds.): ICTCS 2001, LNCS 2202, pp. 123–135, 2001.

work, we will be concerned only with *measure-once 1qfa's.* Thus, the attribute measure-once will always be understood.

The question 1qfa's vs. classical automata is usually tackled from two points of view: the *recognizability* of languages, and the *size* — number of states — of automata when they perform certain works. It is well known that, quite surprisingly, the class of languages accepted by 1qfa's with isolated cut point is a proper subclass (group languages) of regular languages [5]. On the other hand, it is also well known that, in some cases, 1qfa's turn out to be more succinct than 1dfa's and 1pfa's. For instance, in [4], it is proved that accepting the language $L_p = \{1^{kp} \mid k \in \mathbf{N}$ and p is a fixed prime$\}$ with isolated cut point requires at least p states on 1pfa's, while a 1qfa with $\mathcal{O}(\log p)$ states for L_p is exhibited. Several other results are pointed out in the literature, that witness strength and weakness of 1qfa's (see, e.g., [4,8,10]). Almost all of them are obtained by constructing 1qfa's accepting *ad hoc* languages or solving suitably defined problems.

Here, we aim to give a general method for building succinct 1qfa's that have a "periodic behavior". More precisely: The stochastic event induced by a unary (i.e., with a single letter input alphabet) 1qfa $\mathcal{A}$ is the function $p : \mathbf{N} \to [0,1]$ defined, for any $k \in \mathbf{N}$, as $p(k)$ = probability that $\mathcal{A}$ accepts the string 1^k. We are interested in designing unary 1qfa's inducing *periodic events*, i.e., events satisfying $p(k) = p(k+n)$, for a fixed period $n > 0$ and any $k \in \mathbf{N}$. Actually, we will be content with obtaining a "linear approximation" of p, that is, an event of the form $ap + b$, for some constants $a > 0$, $b \geq 0$, with $a + b \leq 1$. It is not hard to verify that, from a language acceptance point of view, the events p and $ap + b$ are fully equivalent.

We prove that, *for any stochastic event p of period n taken as input, there exists a unary 1qfa $\mathcal{A}$ with at most $2\sqrt{6n} + 25$ states which induces $ap + b$, for some constants $a > 0$, $b \geq 0$, with $a + b \leq 1$.* More precisely, we provide an algorithm which actually constructs $\mathcal{A}$ in polynomial time. To this purpose, we first show that any event induced by a unary 1qfa has a sort of *normal form.* We then display an algorithm which, in a first phase, computes some parameters in this normal form so to reproduce the *harmonic structure* of p. In a second phase, the algorithm turns the computed parameters into a well formed unary 1qfa $\mathcal{A}$ with at most $2\sqrt{6n}+25$ states that induces $ap+b$. It is interesting to notice that the size of $\mathcal{A}$ is bounded by the size of *difference covers* for $\mathbf{Z}_n$, i.e., sets $\Delta \subseteq \mathbf{Z}_n$ such that each element in $\mathbf{Z}_n$ can be obtained as the difference modulo n of two elements in Δ.

This result allows us to give an upper bound on the size of 1qfa's accepting *periodic languages.* A unary languages L is said to be periodic if it can be written as $L = \{1^k \mid k \in \mathbf{N}$ and $(k \bmod n) \in S\}$, for a fixed $S \subseteq \mathbf{Z}_n$. The reader is referred to, e.g., [9,15] where the relevance of periodic languages is emphasized. We show that *any periodic language of period n can be accepted with isolated cut point by a unary 1qfa with no more than $2\sqrt{6n} + 26$ states.*

Our results once more witness the strength, by a quadratic state decreasing, of 1qfa's with respect to classical automata. It is well known, for instance, that accepting n-periodic languages on 1dfa's requires at least n states. Furthermore, when n is prime, we cannot hope to save states even by using 1pfa's [4,16] or two-way nondeterminism [15]. For a more extensive discussion on these and other topics in relation to the question quantum vs. classical devices, we refer the reader to Section 5.

The paper is organized as follows: In Section 2, we give basics on linear algebra, quantum finite automata, and difference cover. In Section 3, we present the polynomial algorithm to construct a $\mathcal{O}(\sqrt{n})$-state 1qfa inducing a linear approximation of a periodic stochastic event given as input. In Section 4, we show how to recognize with isolated cut point periodic languages with $\mathcal{O}(\sqrt{n})$ quantum states. Finally, in Section 5, we discuss our results in the light of quantum vs. classical question, and we point out some possible directions for future researches.

2 Preliminaries

2.1 Linear Algebra

Here, we recall some basic notions on vector spaces and linear algebra. For more details, we refer the reader to any of the standard books on the subject, such as [13,14]. Given a complex number $z \in \mathbf{C}$, its *complex conjugate* is denoted by z^*, and its *modulus* is $|z| = \sqrt{zz^*}$. Let $\mathcal{V}$ be a vector space of finite dimension n on $\mathbf{C}$. The *inner product* of vectors $x, y \in \mathcal{V}$, with $x = (x_1, x_2, \ldots, x_n)$ and $y = (y_1, y_2, \ldots, y_n)$, is defined as $\langle\, x, y \,\rangle = \sum_{i=1}^{n} x_i\, y_i^*$. The *norm* of x is defined as $\parallel x \parallel = \sqrt{\langle\, x, x \,\rangle}$. If $\langle\, x, y \,\rangle = 0$ (and $\parallel x \parallel = \parallel y \parallel = 1$) then x and y are *orthogonal* (*orthonormal*). A *decomposition of* $\mathcal{V}$ is a set $\{\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_k\}$ $(k \leq n)$ of mutually orthogonal subspaces of $\mathcal{V}$ such that each $x \in \mathcal{V}$ can be written as the sum of the projections of x onto each $\mathcal{S}_i$.

We denote by $\mathbf{C}^{m \times n}$ the set of complex matrices having m rows and n columns. Given two matrices $M \in \mathbf{C}^{m \times m}$ and $N \in \mathbf{C}^{n \times n}$, their *direct sum* is the block diagonal matrix $M \oplus N \in \mathbf{C}^{(m+n) \times (m+n)}$ having M and N on its main diagonal and 0 elsewhere. Let us introduce some properties of normal matrices that will turn out to be useful in what follows. We denote by $M^\dagger \in \mathbf{C}^{m \times m}$ the *conjugate transpose* of the matrix M. If $MM^\dagger = M^\dagger M$ then M is said to be *normal.* Two important subclasses of normal matrices are the unitary and the Hermitian matrices. A matrix M is said to be *unitary* whenever $MM^\dagger = I = M^\dagger M$, where I is the identity matrix. The eigenvalues of unitary matrices are complex numbers of modulus 1, i.e., they are in the form $\mathrm{e}^{i\vartheta}$, for some real ϑ. This fact characterizes the class of unitary matrices if we restrict to normal matrices. Alternative characterizations of normal and unitary matrices are contained, respectively, in

Proposition 1. [14, Thm. 4.10.3] *A matrix $M \in \mathbf{C}^{m \times m}$ is normal if and only if there exists a unitary matrix $X \in \mathbf{C}^{m \times m}$ such that $M = XDX^\dagger$, where $D = \mathrm{diag}(\nu_1, \nu_2, \ldots, \nu_m)$ is the diagonal matrix of the eigenvalues of M.*

Proposition 2. [14, Thms. 4.7.24, 4.7.14] *A matrix $M \in \mathbf{C}^{m \times m}$ is unitary if and only if:*

(i) its rows are mutually orthonormal vectors;
(ii) $\| xM \| = \| x \|$*, for each vector* $x \in \mathbf{C}^{1\times m}$.

A matrix M is said to be *Hermitian* whenever $M = M^\dagger$. All the eigenvalues of an Hermitian matrix are real. An Hermitian matrix is *positive semidefinite* if and only if all its eigenvalues are non negative. Alternative characterizations are contained in

Proposition 3. [13, Thms. 4.12, 4.8] *An Hermitian matrix* $M \in \mathbf{C}^{m\times m}$ *is positive semidefinite if and only if:*

(i) $xMx^\dagger \geq 0$*, for each vector* $x \in \mathbf{C}^{1\times m}$*;*
(ii) $M = YY^\dagger$*, for some matrix* $Y \in \mathbf{C}^{m\times m}$ *(Cholesky factorization).*

Let $\omega = \mathrm{e}^{i\frac{2\pi}{n}}$ be the n-th root of the unity ($\omega^n = 1$), and define the matrix $W \in \mathbf{C}^{n\times n}$ whose (r,c)-th component is ω^{rc}, for $0 \leq r, c < n$. The *discrete Fourier transform* of a vector $x \in \mathbf{C}^{n\times 1}$ is the vector Wx. The *inverse discrete Fourier transform* of x is the vector $(1/n)W^\dagger x$. It is easy to verify that $(1/n)W^\dagger W = I = W(1/n)W^\dagger$.

Let $f : \mathbf{N} \to \mathbf{C}$ be a periodic function of period n, i.e., for any $k \in \mathbf{N}$, $f(k) = f(k+n)$ holds true. We say that f is n-periodic, for short, and it can be represented by the column vector $(f(0), f(1), \ldots, f(n-1))$. It is well know that f can be expressed as a linear combination of trigonometric functions by using the discrete Fourier transform and its inverse. More precisely:

$$f(k) = \frac{1}{n}\sum_{j=0}^{n-1} F(j)\,\omega^{-kj}, \tag{1}$$

where $(F(0), F(1), \ldots, F(n-1)) = W(f(0), f(1), \ldots, f(n-1))$. We define the support set $\mathrm{Supp}(F) = \{j \in \mathbf{Z}_n \mid F(j) \neq 0\}$. Thus, Equation (1) can be equivalently written as $f(k) = 1/n \sum_{j\in \mathrm{Supp}(F)} F(j)\,\omega^{-kj}$. The reader is referred to, e.g., [1, Chp. 7] for more details on the discrete Fourier transform and its relevance from a computational point of view. Here, we just recall that computing the discrete Fourier transform of n-dimensional vectors requires $\mathcal{O}(n \log n)$ sequential time.

2.2 Difference Cover

The set $\Delta \subseteq \mathbf{Z}_n$ is a *difference cover* for $\mathbf{Z}_n$ if, for each $i \in \mathbf{Z}_n$, there exist two elements $x, \tilde{x} \in \Delta$ such that $i \equiv x - \tilde{x} \,(\bmod\, n)$.

The problem of covering by differences $\mathbf{Z}_n$ is well studied in the literature. Its relevance is also due to connections with some mutual exclusion issues in distributed systems, especially concerning *quorums* [6]. In [20], Wichmann proposes the following sequence of integers, for any $r \geq 0$ (x^r here means $xx\cdots x$ repeated r times): $\sigma = 1^r\,(r{+}1)^1\,(2r{+}1)^r\,(4r{+}3)^{2r+1}\,(2r{+}2)^{r+1}\,1^r$. From σ, construct the set D of $6r{+}4$ integers by setting $a_1 = 0$, and $a_{i+1} = a_i + b_i$ for $1 \leq i \leq 6r{+}3$, where b_i is the i-th element of σ. It is easy to see that $a_{6r+4} = 12r^2{+}18r{+}6$.

The set D has the remarkable property that, for any $1 \leq d \leq 12r^2 + 18r + 6$, there exist $a, b \in D$ such that $d = a - b$.

In [6], Colbourn uses this fact to show that, *for any* $n \leq 24r^2 + 36r + 13$, D *is a difference cover for* $\mathbf{Z}_n$. This is basically due to the fact that, given $d \in \mathbf{Z}_n$, d or $-d$ can be represented by a positive integer less than or equal to $12r^2 + 18r + 6$. Hence, to find a difference cover for any $\mathbf{Z}_n$, it is enough to choose the smallest r satisfying $24r^2 + 36r + 13 \geq n$, and then to construct the corresponding set D with $6r + 4$ elements. Simple arithmetics shows that $6r + 4 \leq \sqrt{1.5n} + 6$, hence:

Theorem 1. [6, Thm. 2.4] *For any* $n \geq 0$, *there exists a difference cover for* $\mathbf{Z}_n$ *of cardinality at most* $\sqrt{1.5n} + 6$.

2.3 Quantum Finite Automata

In this paper, we are interested in *measure-once* quantum finite automata [4,5,17]. Roughly speaking, in this kind of automata, the probability of acceptance is evaluated only at the end of the computation. In the literature, *measure-many* automata are also considered [2,4,5,12], where such an evaluation is taken after each move. Hereafter, the attribute measure-once will always be understood.

The "hardware" of a *one-way quantum finite automaton* is that of a classical finite automaton. Thus, we have an input tape which is scanned by an input head moving one position right at each move[1], plus a finite state control. Formally:

Definition 1. *A one-way quantum finite automaton (1qfa, for short) is a quintuple* $\mathcal{A} = (Q, \Sigma, \pi(0), \delta, F)$, *where*

- $Q = \{s_1, s_2, \ldots, s_q\}$ *is the finite set of states,*
- Σ *is the finite input alphabet,*
- $\pi(0) \in \mathbf{C}^{1 \times n}$, *with* $\| \pi(0) \|^2 = 1$, *is the vector of the initial amplitudes of the states,*
- $F \subseteq Q$ *is the set of accepting states,*
- $\delta : Q \times \Sigma \times Q \to \mathbf{C}$ *is the transition function mapping into the set of complex numbers having square modulus not exceeding* 1; $\delta(s_i, \sigma, s_j)$ *is the amplitude of reaching the state* s_j *from the state* s_i, *upon reading* σ. *The transition function must satisfy the following condition of well-formedness: for any* $\sigma \in \Sigma$ *and* $1 \leq i, j \leq q$, $\sum_{k=1}^{q} \delta(s_i, \sigma, s_k)\, \delta^*(s_j, \sigma, s_k) = 1$ *if* $i = j$, *and* 0 *otherwise.*

It is often useful to express the transition function on every $\sigma \in \Sigma$ as the *transition matrix* $U(\sigma) \in \mathbf{C}^{q \times q}$ whose (i, j)-th entry is the amplitude $\delta(s_i, \sigma, s_j)$. Since δ satisfies the condition of well-formedness, the rows of each $U(\sigma)$ are mutually orthonormal vectors and hence, by Proposition 2(i), $U(\sigma)$*'s are unitary.* The 1qfa $\mathcal{A}$ can thus be represented as a triple $\mathcal{A} = (\pi(0), \{U(\sigma)\}_{\sigma \in \Sigma}, \eta_F)$, where $\eta_F \in \{0, 1\}^{n \times 1}$ is the characteristic vector of the accepting states.

[1] This kind of automata are sometimes referred to as *real time* automata [8,17], stressing the fact that they can never present stationary moves.

Let us briefly discuss how our 1qfa $\mathcal{A}$ works. At any given time t, the *state* of $\mathcal{A}$ is a *superposition* of the states in Q and is represented by a vector $\pi(t)$ of norm 1 in the Hilbert space $l^2(Q)$: the i-th component of $\pi(t)$ is the amplitude of the state s_i. The computation on input $x = x_1x_2 \dots x_n \in \Sigma^*$ starts in the superposition $\pi(0)$. After k steps, i.e., after reading the first k input symbols, the state of $\mathcal{A}$ is the superposition $\pi(k) = \pi(0)U(x_1)U(x_2)\cdots U(x_k)$. Since $\| \pi(0) \| = 1$ and $U(x_i)$'s are unitary matrices, Proposition 2(ii) ensures that $\| \pi(k) \| = 1$. When $\mathcal{A}$ enters the final superposition $\pi(n) = \pi(0)\prod_{i=1}^{n} U(x_i)$, we *observe* $\mathcal{A}$ by the *standard observable* $\mathcal{O} = \{l^2(F), l^2(Q \setminus F)\}$. $\mathcal{O}$ is the decomposition of $l^2(Q)$ into the two subspaces spanned by the accepting and nonaccepting states, respectively. The probability of accepting x is given by the square norm of the projection of $\pi(n)$ onto $l^2(F)$. Formally, $p_{acc}(x) = \sum_{\{j \mid (\eta_F)_j=1\}} |(\pi(0)\prod_{i=1}^{n} U(x_i))_j|^2$, where the subscript j denotes the j-th vector component.

A *stochastic event* is a function $p : \Sigma^* \to [0,1]$. The stochastic event *induced or defined by the 1qfa* $\mathcal{A}$ is the function $p_{\mathcal{A}} : \Sigma^* \to [0,1]$ defined, for any $x \in \Sigma^*$, as $p_{\mathcal{A}}(x) = p_{acc}(x)$. The *language accepted by* $\mathcal{A}$ *with cut point* $\lambda \geq 1/2$ is the set $L_{\mathcal{A},\lambda} = \{x \in \Sigma^* \mid p_{\mathcal{A}}(x) > \lambda\}$. A language L is said to be accepted by $\mathcal{A}$ *with isolated cut point* λ, if there exists $\varepsilon > 0$ such that, for any $x \in L$ ($x \notin L$), we have $p_{\mathcal{A}}(x) \geq \lambda + \varepsilon$ ($\leq \lambda - \varepsilon$).

A 1qfa $\mathcal{A}$ is *unary* if $|\Sigma| = 1$. In this case, we let $\Sigma = \{1\}$, and we can write $\mathcal{A} = (\pi(0), U, \eta_F)$ since we have a unique transition matrix U. With a slight abuse of notation, we write k for the input string 1^k. The probability of accepting k now writes as

$$p_{acc}(k) = \sum_{\{j \mid (\eta_F)_j=1\}} |(\pi(0)U^k)_j|^2. \tag{2}$$

The stochastic event *induced or defined by the unary automaton* $\mathcal{A}$ is the function $p_{\mathcal{A}} : \mathbf{N} \to [0,1]$, with $p_{\mathcal{A}}(k) = p_{acc}(k)$.

A stochastic event $p : \mathbf{N} \to [0,1]$ is said to be *n-periodic* if it is an n-periodic function. A unary language is a set $L \subseteq 1^*$. L is *n-periodic* if there exists a set $S \subseteq \mathbf{Z}_n$ such that $L = \{k \in \mathbf{N} \mid (k \bmod n) \in S\}$.

3 Synthesis of 1qfa's from Periodic Events

The first problem we shall be dealing with is the synthesis of 1qfa's inducing given periodic stochastic events. As a matter of fact, we will consider a relaxed version of this problem where, given a periodic event p, we aim to obtain a 1qfa inducing $ap + b$, for some reals $a > 0$, $b \geq 0$ satisfying $a + b \leq 1$.

If p is taken to be n-periodic, then it can be specified as input for the problem by providing the column vector $(p(0), p(1), \dots, p(n-1))$. Thus, formally we state:

Synthesis from events (SynE)

⋆ Input: An n-periodic stochastic event $(p(0), p(1), \dots, p(n-1))$.

⋆ Output: A 1qfa $\mathcal{A}$ inducing the event $ap + b$, for some reals $a > 0$, $b \geq 0$, with $a + b \leq 1$.

Let us now prepare some tools to approach the problem. First of all, we point out some closure properties on the stochastic events induced by 1qfa's:

Proposition 4. *Let p and $\tilde{p}$ be two stochastic events induced, respectively, by the 1qfa's $\mathcal{A} = (\pi, \{U(\sigma)\}_{\sigma\in\Sigma}, \eta)$ and $\tilde{\mathcal{A}} = (\tilde{\pi}, \{\tilde{U}(\sigma)\}_{\sigma\in\Sigma}, \tilde{\eta})$. Then there exist 1qfa's inducing the stochastic events $1-p$ and $\alpha p + \beta\tilde{p}$, where α and β are non negative reals such that $\alpha + \beta = 1$.*

Proof. It is easy to see that the event $1-p$ is induced by the 1qfa $\overline{\mathcal{A}} = (\pi, \{U(\sigma)\}_{\sigma\in\Sigma}, \neg\eta)$, where $\neg\eta$ is the bitwise negation of η, while the event $\alpha p+\beta\tilde{p}$ is induced by the 1qfa $\alpha\mathcal{A}+\beta\tilde{\mathcal{A}} = ((\sqrt{\alpha}\pi, \sqrt{\beta}\tilde{\pi}), \{U(\sigma) \oplus \tilde{U}(\sigma)\}_{\sigma\in\Sigma}, (\eta, \tilde{\eta}))$. □

At this point, a quick comment on the relevance of SynE is in order. From a language recognition point of view, the events p and $ap+b$ are equivalent in the following sense: Suppose we have a unary 1qfa $\mathcal{A}$ accepting the language $L_{\mathcal{A},\lambda}$ and suppose we are able to construct a unary 1qfa $\mathcal{A}_1$ inducing the event $ap_{\mathcal{A}}+b$. By setting $\lambda_1 = a\lambda + b$, it is easy to see that $L_{\mathcal{A}_1,\lambda_1} = L_{\mathcal{A},\lambda}$. Here, a technical detail should be considered. As stated in Section 2.3, we must require that $\lambda_1 \geq 1/2$. If the opposite is true, by Proposition 4, we construct the 1qfa $\mathcal{A}_2 = \frac{1}{2}\mathcal{A}_1 + \frac{1}{2}\mathcal{U}$, where $\mathcal{U}$ is a single state 1qfa realizing the event $p_{\mathcal{U}}(x) = 1$, for any $x \in \Sigma^*$. We have $p_{\mathcal{A}_2} = 1/2(ap_{\mathcal{A}} + b) + 1/2$ and, by setting $\lambda_2 = (1/2)\lambda_1 + 1/2$, one easily get $L_{\mathcal{A}_2,\lambda_2} = L_{\mathcal{A},\lambda}$. In other words, solving SynE enables us to obtain unary 1qfa's accepting unary languages defined by a precise stochastic event.

Now, we show that the stochastic events induced by unary 1qfa's have a sort of *normal form*. In what follows, we denote by M_{ij} the (i,j)-th entry of the matrix M and by x_i the i-th component of the vector x.

Proposition 5. *Let p be a stochastic event induced by a unary 1qfa $\mathcal{A}=(\pi, U, \eta)$ with q states. Then, for any $k \in \mathbf{N}$, $p(k) = \sum_{1\leq s,t\leq q} \mathrm{e}^{ik(\vartheta_s-\vartheta_t)} B_{st}$, where B is an Hermitian positive semidefinite matrix.*

Proof. From Equation (2) in Section 2.3, the stochastic event induced by $\mathcal{A}$ writes as $p(k) = \sum_{\{j \mid \eta_j=1\}} |(\pi U^k)_j|^2$. Since U is a unitary matrix, by Proposition 1, we can write $U = X\,\mathrm{diag}(\mathrm{e}^{i\vartheta_1}, \mathrm{e}^{i\vartheta_2}, \ldots, \mathrm{e}^{i\vartheta_q})\,X^\dagger$, where X is a unitary matrix and $\mathrm{e}^{i\vartheta}$'s are the norm 1 eigenvalues of U. Thus,

$$U^k = X\,\mathrm{diag}(\mathrm{e}^{ik\vartheta_1}, \mathrm{e}^{ik\vartheta_2}, \ldots, \mathrm{e}^{ik\vartheta_q})\,X^\dagger,$$

and

$$p(k) = \sum_{\{j \mid \eta_j=1\}} |(\pi X\,\mathrm{diag}(\mathrm{e}^{ik\vartheta_1}, \mathrm{e}^{ik\vartheta_2}, \ldots, \mathrm{e}^{ik\vartheta_q})\,X^\dagger)_j|^2 \tag{3}$$

By letting $\xi = \pi X$ and substituting in (3), we get

$$p(k) = \sum_{\{j \mid \eta_j=1\}} ((\xi_1\mathrm{e}^{ik\vartheta_1}, \ldots, \xi_q\mathrm{e}^{ik\vartheta_q})X^\dagger)_j\,((\xi_1\mathrm{e}^{ik\vartheta_1}, \ldots, \xi_q\mathrm{e}^{ik\vartheta_q})X^\dagger)^*_j$$

$$= \sum_{\{j \mid \eta_j=1\}} \left(\sum_{s=1}^{q} \xi_s \mathrm{e}^{ik\vartheta_s} X^\dagger_{sj} \right) \left(\sum_{t=1}^{q} \xi^*_t \mathrm{e}^{-ik\vartheta_t} (X^\dagger_{tj})^* \right)$$

$$= \sum_{1\le s,t\le q} \mathrm{e}^{ik(\vartheta_s-\vartheta_t)} \sum_{\{j \mid \eta_j=1\}} \xi_s X^\dagger_{sj} (\xi_t X^\dagger_{tj})^*.$$

Now, define the matrix B as

$$B_{st} = \sum_{\{j \mid \eta_j=1\}} \xi_s X^\dagger_{sj} (\xi_t X^\dagger_{tj})^*,$$

for $1 \le s,t \le q$. It is easy to verify that $B = B^\dagger$, and hence B is Hermitian. To prove that B is positive semidefinite, by Proposition 3(i), it is enough to show that $xBx^\dagger \ge 0$, for any $x \in \mathbf{C}^{1\times q}$:

$$xBx^\dagger = \sum_{1\le s,t\le q} x_s \left(\sum_{\{j \mid \eta_j=1\}} \xi_s X^\dagger_{sj} (\xi_t X^\dagger_{tj})^* \right) x^*_t$$

$$= \sum_{\{j \mid \eta_j=1\}} \left(\sum_{s=1}^{q} x_s \xi_s X^\dagger_{sj} \right) \left(\sum_{t=1}^{q} x_t \xi_t X^\dagger_{tj} \right)^*$$

$$= \sum_{\{j \mid \eta_j=1\}} \left| \sum_{s=1}^{q} x_s \xi_s X^\dagger_{sj} \right|^2 \ge 0.$$

□

We are now ready to concentrate on SynE. Recall that our aim is to build a unary 1qfa $\mathcal{A}$ which induces $ap + b$, for some reals $a > 0,\ b \ge 0$, with $a + b \le 1$, and an n-periodic stochastic event $p : \mathbf{N} \to [0,1]$ given as input.

We start by observing that the event p is an n-periodic function and hence, according to Equation (1) in Section 2.1, it expands as

$$p(k) = \frac{1}{n} \sum_{j=0}^{n-1} P(j)\, \omega^{-kj}, \tag{4}$$

for $(P(0), P(1), \ldots, P(n-1)) = W(p(0), p(1), \ldots, p(n-1))$. On the other hand, in the light of Proposition 5, to be induced by a unary 1qfa, the event p must have the form

$$p(k) = \sum_{1\le s,t\le q} \mathrm{e}^{ik(\vartheta_s-\vartheta_t)} B_{st}, \tag{5}$$

for some real ϑ's and an Hermitian positive semidefinite matrix B. These observations lead us to design an algorithm consisting of two parts. In the first part, we compute ϑ's and B so that Equation (5) exactly reproduces Equation (4). In the second part, we construct from such ϑ's and B a well formed 1qfa inducing $ap + b$.

First Part of the Algorithm

★ Input $(p(0), p(1), \ldots, p(n-1))$

step 1 Compute $(P(0), P(1), \ldots, P(n-1)) = W(p(0), p(1), \ldots, p(n-1))$, the discrete Fourier transform, and let $\mathrm{Supp}(P) = \{j \in \mathbf{Z}_n \mid P(j) \neq 0\}$.

step 2 Find a difference cover $\Delta = \{a_1, a_2, \ldots, a_q\}$ for $\mathbf{Z}_n$.

step 3 For each $1 \leq t \leq q$, let $\vartheta_t = -\frac{2\pi}{n} a_t$.

step 4 For each $j \in \mathrm{Supp}(P)$, let

$$N(j) = |\{(a_s, a_t) \mid a_s, a_t \in \Delta \text{ and } j \equiv a_s - a_t \,(\operatorname{mod} n)\}|,$$

and, for $1 \leq s, t \leq q$, compute

$$B_{st} = \begin{cases} \frac{1}{n}\frac{P(j)}{N(j)} & \text{if } j \in \mathrm{Supp}(P) \text{ and } j \equiv a_s - a_t \,(\operatorname{mod} n) \\ 0 & \text{otherwise.} \end{cases}$$

It is easy to verify that $B \in \mathbf{C}^{q \times q}$ *is an Hermitian matrix:* to see that $B_{st} = {B_{ts}}^*$, it is enough to notice that $P(j) = P^*(-j \operatorname{mod} n)$ and $N(j) = N(-j \operatorname{mod} n)$, for each $j \in \mathbf{Z}_n$. By plugging ϑ's obtained at step 3 and B obtained at step 4 into Equation (5), we get exactly $p(k)$ as in Equation (4).

Now comes the second part of the algorithm. We show how to build a 1qfa $\mathcal{A} = (\pi, U, \eta)$ inducing $ap+b$ from ϑ's and B computed in the first part. There are two ways of reconstructing $\mathcal{A}$, depending on whether B is positive semidefinite or not.

Second Part of the Algorithm

step 5.a If *B is positive semidefinite* Then

- Find a matrix $Y \in \mathbf{C}^{q \times q}$ satisfying $YY^\dagger = B$. Such Y exists by Proposition 3(ii).
- Construct the $2q \times 2q$ matrix $M = \left(\begin{array}{c|c} Y & E \\ \hline \multicolumn{2}{c}{F} \end{array}\right)$, where the row vectors $M_i \in \mathbf{C}^{1 \times 2q}$ are mutually orthogonal. To get this, the first q rows of M can be computed by setting the lower triangular matrix $E \in \mathbf{C}^{q \times q}$ as

$$E_{ij} = \begin{cases} 1 & \text{if } i = j \\ -\langle Y_i, Y_j \rangle - \sum_{k=1}^{j-1} E_{ik} {E_{jk}}^* & \text{if } i > j \\ 0 & \text{otherwise,} \end{cases}$$

  where Y_i is the i-th row of Y. At this point, we can take the q rows of F as an orthogonal basis of the subspace which is orthogonal to that spanned by the vectors $M_1, M_2, \ldots, M_q$. Such a task can be performed by using standard tools in linear algebra (see, e.g., [13]).
- Define $X_i^\dagger = M_i / \| M_i \|$ the i-th row of the $2q \times 2q$ unitary matrix $X^\dagger$. Unitarity of $X^\dagger$ comes from the fact that we are constructing its rows as mutually orthonormal vectors, and hence Proposition 2(i) applies. Define also the vectors $\tilde{\xi} \in \mathbf{C}^{1 \times 2q}$ and $\eta \in \mathbf{C}^{2q \times 1}$ as

$$\tilde{\xi}_i = \begin{cases} \| M_i \| & \text{for } i \leq q \\ 0 & \text{otherwise} \end{cases} \quad \text{and} \quad \eta_i = \begin{cases} 1 & \text{for } i \leq q \\ 0 & \text{otherwise.} \end{cases}$$

- Compute the $\mathbf{C}^{1\times 2q}$ vector $\pi = \xi X^{\dagger}$, where $\xi = \tilde{\xi}/\|\tilde{\xi}\|$.

★ OUTPUT $\mathcal{A} = (\pi, XDX^{\dagger}, \eta)$, where $D = \text{diag}(\mathrm{e}^{i\vartheta_1}, \ldots, \mathrm{e}^{i\vartheta_q}, \underbrace{1, \ldots, 1}_{q-times})$. Output also $a = \frac{1}{||\tilde{\xi}||^2}$.

Fact. It is not hard to see that $\mathcal{A}$ is a well formed 1qfa: First, notice that π is obtained by multiplying the norm 1 vector ξ by the unitary matrix $X^{\dagger}$. Hence, by Proposition 2(ii), $\|\pi\| = 1$. Next, notice that the transition matrix $XDX^{\dagger}$ is unitary, being the product of unitary matrices. $\mathcal{A}$ has $2q$ states and it is easily seen to induce the event ap, with $0 < a = \frac{1}{||\tilde{\xi}||^2} \leq 1$.

STEP 5.B IF *B is not positive semidefinite* THEN

- Find two Hermitian positive semidefinite matrices $G, H \in \mathbf{C}^{q\times q}$ such that $B = G - H$.

  These two matrices can be constructed as follows. Since B is an Hermitian matrix, by Proposition 1, we can write $B = X\text{diag}(\nu_1, \nu_2, \ldots, \nu_q)X^{\dagger}$, where ν's are the real eigenvalues of B and X is a unitary matrix. Define $D^{+} = \text{diag}(\upsilon_1, \upsilon_2, \ldots, \upsilon_q)$, where $\upsilon_i = \nu_i$ if $\nu_i > 0$, and 0 otherwise. Set $D^{-} = D^{+} - D$. Let $G = XD^{+}X^{\dagger}$ and $H = XD^{-}X^{\dagger}$. It is easy to see that $B = G - H$, and that both G and H are Hermitian. Moreover, one can easily verify that, for each $x \in \mathbf{C}^{1\times q}$, both $xGx^{\dagger} \geq 0$ and $xHx^{\dagger} \geq 0$ hold true. Hence, by Proposition 3(i), G and H are positive semidefinite.
- Perform STEP 5.A by having as input G and H. This yields two $2q$-state 1qfa's $\mathcal{A}_1$ and $\mathcal{A}_2$ inducing, respectively, the events a_1p_1 and a_2p_2, with $0 < a_1, a_2 \leq 1$ and $p_1 - p_2 = p$.
- Let $\mathcal{U}$ be the 1-state 1qfa inducing the event $p_{\mathcal{U}}(k) = 1$, for any $k \geq 0$. Use Proposition 4 to construct the following 1qfa's:

  IF $a_1 \leq a_2$ THEN

  - construct $\mathcal{A}_3 = \frac{a_1}{a_2}\mathcal{A}_2 + (1 - \frac{a_1}{a_2})\mathcal{U}$. $\mathcal{A}_3$ has $2q+1$ states and induces the event $a_1p_2 + (1 - \frac{a_1}{a_2})$.
  - construct $\overline{\mathcal{A}_3}$, i.e., the $(2q{+}1)$-state 1qfa inducing $1 - p_{\mathcal{A}_3} = \frac{a_1}{a_2} - a_1p_2$.

  ★ OUTPUT $\mathcal{A}_4 = \frac{1}{2}\mathcal{A}_1 + \frac{1}{2}\overline{\mathcal{A}_3}$. Output also $a = \frac{a_1}{2}$ and $b = \frac{a_1}{2a_2}$.

  IF $a_1 > a_2$ THEN

  - construct $\mathcal{A}_3 = \frac{a_2}{a_1}\mathcal{A}_1 + (1 - \frac{a_2}{a_1})\mathcal{U}$. $\mathcal{A}_3$ has $2q+1$ states and induces the event $a_2p_1 + (1 - \frac{a_2}{a_1})$.
  - construct $\overline{\mathcal{A}_2}$, i.e., the $2q$-state 1qfa inducing $1 - a_2p_2$.

  ★ OUTPUT $\mathcal{A}_4 = \frac{1}{2}\mathcal{A}_3 + \frac{1}{2}\overline{\mathcal{A}_2}$. Output also $a = \frac{a_2}{2}$ and $b = 1 - \frac{a_2}{2a_1}$.

Fact. It is easy to see that, in both cases, $\mathcal{A}_4$ is a $(4q + 1)$-state 1qfa inducing the event $ap + b$, for $a, b > 0$, with $a + b \leq 1$.

In conclusion, the above algorithm provides a constructive proof of the following

Theorem 2. *For any n-periodic event p, there exists a unary 1qfa with at most $2\sqrt{6n} + 25$ states inducing $ap + b$, for some reals $a > 0$, $b \geq 0$, with $a + b \leq 1$.*

Proof. We use our algorithm to construct a 1qfa $\mathcal{A}$ for $ap+b$. As one may easily see, $\mathcal{A}$ has at most $4q+1$ states, where q is the cardinality of a difference cover for $\mathbf{Z}_n$. By Theorem 1, q is bounded above by $\sqrt{1.5n}+6$, whence the result. □

We end with a quick evaluation of the complexity of our algorithm.

First Part of the Algorithm: Computing the discrete Fourier transform at step 1 requires $\mathcal{O}(n \log n)$ time, as observed in Section 2.1. The operations at steps 3, 4 are easily seen to require polynomial time. Finally, at step 2, we can construct a difference cover for $\mathbf{Z}_n$ in polynomial time by using Wichmann's sequence, as addressed in Section 2.2.

Second Part of the Algorithm: The hardest tasks at steps 5.a, 5.b are basically to solve some problems from linear algebra, such as: Cholesky factorization, computation of basis for orthogonal subspaces, decomposition of Hermitian matrices. For all these tasks, polynomial time algorithms can be obtained from the literature (see, e.g., [11]).

This enables us to conclude that a $(2\sqrt{6n}+25)$-state 1qfa for the event $ap+b$ can be constructed in polynomial time.

4 Synthesis of 1qfa's from Periodic Languages

We now focus on accepting periodic languages, i.e., unary languages in the form $L = \{k \in \mathbf{N} \mid (k \bmod n) \in S\}$, for a fixed $S \subseteq \mathbf{Z}_n$. As recalled in the introduction, recognizing n-periodic languages by 1dfa's takes at least n states. Moreover, in some cases, e.g. when n is a prime, even using 1pfa's (or also two-way nondeterminism) does not help in saving states.

By using the results in the previous section, we are always able to design 1qfa's with $\mathcal{O}(\sqrt{n})$ states and isolated cut point for n-periodic languages, as proved in the following

Theorem 3. *Any n-periodic language can be accepted with* isolated cut point *on a 1qfa having no more than $2\sqrt{6n}+26$ states.*

Proof. With each n-periodic language $L = \{k \in \mathbf{N} \mid (k \bmod n) \in S\}$, for some $S \subseteq \mathbf{Z}_n$, we can associate the n-periodic event p defined, for each $k \geq 0$, as $p(k) = 1$ if $(k \bmod n) \in S$, and 0 otherwise. By Theorem 2, there exists a 1qfa $\mathcal{A}$, with no more than $2\sqrt{6n}+25$, that induces $ap+b$, for some reals $a > 0$, $b \geq 0$, $a+b \leq 1$. If $b + a/2 \geq 1/2$, we let $\lambda = b + a/2$ and $\varepsilon = a/2$. Otherwise, we construct the automaton $\mathcal{A}_1 = \frac{1}{2}\mathcal{A} + \frac{1}{2}\mathcal{U}$ by adding one more state to the states of $\mathcal{A}$, and we let $\lambda = b/2 + a/4 + 1/2$ and $\varepsilon = a/4$. It is easy to verify that $\mathcal{A}$ or $\mathcal{A}_1$ accepts L with cut point $\lambda \geq 1/2$ isolated by ε. □

5 Some Concluding Remarks and Open Problems

In this work, we have provided a polynomial time algorithm for constructing small 1qfa's that induce periodic stochastic events or accept periodic languages. More precisely, we have shown that any n-periodic event can be induced by a 1qfa with at most $2\sqrt{6n}+25$ states, while any n-periodic language is accepted with no more than $2\sqrt{6n}+26$ states.

These results point out that, on a wide class of problems, 1qfa's are quadratically more succinct than corresponding deterministic automata[2]. In fact, it is well known that any 1dfa recognizing an n-periodic language must have at least n states.

More can be said even on the question quantum vs. probabilistic automata. As proved in [4], for any given prime p, the language L_p requires at least p states to be accepted on 1pfa's with isolated cut point. This clearly implies the same state lower bound to induce p-periodic events by 1pfa's. Our results show that 1qfa's can be built that induce p-periodic events using only $\mathcal{O}(\sqrt{p})$ states. Moreover, we have used this fact to accept p-periodic languages with isolated cut point on $\mathcal{O}(\sqrt{p})$-state 1qfa's.

It should be noticed that, by using *ad hoc* techniques on *ad hoc* problems, we can sometimes obtain even more succinct 1qfa's. For instance, in [4], a $\mathcal{O}(\log p)$-state 1qfa for L_p is exhibited. However, due to its generality, we cannot expect our method to be so "state-saving". Nevertheless, it can be used as a tool to generate small quantum machines that can eventually serve as starting points for further refinements. Yet, we feel that our method could be of help in approaching open questions on quantum finite automata, some of which are quickly suggested hereafter:

- How to construct 1qfa's exactly inducing given periodic stochastic events?
- How to obtain Monte Carlo 1qfa's (a more "reliable" version of isolated cut point 1qfa's, see [8]) accepting periodic languages?
- What about the size of the resulting 1qfa's?
- What about the size of minimal 1qfa's inducing periodic stochastic events or accepting periodic languages?

Acknowledgments. The authors wish to thank Alberto Bertoni for stimulating discussions, and anonymous referees for their comments.

References

1. A. Aho, J. Hopcroft and J. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974. 126
2. A. Ambainis, A. Ķikusts and M. Valdats. On the Class of Languages Recognizable by 1-way Quantum Finite Automata. In *Proc. 18th Annual Symposium on Theoretical Aspects of Computer Science,* Lecture Notes in Computer Science 2010, Springer, pp. 305–316, 2001. 127

[2] A similar quadratic decrease is proved in [10] for a particular binary language.

3. A. Ambainis and J. Watrous. Two-way Finite Automata with Quantum and Classical States. Technical Report `quant-ph/9911009`, 1999. 123
4. A. Ambainis and R. Freivalds. 1-way Quantum Finite Automata: Strengths, Weaknesses and Generalizations. In *Proc. 39th Annual Symposium on Foundations of Computer Science,* pp. 332–342, 1998. 123, 124, 125, 127, 134
5. A. Brodsky and N. Pippenger. *Characterizations of 1-Way Quantum Finite Automata.* Technical Report, Department of Computer Science, University of British Columbia, TR-99-03, 2000 Revised. 123, 124, 127
6. C. Colbourn and A. Ling. Quorums from difference covers. *Information Processing Letters,* 75:9–12, 2000. 126, 127
7. L. Grover. A Fast Quantum Mechanical Algorithm for Database Search. In *Proc. 28th ACM Symposium on Theory of Computing,* pp. 212–219, 1996. 123
8. J. Gruska. Descriptional complexity issues in quantum computing. *J. Automata, Languages and Combinatorics,* 5:191–218, 2000. 124, 127, 134
9. T. Jiang, E. McDowell and B. Ravikumar. The structure and complexity of minimal nfa's over a unary alphabet. *Int. J. Found. Comp. Sc.,* 2:163–182, 1991. 124
10. A. Ķikusts. A Small 1-way Quantum Finite Automaton. Technical Report `quant-ph/9810065`, 1998. 124, 134
11. M. Kohn. *Practical Numerical Methods: Algorithms and Programs.* The Macmillan Company, 1987. 133
12. A. Kondacs and J. Watrous. On the Power of Quantum Finite State Automata. In *Proc. 38th Annual Symposium on Foundations of Computer Science,* pp. 66–75, 1997. 123, 127
13. M. Marcus and H. Minc. *Introduction to Linear Algebra.* The Macmillan Company, 1965. Reprinted by Dover, 1988. 125, 126, 131
14. M. Marcus and H. Minc. *A Survey of Matrix Theory and Matrix Inequalities.* Prindle, Weber & Schmidt, 1964. Reprinted by Dover, 1992. 125
15. C. Mereghetti and G. Pighizzini. Two-Way Automata Simulations and Unary Languages. *J. Automata, Languages and Combinatorics,* 5:287-300, 2000. 124, 125
16. C. Mereghetti, B. Palano and G. Pighizzini. On the succinctness of deterministic, nondeterministic, probabilistic and quantum finite automata. In *Pre-Proc. Descriptional Complexity of Automata, Grammars and Related Structures (DCAGRS 2001).* To appear. 125
17. C. Moore and J. Crutchfield. Quantum automata and quantum grammars. *Theoretical Computer Science,* 237:275–306, 2000. 123, 127
18. P. Shor. Algorithms for Quantum Computation: Discrete Logarithms and Factoring. In *Proc. 35th Annual Symposium on Foundations of Computer Science,* pp. 124–134, 1994. 123
19. P. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM J. Computing*, 26:1484–1509, 1997. 123
20. B. Wichmann. A note on restricted difference bases. *J. London Math. Soc.,* 38:465–466, 1963. 126

P Systems with Gemmation of Mobile Membranes

Daniela Besozzi[1], Claudio Zandron[2], Giancarlo Mauri[2], and
Nicoletta Sabadini[1]

[1] Università degli Studi dell'Insubria
Via Valleggio 11, 22100 Como, Italy
{daniela.besozzi,nicoletta.sabadini}@uninsubria.it

[2] Università degli Studi di Milano-Bicocca, Dipartimento di Informatica, Sistemistica e Comunicazione
Via Bicocca degli Arcimboldi 8, 20136 Milano, Italy
{zandron,mauri}@disco.unimib.it

Abstract. P systems are computational models inspired by some biological features of the structure and the functioning of real cells. In this paper we introduce a new kind of communication between membranes, based upon the natural budding of vesicles in a cell. We define the operations of gemmation and fusion of mobile membranes, and we use membrane structures and rules over strings of biological inspiration only. We prove that P systems of this type can generate all recursively enumerable languages and, moreover, the Hamiltonian Path Problem can be solved in a quadratic time. Some open problems are also formulated.

1 Introduction

The P systems were recently introduced in [7] as a class of distributed parallel computing devices of a biochemical type. The basic model consists of a membrane structure composed by several cell-membranes, hierarchically embedded in a main membrane called the skin membrane. The membranes delimit regions and can contain objects, which evolve according to given evolution rules associated with the regions. Such rules are applied in a nondeterministic and maximally parallel manner: at each step, all the objects which can evolve should evolve.
A computation device is obtained: we start from an initial configuration and we let the system evolve. A computation halts when no further rule can be applied. The objects in a specified output membrane (or expelled through the skin membrane) are the result of the computation.

Many basic variants are considered in [7], [8], and [10]. Further extensions are defined in [6], [9], [15], where polynomial (or even linear) time solution for some NP-complete problems are proposed.
A survey and an up-to-date bibliography can be found at the web address http://bioinformatics.bio.disco.unimib.it/psystems.

Up to now, in all the variants of P systems only the direct communication of objects through membranes has been considered: an object can exit the membrane where it is placed and enter the upper level region, or it can enter a lower

A. Restivo, S. Ronchi Della Rocca, L. Roversi (Eds.): ICTCS 2001, LNCS 2202, pp. 136–153, 2001.

level region. Such communications are defined by means of target indications *in/out* attached to the evolution rules of the system. The aim of the present work is to introduce a new kind of communication between membranes and to keep the definition of P systems closer to the real structure of cells.

The notion of mobility was first introduced into P systems area in [11], where a link was established between P systems and Ambient Calculus (see, e.g., [1]). In that paper, the creation of travelling cells was proposed in order to get direct and secure communications of objects between non-adjacent membranes, both provided with a common shared key. The passage and the possible consequent modification of objects along the path between the two membranes was thus avoided.

In this paper we do not consider any security feature, instead we want to introduce a different definition of *mobile membranes*, based upon a biological process of alive cells. Cellular membranes are selectively permeable to many small substances as water and ions, but not to bigger ones as proteins (see, e.g., [14]). Such substances are communicated inside or outside the cell by means of vesicles, which are little parts of a membrane, encased on their cytosolic face by a specific protein that causes their budding from the membrane. When the vesicle fuses with its target membrane, the carried proteins are introduced inside it, where they can be modified by other chemical reactions. Many cellular compartments use this kind of communication, in particular this is the case of the Golgi apparatus ([12]), a stack of distinct elementary membranes (i.e. membranes without other membranes inside) where, in sequence, many proteins are stepwise modified and then sent to another Golgi-region. Specifically, only the substances that have completed their “maturation path” inside the current region can be communicated by a vesicle to the next destination. For example, only the proteins that have reached their exact folding can enter a budding vesicle, otherwise they will remain inside the current Golgi-region.

In order to simulate all these natural features, we consider P systems with simple membrane structures (the skin membrane can only contain elementary membranes) and with operations on strings of a biochemical inspiration, such as mutation, replication and splitting rules. Moreover, we define a *meta-priority* between the set of classical evolution rules and the set of *pre-dynamical rules*, which are the rules that give rise to the *gemmation* of mobile membranes (that is, the budding of vesicles in the cell). The meta-priority is needed to the aim of simulating the completion of the maturation path of an object. After a pre-dynamical rule has been used, the phases of gemmation and fusion of mobile membranes take place. In particular, the output of the system is due to the fusion of a mobile membrane with the skin membrane: this process causes the release of the objects outside the system. The set of strings that exit the skin membrane is the language generated by the system, as usual in P systems with external output ([10]).

We show that the obtained system is able to generate every recursively enumerable language and that it can be used to solve NP-complete problems in polynomial time. In particular, we prove this by showing how to solve the Hamiltonian Path Problem in a quadratic time with respect to the input length. A solution to this problem was also proposed, for a different variant of P systems, in [2].

2 Definition

We refer to [13] for formal language theory prerequisites, we only mention here that we denote by V^* the free monoid generated by the alphabet V under the operation of concatenation. The empty string is denoted by λ and $V^+ = V^* - \{\lambda\}$ is the set of non-empty strings over V.

A membrane structure μ is a construct consisting of several membranes hierarchically embedded in a unique membrane, called a *skin membrane*. We identify a membrane structure with a string of correctly matching square parentheses, placed in a unique pair of matching parentheses; each pair of matching parentheses corresponds to a membrane. We can also associate a tree with the structure, in a natural way; the height of the tree is the *depth* of the structure itself. In order to stay close to the structure of a real cell, in this paper we will consider only membrane structures of depth 2, with the skin membrane always labelled with 0 and the inner membranes injectively labelled with numbers in the set $\{1, \ldots, n\}$.

Each membrane identifies a region, delimited by it and the membrane immediately inside it. If we place multisets of objects in the region from a specified finite set V, we get a super-cell. A super-cell system (or P-system) is a super-cell provided with evolution rules for its objects.

We will work with string-objects, so with every region $i = 0, 1, \ldots, n$ of μ we associate a multiset of finite support over V^*, that is a map $M_i : V^* \to N$ where $M_i = \{(x_1, M_i(x_1)), \ldots, (x_p, M_i(x_p))\}$, for some $x_k \in V^+$ such that $M_i(x_k) > 0 \; \forall k = 1, \ldots, p$.
We will use three types of operations on strings over V, which were first considered in [2]:

1. *Mutation*: a *mutation rule* is a context-free rewriting rule $r_m : a \to u$, where $a \in V$ and $u \in V^*$. For strings $w_1, w_2 \in V^+$ we write $w_1 \Rightarrow_{r_m} w_2$ if $w_1 = x_1 a x_2$ and $w_2 = x_1 u x_2$, for some x_1, $x_2 \in V^*$.
2. *Replication*: if $a \in V$ and $u_1, u_2 \in V^+$, then $r_r : a \to u_1 \parallel u_2$ is called a *replication rule*. For strings $w_1, w_2, w_3 \in V^+$ we write $w_1 \Rightarrow_{r_r} (w_2, w_3)$ (and we say that w_1 is replicated with respect to rule r_r) if $w_1 = x_1 a x_2$, $w_2 = x_1 u_1 x_2$ and $w_3 = x_1 u_2 x_2$ for some $x_1, x_2 \in V^*$.
3. *Splitting*: if $a \in V$ and $u_1, u_2 \in V^+$, then $r_s : a \to u_1 : u_2$ is called a *splitting rule*. For strings $w_1, w_2, w_3 \in V^+$ we write $w_1 \Rightarrow_{r_s} (w_2, w_3)$ (and we say that w_1 is split with respect to rule r_s) if $w_1 = x_1 a x_2$, $w_2 = x_1 u_1$ and $w_3 = u_2 x_2$ for some $x_1, x_2 \in V^*$.

Note that only replication and splitting rules increase the number of strings, while mutation rules can delete symbols. When using such operations in P systems, we will add target indications to rules, indicating the regions where the

resulting strings will be communicated at the next step.
With each region $i = 0, 1, \ldots, n$ of the membrane structure we associate two distinct sets of rules:

1. A set C_i of *classical* evolution rules, that is a set of mutation, replication and splitting rules of the form $a \to \alpha$, with $\alpha \in \{(u, tar), (u_1 \| u_2; tar_1, tar_2), (u_1 : u_2; tar_1, tar_2)\}$, where u, u_1, u_2 are strings over V (as defined above) and $tar, tar_1, tar_2 \in \{here, out\}$ if $i = 1, \ldots, n$, $tar, tar_1, tar_2 \in \{here, out\} \cup \{in_1, \ldots, in_n\}$ if $i = 0$.

2. A set D_i of *pre-dynamical* evolution rules, that is a set of mutation, replication and splitting rules of the form $a \to \alpha'$, with $\alpha' \in \{(u, here), (u_1 \parallel u_2; here, here), (u_1 : u_2; here, here)\}$, where, following the above notations for strings and substrings, it holds that $x_1 = \lambda$ (or $x_2 = \lambda$), u and at least one string between u_1, u_2 belong to $\{@_j\} \cdot V^*$ (respectively $V^* \cdot \{@_j\}$), where $@_j$ is a special symbol not in V and $j \in \{0, 1, \ldots, n\}, j \neq i$.
   We point out that a pre-dynamical rule can introduce the special symbol $@_j$ *only* at the ends of the string, that is the reason why we ask for x_1 or x_2 to be empty words. Note that if $i = 0$, then the set D_0 is empty, that is no pre-dynamical rule is defined inside the skin membrane.

Once the symbol $@_j$ as been introduced by a pre-dynamical rule in membrane i, for $j \neq i$, inside the P system we have two sequential and dynamical communication processes carried out by a *mobile membrane*, which we write as a couple of well-matching round brackets $(_{i,j} \ldots)_{i,j}$, where i is the label of the originating membrane and j is the label of the target membrane. The communication steps are defined by means of the following rules:

1. *Gemmation* of a mobile membrane:

$$[_0 \cdots [_i \ldots, w@_j, \ldots]_i \ldots]_0 \to_G [_0 \cdots [_i \cdots]_i (_{i,j}\, w\,)_{i,j} \ldots]_0$$

   for some $i \in \{1, \ldots, n\}, j \in \{0, 1, \ldots, n\}, j \neq i, w \in V^+$.
   During this first phase the symbol $@_j$ is removed, its subscript becomes the second label of the mobile membrane, the string w leaves membrane i and enters the freshly created mobile membrane.
   If there are more than one string as $w_1@_j, \ldots, w_k@_j$ inside membrane i, all of which directed to the same target membrane j, then a single common mobile membrane will be budded off from membrane i:

$$[_0 \cdots [_i \ldots, w_1@_j, \ldots, w_k@_j, \ldots]_i \ldots]_0 \to_G [_0 \cdots [_i \cdots]_i (_{i,j} w_1, \ldots, w_k)_{i,j} \ldots]_0.$$

   Otherwise, if inside membrane i there are strings $w_1@_{j_1}, \ldots, w_h@_{j_k}$ $(h \geq k)$ such that $j_1, \ldots, j_k$ are pairly distinct, then k distinct mobile membranes will be gemmated, each one containing the strings directed to the specified membrane:

$$[_0 \cdots [_i \ldots, w_1@_{j_1}, \ldots, w_{h_1}@_{j_1}, \ldots, w_{h_k}@_{j_k}, \ldots, w_h@_{j_k}, \ldots]_i \ldots]_0 \to_G$$

$$[_0 \dots [_i \dots]_i (_{i,j_1} w_1, \dots, w_{h_1})_{i,j_1} \dots (_{i,j_k} w_{h_k}, \dots, w_h)_{i,j_k} \dots]_0.$$

Exactly analogous is the symmetrical case when a membrane i, for some $i \in \{1, \dots, n\}$, contains one or more strings of the form $@_j w$, for some $j \in \{0, 1, \dots, n\}$. Obviously the same holds when a membrane i contains some strings of both forms.

2. *Fusion* of the mobile membrane:

$$[_0 \dots (_{i,j} w)_{i,j} [_j \dots]_j \dots]_0 \rightarrow_F [_0 \dots [_j \dots, w, \dots]_j \dots]_0$$

for some $i \in \{1, \dots, n\}, j \in \{1, \dots, n\}, j \neq i, w \in V^+$.
During this second phase the mobile membrane becomes a part of the target membrane, leaving its contents inside it.
In particular, if $j = 0$ the mobile membrane fuses with the skin membrane and the objects exit the system. In this way we simulate the biological process of exocytosis and hence we have the (external) output of the string:

$$[_0 \dots (_{i,0} w)_{i,0} \dots]_0 \rightarrow_F [_0 \dots]_0 w.$$

The processes of gemmation and fusion of a mobile membrane are illustrated in figure 1, where Euler-Venn diagrams of two types are used: rectangular boxes represent membranes in the membrane structure μ, while a circle box represents a mobile membrane.

One more theoretical feature has to be introduced to the aim of keeping this variant closer to the functioning of real cells. We define a *meta-priority* between the whole set C_i and the whole set D_i, $\forall i = 1, \dots, n$, meaning that all applicable classical rules in C_i must be used before any other applicable pre-dynamical rule in D_i. The meta-priority is used to simulate the completion of the maturation path of a substance inside the Golgi apparatus. On the contrary, we do not define any priority relation between rules in the set C_i neither between rules in the set D_i, as it has been previously done in [7] in the form of a partial order relation between evolution rules.

Finally, a *P system* (of degree $n + 1$) with *gemmation of mobile membranes* (or *gemmating P system*, in short) is defined by the construct

$$\Pi_{(G,F)} = (V, T, \mu, M_0, \dots, M_n, (C_0, \emptyset), (C_1, D_1), \dots, (C_n, D_n), \infty)$$

with the following components:

(*i*) V is an alphabet such that $V \cap \{@_j\} = \emptyset, \forall j = 0, 1, \dots, n$;
(*ii*) $T \subseteq V$ is the output alphabet;
(*iii*) $\mu = [_0 [_1]_1 [_2]_2 \dots [_{n-1}]_{n-1} [_n]_n]_0$ is a membrane structure of depth 2 and degree $n + 1$;
(*iv*) $M_0, \dots, M_n$ are multisets of finite support over V^*;

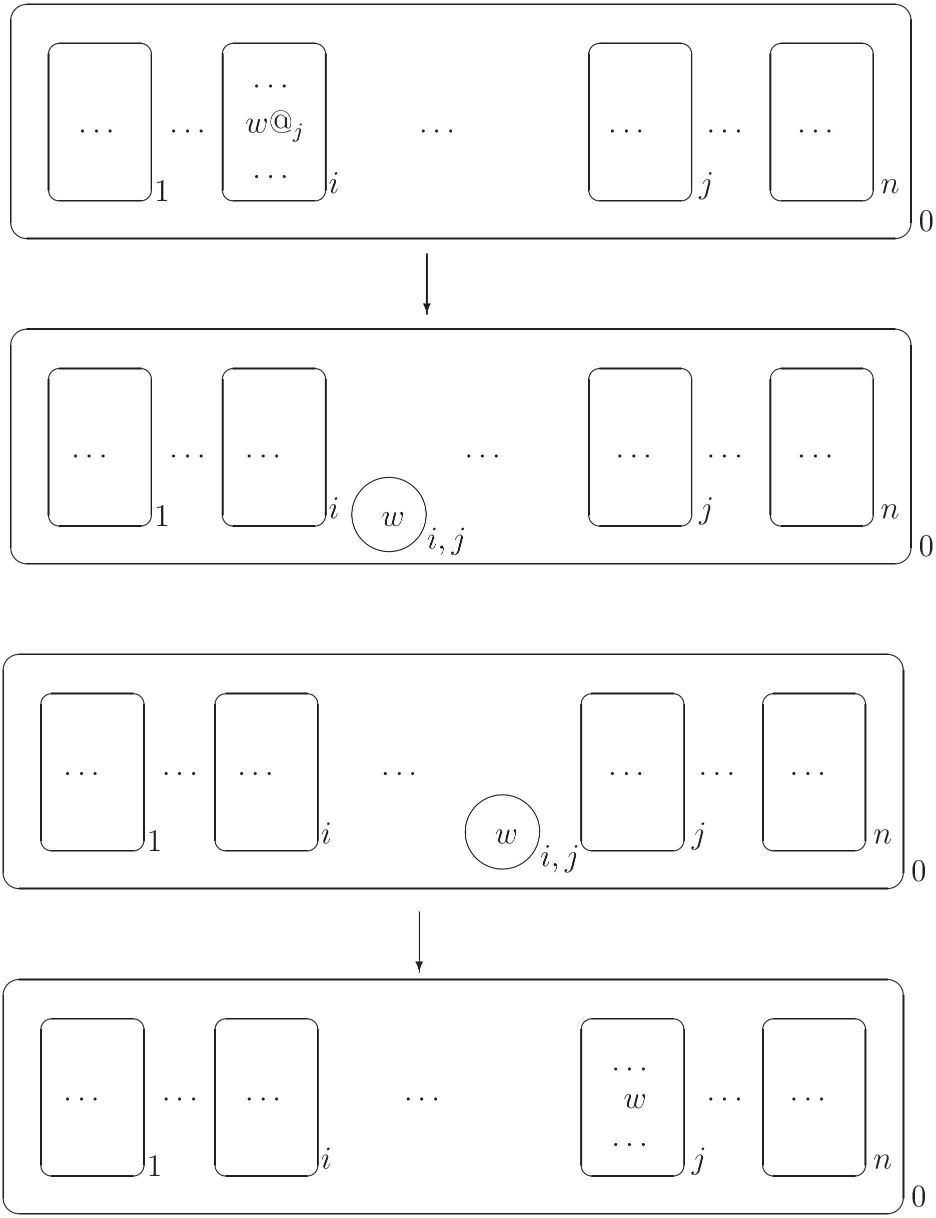


Fig. 1. Gemmation of a single mobile membrane from membrane i and fusion with target-membrane j

(*v*) $(C_i, D_i)\ \forall\, i = 0, 1, \ldots, n$ are the set of classical evolution rules and the set of pre-dynamical evolution rules, respectively. C_i has a meta-priority above D_i as far as the application of all of its rules is concerned, $\forall\, i = 1, \ldots, n$. The set D_0 is empty;

(*vi*) ∞ means that the system has external output.

The application of the rules is done as usual in P system area: in one step all regions are processed simultaneously by using the rules in a nondeterministic and maximally parallel manner. This means that in each region the objects to evolve and the rules to be applied to them are nondeterministically chosen, but all objects which can evolve should evolve. Specifically, at each step of a computation a string can be processed by one rule only, and its multiplicity is decreased by one. The multiplicity of strings produced by an operation is accordingly increased. The strings resulting after the application of a rule are communicated by mobile membranes or by in/out communication to the regions specified by the target indications (the target indication *here* will often be omitted).

The membrane structure at a given time, together with all multisets of objects associated with the regions defined by the membrane structure, is the *configuration* of the system at that time. The $(n+1)$-tuple $(\mu, M_1, \ldots, M_n)$ constitutes the *initial configuration* of the system. For two configurations $C_1 = (\mu, M_1', \ldots, M_n'), C_2 = (\mu, M_1'', \ldots, M_n'')$ of $\Pi_{(G,F)}$ we say that we have a *transition* from C_1 to C_2 by applying the rules present in the sets (C_i, D_i), $0 \leq i \leq n$, according to the meta-priority relation. A sequence of transitions forms a *computation*. A computation *halts* when there is no rule which can be further applied in the current configuration. On the contrary, we say that a computation is non-halting if there is at least one rule which can be applied forever. The *output* is the set of strings over T sent out of the system during the computation. The language generated in this way by a P system $\Pi_{(G,F)}$ is denoted by $L(\Pi_{(G,F)})$. Non-halting computations provide no output.

3 Examples

We show that we can easily generate non context-free languages using P systems with only mutation rules and only communications performed by gemmation of mobile membranes. Moreover, we will consider only sets of strings without multiplicities, as no replication nor splitting rules will be used.

3.1 Example 1

We construct the P system Π_1 with gemmation of mobile membranes

$$\Pi_1 = (V, T, \mu, M_0, \ldots, M_3, C_0, (C_1, D_1), \ldots, (C_3, D_3), \infty)$$

where:

$V = \{A, A', B, B', C, a, b, c\};$

$$\begin{aligned}
T &= \{a,b,c\};\\
\mu &= [_0\ [_1\ ]_1\ [_2\ ]_2\ [_3\ ]_3\ ]_0;\\
M_1 &= \{ABC\}, \text{ all other sets are empty;}\\
C_0 &= \emptyset;\\
C_1 &= \{A \rightarrow aA', B \rightarrow bB'\},\\
D_1 &= \{C \rightarrow cC@_2, C \rightarrow cC@_3\};\\
C_2 &= \{A' \rightarrow aA, B' \rightarrow bB\},\\
D_2 &= \{C \rightarrow cC@_1, C \rightarrow cC@_3\};\\
C_3 &= \{A \rightarrow \lambda, A' \rightarrow \lambda, B \rightarrow \lambda, B' \rightarrow \lambda\},\\
D_3 &= \{C \rightarrow \lambda@_0\}.
\end{aligned}$$

The computation starts in membrane 1, where both classical rules $A \rightarrow aA', B \rightarrow bB'$ must be used before any pre-dynamical rule can be applied. The symbols A', B' guarantee that an equal number of a and b will be generated, and they are necessary in order to obtain halting computation. In fact, if we would substitute the two rules $A \rightarrow aA', B \rightarrow bB'$ with $A \rightarrow aA, B \rightarrow bB$ respectively, then we should apply each one forever because of the meta-priority relation. Hence, by making cycles between membranes 1 and 2 we generate all the strings of the form $a^nXb^nYc^nC$, with $(X,Y) \in \{(A,B),(A',B')\}$. When the current string reaches membrane 3, all nonterminal symbols are erased and the string leaves the systems.

It is easy to see that the language generated by the system is $L(\Pi_1) = \{a^nb^nc^n \mid n \geq 1\}$.

3.2 Example 2

We define the system

$$\Pi_2 = (V, T, \mu, M_0, \ldots, M_7, C_0, (C_1, D_1), \ldots, (C_7, D_7), \infty)$$

with the following components:

$$\begin{aligned}
V &= \{A, B, A', B', C, a, b\};\\
T &= \{a,b\};\\
\mu &= [_0\ [_1\ ]_1\ [_2\ ]_2\ [_3\ ]_3\ [_4\ ]_4\ [_5\ ]_5\ [_6\ ]_6\ [_7\ ]_7\ ]_0;\\
M_1 &= \{ABC\}, \text{ all other sets are empty;}\\
C_0 &= \emptyset;\\
C_1 &= \emptyset,\\
D_1 &= \{C \rightarrow \lambda@_6, C \rightarrow \lambda@_7, C \rightarrow C@_2, C \rightarrow C@_3\};\\
C_2 &= \{A \rightarrow aA', B \rightarrow aB'\},\\
D_2 &= \{C \rightarrow \lambda@_6, C \rightarrow \lambda@_7, C \rightarrow C@_4, C \rightarrow C@_5\};\\
C_3 &= \{A \rightarrow bA', B \rightarrow bB'\},\\
D_3 &= \{C \rightarrow \lambda@_6, C \rightarrow \lambda@_7, C \rightarrow C@_4, C \rightarrow C@_5\};\\
C_4 &= \{A' \rightarrow aA, B' \rightarrow aB\},\\
D_4 &= \{C \rightarrow \lambda@_6, C \rightarrow \lambda@_7, C \rightarrow C@_2, C \rightarrow C@_3\};\\
C_5 &= \{A' \rightarrow bA, B' \rightarrow bB\},\\
D_5 &= \{C \rightarrow \lambda@_6, C \rightarrow \lambda@_7, C \rightarrow C@_2, C \rightarrow C@_3\};
\end{aligned}$$

$$\begin{aligned} C_6 &= \{A \to a, A' \to a\}, \\ D_6 &= \{B \to a@_0, B' \to a@_0\}; \\ C_7 &= \{A \to b, A' \to b\}, \\ D_7 &= \{B \to b@_0, B' \to b@_0\}. \end{aligned}$$

The computation starts in membrane 1 which contains only pre-dynamical rules that non deterministically redirect the current string to membrane 2, 3, 6 or 7. In each membrane of the system the gemmation phase is controlled by the support-symbol C. If we make cycles between membranes 2 and 4, then we generate strings of the form $xaAxaBC$, with $x \in \{a, b\}^*$. Observe that in membrane 2 we could use no rules such as $A \to aA, B \to aB$ or $C \to C$ in order to get an immediate generation of such strings, because we should apply these rules forever and hence the computation would never halt. In the same way, we need a cycle between membranes 3 and 5 for further rewriting of the terminal symbol b; the generated strings have now the form $xbA'xbB'C$, with $x \in \{a, b\}^*$. Finally, membranes 6 and 7 cause the terminal rewriting of the string and its output from the system.

It follows that the generated language is $L(\Pi_2) = \{xx \mid x \in \{a, b\}^+\}$.

4 Computational Completeness

We show that P systems with gemmation of mobile membranes and in/out communications are able to generate any recursively enumerable language. Moreover, as only mutation rules are used in the proof, there is no need of using multisets of strings. Hence, we will not indicate the multiplicity of the string, being understood that it is always equal to one.

In the proof we need the notion of a *matrix grammar with appearance checking*; such a grammar is a construct $G = (N, T, S, M, F)$, where N, T are disjoint alphabets of nonterminal and terminal symbols, $S \in N$ is the axiom, M is a finite set of matrices, which are sequences of context-free rules of the form $(A_1 \to x_1, \ldots, A_n \to x_n)$, $n \geq 1$, (with $A_i \in N, x_i \in (N \cup T)^*$, in all cases), and F is a set of occurrences of rules in M.
For $w, z \in (N \cup T)^*$ we write $w \Rightarrow z$ if there are a matrix $(A_1 \to x_1, \ldots, A_n \to x_n)$ in M and strings $w_i \in (N \cup T)^*, 1 \leq i \leq n+1$, such that $w = w_1, z = w_{n+1}$, and, for all $1 \leq i \leq n$, either $w_i = w_i' A_i w_i'', w_{i+1} = w_i' x_i w_i''$, for some $w_i', w_i'' \in (N \cup T)^*$, or $w_i = w_{i+1}$, A_i does not appear in w_i, and the rule $A_i \to x_i$ appears in F. (The rules of a matrix are applied in order, possibly skipping the rules in F if they cannot be applied – one says that these rules are applied in the *appearance checking* mode). The language generated by G is defined by $L(G) = \{w \in T^* \mid S \Rightarrow^* w\}$. The family of languages of this form is denoted by MAT_{ac}. When $F = \emptyset$ (hence we do not use the appearance checking feature), the generated family is denoted by MAT.
A matrix grammar with appearance checking $G = (N, T, S, M, F)$ is said to be in the *binary normal form* if $N = N_1 \cup N_2 \cup \{S, \dagger\}$ is the union of mutually disjoint sets, and the matrices in M are of one of the following forms:

1. $(S \to XA)$ with $X \in N_1, A \in N_2$;
2. $(X \to Y, A \to x)$ with $X, Y \in N_1, A \in N_2, x \in (N_2 \cup T)^*$;
3. $(X \to Y, A \to \dagger)$ with $X, Y \in N_1, A \in N_2$;
4. $(X \to \lambda, A \to x)$ with $X \in N_1, A \in N_2, x \in T^*$.

Moreover, there exists only one matrix of type 1, F exactly consists of all rules $A \to \dagger$ appearing in matrices of type 3 and $\dagger$ is a trap-symbol (once introduced, it can never be removed). Finally, each matrix of type 4 is used only once, at the last step of a derivation. According to Lemma 1.3.7 in [3], for each matrix grammar there exists an equivalent one in the binary normal form.

We denote by CF and RE the families of context free and recursively enumerable languages. The following proper inclusions hold:$CF \subset MAT \subset MAT_{ac} = RE$. Further details about matrix grammars can be found in [13] and [5]. Moreover, in [5] it is shown that all one-letter languages in MAT are regular.

We denote by $GemP_n(MPri, (in/out))$ the family of languages generated by gemmating P systems of degree n, for $n \geq 1$, with relation of meta-priority and communications of type in/out. If the number of membranes is not limited, then the subscript n is replaced by $*$.

Theorem 1. *$GemP_*$(MPri, (in/out)) = RE.*

Proof. The inclusion $GemP_*(MPri, (in/out)) \subseteq RE$ directly follows from Church-Turing thesis. So, we only have to prove the opposite inclusion; to this aim, we make use of the equality $RE = MAT_{ac}$ and we consider a matrix grammar with appearance checking $G = (N, T, S, M, F)$, in the binary normal form previously described. Let p be the number of matrices of type 2 in G, q the number of matrices of type 3 and r the number of matrices of type 4, with $p \geq 1, q \geq 1, r \geq 1$.
We show how to construct a gemmating P system of degree $s = 2p + q + 2r + 3$ that generates the same language as G:

$$\begin{aligned} \Pi_{(G,F)} = (&V, \mu, M_0, M_1, M_{(2)_1}, M_{(2)_2}, \dots, M_{(p+1)_1}, M_{(p+1)_2}, M_{(p+2)_1}, \dots, \\ &M_{(p+q+1)_1}, M_{(p+q+2)_1}, M_{(p+q+2)_2}, \dots, M_{(p+q+r+1)_1}, M_{(p+q+r+1)_2}, M_c, \\ &C_0, (C_1, D_1), (C_{(2)_1}, D_{(2)_1}), \dots, (C_{(p+q+r+1)_2}, D_{(p+q+r+1)_2}), (C_c, D_c)) \end{aligned}$$

with

$$V = N_1 \cup N_2 \cup T \cup \{P, J, J', \sharp\}$$

(we use the symbols of G plus four support-symbols P, J, J' and $\sharp$);

$$\begin{aligned} \mu = &[_0\ [_1]_1\ [_{(2)_1}\ ]_{(2)_1}\ [_{(2)_2}\ ]_{(2)_2}\ \cdots\ [_{(p+1)_1}\ ]_{(p+1)_1}\ [_{(p+1)_2}\ ]_{(p+1)_2}\ [_{(p+2)_1}\ ]_{(p+2)_1} \\ &\cdots\ [_{(p+q+1)_1}\ ]_{(p+q+1)_1}\ [_{(p+q+2)_1}\ ]_{(p+q+2)_1}\ [_{(p+q+2)_2}\ ]_{(p+q+2)_2}\ \cdots \\ &[_{(p+q+r+1)_1}\ ]_{(p+q+r+1)_1}\ [_{(p+q+r+1)_2}\ ]_{(p+q+r+1)_2}\ [_c\ ]_c\ ]_0 \end{aligned}$$

(membrane 1 simulates the matrix of type 1 in G, each couple of membranes labelled with $(i)_1, (i)_2, 2 \leq i \leq p+1$, simulate a matrix of type 2 in G, each membrane labelled with $(j)_1, p+2 \leq j \leq p+q+1$, simulate a matrix of type 3 in G, each couple of membranes labelled with $(k)_1, (k)_2, p+q+2 \leq k \leq p+q+r+1$, simulate a matrix of type 4 in G; we also use a control-membrane labelled with c);

$M_1 = \{XAP \mid S \to XA$ is the rule of the matrix of type $1\}$, all other sets are empty;

$C_0 = \{J \to (\lambda, in_1), J' \to (\lambda, in_c)\}$
(the skin membrane contains two classical rules, one for each support-symbol J, J', which redirect the received strings to membrane 1 or to the control-membrane);

$C_1 = \emptyset$ and $D_1 = \{P \to P@_{(i)_1}, P \to P@_{(j)_1}, P \to P@_{(k)_1}\}$,
$\forall\, i = 2, \ldots, p{+}1, \forall\, j = p{+}2, \ldots, p{+}q{+}1, \forall\, k = p{+}q{+}2, \ldots, p{+}q{+}r{+}1$
(membrane 1 sends the current string to the first membrane $(i)_1, (k)_1$ of any couple of membranes simulating matrices of type 2 and 4, or to any of the single membranes $(j)_1$ simulating the matrices of type 3);

$\forall\, i = 2, \ldots, p + 1$ we define:

$C_{(i)_1} = \emptyset$ and $D_{(i)_1} = \{X \to @_{(i)_2}Y\}$
(for each matrix of type 2, in the first membrane we simulate the first rule of the matrix with one pre-dynamical rule),

$C_{(i)_2} = \{A \to (Jx, out)\}$ and $D_{(i)_2} = \emptyset$
(for each matrix of type 2, in the second membrane we simulate the second rule of the matrix with one classical rule, which introduces the support-symbol J in the string);

$\forall\, j = p + 2, \ldots, p + q + 1$ we define:

$C_{(j)_1} = \{A \to A\}$ and $D_{(j)_1} = \{X \to @_1Y\}$
(for each matrix of type 3, in a unique membrane we simulate the second rule of the matrix by one classical evolution rule, and the first rule of the matrix by one pre-dynamical rule);

$\forall\, k = p + q + 2, \ldots, p + q + r + 1$ we define:

$C_{(k)_1} = \emptyset$ and $D_{(k)_1} = \{X \to @_{(k)_2}\lambda\}$
(for each matrix of type 4, in the first membrane we simulate the first rule of the matrix with one pre-dynamical rule),

$C_{(k)_2} = \{A \to (J'x, out)\}$ and $D_{(k)_2} = \emptyset$
(for each matrix of type 4, in the second membrane we simulate the second rule of the matrix with one classical rule. The support-symbol J' is introduced in the string);

and finally

$C_c = \{B \to (\sharp, here), \sharp \to (\sharp, here)\}\ \forall B \in N_1 \cup N_2$
(in the control-membrane we define a classical rule for each nonterminal symbol in $N_1 \cup N_2$, and one classical mutation rule over $\sharp$ which causes the non termination of a computation),

$D_c = \{P \to \lambda @_0\}$
(this pre-dynamical rule erases the support-symbol P and sends the string outside the system).

The system works as follows: consider the string ZwP in membrane 1 with $Z \in N_1$ and $w \in (N_2 \cup T)^*$. Initially we have $Z = X$ and $w = A$.
We nondeterministically choose between any of the pre-dynamical rules defined in membrane 1, the string ZwP is so rewritten as $ZwP@_t$, with $t \in \{(i)_1, (j)_1, (k)_1\}$. This string is sent to membrane $(i)_1$, with $2 \leq i \leq p+1$, or $(k)_1$, with $p+q+2 \leq k \leq p+q+r+1$, or $(j)_1$, with $p+2 \leq j \leq p+q+1$, where we simulate the first rule of the matrices of type 2 and 4, or both rules of matrices of type 3, respectively.

If the string ZwP enters a membrane $(i)_1$, for any $i = 2, \ldots, p+1$, and $Z = X$, then we can apply the pre-dynamical rule $X \rightarrow @_{(i)_2}Y$: the rule of the corresponding matrix is correctly simulated and the string enters the second membrane $(i)_2$. On the contrary, if $Z \neq X$, then the rule cannot be applied and the computation halts, no string will be generated. In the first case, when the string YwP enters membrane $(i)_2$, if the symbol $A \in w$ then we can apply the rule $A \rightarrow (Jx, out)$. The string Yw_1Jxw_2P, with $w_1, w_2 \in (N_2 \cup T)^*$ such that $w = w_1Aw_2$, enters membrane 0 and then it returns to membrane 1 by means of the rule $J \rightarrow (\lambda, in_1)$. Observe that the support-symbol J is immediately erased and it will never appear in any terminal string. From membrane 1 we can now start the simulation of another matrix. In membrane $(i)_2$, if the symbol $A \notin w$, then the string will never exit the current membrane, the computation halts and no string will be generated. Thus, with two membranes we are able to simulate the productions of any matrix of type 2, and we can correctly do it.

If the string ZwP enters a membrane $(j)_1$, for any $j = p+2, \ldots, p+q+1$, and $A \in w$, then the computation will never stop: the rule $A \rightarrow A$ will be applied forever because of the meta-priority relation. No string will be generated, thus we correctly simulate the introduction of the symbol † in a production of G. On the contrary, if $A \notin w$, then the classical rule $A \rightarrow A$ cannot be applied and we pass to the pre-dynamical rule $X \rightarrow @_1Y$. If $Z = X$ then the string YwP will be sent to membrane 1, otherwise the rule cannot be applied and the computation stops. Thus we only need one membrane for every matrix in G whose rules are to be applied in the appearance checking mode. Observe that the order of the rules in the membrane is opposite to the order of the rules in the matrix, but this fact does not change the set of generated strings.

Finally, if the string ZwP enters a membrane $(k)_1$, for any $k = p+q+2, \ldots, p+q+r+1$, and $Z = X$, then we can apply the pre-dynamical rule $X \rightarrow @_{(k)_2}\lambda$: the rule of the corresponding matrix is correctly simulated and the string enters the second membrane $(k)_2$. If $Z \neq X$, then the rule cannot be applied and the computation halts, no string will be generated. If the string wP enters membrane $(k)_2$ and if the symbol $A \notin w$, then the string will never exit the current membrane, the computation halts and no string will be generated. On the contrary, if $A \in w$ we can apply the rule $A \rightarrow (J'x, out)$: the string $w_1J'xw_2P$, with $w_1, w_2 \in (N_2 \cup T)^*$ such that $w = w_1Aw_2$, enters membrane 0. As for matrices of type 2, we are therefore able to simulate the productions of any matrix of type 4 with two membranes, and we can do it in the correct order.

When a string reaches a membrane labelled with $(k)_2$, for any $k=p+q+2, \ldots, p+q+r+1$, the simulation of the matrices of G has to be ended. To this aim, we make use of the support-symbol J': in membrane 0 we define the rule $J' \to (\lambda, in_c)$ which will send the received string w_1xw_2P to membrane c. Again, the support-symbol J' is immediately erased and it will never appear in any terminal string. Inside the control-membrane we check that the string does not contain any nonterminal symbol: if it is so, then the string w_1xw_2 will exit the system by a final gemmation due to the rule $P \to \lambda @_0$. Otherwise, if w_1xw_2P contains a symbol $B \in (N_1 \cup N_2)$, then the classical rules $B \to \sharp$ will introduce the trap symbol $\sharp$ that causes never halting computations. No string will be generated and, once more, we can correctly simulate any production in G.

It follows that we exactly generate the strings of terminal symbols generated by G, that is $L(\Pi_{(G,F)}) = L(G)$. □

We want to point out that, as seen in the proof, a unique membrane suffices for simulating each matrix of type 3, while we need two membranes and in/out communications for each matrix of type 2 and 4. The meta-priority relation and the gemmation of mobile membranes yield here an easy and immediate simulation of the appearance checking mode, unlike all other variants of P systems where this aspect of matrix grammars is harder to be proved.

We stress the fact that in/out communications are essential in order to get a correct simulation of matrices of type 2 and 4. In fact, let us consider the second rule $A \to x$ (with $x \in (N_2 \cup T)^*$ or $x \in T^*$) of such matrices and analyse the following cases:

1. this rule could not be simulated using a pre-dynamical rule of the form $A \to x@_1$, because the symbol $@_1$ can be introduced only at one end of the string but we do not know where the symbol A is placed in the current string;
2. we could not use a single membrane and two rules of the form $A \to x, P \to P@_1$ because the meta-priority forces the application of the classical rule to all occurrences of the symbol A in the string. Hence, the simulation of the corresponding matrix could not be correct.

If we do not use in/out communications, it is possible to show that the family of languages MAT is properly included in the family of languages generated by gemmating P systems of degree 4.

Theorem 2. *$GemP_4(MPri, n(in/out)) - MAT \neq \emptyset$.*

Proof. Consider the P system

$$\Pi = (V, T, \mu, M_0, \ldots, M_3, C_0, (C_1, D_1), (C_2, D_2), (C_3, D_3), \infty)$$

with components

$$V = \{A, A', B, a\};$$
$$T = \{a\};$$

$$\begin{aligned}
\mu &= [_0[_1]_1[_2]_2[_3]_3]_0;\\
M_1 &= \{AB\}, \text{all other sets are empty};\\
C_0 &= \emptyset;\\
C_1 &= \{A \rightarrow A'A'\},\\
D_1 &= \{B \rightarrow B@_2, B \rightarrow B@_3\};\\
C_2 &= \{A' \rightarrow AA\},\\
D_2 &= \{B \rightarrow B@_1, B \rightarrow B@_3\};\\
C_3 &= \{A \rightarrow a, A' \rightarrow a, \},\\
D_3 &= \{B \rightarrow \lambda@_0\}.
\end{aligned}$$

The system works as follows: in membrane 1 and 2 we duplicate the number of the symbols A and A', respectively, and we generate strings of the form $(A')^{2^n}B$ or $(A)^{2^n}B\ \forall n \geq 1$. When the current string is sent to membrane 3, each nonterminal symbol A or A' becomes a terminal symbol a, while the support-symbol B is erased and the string leaves the system. Thus we generate the language $L(\Pi) = \{a^{2^n} \mid n \geq 1\}$, which is a non regular language over one-letter alphabet.
It follows that $MAT \subset GemP_4(MPri, n(in/out))$. □

5 Solving the HPP in Quadratic Time

Consider a directed graph $\gamma = (N, A)$ where N is a finite set of n vertices, identified with the numbers $1, 2, \ldots, n$, and A is a set of ordered pairs of vertices (v_i, v_j), for $i, j \in \{1, \ldots, n\}$. The Hamiltonian Path Problem (in short HPP) for γ asks whether or not there exists a path from a given initial vertex v_1 to a final vertex v_n which passes exactly once through each and every vertex of the graph ([4]). We write as r_i the outdegree of the vertex v_i, $\forall\, i \in \{1, \ldots, n\}$, and we ignore useless arcs of the form (v_i, v_i), so r_i will be at most equal to $n - 1$.

We show how to construct a P system with gemmation of mobile membranes, with rules similar to those used in [2], which actually finds all Hamiltonian paths in a given graph and not only their existence (if any). The computation halts for all inputs and the problem is solved at most in a quadratic time with respect to the number of vertices.

Theorem 3. *The HPP can be solved by P systems with gemmation of mobile membranes in a quadratic time with respect to the number of vertices.*

Proof. We define a gemmating P system of degree $n + 1$ associated with γ

$$\Pi_{HPP} = (V, T, \mu, M_0, \ldots, M_n, C_0, (C_1, D_1), \ldots, (C_n, D_n), \infty)$$

with the following components:

$$V = \{\langle i, k\rangle, [i, k], \langle i, k; j_1, \ldots, j_{r_i}\rangle, \sharp \mid 1 \leq i \leq n, 0 \leq k \leq n - 1 \text{ and } j_1, \ldots, j_{r_i}$$

are labels in $\{1,\ldots,n\}$ such that $(v_i,v_{j_h})\in A, \forall\, h=1,\ldots,r_i\}$,
i is the label of the vertex v_i, r_i is the outdegree of the vertex v_i,
while k is used to count the steps of computation;

$T=\{[i,k]\mid 1\le i\le n, 0\le k\le n-1\}$;

$\mu=[_0[_1]_1[_2]_2\ldots[_{n-1}]_{n-1}[_n]_n]_0$, that is we define an inner membrane i for each vertex v_i in N;

$M_1=\{(\langle 1,0\rangle,1)\}$, all other multisets are empty;

$C_0=\emptyset$,

and with the following sets of rules which, starting from the object $\langle 1,0\rangle$ in membrane 1 and by repeatedly using replication rules in the inner membranes, create all the strings that correspond to paths in γ:

C_1: $\langle\mathbf{1},\mathbf{k}\rangle\to(\sharp;\mathbf{here})\ \forall\, k=1,\ldots,n-1$
(in membrane 1 if a string contains the symbol $\langle 1,k\rangle$ for $k\neq 0$, then it codifies a wrong path because it surely visited the current membrane twice. We stop such strings by the introduction of the symbol $\sharp$);

D_1: $\langle\mathbf{1},\mathbf{0}\rangle\to([\mathbf{1},\mathbf{0}]\langle\mathbf{j_1},\mathbf{1}\rangle@_{\mathbf{j_1}};\mathbf{here})$ if $r_1=1$
(if there is a single arc from vertex v_1 to vertex v_{j_1}, then the nonterminal symbol $\langle 1,0\rangle$ is rewritten as the corresponding terminal symbol $[1,0]$, and the string is prolonged by adding $\langle j_1,1\rangle$, which denotes the label of the membrane to be visited and the next step in the path);
$\langle\mathbf{1},\mathbf{0}\rangle\to([\mathbf{1},\mathbf{0}]\langle\mathbf{j_1},\mathbf{1}\rangle@_{\mathbf{j_1}}\parallel[\mathbf{1},\mathbf{0}]\langle\mathbf{j_2},\mathbf{1}\rangle@_{\mathbf{j_2}};\mathbf{here},\mathbf{here})$ if $r_1=2$
(if there are two arcs exiting from vertex v_1, then we replicate the initial object into two strings at the same step);
$\langle\mathbf{1},\mathbf{0}\rangle\to([\mathbf{1},\mathbf{0}]\langle\mathbf{j_1},\mathbf{1}\rangle@_{\mathbf{j_1}}\parallel\langle\mathbf{1},\mathbf{0};\mathbf{j_2},\ldots,\mathbf{j_{r_1}}\rangle;\mathbf{here},\mathbf{here})$ if $r_1>2$
(if there are more than two vertices exiting vertex v_1, then we use a replication rule to prolong one string and to memorize all the others vertex-labels in the nonterminal symbol $\langle 1,0;j_2,\ldots,j_{r_1}\rangle$);
$\langle\mathbf{1},\mathbf{0};\mathbf{j_h},\ldots,\mathbf{j_{r_1}}\rangle\to([\mathbf{1},\mathbf{0}]\langle\mathbf{j_h},\mathbf{1}\rangle@_{\mathbf{j_h}}\parallel\langle\mathbf{1},\mathbf{0};\mathbf{j_{h+1}},\ldots,\mathbf{j_{r_1}}\rangle;\mathbf{here},\mathbf{here})$
$\forall\, h=2,\ldots,r_1-2$
(if there are more than two memorized vertices, then in a step we prolong only one of them, while keeping memorized all the others);
$\langle\mathbf{1},\mathbf{0};\mathbf{j_{r_1-1}},\mathbf{j_{r_1}}\rangle\to([\mathbf{1},\mathbf{0}]\langle\mathbf{j_{r_1-1}},\mathbf{1}\rangle@_{\mathbf{j_{r_1-1}}}\parallel[\mathbf{1},\mathbf{0}]\langle\mathbf{j_{r_1}},\mathbf{1}\rangle@_{\mathbf{j_{r_1}}};\mathbf{here},\mathbf{here})$
(if there are exactly two memorized vertices, then we replicate the single object into a new couple of strings in a single step);

C_i, $\forall\, i=2,\ldots,n-1$, consists of:
$[\mathbf{i},\mathbf{k}]\to(\sharp\ :\ \sharp;\mathbf{here},\mathbf{out})\ \forall\, k=1,\ldots,n-2$
(if a string containing the symbol $[i,k]$ enters membrane i, then such string must be stopped because it codifies a wrong path and we break it into two substrings by a splitting rule);

D_i, $\forall\, i = 2, \ldots, n-1$, consists of rules analogous to those defined for the set D_1 (here the range of the step counter is $1 \leq k \leq n-2$ for all rules):
$\langle \mathbf{i}, \mathbf{k} \rangle \rightarrow ([\mathbf{i}, \mathbf{k}]\langle \mathbf{j_1}, \mathbf{k}+\mathbf{1} \rangle @_{\mathbf{j_1}}; \mathbf{here})$ if $r_i = 1$;
$\langle \mathbf{i}, \mathbf{k} \rangle \rightarrow ([\mathbf{i}, \mathbf{k}]\langle \mathbf{j_1}, \mathbf{k}+\mathbf{1} \rangle @_{\mathbf{j_1}} \parallel [\mathbf{i}, \mathbf{k}]\langle \mathbf{j_2}, \mathbf{k}+\mathbf{1} \rangle @_{\mathbf{j_2}}; \mathbf{here}, \mathbf{here})$ if $r_i = 2$;
$\langle \mathbf{i}, \mathbf{k} \rangle \rightarrow ([\mathbf{i}, \mathbf{k}]\langle \mathbf{j_1}, \mathbf{k}+\mathbf{1} \rangle @_{\mathbf{j_1}} \parallel \langle \mathbf{i}, \mathbf{k}; \mathbf{j_2}, \ldots, \mathbf{j_{r_i}} \rangle; \mathbf{here}, \mathbf{here})$ if $r_i > 2$;
$\langle \mathbf{i}, \mathbf{k}; \mathbf{j_h}, \ldots, \mathbf{j_{r_i}} \rangle \rightarrow ([\mathbf{i}, \mathbf{k}]\langle \mathbf{j_h}, \mathbf{k}+\mathbf{1} \rangle @_{\mathbf{j_h}} \parallel \langle \mathbf{i}, \mathbf{k}; \mathbf{j_{h+1}}, \ldots, \mathbf{j_{r_i}} \rangle; \mathbf{here}, \mathbf{here})$
$\forall\, h = 2, \ldots, r_i - 2$;
$\langle \mathbf{i}, \mathbf{k}; \mathbf{j_{r_i-1}}, \mathbf{j_{r_i}} \rangle \rightarrow$
$([\mathbf{i}, \mathbf{k}]\langle \mathbf{j_{r_i-1}}, \mathbf{k}+\mathbf{1} \rangle @_{\mathbf{j_{r_i-1}}} \parallel [\mathbf{i}, \mathbf{k}]\langle \mathbf{j_{r_i}}, \mathbf{k}+\mathbf{1} \rangle @_{\mathbf{j_{r_i}}}; \mathbf{here}, \mathbf{here})$;

C_n: $\langle \mathbf{n}, \mathbf{k} \rangle \rightarrow (\sharp; \mathbf{here})\ \forall\, k = 1, \ldots, n-2$
(in membrane n, if $k \neq n-1$ we introduce the symbol $\sharp$ to stop every string containing the symbol $\langle n, k \rangle$, which codifies a wrong path);

D_n: $\langle \mathbf{n}, \mathbf{n}-\mathbf{1} \rangle \rightarrow ([\mathbf{n}, \mathbf{n}-\mathbf{1}]@_{\mathbf{0}}; \mathbf{here})$
(if a string reaches membrane n and if it contains the symbol $\langle n, n-1 \rangle$, then it surely codifies a correct path and it can leave the system).

The computation starts in membrane 1 from the unique object $\langle 1, 0 \rangle$: we start from vertex v_1 at the step 0 and, by repeatedly using pre-dynamical replication rules, we prolong all the strings which correspond to paths in γ. The paths can be correctly continued if either no vertex label is repeated in the string or we reach membrane n (that is, the final vertex in γ) at the step $n-1$.
The special symbol $\sharp$ is thus needed in order to break and stop every wrong Hamiltonian path, $\sharp$ is introduced in membrane 1 and n by classical rules, which have meta-priority above all other rules defined inside the membrane, so we can assure that no wrong path will be prolonged in the system.
Observe that, to this aim, we could not use a similar mutation rule in membranes $2, \ldots, n-1$, in fact if a string containing the symbol $[i, k]$ enters membrane i, then it will certainly be of the form $[1, 0]x_1[i, k]x_2\langle i, k' \rangle$, where $k' > k$, x_1 is a (possible empty) string over $\{[j, k]\}$, and x_2 is a non-empty string over $\{[j, k]\}$, for $j \in \{2, \ldots, n-1\}, j \neq i$. If we would use the rule $[i, k] \rightarrow (\sharp, \mathit{here})$ then the last symbol $\langle i, k' \rangle$ would cause the continuation of the path and we would finally have a wrong output. We choose not to use the similar rule $[i, k] \rightarrow (\sharp, \mathit{out})$ because we try to simulate the direct transport through a membrane only for those objects which do not have "too long" length. So we break any wrong string by a splitting rule and then we send to the skin membrane the second halves of such strings, which would otherwise be processed again. As the first half of any wrong string is not dangerous at all, we can decide both to send it out or to keep it inside the current membrane.
We continue in this way only those paths that pass exactly one time through each and every membrane, the computation always stops and we send every Hamiltonian path (if existing) outside the system by a mobile membrane gemmated from membrane n.

Let's now compute the maximum number of steps until a computation halts. We suppose that the outdegree of each vertex is equal to $n-1$. In membrane 1

all the possible continuations of the starting string are generated and sent to the destination membrane step by step, hence the worst complexity case corresponds to the last generated strings, which take $n-2$ steps to be ready to leave the membrane. With two more steps (gemmation and fusion phases) such strings are communicated to any membrane i, for $i = 2, \ldots, n-1$, where again other $n-2$ steps (plus two communication steps) are needed for the local last generated strings to reach their destination membranes. So it takes $n^2 - n$ steps until the last generated strings reach membrane n. Here after three more steps (evolution, gemmation, fusion) the strings codifying Hamiltonian paths (if any) will be sent out of the system. Hence, in total we perform at most $n^2 - n + 3$ steps before the system stops its computation.
The exact number of steps is given by the formula $\left[\sum_{i=1}^{n-1}((r_i - 1) + 2)\right] + 3$, when $r_i \leq n-1$. □

Note that, in particular, if the maximum outdegree of each vertex of the graph is bounded by 2, then the computation always halts after $3n$ steps. The quadratic time would collapse to linear time also if parallel replication rules were used, as introduced in [6]. In this case, in fact, we could prolong all the paths from each vertex in a single step, then in other two communication steps (gemmation and fusion of the mobile membranes) the strings would be sent to the target membranes. It follows that we would only need $3n$ steps to prolong and output all the Hamiltonian paths in the graph.

6 Final Remarks

We have introduced a new kind of communication for P systems and worked with membrane structures and evolution rules of biological inspiration, keeping the model as close as possible to the real structure of cells, in order to make easier an implementation of the model. We have proved that P systems with such features characterize the recursively enumerable languages and they can solve the Hamiltonian Path Problem in a quadratic time. We close the paper with three topics for further research.

As no priority is defined between classical evolution rules, we could think about a parallel application of all applicable rules over the same string, as in Lindenmayer systems ([13]). The generative power of this variant is still to be analyzed.

The second problem concerns the fact that, for the moment, no pre-dynamical rule can be defined in the skin membrane: up to now P systems have been "isolated" structures and no rules have ever been defined for letting an object entering the skin membrane from outside. Moreover, no object can be ejected from the system by mobile membranes gemmated from the skin membrane: the mobile membrane could never reach any other P system and the objects would remain forever inside it (no language would be generated in this way). Hence, it would be interesting to define either an *external ambient* for P systems, either *colonies* of P systems (of depth 2) which can communicate by mobile membranes or by getting the objects from outside with a new kind of evolution rules. We remark here again that the goal is to keep the model as realistic as possible, hence it is different to think about a colony of P systems of depth 2 (which stands as a formal model for a real multi-cellular tissue), or about a unique system with a big skin membrane enclosing inner P systems of any depth (which has no realistic counterpart).

Finally, we have seen that P systems which use gemmation of mobile membranes and no in/out communications can generate at least all languages in MAT, but it is still an open problem knowing if the family of generated languages can ever be enlarged using this new communication feature only.

References

1. L. Cardelli, A. G. Gordon, Mobile ambients, *Proceedings of FoSSaCS'98* (M. Nivat, ed.), LNCS 1378, 140–155. 137
2. J. Castellanos, A. Rodriguez-Paton, Gh. Păun, Computing with membranes: P systems with worm-objects, *IEEE 7th. Intern. Conf. on String Processing and Information Retrieval, SPIRE 2000*, La Coruna, Spain, 64–74. 138, 149
3. J. Dassow, Gh. Păun, *Regulated Rewriting in Formal Language Theory*, Springer-Verlag, Berlin, 1989. 145
4. M. R. Garey, D. S. Johnson, *Computers and intractability. A guide to the theory of NP-completeness*, 1979, W. H. Freeman and Company. 149
5. D. Hauschild, M. Jantzen, Petri nets algorithms in the theory of matrix grammars, *Acta Informatica*, 31 (1994), 719–728. 145
6. S. N. Krishna, R. Rama, P systems with replicated rewriting, *J. Automata, Languages, Combinatorics*, to appear. 136, 152
7. Gh. Păun, Computing with membranes, *Journal of Computer and System Sciences*, 61, 1 (2000), 108–143 (see also *Turku Center for Computer Science-TUCS Report* No 208, 1998, www.tucs.fi). 136, 140
8. Gh. Păun, Computing with membranes – A variant: P systems with polarized membranes, *Intern. J. of Foundations of Computer Science*, 11, 1 (2000), 167–182. 136
9. Gh. Păun, P systems with active membranes: Attacking NP complete problems, *J. Automat, Languages and Combinatorics*, 6, 1 (2001), 75–90. 136
10. Gh. Păun, G. Rozenberg, A. Salomaa, Membrane computing with external output, *Fundamenta Informaticae*, 41, 3 (2000), 259–266, and *Turku Center for Computer Science-TUCS Report* No 218, 1998 (www.tucs.fi). 136, 137
11. I. Petre, L. Petre, Mobile ambients and P systems, *Workshop on Formal Languages*, FCT99, Iaşi, 1999, *J. Universal Computer Sci.*, 5, 9 (1999), 588–598 (see also *Turku Center for Computer Science-TUCS Report* No 293, 1999, www.tucs.fi). 137
12. J. E. Rothman, L. Orci, Budding vesicles in cells, March 1996, *Scientific American*. 137
13. G. Rozenberg, A. Salomaa, eds., *Handbook of Formal Languages*, Springer-Verlag, Heidelberg, 1997. 138, 145, 152
14. D. Voet, J. G. Voet, *Biochemistry* (second edition), 1995, John Wiley and Sons, Inc. 137
15. C. Zandron, C. Ferretti, G. Mauri, Solving NP-complete problems using P systems with active membranes, in vol. *Unconventional Models of Computation* (I, Antoniou, C. S. Calude, M. J. Dinneen, eds.), Springer-Verlag, London, 2000, 289–301. 136

Instantaneous Actions vs. *Full Asynchronicity*: Controlling and Coordinating a Set of Autonomous Mobile Robots

Giuseppe Prencipe

Dipartimento di Informatica, Università di Pisa
Corso Italia, 40, 56100 - Pisa, Italy
prencipe@di.unipi.it

Abstract. Over the past few years, the focus of robotic design has been moving from a scenario where few, specialized (and expensive) units were used to solve a variety of tasks, to a scenario where many, general purpose (and cheap) units were used to achieve some common goal. Consequently, part of the focus has been to better understand how to efficiently coordinate and control a set of such "simpler" mobile units. Studies can be found in different disciplines, from engineering to artificial life: a shared feature of the majority of these studies has been the design of algorithms based on heuristics, without mainly being concerned with correctness and termination of such algorithms. Few studies have focused on trying to formally model an environment constituted by mobile units, studying which kind of capabilities they must have in order to achieve their goals; in other words, to study the problem from a *computational* point of view. This paper focuses on two of these studies [1,6,14] (the only ones, to our knowledge, that analyze the problem of coordinating and controlling a set of autonomous, mobile units from this point of view). First, their main features are described. Then, the main differences are highlighted, showing the relationship between the class of problems solvable in the two models.

Keywords: Mobile Robots, Distributed Coordination, Distributed Models, Computability.

1 Introduction

In a system consisting of a set of totally distributed agents the goal is generally to exploit the multiplicity of the elements in the system so that the execution of a certain number of predetermined tasks occurs in a coordinated and distributed way. Such a system is preferable to one made up of just one powerful robot for several reasons: the advantages that can arise from a distributed and parallel solution to the given problems, such as a faster computation; the ability to perform tasks which are unable to be executed by a single agent; increased fault tolerance; and, the decreased cost through simpler individual robot design. On the other hand, the main concern in such a system is to find an efficient way to

A. Restivo, S. Ronchi Della Rocca, L. Roversi (Eds.): ICTCS 2001, LNCS 2202, pp. 154–171, 2001.

coordinate and control the mobile units, in order to exploit to the utmost the presence of many elements moving independently.

Several studies have been conducted in recent years in different fields. In the engineering area we can cite the Cellular Robotic System (CEBOT) of Kawaguchi *et al.* [9], the Swarm Intelligence of Beni *et al.* [3], and the Self-Assembly Machine ("fructum") of Murata *et al.* [11]. In the AI community there has been a number of remarkable studies: social interaction leading to group behavior by Matarić [10]; selfish behavior of cooperative robots in animal societies by Parker [12]; and primitive animal behavior in pattern formation by Balch and Arkin [2].

The shared feature of all these approaches is that they do not deal with formal correctness and they are only analyzed empirically. Algorithmic aspects were somehow implicitly an issue, but clearly not a major concern - let alone the focus - of the study.

A different approach is to analyze an environment populated by a set of autonomous, mobile robots, aiming to identify the algorithmic limitations of what they can do. In other words, the approach is to study the problem from a *computational point of view*. This paper deals with two studies leading in this direction (the only ones, to our knowledge, that analyze the problem of coordinating and controlling a set of autonomous, mobile units from this point of view). The first study is by Suzuki *et al.* [1,13,14]. It gives a nice and systematic account on the algorithmics of pattern formation for robots, operating under several assumptions on the power of the individual robot. The second is by Flocchini *et al.* [6,8]: they present a model (that we will refer to as CORDA – COordination and control of a set of RObots in a totally Distributed and Asynchronous environment), that has as its primary objective to describe a set of simple mobile units, which have no central control, hence move independently from each other, which are totally asynchronous, and which execute the same deterministic algorithm in order to achieve some goal. In both studies, the modeled robots are rather *weak* and simple, but this simplicity allows us to formally highlight by an algorithmic and computational viewpoint the minimal capabilities they must have in order to accomplish basic tasks and produce interesting interactions. Furthermore, it allows us to better understand the power and limitations of the distributed control in an environment inhabited by mobile agents, hence to formally prove what can be achieved under the "weakness" assumptions of the models, that will be described later in more detail (see [7] for more detailed motivations).

An investigation with an algorithmic flavor has been undertaken within the AI community by Durfee [5], who argues in favor of limiting the knowledge that an intelligent robot must possess in order to be able to coordinate its behavior with others.

Although the model of Suzuki *et al.* (which we will refer to as SYm) and CORDA share some features, they differ in some aspects that render the two models quite different. In this paper we highlight these differences, focusing in particular on the different approach in modeling the asynchronicity of the environment in which the robots operate, and showing that the algorithms designed on SYm do not work in general on CORDA.

In Section 2.1, SYm and CORDA are described, highlighting the features that render the two models different. In Section 3, we show that the class of problems solvable in CORDA is strictly contained in the class of problems solvable in SYm. In Section 4, we present a case study: we analyze the *oblivious gathering* problem, showing that the algorithmic solutions designed for SYm do not work in CORDA. Finally, in Section 5 we draw some conclusions and present open problems and suggestions for further study.

2 Modeling Autonomous Mobile Robots

In this section we present the approaches used in SYm and CORDA to model the control and coordination of a set of autonomous mobile robots. In particular, we first present the common features in the two models, and successively present in detail the *instantaneous action* of SYm, and the *full asynchronicity* of CORDA, that model the interactions between the robots.

2.1 Common Features

The two models discussed in this paper share some basic features. The robots are modeled as units with computational capabilities, which are able to freely move in the plane. They are viewed as points, and they are equipped with sensors that let them observe the positions of the other robots in the plane. Depending on whether they can observe all the plane or just a portion of it, two different models can arise: *Unlimited* and *Limited Visibility* model (each robot can see only whatever is at most at distance V from it). The robots are *anonymous*, meaning that they are a priori indistinguishable by their appearances, and they do not have any kind of identifiers that can be used during the computation. They are *asynchronous* and no central control is allowed. Each robot has its own *local view* of the world. This view includes a local Cartesian coordinate system with origin, unit of length, and the *directions* of two coordinate axes, identified as x axis and y axis, together with their *orientations*, identified as the positive and negative sides of the axes. The robots do not necessarily share the same $x - y$ coordinate system, and do not necessarily agree on the location of the origin (that we can assume, without loss of generality, to be placed in the current position of the robot), or on the unit distance. They execute, however, the same deterministic algorithm, which takes in input the positions of the robots in the plane observed at a time instant t, and returns a destination point towards which the executing robot moves. The algorithm is *oblivious* if the new position is determined only from the positions of the others at t, and not on the positions observed in the past[1]; otherwise, it is called *non oblivious*. Moreover, there are no

[1] We also refer to the robots as *oblivious* because of this feature of the algorithms they execute.

explicit means of communication: the communication occurs in a totally implicit manner. Specifically, it happens by means of observing the change of robots' positions in the plane while they execute the algorithm.

Clearly, these basic features render the modeled robots simple and rather "weak", especially considering the current engineering technology. But, as already noted, the main interest in the studies done in [6,14], is to approach the problem of coordinating and controlling a set of mobile units from a *computational* point of view. The robots are modeled as "weak robots" because in this way it is possible to formally analyze the strengths and weaknesses of the distributed control. Furthermore, this simplicity can also lead to some advantages. For example, avoiding the ability to remember what has been computed in the past gives the system the nice property of self-stabilization [7,14].

During its life, each robot cyclically is in three *states*: (i) it observes the positions of the others in the world, (ii) it computes its next destination point, and (iii) it moves towards the point it just computed. As already stated, the robots execute these phases *asynchronously*, without any central control: in this feature the two models drastically differ. In fact, in SYm states (i) to (iii) are executed atomically (instantaneously), while this assumption is dropped in Corda. In the following we better describe how the asynchronicity is approached in the two models.

2.2 The *Instantaneous Actions* of SYm

In this section we better describe how the movement of the robots is modeled in SYm [1,14]. The authors assume discrete time $0, 1, 2, \ldots$. At each time instant t, every robot r_i is either *active* or *inactive*. At least one robot is active at every time instant, and every robot becomes active at infinitely many unpredictable time instants. A special case is when every robot is active at every time instant; in this case the robots are *synchronized*, but this case is not interesting for the purpose of this paper.

Let $p_i(t)$ indicate the position of robot r_i at time instant t, and ψ the algorithm every robot uses. Since the robots are viewed as points, in SYm it is assumed that two robots can occupy the same position simultaneously and never collide. ψ is a function that, given the positions of the robots at time t (or, in the non oblivious case, all the positions the robots have occupied since the beginning of the computation[2]), returns a new destination point p. For any $t \geq 0$, if r_i is inactive, then $p_i(t+1) = p_i(t)$; otherwise $p_i(t+1) = p$, where p is the point returned by ψ. The maximum distance that r_i can move in one step is bounded by a distance $\epsilon_i > 0$ (this implies that every robot is then capable of traveling at least a distance $\epsilon = \min\{\epsilon_1, \ldots, \epsilon_n\} > 0$). The reason for such a constant is to simulate a continuous monitoring of the world by the robots.

Thus, r_i executes the three states (i)–(iii) *instantaneously*, in the sense that a robot that is active and observes at t, has already reached its destination

[2] Note that the non obliviousness feature does not imply the possibility for a robot to find out which robot corresponds to which position it stored, since the robots are anonymous.

point p at $t+1$. Therefore, a robot takes a certain amount of time to move (the time elapsed between t and $t+1$), but no fellow robot can see it *while* it is moving (or, alternatively, the movement is *instantaneous*).

2.3 The *Full Asynchronicity* of CORDA

Similarly to SYm, each robot repeatedly executes four states. A robot is initially in a waiting state (*Wait*); at any point in time, asynchronously and independently from the other robots, it observes the environment in its area of visibility (*Look*), it calculates its destination point based only on the current locations of the observed robots (*Compute*), it then moves towards that point (*Move*) and goes back to a waiting state. The states are described more formally in the following.

1. **Wait** The robot is idle. A robot cannot stay infinitely idle.
2. **Look** The robot observes the world by activating its sensors which will return a snapshot of the positions of all other robots with respect to its local coordinate system. Each robot r is viewed as a point, and therefore its position in the plane is given by its coordinates. In addition, the robot cannot in general detect whether there is more than one fellow robot on any of the observed points, included the position where the observing robot is. We say it cannot detect *multiplicity.* If, on the other hand, a robot can recognize that there is more than one fellow on the positions where it is, we say that it can detect a *weak multiplicity.*
3. **Compute** The robot performs a *local computation* according to its deterministic algorithm. The result of the computation can be a destination point or a *null movement* (i.e., the robot decides to not move).
4. **Move** If the result of the computation was a *null movement*, the robot does not move; otherwise it moves towards the point computed in the previous state. The robot moves towards the computed destination of an unpredictable amount of space, which is assumed neither infinite, nor infinitesimally small (see Assumption A2 below). Hence, the robot can only go towards its goal, but it cannot know how far it will go in the current cycle, because it can stop anytime during its movement [3].

A computational cycle is defined as the sequence of the *Wait-Look-Compute-Move* states; the "life" of a robot is then a sequence of computational cycles.

In addition, we have the following assumptions on the behavior of a robot:

A1(Computational Cycle) The amount of time required by a robot r to complete a computational cycle is not infinite, nor infinitesimally small.

A2(Distance) The distance traveled by a robot r in a *Move* is not infinite. Furthermore, it is not infinitesimally small: there exists an arbitrarily small constant $\delta_r > 0$, such that if the result of the computation is not a *null*

[3] That is, a robot can stop before reaching its destination point, e.g. because of limits to the robot's motorial autonomy.

movement and the destination point is closer than δ_r, r will reach it; otherwise, r will move towards it of at least δ_r. In the following, we shall use $\delta = \min_r \delta_r$.

Therefore, in CORDA there is no assumption on the maximum distance a robot can travel before observing again (apart from the bound given from the destination point that has to be reached), while in SYm an active robot r_i always travels at most a distance ϵ_i in each step. The only assumption in CORDA is that there is a lower bound on such distance: when a robot r moves, it moves at least some positive, small constant δ_r. The reason for this constant is to better model reality: it is not realistic to allow the robots to move an infinitesimally small distance.

The main difference between the two models is, as stated before, in the way the asynchronicity is regarded. In CORDA the environment is *fully asynchronous*, in the sense that there is no common notion of time, and a robot observes the environment at unpredictable time instants. Moreover, no assumptions on the cycle time of each robot, and on the time each robot elapses to execute each state of a given cycle are made. It is only assumed that each cycle is completed in finite time, and that the distance traveled in a cycle is finite. Thus, each robot can take its own time to compute, or to move towards some point in the plane: in this way, it is possible to model different computational and motorial speeds of the units. Moreover, every robot can be seen *while* it is moving by other robots that are observing. This feature renders more difficult the design of an algorithm to control and coordinate the robots. For example, when a robot starts a *Move* state, it is possible that the movement it will perform will not be "coherent" with what it observed, since, during the *Compute* state, other robots can have moved.

3 *Instantaneous Action* vs. *Full Asynchronicity*

In this section, we highlight the relationship between the two models. In particular, we first show that any algorithm designed in CORDA to solve some problem $\mathcal{P}$ can be used in SYm to let the robots accomplish the task defined by $\mathcal{P}$. The vice versa is not true. In fact, we will give strong evidence that the differences pointed out in the previous sections, in particular the way in which the asynchronicity is modeled, render the two models *really* different, both in the oblivious and non oblivious case, and that the algorithms designed in SYm do not work in CORDA.

Let us first introduce the definition of a valid *activation schedule* for an algorithm in CORDA.

Definition 1. *Given an algorithm $\mathcal{A}$, an activation schedule for $\mathcal{A}$ in* CORDA *is defined as a function $\mathcal{F}(t) =< \mathbb{W}(t), \mathbb{L}(t), \mathbb{C}(t), \mathbb{M}(t) >$, where $\mathbb{W}(t)$ is a set of pairs (r, t'), such that*

1. *r is a robot that is in the* Wait *state at time t,*
2. *$t' > t$, and*

3. in $\mathbb{W}(t)$ there is at most one pair per each robot in the system

($\mathbb{L}(t)$, $\mathbb{C}(t)$, and $\mathbb{M}(t)$ are defined similarly for the Look, Compute, *and* Move *states, respectively).*

Definition 2. *An activation schedule is* valid*, if the following conditions hold: (i) $(r, t') \in \mathbb{W}(t) \Rightarrow \forall\, t \leq t'' < t'$, $(r, t') \in \mathbb{W}(t'')$ (a similar condition applies also for $\mathbb{L}(t)$, $\mathbb{C}(t)$, and $\mathbb{M}(t)$); (ii) for all t, $\mathbb{W}(t)$, $\mathbb{L}(t)$, $\mathbb{C}(t)$, and $\mathbb{M}(t)$ constitute a partition of all the robots in the system.*

An algorithm $\mathcal{A}$ correctly solves a problem $\mathcal{P}$ in Corda, if, given any valid activation schedule for $\mathcal{A}$, the robots accomplish the task defined by $\mathcal{P}$ in a finite number of cycles. Let us denote by $\mathfrak{C}$ and $\mathfrak{Z}$ the class of problem that are solvable in Corda and SYm, respectively. We are now ready to show that SYm is at least as powerful as Corda, that is $\mathfrak{C} \subseteq \mathfrak{Z}$.

Theorem 1. *Any algorithm that correctly solves a problem $\mathcal{P}$ in* Corda*, correctly solves $\mathcal{P}$ also in* SYm*.*

Proof. Let $\mathcal{A}$ be an algorithm that solves a given problem $\mathcal{P}$ in Corda. In order to prove that $\mathcal{A}$ solves $\mathcal{P}$ also in SYm, we show that any execution of $\mathcal{A}$ in SYm corresponds to an activation schedule in Corda. Hence, since by hypothesis $\mathcal{A}$ correctly solves $\mathcal{P}$ in Corda, the theorem follows.

Let us execute $\mathcal{A}$ in SYm, and let $\mathcal{E}(\overline{t})$ be the set of robots that are active at time $\overline{t}$. Therefore, all the robots $\mathcal{E}(\overline{t})$ finish to execute their cycle at time $\overline{t}+1$. The activation schedule $\mathcal{F}(t)$, for all $\overline{t} \leq t < \overline{t}+1$, in Corda for $\mathcal{A}$ corresponding to the portion of the execution of $\mathcal{A}$ in SYm starting at time $\overline{t}$ and ending at time $\overline{t}+1$, is defined as follows (see Figure 1). If $r \in \mathcal{E}(\overline{t})$, then for all $\overline{t} \leq t < t_1$, $(r, t_1) \in \mathbb{L}(t)$; for all $t_1 \leq t < t_2$, $(r, t_2) \in \mathbb{C}(t)$; for all $t_2 \leq t < t_3$, $(r, t_3) \in \mathbb{M}(t)$; and for all $t_3 \leq t < \overline{t}+1$, $(r, \overline{t}+1) \in \mathbb{W}(t)$. Otherwise, for all $\overline{t} \leq t < \overline{t}+1$, $(r, \overline{t}+1) \in \mathbb{W}(t)$. In other words, all the robots in $\mathcal{E}(\overline{t})$ start their *Look* state, while all the others are in *Wait*. Moreover, all these robots execute their three states perfectly synchronized, so that they start their next cycle all together. Inductively, $\mathcal{F}(t)$, for all $\overline{t}+1 \leq t < \overline{t}+2$, corresponding to the next cycle (from time $\overline{t}+1$ to $\overline{t}+2$) of the execution of $\mathcal{A}$ in SYm is constructed.

Therefore, any execution of $\mathcal{A}$ in SYm corresponds to a valid activation schedule for $\mathcal{A}$ in Corda. Since by hypothesis $\mathcal{A}$ correctly solves $\mathcal{P}$ on Corda, the robots will correctly accomplish their task in SYm, and the theorem follows.

Corollary 1. *Any problem that can be solved in* Corda*, can be solved in* SYm*; hence $\mathfrak{C} \subseteq \mathfrak{Z}$.*

To prove that the inclusion is strict, we place ourselves in the *non oblivious* setting: the robots have an unlimited amount of memory, hence they can remember the positions of all the other robots since the beginning of the execution, and they can use this information while computing.

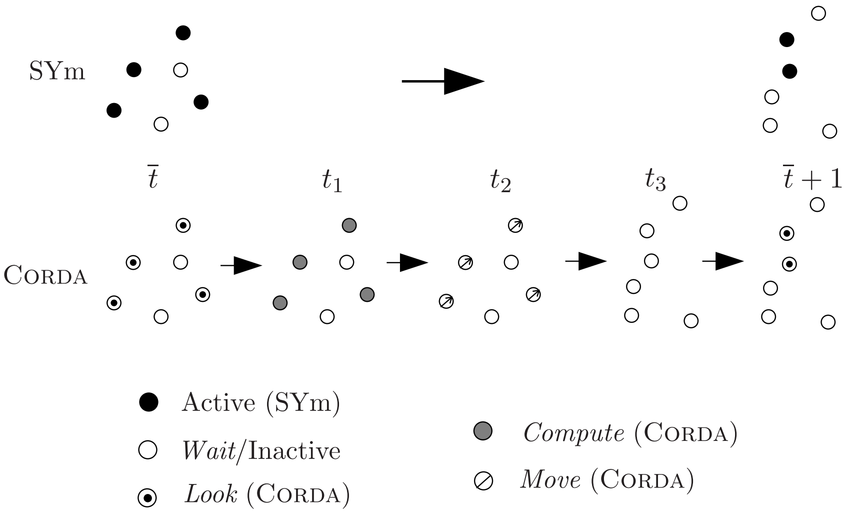


Fig. 1. The activation schedule defined in Theorem 1

Definition 3 (Movement Awareness). *The* Movement Awareness *problem $\mathcal{MA}$ is divided in two subtasks $\mathcal{T}_1$ and $\mathcal{T}_2$. In $\mathcal{T}_1$, robot r_i, $1 \leq i \leq n$, simply moves along a direction it chooses arbitrarily; r_i can start $\mathcal{T}_2$ only after it observed r_j in at least three different positions, and after r_j observed r_i in at least three different positions, for all $j \neq i$.*

Theorem 2. *There exists no algorithm that solves $\mathcal{MA}$ in* CORDA *in the non oblivious setting.*

Proof. By contradiction, let us assume that there exists an algorithm $\mathcal{A}$ that correctly solves $\mathcal{MA}$ in CORDA. The generic robot r starts its execution by moving along the direction it chooses. By hypothesis, it will eventually and within a finite number of cycles start the second subtask. Let t be the time when r decides to switch to $\mathcal{T}_2$. Since the robots operate in full asynchronicity, there can exist a robot r' that started its first *Move* state at time $t' < t$, and is still moving at time t (that is r' is still executing its first cycle). Then $\mathcal{MA}$ is not correctly solved, since r' has not started its second cycle at time t yet, hence r' has not observed r in at least three different positions yet, having a contradiction.

An algorithm similar to the one used in [14] to discover the initial configuration ("distribution") of the robots in the system, can be used to solve in SYm $\mathcal{MA}$. Namely, each robot starts moving along the direction it locally chooses, e.g. the direction of its local y axis. When a robot r observes another robot r' in at least three different positions, r moved at least twice. Moreover, since in SYm the actions are instantaneous, r can correctly deduce that r' observed at least twice, hence that r' observed r in at least three different positions.

Therefore, r can correctly start $\mathcal{T}_2$ when it observes all $r' \neq r$ in at least three different positions. Hence, we can state the following

Theorem 3. *$\mathcal{MA}$ is solvable in* SYm, *in the non oblivious setting.*

Corollary 2. $\mathfrak{C} \subset \mathfrak{Z}$.

A question that arises is: what does it happen in the oblivious case? Unfortunately, we do not yet have an answer. Our conjecture, however, is that the result stated in Corollary 2 holds also in the oblivious case. In the non oblivious setting, the fact that in CORDA a robot can be seen by its fellows *while* it is moving is crucial to prove $\mathfrak{C} \subset \mathfrak{Z}$. This is not the case in the oblivious setting. In fact, since the robots have no memory of robots' positions observed in the past, every time a robot r observes another robot r', r can not tell if r' moved since last cycle or not, and every observation is like the first one (that is every time r observes, is like the execution begins). Hence, we believe that the key to prove $\mathfrak{C} \subset \mathfrak{Z}$ in the oblivious case is related to the fact that in CORDA the positions of the robots between a *Look* and a *Compute* can change, hence the computation can be done on "outdated" data. In other words, if r executes the *Look* at time t and the *Compute* at time $t' > t$, the set of robots' positions at t and at t' can be clearly different; hence r computes its destination point on the old data sensed at time t, implying that the movement will not be "choerent" with what it observed at time t. This clearly does not happen in SYm, where the possible states a robot can be in are executed instantaneously.

4 Case Study: Oblivious Gathering

In this section, we will give evidence that the algorithms designed in SYm in the oblivious setting do not work in general in CORDA.

The problem we consider is the *gathering problem*: the robots are asked to gather in a not predetermined point in the plane in a finite number of cycles. An algorithm is said *to solve* the gathering problem if it lets the robots gather in a point, given any *initial configuration*. An initial configuration is the set of robots' positions when the computation starts, one position per robot, with no position occupied by more than one robot. This is the only problem, to our knowledge, solved with an oblivious algorithm in SYm [1,14]. In the following, we will analyze both the unlimited and limited visibility setting.

4.1 The Unlimited Visibility Setting

An algorithm for solving the gathering problem in SYm in the unlimited visibility setting (called Algorithm 1 in Appendix A.1) is presented in [14]. The idea is as follows. Starting from distinct initial positions, the robots are moved in such a way that eventually there will be exactly one position, say p, that two or more robots occupy. Once such a situation has been reached, all the robots move towards p. It is clear that such a strategy works only if the robots in the

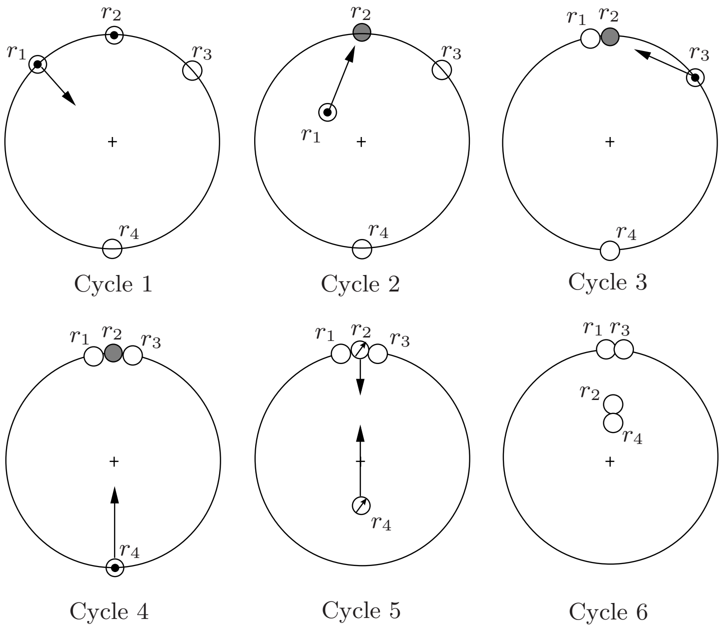


Fig. 2. Proof of Theorem 4. The symbols used for the robots are the same as in Figure 1. The dotted circles indicate the robots in the *Look* state; the grey ones the robots in the *Compute* state; the circle with an arrow inside are the robots that are moving; the white circles represent the robots in *Wait*. The arrows indicate the direction of the movement computed in the *Compute* state

system have the ability to detect the multiplicity. In SYm this capability is never mentioned, but it is clearly used implicitly.

Theorem 4. *Algorithm 1 does not solve the gathering problem in* CORDA, *in the unlimited visibility setting.*

Proof. In order to show that Algorithm 1 does not solve the gathering problem in CORDA, we give an initial configuration of the robots and describe an activation schedule that leads to having two points in the plane with multiplicity greater than two, thus violating the invariant proven for Algorithm 1, that "eventually there will be exactly one position that two or more robots occupy" [14].

Let us suppose to have 4 robots r_i, $i = 1, 2, 3, 4$, that at the beginning are on a circle C, with r_2 and r_4 that occupy the ending points of a diameter of C (as pictured in Figure 2, Cycle 1). In the following, the positions of the robots are indicated by p_i, $i = 1, 2, 3, 4$. Executing Algorithm 1, but assuming the features of CORDA, a possible run (activation schedule) is described in the following.

Cycle 1 At the beginning the four robots are in distinct positions, on a circle C.
r_1 and r_2 enter the *Look* state, while the others are in *Wait*. After having

observed, both of them enter the *Compute* state, and let us assume that r_2 is computationally very slow (or, alternatively, that r_1 is very fast). Therefore, r_1 decides to move towards the center of C (part 2.3 of Algorithm 1), while r_2 is stuck in its *Compute* state. r_1 starts moving towards the center, while r_2 is still in *Compute*, and r_3 and r_4 are in *Wait*.

Cycle 2 r_1 is inside C, while the other robots are still on C. Now r_1 observes again (already in its second cycle) and, according to part 2.1 of the algorithm, decides to move toward a robot that is on the circle, say r_2. Moreover, r_2 is still in the *Compute* state of its first cycle, and r_3 and r_4 are in *Wait*.

Cycle 3 r_1 reaches r_2 and enters the *Wait* of its third cycle: at this point, there is one position in the plane with two robots, namely $p = p_1 = p_2$. Now, r_3 enters its first *Look* state, looks at the situation and, according to the algorithm, decides to move towards p, that is the only point in the plane with more than ore robots on it. r_2 is still in its first *Compute*, and r_4 in *Wait*.

Cycle 4 r_3 reaches r_1 and r_2 on p, and it starts waiting. r_1 is in *Wait*, r_2 still in its first *Compute* state, and r_4 starts its first *Look* state, decides to move towards p, and starts moving.

Cycle 5 While r_4 is on its way towards p, r_2 ends its first *Compute* state. Since the computation is done according to what it observed in its previous *Look* state (Cycle 1), it decides to move towards the center of C (part 2.3 of the algorithm). r_2 starts moving towards the center of C after r_4 passes over the center of C, and while r_4 is still moving towards p; r_1 is in *Wait*.

Cycle 6 r_2 and r_4 are moving in opposite directions on the same diameter of C, and they stop exactly on the same point p' (in CORDA a robot can stop before reaching its final destination). There are two points in the plane, namely p and p' with $p \neq p'$, with two robots on each. Therefore, the invariant proven for Algorithm 1, that "eventually there will be exactly one position that two or more robots occupy" [14], is violated.

Remark 1. We note that in Cycle 6 we made use of the possibility that a robot stops before reaching the destination point it computed. The proof, however, works even if we do not assume this; that is, if r_2 and r_4 do not stop before reaching their respective destination points. In fact, if we assume, as in SYm, that the robots simply cross each other without stopping, if (i) the crossing happens in a point $p' \neq p$, and (ii) r_1 enters its Observe phase exactly when the crossing happens, we have that r_1 sees two points in the plane with two robots on each, namely p and p', and does not know what to do, since this possibility is not mentioned in SYm's algorithm. Therefore, Theorem 4 still holds.

4.2 The Limited Visibility Setting

In [1], an algorithm to solve the gathering problem in SYm in the limited visibility setting (called Algorithm 2 in Appendix A.2) is presented. We recall that, in this setting a robot can see only whatever is at distance V from it. In the following we shortly describe it.

Let us denote by $r_i(t)$ the position of robot r_i at time instant t. The set $P(t) = \{r_1(t), \ldots, r_n(t)\}$ then denotes the set of the robots' positions at t. Define $G(t) = (R, E(t))$, called the *Proximity Graph* at time t, by $(r_i, r_j) \in E(t) \leftrightarrow dist(r_i(t), r_j(t)) \leq V$, where $dist(p, q)$ denotes the Euclidean distance between points p and q. It can be proven that, if the proximity graph is not connected at the beginning, the robots can not gather in a point [1] (*form* a point, in SYm language).

Let $S_i(t)$ denote the set of robots that are within distance V from r_i at time t; that is, the set of robots that are visible from r_i (note that $r_i \in S_i(t)$). $C_i(t)$ denotes the smallest enclosing circle of the set $\{r_j(t) | r_j \in S_i(t)\}$ of the positions of the robots in $S_i(t)$ at t. The center of $C_i(t)$ is denoted $c_i(t)$.

Every time a robot r_i becomes active, the algorithm moves r_i toward $c_i(t)$, but only over a certain distance $MOVE$. Specifically, if r_i does not see any robot other than itself, then r_i does not move. Otherwise, the algorithm chooses x to be the point on the segment $\overline{r_i(t)c_i(t)}$ that is closest to $c_i(t)$ and that satisfies the following conditions:

1. $dist(r_i(t), x) \leq \sigma$. An arbitrary small constant $\sigma > 0$ is fixed, and it is assumed that the distance a robot can travel in one state is bounded by σ (similarly to the ϵ introduced in Section 2.2).
2. For every robot $r_j \in S_i(t)$, x lies in the disk D_j whose center is the midpoint m_j of $r_i(t)$ and $r_j(t)$, and whose radius is $V/2$. This condition ensures that r_i and r_j will still be visible after the movement of r_i (and possibly of r_j, see Figure 3.a).

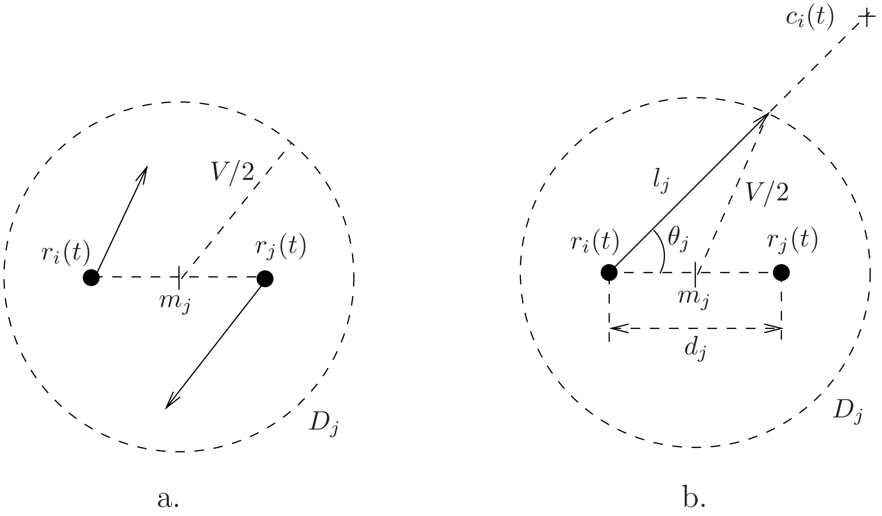


Fig. 3. The algorithm for the gathering problem in SYm, limited visibility setting

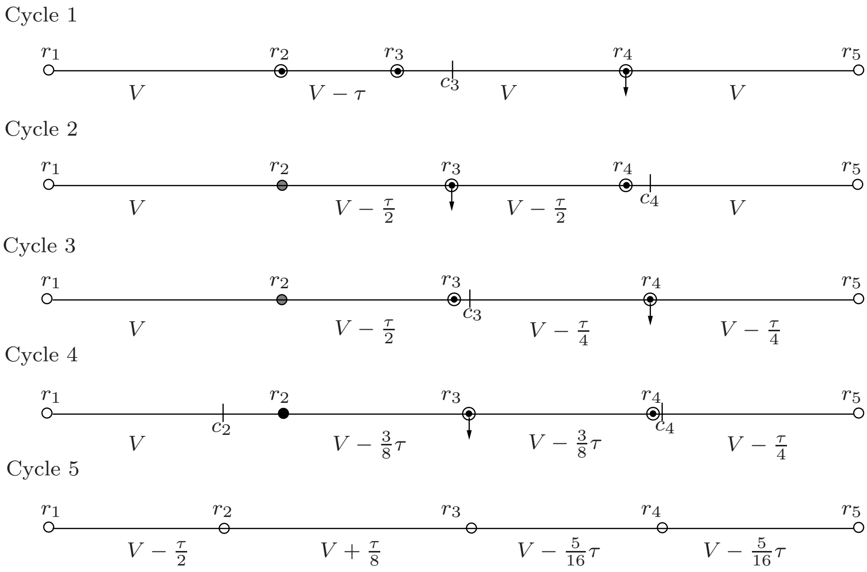


Fig. 4. Proof of Theorem 5. The symbols used for the robots are the same as in Figure 2. A vertical arrow means that a robot decided not to move ($Move = 0$). A robot r_i moves always towards the center c_i of the smallest enclosing circle of all the robots it can see

We note that, since by condition 1. the algorithm uses the constant σ to compute the destination point of a robot, all the robots must agree on the value of this constant, and thus it must be a priori known.

In [1] it is proven that, executing Algorithm 2, two robots that are connected in $G(t)$, will be connected in $G(t+1)$. In the following theorem we prove that it does not solve the gathering problem in CORDA, in the limited visibility setting. Specifically, we give an initial configuration of the robots and describe a possible run of the algorithm that leads to partitioning the proximity graph: two robots that were visible until time t, are not visible any more at $t+1$, contradicting the result proven in [1].

Theorem 5. *The algorithm presented in [1] does not solve the gathering problem in* CORDA*, in the limited visibility setting.*

Proof. In order to show that Algorithm 2 does not solve the gathering problem in CORDA, we give an initial configuration of the robots and describe an activation schedule that leads to partitioning the proximity graph: two robots that were visible until time t, are not visible any more at $t+1$, contradicting the result proven in [1].

Let us suppose to have at the beginning 5 robots on a straight line, as shown in Figure 4. Moreover, let τ be a constant such that $\tau \leq \sigma$ and $\delta = \tau/16$, where δ is the constant introduced in the Assumption A2 in Section 2.3. At the beginning, we have the following *visibility situation*: r_1 can see r_2 ($dist(r_1, r_2) = V$), r_2 can see r_1 and r_3 ($dist(r_2, r_3) = V - \tau$), r_3 can see r_2 and r_4 ($dist(r_3, r_4) = V$), r_4 can see r_3 and r_5 ($dist(r_4, r_5) = V$), r_5 can see r_4. We recall that a robot r_i always move towards the center c_i of the smallest circle enclosing all the robots it can see. Executing Algorithm 2, but assuming the features of Corda, a possible run is described in the following.

Cycle 1 All the robots, except r_1 and r_5 (that we assume in *Wait*), execute their first *Look*, and start the *Compute* state. Let us suppose that r_3 and r_4 are faster than r_2 in computing. The values they compute are:

$$r_3\colon \quad \begin{cases} Goal = dist(r_3, c_3) = \left|\frac{V-\tau+V}{2} - V + \tau\right| = \frac{\tau}{2} \\ Limit = \min\{-\frac{V-\tau}{2} + \frac{V}{2}, V\} = \frac{\tau}{2} \end{cases} \Rightarrow Move = \frac{\tau}{2}$$

$$r_4\colon \quad Goal = 0 \;\Rightarrow Move = 0$$

Moreover, r_3 and r_4 also start moving while r_2 is still computing; r_1 and r_5 are in *Wait*.

Cycle 2 After r_3 and r_4 move, the visibility situation is the same as it was in the beginning. r_3 and r_4 *Look* and *Compute* again, as follows:

$$r_3\colon \quad Goal = 0 \;\Rightarrow Move = 0$$

$$r_4\colon \quad \begin{cases} Goal = dist(r_4, c_4) = \left|\frac{V+V-\frac{\tau}{2}}{2} - V + \frac{\tau}{2}\right| = \frac{\tau}{4} \\ Limit = \min\{-\frac{V-\frac{\tau}{2}}{2} + \frac{V}{2}, V\} = \frac{\tau}{4} \end{cases} \Rightarrow Move = \frac{\tau}{4}$$

r_3 and r_4 move again, while r_2 is still in its first *Compute* state, and r_1 and r_5 in their first *Wait*.

Cycle 3 After the movement of the previous cycle, the visibility situation is still unchanged, that is, the proximity graph is still connected. r_3 and r_4 enter their third *Look* and *Compute* states.

$$r_3\colon \quad \begin{cases} Goal = dist(r_3, c_3) = \left|\frac{V-\frac{\tau}{2}+V-\frac{\tau}{4}}{2} - V + \frac{\tau}{2}\right| = \frac{\tau}{8} \\ Limit = \min\{-\frac{V-\frac{\tau}{2}}{2} + \frac{V}{2}, \frac{V-\frac{\tau}{4}}{2} + \frac{V}{2}\} = \frac{\tau}{4} \end{cases} \Rightarrow Move = \frac{\tau}{8}$$

$$r_4\colon \quad Goal = 0 \;\Rightarrow Move = 0$$

r_3 and r_4 move again. The other robots are in the same states as in the previous cycle.

Cycle 4 The proximity graph is still connected. r_3 and r_4 *Look* and *Compute* again (this is their fourth cycle).

$$r_3\colon \quad Goal = 0 \;\Rightarrow Move = 0$$

$$r_4\colon \quad \begin{cases} Goal = dist(r_4, c_4) = \left|\frac{V-\frac{3}{8}\tau+V-\frac{\tau}{4}}{2} - V + \frac{3}{8}\tau\right| = \frac{\tau}{16}\tau \\ Limit = \min\{-\frac{V+\frac{3}{8}\tau}{2} + \frac{V}{2}, \frac{V-\frac{\tau}{4}}{2} + \frac{V}{2}\} = \frac{3}{16}\tau \end{cases} \Rightarrow Move = \frac{\tau}{16}$$

r_3 and r_4 enter the *Move* state. Meanwhile, r_2 finishes its first *Compute*. The values it computes refer to what was the situation when it observed, in Cycle 1.

$$r_2\colon \quad \begin{cases} Goal = dist(r_2, c_2) = \left|\frac{V+V-tau}{2} - V\right| = \frac{\tau}{2} \\ Limit = \min\{V, -\frac{V-\tau}{2} + \frac{V}{2}\} = \frac{\tau}{2} \end{cases} \Rightarrow Move = \frac{\tau}{2}$$

r_2 starts moving according to the destination point it just computed (it enters it first *Move* state).

Cycle 5 The distance between r_2 and r_3 is $V + \tau/8 > V$; so r_2 and r_3 can not see each other anymore, breaking the proximity graph connectivity that we had at the beginning of the cycle. So, the invariant that "robots that are mutually visible at t remain within distance V of each other thereafter" asserted in [1] is violated. Therefore, the theorem follows.

5 Conclusions

In this paper we discussed two models, SYm [1,14], and CORDA [6,7,8], whose main focus is to study the algorithmic problems that arise in an asynchronous environment populated by a set of autonomous, anonymous, mobile units that are requested to accomplish some given task. These studies want to gain a better understanding of the power of the distributed control from an algorithmic point of view; specifically, the goal is to understand what kind of goals such a set of robots can achieve, and what are the minimal requirements and capabilities that they must have in order to do so. To our knowledge, these are the only approaches to the study of the control and coordination of mobile units in this perspective.

We showed that the different way in which the asynchronicity is modeled in SYm and CORDA, is the key feature that renders the two models different: in SYm the robots operate executing *instantaneous actions*, while in CORDA *full asynchronicity* is modeled, and the robots elapses finite, but otherwise unpredictable, amount of time to execute their states. In particular, we showed that $\mathfrak{C} \subset \mathfrak{Z}$ in the non oblivious setting. Therefore, one open issue is to prove this result also in the oblivious setting.

We feel that the approach used in CORDA better describes the way a set of independently-moving units operates in a totally asynchronous environment; hence the motivation to further investigate coordination problems in a distributed, asynchronous environment using the *fully asynchronous* approach. Issues which merit further research, regard the operating capabilities of the robots modeled. In fact, it would be interesting to look at models where robots have different capabilities. For instance, we could equip the robots with just a bounded amount of memory (*semi-obliviousness*), and analyze the relationship between amount of memory and solvability of the problems, or how it would affect the self-stability property of the oblivious algorithms [7].

Other features that would inspire further study include giving a dimension to the robots, and adding stationary obstacles to the environment, thus adding the possibility of collision between robots or between moving robots and obstacles. Furthermore, we could also study how the robots can use some kind of direct communication, and we could introduce different kinds of robots that move in the environment (as in the *intruder* problem, where all the robots in the environment must chase and "catch" a "designated" robot).

Relationship between memory and ability of the robots to complete given tasks, dimensional robots, obstacles in the environment that limit the visibility and that moving robots must avoid or push aside, suggest that the algorithmic nature of distributed coordination of autonomous, mobile robots merits further investigation.

Acknowledgments

I would like to thank Paola Flocchini, Nicola Santoro and Vincenzo Gervasi for the discussions and comments that helped with the writing of this paper.

References

1. H. Ando, Y. Oasa, I. Suzuki, and M. Yamashita. A Distributed Memoryless Point Convergence Algorithm for Mobile Robots with Limited Visibility. *IEEE Trans. on Robotics and Automation*, 15(5):818–828, 1999. 154, 155, 157, 162, 164, 165, 166, 168, 170, 171
2. T. Balch and R. C. Arkin. Behavior-based Formation Control for Multi-robot Teams. *IEEE Trans. on Robotics and Automation*, 14(6), December 1998. 155
3. G. Beni and S. Hackwood. Coherent Swarm Motion Under Distributed Control. In *Proc. DARS'92*, pages 39–52, 1992. 155
4. Y. U. Cao, A. S. Fukunaga, A. B. Kahng, and F. Meng. Cooperative Mobile Robotics: Antecedents and Directions. In *Int. Conf. on Intel. Robots and Sys.*, pages 226–234, 1995.
5. E. H. Durfee. Blissful Ignorance: Knowing Just Enough to Coordinate Well. In *ICMAS*, pages 406–413, 1995. 155
6. P. Flocchini, G. Prencipe, N. Santoro, and P. Widmayer. Hard Tasks for Weak Robots: The Role of Common Knowledge in Pattern Formation by Autonomous Mobile Robots. In *ISAAC '99*, pages 93–102, 1999. 154, 155, 157, 168
7. P. Flocchini, G. Prencipe, N. Santoro, and P. Widmayer. Distributed Coordination of a Set of Autonomous Mobile Robots. In *IEEE Intelligent Veichle 2000*, pages 480–485, 2000. 155, 157, 168
8. P. Flocchini, G. Prencipe, N. Santoro, and P. Widmayer. Gathering of Asynchronous Mobile Robots with Limited Visibility. In *STACS 2001*, volume 2010 of *Lecture Notes in Computer Science*, pages 247–258, 2001. 155, 168
9. Y. Kawauchi and M. Inaba and T. Fukuda. A Principle of Decision Making of Cellular Robotic System (CEBOT). In *Proc. IEEE Conf. on Robotics and Autom.*, pages 833–838, 1993. 155
10. M. J. Matarić. *Interaction and Intelligent Behavior*. PhD thesis, MIT, May 1994. 155
11. S. Murata, H. Kurokawa, and S. Kokaji. Self-Assembling Machine. In *Proc. IEEE Conf. on Robotics and Autom.*, pages 441–448, 1994. 155
12. L. E. Parker. On the Design of Behavior-Based Multi-Robot Teams. *Journal of Advanced Robotics*, 10(6), 1996. 155
13. K. Sugihara and I. Suzuki. Distributed Algorithms for Formation of Geometric Patterns with Many Mobile Robots. *Journal of Robotics Systems*, 13:127–139, 1996. 155
14. I. Suzuki and M. Yamashita. Distributed Anonymous Mobile Robots: Formation of Geometric Patterns. *Siam J. Comput.*, 28(4):1347–1363, 1999. 154, 155, 157, 161, 162, 163, 164, 168, 170, 171

Appendix

A Oblivious Gathering in SYm

In this appendix, we report the oblivious algorithms described in [1,14] that let the robots gather in a point in SYm, in both the unlimited and limited visibility settings.

A.1 Unlimited Visibility

In the following we report the oblivious algorithm described in [14] that lets the robots achieve a configuration where a unique point p with multiplicity greater than one is determined.

Algorithm 1 (Point Formation Algorithm in SYm**, Unlim. Visib.**[14]**)**

Case 1. $n = 3$; p_1, p_2, and p_3 denote the positions of the three robots.

- **1.1.** If $n = 3$ and p_1, p_2, and p_3 are collinear with p_2 in the middle, then the robots at p_1 and p_3 move towards p_2 while the robot at p_2 remains stationary. Then eventually two robots occupy p_2.
- **1.2.** If $n = 3$ and p_1, p_2, and p_3 form an isosceles triangle with $|\overline{p_1p_2}| = |\overline{p_1p_3}| \neq |\overline{p_2p_3}|$, then the robot at p_1 moves toward the foot of the perpendicular drop from its current position to $\overline{p_2p_3}$ in such a way that the robots do not form an equilateral triangle at any time, while the robots at p_2 and p_3 remain stationary. Then eventually the robots become collinear and the problem is reduced to part 1.1.
- **1.3.** If $n = 3$ and the lengths of the three sides of triangle p_1, p_2, p_3 are all different, say $|\overline{p_1p_2}| > |\overline{p_1p_3}| > |\overline{p_2p_3}|$, then the robot at p_3 moves toward the foot of the perpendicular drop from its current position to $\overline{p_1p_2}$ while the robots at p_1 and p_2 remain stationary. Then eventually the robots become collinear and the problem is reduced to part 1.1.
- **1.4.** If $n = 3$ and p_1, p_2, and p_3 form an equilateral triangle, then every robot moves towards the center of the triangle. Since all robots can move up to at least a constant distance $\epsilon > 0$ in one step, if part 1.4. continues to hold then eventually either the robots meet at the center, or the triangle they form becomes no longer equilateral and the problem is reduced to part 1.2 or part 1.3.

Case 2. $n \geq 4$; C_t denotes the smallest enclosing circle of the robots at time t.

- **2.1.** If $n \geq 4$ and there is exactly one robot r in the interior of C_t, then r moves toward the position of any robot, say r', on the circumference of C_t while all other robots remain stationary. Then eventually r and r' occupy the same position.
- **2.2.** If $n \geq 4$ and there are two or more robots in the interior of C_t, then these robots move toward the center of C_t while all other robots remain stationary (so that the center of C_t remains unchanged). Then eventually at least two robots reach the center.

2.3. If $n \geq 4$ and there are no robots in the interior of C_t, then every robot moves toward the center of C_t. Since all robots can move up to at least a constant distance $\epsilon > 0$ in one step, if part 2.3 continues to hold,then eventually the radius of C_t becomes at most ϵ. Once this happens, then the next time some robot moves, say, at t', either (i) two or more robots occupy the center of C_t or (ii) there is exactly one robot r at the center of C_t, and therefore there is a robot that is not on $C_{t'}$ (and the problem is reduced to part 2.1 or part 2.2) since a cycle passing through r and a point on C_t intersects with C_t at most at two points.

A.2 Limited Visibility

In the following we report the oblivious algorithm described in [14] that lets the robots gather in a point (refer to Figure 3.b).

Algorithm 2 (Point Formation Algorithm in SYm**, Lim. Visib.** [1] **)**

```
1. If $S_i(t) = \{r_i\}$, then $x = r_i(t)$.
2. $\forall r_j \in S_i(t) - \{r_i\}$,
   2.1.  $d_j = dist(r_i(t), r_j(t))$,
   2.2.  $\theta_j = c_i(t)\widehat{r_i(t)}r_j(t)$,
   2.3.  $l_j = (d_j/2)\cos\theta_j + \sqrt{(V/2)^2 - ((d_j/2)\sin\theta_j)^2}$,
3. $LIMIT = \min_{r_j \in S_i(t)-\{r_i\}}\{l_j\}$,
4. $GOAL = dist(r_i(t), c_i(t))$,
5. $MOVE = \min\{GOAL, LIMIT, \sigma\}$,
6. $x =$ point on $\overline{r_i(t)c_i(t)}$ at distance $MOVE$ from $r_i(t)$.
```

Some Structural Properties of Associative Language Descriptions[*]

Alessandra Cherubini[1], Stefano Crespi Reghizzi[2,°], Pierluigi San Pietro[2]

[1] Dipartimento di Matematica, Politecnico di Milano
P.za L. da Vinci 32, 20133 Milano, Italia
aleche@mate.polimi.it
[2] Dipartimento di Elettronica e Informazione, Politecnico di Milano
P.za Leonardo da Vinci 32,
20133 Milano, Italia
crespi@elet.polimi.it
pierluigi.sanpietro@polimi.it

Abstract. The Associative Language Description model (ALD), a combination of locally testable and constituent structure ideas, has been recently proposed to overcome some criticisms relative to context-free languages. This approach is consistent with current views on brain organization and can conveniently describe typical technical languages such as Pascal or HTML. ALD languages are strictly enclosed in context-free languages but in practice the ALD model equals context-free grammars in explanatory adequacy. Moreover, it excludes mathematical sets based on counting properties that are never used in the definition of artificial languages. Many properties of ALD are still to be investigated. Here, a characterization of context free languages in term of ALD languages is proved and a new hierarchy in the ALD family is given.

1 Introduction

In spite of their universal adoption in language reference manuals and compilers, Context-Free (CF) Grammars have a generative capacity that is partly misdirected: it affords languages that are unsuitable for practical use, like counting languages, which characterize the legal strings by some numerical congruence. Clearly, nobody has ever proposed a language where grammaticality depends on the number of certain items being congruous to some integer value. In an attempt to rule out counting, years ago the class of NC CF languages has been introduced for parenthesis grammars [1], and later on reformulated within the theory of tree languages [2].

[*] Work partially supported by CNR-CESTIA.

[°] Lecturer of Formal Languages, Università Svizzera Italiana

A. Restivo, S. Ronchi Della Rocca, L. Roversi (Eds.): ICTCS 2001, LNCS 2202, pp. 172-183, 2001.

Another defect of CF grammars that has been often pointed out (for instance by the Marcus' school of Contextual Grammars) is that CF grammars require an unbounded number of metasymbols, the nonterminals. A "pure" grammar should not use metavariables, which are "external" to the language, but it should rely instead on structural and distributional properties.

In [3], a language definition technique has been presented that addresses both criticisms, but does not extend the capacity of CF grammars: the Associative Language Description model (ALD), originally motivated by the want of a brain compatible theory of language. In essence, this definition combines the concepts of local testability and of phrase structure in as simple a way as possible and it is related with Z. Harrys's linguistic models of word distribution in sentences. Such approaches, also known as Skinner's associative models, were antagonized by Chomsky's generative grammars and had no comparable success. Yet, associative models on the one hand provide an intuitively appealing explanation of many linguistic regularities, on the other they are aligned with current views on information processing in the brain [4].

The ALD model has been studied more in depth in [5], where basic properties of the model were established, such as nonclosure under union, concatenation and homomorphism, and strict inclusion in the CF family; moreover, the ALD family of languages was compared with CF, Non-Counting (NC) CF, locally testable, non-contextual families of languages, and other families. However, many problems still remain open, the main one being the inclusion of regular languages in the ALD family. The expressive adequacy of the ALD family for common artificial languages, such as Pascal and HTML, has been shown in [6].

The aim of this paper is to solve a few of the open questions, by establishing a new hierarchy in the ALD family and giving a characterization of CF languages in terms of the ALD family. Section 2 recalls the basic definitions and some properties of the model, while Section 3 proves the main theorems of the paper. Section 4 draws a few conclusions.

2 Basic Definitions

Let Σ be a finite alphabet, and let $\Delta \notin \Sigma$ be the *placeholder*.

Definition 2.1. (stencil trees, frontier, constituents)

A *stencil tree* is a tree such that its internal nodes are labeled by Δ and its leaves have labels in $\Sigma \cup \{\varepsilon\}$. The *constituents* of a stencil tree are its subtrees of height one and leaves with labels in $\Sigma \cup \{\varepsilon\} \cup \{\Delta\}$. The *frontier* of a stencil tree T or of a constituent K is denoted, respectively, by τ (T) and τ (K).

Definition 2.2*. (maximal subtree)*

Given a stencil tree T, a *maximal subtree* of T is a subtree of T whose leaves are also leaves of T.

Definition 2.3. *(left and right contexts)*

Let T be a stencil tree. For an internal node i of T, let K_i and T_i be respectively the constituent and the maximal subtree of T having root i. Consider the tree T' obtained by excising the subtree T_i from T, leaving only the root, labeled Δ, of T_i behind. Let s, t $\in \Sigma^*$ be two strings such τ (T') = s Δ t.

The *left context* of K_i in T and of T_i in T is *left*(K_i, T) = *left*(T_i, T) = s; the *right context* of K_i in T and of T_i in T is *right*(K_i, T) = *right*(T_i, T) = t.

Definition 2.4. *(ALD, pattern, permissible contexts of a rule)*

Let $\perp \notin \Sigma$ be the *left/right terminator*. An Associative Language Description (ALD) A is a finite collection of triples (x, z, y), called *rules*, where x $\in \{\varepsilon \cup \perp\}\Sigma^*$, y$\in$ $\Sigma^*\{\perp \cup \varepsilon$ }, and z $\in$ $(\Sigma \cup \Delta)^* \setminus \{\Delta\}$.

The string z is called the *pattern* of the rule (x, z, y) and the strings x and y are called the *permissible left/right contexts*.

Shorthand notations
When a left/right context is irrelevant for a pattern, it is represented by the empty string ε or it is omitted. The new symbol Λ may be used to denote the optionality of one occurrence of Δ, that is to merge two rules (x, z' Δ z", y) and (x, z' z", y) into the rule (x, z' Λz", y). Other convenient shorthands have been defined in [6], but they are not used in this paper.

An ALD defines a set of constraints or test conditions that a stencil tree must satisfy, in the following sense.

Definition 2.5. *(Constituent matched by a rule, valid trees)*

Let A be an ALD. A constituent K_i of a stencil tree T is *matched* by a rule (x, z, y) of an ALD A iff:

1) z = $\tau(K_i)$,
2) x is a suffix of $\perp$left(K_i, T), and
3) y is a prefix of right(K_i, T)$\perp$.
A stencil tree T is *valid* for A iff each constituent K_i of T is matched by a rule of A.

Therefore, an ALD is a device for defining a set of stencil trees and a string language, corresponding to the set of their frontiers. The validity of a stencil tree is determined by means of a derivation but by a test. Hence, an ALD is not a generative grammar.

Definition 2.6. *(Tree language and string language of an ALD)*

The *(stencil) tree language* defined by an ALD A, denoted by T_L(A), is the set of all stencil trees valid for A. The *(string) language* defined by an ALD A, denoted by L(A), is the set $\{x \in \Sigma^* \mid x = \tau(T)$ for some tree T$\in$ T_L(A) }.

Example 2.1. Let L be the CF language $\{a^n b^n \mid n>0\}^+$, which is generated for example by the CF grammar G with axiom S, nonterminals X and S, and productions S$\rightarrow$XS | aXb, X$\rightarrow$aXb | ε. L is also defined by the ALD {($\perp$, aΔbΛ, $\perp$), (b, aΔbΛ, $\perp$), (a, aΔb, b), (a, ε, b)}. For instance, the string a^2b^2ab of L is the frontier of the valid tree shown in Fig. 1, where the constituent K_1 is matched by the rule ($\perp$, aΔbΔ, $\perp$), the constituent K_2 is matched by the rule (a, aΔb, b), the constituent K_3 is matched by the rule (b, aΔb, $\perp$) and both constituents labeled K_4 are matched by the rule (a, ε, b). Notice that this tree is different from the derivation tree in G of the string a^2b^2ab.

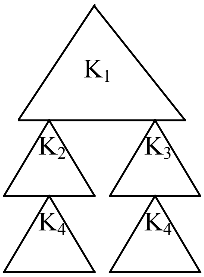


Fig. 1. A valid tree for Example 2.1

Definition 2.7. *(degree, width of an ALD)*

For every ALD A and every rule (x, z, y) in A the *degree* of the rule (x, z, y) is max (|x|, |y|), the maximum length of the permissible left/right contexts; the *width* of the rule (x, z, y) is |z|, the length of the pattern. For an ALD A the *degree* is the maximum degree and the *width* is the maximum width of its rules.

Definition 2.8. *(Left and Right Contexts)*

Let A be an ALD of degree k. LC_k *(Left Contexts)* is the set: $\Sigma^k \cup \bigcup_{0\leq j\leq k-1} \perp \Sigma^j$, RC_k *(Right Contexts)* is the set: $\Sigma^k \cup \bigcup_{0\leq j\leq k-1} \Sigma^j \perp$.

Definition 2.9. *(homogenous and reduced ALD)*

An ALD A of degree k is:

homogeneous iff $A \subseteq LC_k \times (\Sigma \cup \{\Delta\})^* \times RC_k$;
reduced iff each rule matches some constituent, in some valid tree.

Definition 2.10. *(equivalent, structurally equivalent ALD's)*

Two ALD's A_1, A_2 are called *equivalent (*resp. *structurally equivalent)* iff $L(A_1) = L(A_2)$ (resp. $T_L(A_1) = T_L(A_2)$).

The assumption that an ALD is homogeneous and reduced does not violate generality, as shown by the following proposition.

Lemma 2.11 [6] For every ALD there exists a structurally equivalent, homogeneous and reduced ALD.

2.1 Examples of ALD Languages

To better assess the expressive power of the ALD family, it is useful to look at various examples of ALD and non-ALD languages.

While it is unknown, at the present, whether the ALD family includes the regular languages, there are various examples of regular languages that are in ALD. A non-trivial example is the following regular language. Let L_1 and L_2 be two languages: the *shuffle* $L_1 \parallel L_2$ is defined as the language: $\{x_1\, y_1\, x_2\, y_2 \ldots x_n\, y_n \mid n \geq 1, x_j \in \Sigma^*, y_j \in \Sigma^*, x_1\, x_2 \ldots x_n \in L_1, y_1\, y_2 \ldots y_n \in L_2\}$. The regular language $L = a^* \parallel b^* c^* d^*$ is described by the ALD rules:

$(\perp, \Lambda\, c\, \Lambda\, d\, \Lambda, \perp)$, $(\perp, \Lambda\, c\, \Lambda, \perp)$, $(\perp, \Lambda\, d\, \Lambda, \perp)$, $(\perp,\varepsilon,\perp)$, $(\varepsilon, \Lambda\, a, \varepsilon)$, $(\perp, \Lambda\, b, \varepsilon)$, $(c, \Lambda\, c, \varepsilon)$, $(d, \Lambda\, d, \varepsilon)$

Many CF languages are ALD, such as the cited Pascal and HTML. We also provide here a few examples of CF languages not in the ALD family:

$\{a^n\, b^n \mid n \geq 1\} \cup \{a^n\, b^{2n} \mid n \geq 1\}$,
$\{a^n\, b^n\, a^m\, b^{2m} \mid n, m \geq 1\}$,
$\{a^n\, c\, b^n\, a^m\, d\, b^m \mid n, m \geq 0\}$.

The proof that these languages are not ALD is omitted, since it is a simple variation of similar proofs in [5]. Notice that simple changes to the alphabet of a language may make it ALD: for instance, $\{a^n b^n \mid n \geq 1\} \cup \{a^n c^{2n} \mid n \geq 1\}$ is ALD. This fact will be explained in Theorem 3.9 below.

3 A Hierarchy of ALD Languages

In [5] it has been proved that the degree classifies the ALD family in an infinite, strict hierarchy. A similar, but weaker, result holds also with respect to the width. Let $ALD_{W=k}$, $k > 0$, be the subfamily of ALD having width k.

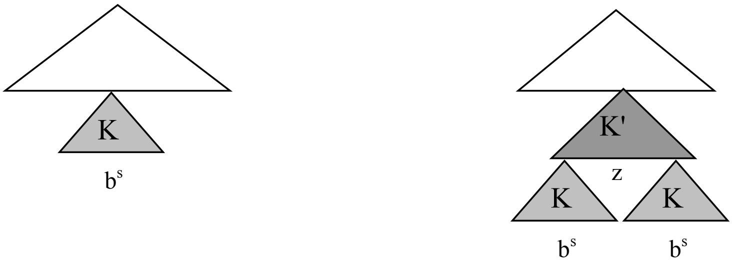
Proposition 3.1. For all $k > 0$, there exist an ALD language L such that L is not $ALD_{W=k}$ but is in $ALD_{W=k^2}$.

Proof. For every $i > 0$, let $L_i = \{b^{in} \mid n \geq 1\}$, where each L_i is defined by the ALD rule $(\perp, \Lambda b^i, \varepsilon)$. Assume by contradiction that there is $k > 0$ such that every language in the ALD family is also $ALD_{W=k}$. Let $j = k^2$: $L_j \in ALD_{W=k}$. Let A be an ALD of width k defining L_j. Without loss of generality, we can assume that A is homogeneous and reduced of degree r for some $r \geq 0$. Hence, every rule with contexts not including the endmarker must be of the form (b^r, z, b^r) with $z \in \{b, \Delta\}^*$, $0 \leq |z| \leq k$.

Claim 1: We can assume that for each rule of the form (b^r, z, b^r), with $z \in \{b, \Delta\}^*$, the rule is in A only if $z \in b^*$.

To prove the claim, first we notice that if $(b^r, z, b^r) \in A$, with $z \in \{b, \Delta\}^* \Delta \{b, \Delta\}^*$, then the rule must occur in a valid tree, since A is reduced. But then also a rule of the form (b^r, b^s, b^r) for some s, $0 \leq s \leq k$, must occur in the same valid tree (since a node labeled Δ cannot be on the frontier of a valid tree). We now claim that in this case $s = 0$ and z does not contain any occurrence of b. As a consequence, if there is a constituent K matched by the rule (b^r, z, b^r) then the maximal subtree with the same root of K has an empty frontier. In fact, take a valid tree T where the rule (b^r, z, b^r) matches some constituent (Fig.2, case (a)), i.e., a valid tree having a constituent K whose frontier is b^s and whose right and left contexts are both b^r, i.e., it is matched by

(b^r, b^s, b^r). The frontier $\tau(T)$ of T is b^{jh} for some $h > 0$. Let T' be a new tree obtained from T by replacing the constituent K with a constituent K' matched by the rule (b^r, z, b^r), and by appending the constituent K to each Δ occurring on the frontier of K'. T' is a valid tree because the new constituents K' and K are matched respectively by the rules (b^r, z, b^r) and (b^r, b^s, b^r). The frontier of the maximal subtree with root K' is $b^{|z|-t+ts}$ where $t\leq |z| \leq k$ is the number of Δ occurring in z (see Fig. 2, (b)). Hence, τ (T') = $b^{jh-s+|z|-t+ts}$, where $0 \leq |z| - t + (t-1)s < |z| + (t-1)s \leq k + (k-1)k = k^2$. Since $j = k^2$, then $\tau (T') \in L_j$ necessarily implies $|z| - t + (t-1) s = 0$: since $t\geq 1$ and $|z|\geq t$, it must be $|z| = t$ and $s = 0$. This means that if there is a rule of the form (b^r, z, b^r), with $z \in \{b\}^*\Delta\{b, \Delta\}^*$, then in A there cannot be any rule of the form (b^r, b^s, b^r) with $s > 0$, and z must be of the form Δ^t for some t, $0\leq t\leq k$. Since there is no way to add b's by using rules of the form (b^r, Δ^t, b^r), for some $t\leq k$, we can safely remove all those rules from A, obtaining an equivalent ALD that is homogeneous, reduced, of width k, of degree r and where the rules of the form (b^r, z, b^r) have $z \in b^*$. Hence, we can assume that A is already in this form.



(a) The tree T (b) The tree T'

Fig. 2. T' is obtained from T by replacing K, in the left and right contexts b^r, with a constituent K' matched by (b^r, z, b^r), $z \in \{b\}^*\Delta\{b,\Delta\}^*$. In the picture, it is shown the case where z includes two placeholders

Claim 2. For all t, $0\leq t\leq k$, and for all s, $1\leq s\leq k$, there is no rule in A of the form ($\perp b^t$, Δb^s, b^r) or of the form (b^r, $b^s\Delta$, $b^t\perp$). Suppose by contradiction that in A there is at least a rule of the former form, the latter case being analogous. Then, since the ALD is reduced there is a valid tree T with a constituent K matched by the rule. Let τ (T) = b^{jn} for some $n > 0$. Let T_1 be the maximal subtree of T with K at its root. Denote with T_2 a tree with the constituent K at its root and with the tree T_1 appended at the node labeled Δ (hence, there are two constituents of T_2 matched by the rule ($\perp b^t$, Δb^s, b^r)). Let T' be the tree obtained by replacing in T the subtree T_1 with the tree T_2. T' is a valid tree because the contexts of K and T_1 in T' are equal to the contexts of T_1 in T, but T' is longer than T of exactly s characters: τ (T') = b^{jn+s} with $0 < s\leq k < j$. Then $\tau(T') \notin L(A)$.

Claim 3. There is a constant $p > 0$ such that every valid tree T of A, whose frontier is larger than p, has at least a constituent K matched by one rule of A either of the form ($\perp b^t$, Δv, b^r) or of the form (b^r, vΔ, $b^t\perp$), for some $t < r$ and $v\in \{b, \Delta\}^*$.

The constituent K of part (a) is at the root of a maximal subtree whose frontier is larger than b^r.

In A there is at least a rule (b^r, b^s, b^r) with $0 \leq s \leq k$.

If the constituent K of part (a) is matched by a rule such that $v \in \Delta^*$ then in A there is at least a rule of the form (b^r, b^s, b^r) with $0 < s \leq k$.

Part (a). By Claim 1, in the ALD A there is at least a rule ($\bot b^t$, z, b^r) with $z \in \{b\}^* \Delta \{b, \Delta\}^*$, otherwise it would be impossible to get valid trees whose frontier is b^{jn}, for n > k. Moreover, if the pattern of each rule ($\bot b^t$, z, b^r) (resp. (b^r, z, $b^t \bot$)) had prefix (resp. suffix) b then the rule could match a constituent at most once in a valid tree. Hence, if part (a) of Claim 3 were false, the height of the valid tree T would be less or equal to the number |A | of rules in A: since the width is limited, the maximum length of the frontier of a tree without such constituent is k |A |. Select p = max (r, |τ (T) |) > max (r, k |A |) to have a contradiction. Hence, in every valid tree of frontier larger than p there must be an occurrence of either a constituent K matched by ($\bot b^t$, Δv, b^r) or by (b^r, $v\Delta$, $b^t \bot$), for some t < r and $v \in \{b, \Delta\}^*$.

Part (b). If p is large enough, the occurrence of the constituent K can be chosen high enough in T to be the root of maximal subtree T' whose frontier is larger than p_1 = max (r, k |A |) (by applying the same line of reasoning).

Part (c). By combining Claim 3, part (a), and Claim 2, the constituent K has $v \in \{b, \Delta\}^* \Delta \{b, \Delta\}^*$ and in A there is no rule of the same form with $v \in b^+$. Then each occurrence of a placeholder in v must be in the left context b^r and in the right context b^r. Hence, to be able to "close" those rules, it is necessary that in A there are rules with the same contexts. But by Claim 1, only rules of the form (b^r, b^s, b^r) with $0 \leq s \leq k$ are possible with left and right contexts b^r.

Part (d). If $v \in \Delta^*$ and all productions of the form (b^r, b^s, b^r) have s = 0, then all the placeholders at the right of the leftmost one in the constituent, with both left and right contexts b^r, would be roots of trees whose frontier is the empty string. Since no other rule can match a constituent at the root of a tree with more than p b's, the frontier of the maximal subtree T' would be shorter than p_1.

Now we prove the statement of the proposition. Let T be a valid tree whose frontier is b^{jn} with n larger than the constant p of Claim 3. By Claim 3, T has a maximal subtree T_1 such that the left and right contexts of T_1 are, respectively, either $\bot b^t$ and b^r, or b^r and $b^t \bot$, for some t, $0 \leq t < r$, with |τ (T1) | > r. Assume that the contexts of T_1 are $\bot b^t$ and b^r, respectively. The constituent K of T at the root of T_1 is matched by a rule ($\bot b^t$, Δv, b^r), for some v. By claim 3, either there is s > 0 such that (b^r, b^s, b^r) is in A or in v there is at least one occurrence of the letter b. As shown in Fig. 3, let T_2 be the tree that has the constituent K at its root, where, at the node labeled by the leftmost occurrence of Δ, the tree T_1 is appended, and where at each remaining occurrence of Δ a constituent K1, matched by (b^r, b^s, b^r) of A, with s > 0, is appended. Since the constituent K has at most k-1 placeholders, and $0 < s \leq k$, then $\tau(T_2) = b^{i+q}$, where $q \leq k(k-1) < j$. By Claim 3, part (d), q > 0. Replace T_1 in T with the tree T_2, obtaining a tree T'. T' is a valid tree because the contexts of K and T_1 in T' are equal to the contexts of T_1 in T, and both of the contexts of K1 in T' are b^r. Then the frontier of T' is $\tau(T') = b^{jn+q}$ with $0 < q < k^2 = j$, which is not a word in L_j: a contradiction. ■

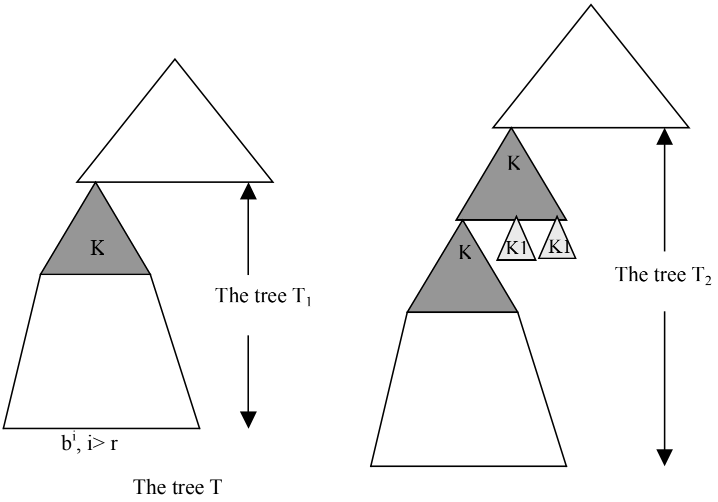


Fig. 3. The trees T and T' of the proof of Theorem 3.9

Remark. The above proof shows that for every k>1 there is a regular language $L_j = \{b^{jn} \mid n > 0\}$, with $j = k^2$, that is in $\mathrm{ALD}_{W = j}$ but not in $\mathrm{ALD}_{W = k}$. Hence, the theorem cannot state that there is a proper hierarchy for the width. i.e., that each level is properly contained in the following level, but only that the hierarchy is infinite. This derives from the usage of L_j as typical language of $\mathrm{ALD}_{W = j}$. A strict hierarchy theorem could be proved only if L_j is in $\mathrm{ALD}_{W = k+1}$. Actually, we can prove that $L_j \in \mathrm{ALD}_{W = h}$ with h < j. Namely, L_j is defined for j even by the ALD A: $(\perp, \Lambda b^{j/2}\Delta, \varepsilon)$, $(b, b^{j/2}, \varepsilon)$ and for j odd by the ALD A': $(\perp, \Lambda b^{\lfloor j/2 \rfloor}\Delta, \varepsilon)$, $(b, b^{\lceil j/2 \rceil}, \varepsilon)$, both having width $\lfloor j/2 \rfloor + 2$. In most cases, it seems possible to improve further this bound: for instance, L_9 is defined by the following ALD of width 4: $(\perp, \Lambda b\Delta\Delta, \varepsilon)$, (b, b^4, ε). This means that the language L_9 is at the level 4 of the hierarchy, while, by Proposition 3.1, it is not at the level 3: level 3 is strictly included in level 4. Similarly, L_4 is not in $\mathrm{ALD}_{W = 2}$, though it is in $\mathrm{ALD}_{W = 3}$. It could then be possible to generalize this fact and prove that that for every k > 1 there exists a language that is in $\mathrm{ALD}_{W = k}$ but not in $\mathrm{ALD}_{W = k-1}$.

The previous examples show also a general fact about ALD's: to decrease the width it is often necessary to increase the maximum number of placeholders in the rules. So the following problem naturally arises: does the maximum number of placeholders in ALD rules classify ALD's in a hierarchy? or may each ALD language be defined by an ALD whose rules contain at most k placeholders? In the latter case, an analogy with the CF case suggests that a possible value for k is 2.

As pointed out in the introduction, it is yet unknown whether all regular languages are also ALD languages. However, all CF unary languages (hence, regular) can be described by ALD.

Proposition 3.2 Every context-free language on a one-letter alphabet is an ALD language.

Proof: Each CF unary language is semilinear: hence, it is the finite union of the languages of the form $L_j = \{b^{jn} \mid n > 0\}$, for every j (and possibly also with the empty string). Let L be a CF language on a one-letter alphabet. Hence, there exists n such that L = $L_{k_1} \cup L_{k_2} \cup \ldots L_{k_n}$. L is defined by the following ALD: { $(\perp, a^{k_i}\,\Lambda, \perp)$ | $1 \le i \le n$} ∪ { $(\perp a^{k_i}, \Lambda a^{k_i}, \perp)$ | $1 \le i \le n$}.■

In [5], it has been proved that the ALD family is strictly included in the CF one and that it is not closed under homomorphism. Here we prove that the closure of ALD under homomorphism gives exactly the family of CF languages.

We recall the following

Definition 3.3. *[7] (Operator form of CF grammars)*

A context free grammar $G = (V_N, \Sigma, P, S)$ is said in *operator form* if $P \subseteq V_N \times (V_N \cup \Sigma)^* \setminus (V_N \cup \Sigma)^* V_N^2 (V_N \cup \Sigma)^*$.

This means that each rule of P is either in the form:

$A \to B$ for $A \in V_N$, $B \in V_N \cup \Sigma \cup \{\varepsilon\}$,

or in the form $A \to \alpha BC\beta$, where for all $\alpha, \beta \in (V_N \cup \Sigma)^*$, $A, B \in (V_N \cup \Sigma)$ either $B \in \Sigma$ or $C \in \Sigma$.

Roughly speaking, in the right-hand side of a rule two nonterminals are never adjacent.

The following theorem is well-known:

Theorem 3.4 ([7], th.4.8.2). For any context-free grammar G there is an equivalent grammar G' in operator form.

Looking at the proof of Theorem 3.4 in [7], it is easy to check that, in the grammar G', the right-hand side of a production contains at most two nonterminals; moreover, the productions of the form $A \to B$ for $A \in V_N$, $B \in V_N \cup \{\varepsilon\}$ can be eliminated: add the productions obtained by replacing with B all occurrences of A in the right-hand side of a rule to the productions of G'. So we can state the following

Corollary 3.5. For every context-free grammar G there is an equivalent grammar G' in operator form whose productions are in one of the following forms:

$A \to a$ for $A \in V_N$, $a \in \Sigma$, or
$A \to \alpha B\beta$ with $\alpha\beta \in \Sigma^+$, $A, B \in V_N$.
$A \to \alpha B\gamma C\beta$ with $\alpha, \beta \in \Sigma^*$, $\gamma \in \Sigma^+$, $A, B \in V_N$.

We recall two definitions and a lemma from [5], stated here for the case of degree k =1.

Definition 3.6 *(Contexts of the nonterminals of a CF grammar)*

Let $G = (V_N, \Sigma, P, S)$ be a CF grammar. For every $X \in V_N$, *Con(X)* is the set:
$\{(x, y) \mid (x, y) \in LC_1 \times RC_1 \wedge \exists u \in \perp\Sigma^* \wedge \exists v \in \Sigma^*\perp \wedge \exists\, z \in \Sigma^*: \perp S \perp \Rightarrow^*_G uxXyv \Rightarrow^*_G uxzyv\}$

Definition 3.7 *(Disjoint operator form of CF grammars)*

Let $G = (V_N, \Sigma, P, S)$ be a CF grammar. G is said to be in *disjoint operator form* if, and only if:

G is in operator form;
Con(X) $\neq\emptyset$ for all $X\in V_N$;
For all X, $Y\in V_N$, with $X\neq Y$, Con(X) $\cap$Con(Y) = $\emptyset$.

A disjoint operator form grammar can always be transformed into an equivalent ALD of degree 1, as stated in the next lemma.

Lemma 3.8 ([5]) Let G = (V_N, Σ, P, S) be a CF grammar in disjoint operator form. Let h: $V_N \cup \Sigma \rightarrow \Delta\cup\Sigma$ be the homomorphism defined by h(a) = a for $a \in \Sigma$, h(X) = Δ for $X \in V_N$.
Let A be the following homogeneous ALD of degree 1:

$$\{(x, w, y)\ \mid \exists\, X \in V_N, z \in (V_N\cup\Sigma)^*: X \rightarrow z \in P, w = h(z), (x, y) \in \text{Con}(X)\ \}.$$

A is structurally equivalent to G. Then we can prove the following:

Theorem 3.9. A language L is context-free if and only if it is a (non erasing) homomorphic image of an ALD of degree 1.

Proof. Obviously, the homomorphic image of an ALD is a CF language because every ALD language is CF and the CF family is closed under homomorphism.

Conversely, let L be a CF language. Without loss of generality we can assume that it is generated by a grammar G = (V_N, Σ, P, S) in operator form with productions of types described in Corollary 3.5. We construct from G a new grammar G' by renaming the terminal symbols which precede or follow a nonterminal, according to following rules:

$P' = \ \{A\rightarrow a\ \mid A\rightarrow a \in P\} \cup$
$\{A\rightarrow\alpha\, (a, B)\, B\, (b, B)\, \beta \mid A\rightarrow\alpha aBb\beta \in P\} \cup$
$\{A\rightarrow B\, (b, B)\, \beta\ \mid A\rightarrow Bb\beta \in P\} \cup$
$\{A\rightarrow\alpha\, (a, B)\, B\ \mid A\rightarrow\alpha aB \in P\} \cup$
$\{A\rightarrow\alpha\, (a, B)\, B\, (c, B)\, \gamma\, (d, C)\, C(b, C)\, \beta\ \mid A\rightarrow\alpha aBc\gamma dC\, b\beta \in P\} \cup$
$\{A\rightarrow B\, (c, B)\, \gamma\, (d, C)\, C(b, C)\, \beta\ \mid A\rightarrow Bc\gamma dC\, b\beta \in P\} \cup$
$\{A\rightarrow\alpha\, (a, B)\, B\, (c, B)\, \gamma\, (d, C)\, C\ \mid A\rightarrow\alpha aBc\gamma dC \in P\} \cup$
$\{A\rightarrow B\, (c, B)\, \gamma\, (d, C)\, C \mid A\rightarrow Bc\gamma dC \in P\} \cup$
$\{A\rightarrow\alpha\, (a, B)\, B\, (c, BC)\, C(b, C)\, \beta\ \mid A\rightarrow\alpha aBcCb\beta \in P\}\} \cup$
$\{A\rightarrow B\, (c, BC)\, \gamma C\, (b, C)\, \beta\ \mid A\rightarrow BcCb\beta \in P\}\ \} \cup$
$\{A\rightarrow\alpha\, (a, B)\, B\, (c, BC)\, C \mid A\rightarrow\alpha aBcC \in P\}\} \cup$
$\{A\rightarrow B\, (c, BC)\, C\ \mid A\rightarrow BcC \in P\}$

where A, B, C are in V_N, a, b, c are in Σ and α, β, γ are in Σ^*.

Then G' = (V_N, $\Sigma \cup \{\Sigma \times (V_N \cup V_N^2)\ \}$, P', S) is a CF grammar in disjoint operator form. Then by Lemma 3.8 there is a homogeneous ALD A of degree 1 structurally equivalent to G'. Let L' = L(G') = L(A) and let h be the homomorphism from $\Sigma \cup \{\Sigma \times (V_N \cup V_N^2)\ \}$ to Σ that acts as identity on Σ and as natural projection on $\Sigma\times (V_N \cup V_N^2)$. It is obvious that h(L') = L. ■

Corollary 3.10. A language L defined by an ALD of degree k is a (non erasing) homomorphic image of an ALD of degree 1.

Theor. 3.9 shows that by a suitable change of alphabet any CF language can be turned into an ALD language of degree one, preserving its structure. As a practical case consider the language Pascal, which is an ALD language of degree 3 [6]. By a change of alphabet, the degree can be lowered to 1. In practice, one does not need to turn the grammar into operator form, nor to rename all terminals surrounding nonterminal symbols, as in the proof of Theorem 3.9; it suffices to rename the rare terminal occurrences where the contexts do not meet the disjointness hypothesis of Definition 3.7. Similar transformations of the surface representation of a language have been applied in the early days for obtaining grammars suitable for parsing using precedence algorithms. Another remark relates Theor. 3.9 to modern mark-up languages such as XML. The terminal symbols introduced in the proof can be viewed as "tags" that mark-up or delimit a piece of text.

4 Conclusions

In spite of the simplicity of the model, various theoretical questions on ALD are still open or under investigation, e.g. inclusion of the regular set, some decidability properties, minimization w.r.t. degree or width, hierarchy with respect to the number of placeholders in a pattern. Comparisons with related models, such as the contextual grammars of S. Marcus [8] and [9], are given in [5]. A similar, but more complex, model has been introduced in [10].

We hope that ALD could be a good model both as an explanation of fundamental syntactic phenomena and as a practical technique for language specification. To explore to what extent existing technical languages can be defined by ALD, in [6] the CF syntax of Pascal has been completely defined by ALD rules; it was checked that the main features of HTML can be described conveniently by ALD.

References

1. S. Crespi Reghizzi, G. Guida, D. Mandrioli, *Non-counting context-free languages*, Journ. ACM, 25 (1978), 4, 571-580.
2. W. Thomas, *On Noncounting Tree Languages*, Grundl. Theor. Informatik, Proc. First Int. Workshop, Paderborn 1982, 234-242.
3. A. Cherubini, S. Crespi, P. San Pietro, *Languages Based on Structural Local Testability*, in C.S. Calide, M.J.Dinneen, Proceedings of DMTCS'99, Auckland, New Zealand, 18-21, January 1999, Springer, Singapore.
4. V. Braitenberg and F. Pulvermüller, *Entwurf einer neurologischen Theorie der Sprache*, Naturwissenschaften 79 (1992), 103-117.
5. A. Cherubini, S. Crespi Reghizzi, P. San Pietro, *Associative Language Descriptions*, Theoretical Computer Science, 2001 (to appear)
6. S. Crespi Reghizzi, M. Pradella, P. San Pietro, *Associative definitions of programming languages*, Computer Languages, 26 (2:4), 2001.
7. M. Harrison, *Introduction to Formal Language Theory*, Addison Wesley (1978).

8. S. Marcus, *Contextual Grammars and Natural Languages*, Handbook of formal languages (Eds. G. Rozenberg, A. Saloma), Vol.II, Ch.6, Springer (1997), 215-132.
9. A. Ehrenfeucht, G. Päun and G. Rozenberg, *Contextual Grammars and Formal Languages*, Handbook of formal languages (Eds. G. Rozenberg, A. Saloma), Vol.II, Ch.6, Springer (1997), 237-290.
10. S. Crespi-Reghizzi and V. Braitenberg, *Towards a brain compatible theory of syntax based on local testability,* in C. Martin-Vide and V. Mitrana (eds) Grammars and Automata for String Processing: from Mathematics and Computer Science to Biology, and Back. Gordon and Breach, London, 2001.

Block-Deterministic Regular Languages[⋆]

Dora Giammarresi[1], Rosa Montalbano[2], and Derick Wood[3]

[1] Dipartimento di Matematica, Università di Roma "Tor Vergata"
via della Ricerca Scientifica, 00133 Roma, Italy
giammarr@mat.uniroma2.it

[2] Dipartimento di Matematica e Applicazioni, Università di Palermo
via Archirafi 34, 90123 Palermo, Italy
rosalba@altair.math.unipa.it

[3] Department of Computer Science, Hong Kong University of Science and Technology
Clear Water Bay, Kowloon, Hong Kong SAR
dwood@cs.ust.hk

Abstract. We introduce the notions of blocked, block-marked and block-deterministic regular expressions. We characterize block-deterministic regular expressions with deterministic Glushkov block automata. The results can be viewed as a generalization of the characterization of one-unambiguous regular expressions with deterministic Glushkov automata. In addition, when a language L has a block-deterministic expression E, we can construct a deterministic finite-state automaton for L that has size linear in the size of E.

1 Introduction

A regular language is one-unambiguous, according to Brüggemann-Klein and Wood [4], if there is a deterministic Glushkov automaton for the language. An alternative definition of one-unambiguity based on regular expressions is that each position in a regular expression has at most one following position for each symbol in the expression's alphabet. The latter definition is used to define unambiguous content model groups in the Standard Generalized Markup Language (SGML) [13], which are a variant of regular expressions. Indeed, it was the SGML standard that motivated Brüggemann-Klein and Wood's investigation of one-unambiguity. In contrast, to the results of Book and his coworkers [3] on ambiguity of regular expressions, there are regular languages that are not one-unambiguous [4]. It is clear, from the definition of one-unambiguity, that when a regular expression is one-unambiguous it is also unambiguous in the sense of Book and his colleagues. The difference is that one-unambiguity can also be viewed as one-determinism. A lookahead of one symbol when processing a string from left to right determines a unique next position in the given regular expression; they are, essentially, LL(1) regular expressions [1,4].

[⋆] The third author's research was supported under a grant from the Research Grants Council of Hong Kong SAR. It was partially carried out while the first author was visiting HKUST.

A. Restivo, S. Ronchi Della Rocca, L. Roversi (Eds.): ICTCS 2001, LNCS 2202, pp. 184–196, 2001.

These observations lead to two possible generalizations (at least) of one-unambiguous regular expressions. The first is based on a lookahead of at most $k \geq 1$ symbols to determine the next, at most one, matching position in a regular expression. The second is similar except that when we use a lookahead of l symbols we must match the next l positions uniquely. The first notion defines k-unambiguous expressions and the second defines k-block-deterministic expressions. We focus on k-block-deterministic expressions in this work.

Our results have an interesting implication about the regular languages that have block-deterministic expressions. When a language L has a k-block-deterministic expression E, we can construct a deterministic finite-state automaton for L that has size linear in the size of E. Are there other "natural" classes of regular expressions that have this property?

In Section 2, we review basic notation and terminology and, in Section 3, we introduce blocked expressions, block-marked expressions and block-deterministic expressions. In Section 4, we characterize block-deterministic languages in terms of block-deterministic automata.

2 Notation and Terminology

Let Σ be an alphabet of symbols. A **regular expression** over Σ is built from λ, $\emptyset$, and symbols in Σ using the binary operators $+$ and $\cdot$ and the unary operator *. The language specified by a regular expression E is denoted by $\mathrm{L}(E)$ and it is referred to as **regular language**.

To indicate different positions of the same symbol in a regular expression, we mark symbols with unique subscripts. For example, $(a_1 + b_1)^* a_2 (a_3 b_2)^*$ and $(a_4 + b_2)^* a_1 (a_5 b_1)^*$ are both **markings** of the regular expression $(a + b)^* a(ab)^*$. A marking of a regular expression E is denoted by E'. If H is a subexpression of E, we assume that markings H' and E' are chosen in such a way that H' is a subexpression of E'. A **marked regular expression** E' is a regular expression over Π, the alphabet of subscripted symbols, where each subscripted symbol occurs at most once in E'.

The reverse of marking is the dropping of subscripts, indicated by $^\natural$ and defined as follows: If E' is a regular expression over Π, then $(E')^\natural$ is the regular expression over Σ that is obtained by dropping all subscripts in E'. Thus, a marked regular expression H over Π is a **marking** of regular expression E if and only if $H^\natural = E$. Observe that for each regular expression E over Σ, up to an isomorphism on the set of subscripts, set Π is unique and so is the marking E'. Unmarking can also be extended to words and languages: For a word w over Π, let $w^\natural$ denote the word over Σ that is constructed from w by dropping all subscripts. For a language L over Π, let $L^\natural$ denote $\{w^\natural | w \in L\}$. Then, for each regular marked expression E' over Π, $\mathrm{L}((E')^\natural) = \mathrm{L}(E')^\natural$.

Book and his associates [3] and Eilenberg [8] define unambiguous regular expressions as follows. A regular expression E is **unambiguous,** if and only if for all words x and y over Π, the alphabet of subscripted symbols, condition

$x \neq y$ implies $x^\natural \neq y^\natural$. A regular language L is **unambiguous** if it is denoted by an unambiguous regular expression. It is know that all regular languages are unambiguous.

Brüggemann-Klein and Wood [4] defined a more restrictive version of unambiguity motivated by SGML content models [13]. A regular expression E is **one-unambiguous** if and only if, for all words u, v and w over Π and all symbols x and y in Π, the conditions $uxv, uyw \in L(E')$ and $x \neq y$ imply that $x^\natural \neq y^\natural$. A regular language is **one-unambiguous** if it is denoted by some one-unambiguous expression. Brüggemann-Klein and Wood proved that not all regular languages are one-unambiguous.

It is well known that regular languages are those recognized by finite-state automata. Given a regular expression E over an alphabet Σ, we can construct an automaton that recognizes $L(E)$ in many different ways. Many of these automata can be reduced to the **Glushkov automaton** [4,10]. Glushkov first suggested this construction in 1960 [11,12]; it was also suggested by McNaughton and Yamada [14] independently and at about the same time. The construction, given first by Book *et al.* [3], is based on the first, last and follow sets of positions in the marking E' of E. We define the three sets of positions as follows:

$\mathrm{first}(E')$ is the set of all positions that can begin a string in $L(E')$;
$\mathrm{last}(E')$ is the set of all positions that can end a string in $L(E')$;
$\mathrm{follow}(a, E')$ is the set of all positions in E' that can follow position a.

Once we have computed these sets, we can construct the Glushkov automaton G_E as follows: *The states of G_E are $\Pi \cup \{0\}$ where Π is the alphabet of subscripted symbols, $0 \notin \Pi$ is the start state,* $\mathrm{last}(E')$ *(or* $\mathrm{last}(E') \cup \{0\}$*, if the empty word is in the language) is the set of final states, and the transitions in $(\Pi \cup \{0\}) \times \Sigma \times \Pi$ are*

$$\{(x, a, a_j) : {a_j}^\natural = a,\ a_j \in \mathrm{follow}(x, E') \ \textit{or}\ x = 0 \ \textit{and}\ a_j \in \mathrm{first}(E')\}.$$

Caron and Ziadi [5] recently characterized Glushkov automata. Observe that, as consequence of Caron and Ziadi's result, given a finite-state automaton we can estabilish whether it is a Glushkov one or not, without any knowledge on E, its generating regular expression. Moreover, E can be computed from G_E.

Finite-state automata admit a generalization in terms of block automata, that also describe all and only regular languages. Block automata [1] were introduced by Eilenberg [8]. They allow the transition labels to be nonempty strings or **blocks** over the input alphabet rather than just symbols. Formally, a **block automaton** A is specified by a tuple $(Q, \Sigma, \Gamma, \delta, s, F)$, where Q is a finite set of states, Σ is an input alphabet, Γ is a finite subset of Σ^+ called the **block alphabet,** $\delta \subseteq Q \times \Gamma \times Q$ is a transition relation, $s \in Q$ is a start state and $F \subseteq Q$ is a set of final states. If the maximum block length in A is k, then we refer to A as a **k-block automaton.**

[1] Block automata are called generalized automata by Eilenberg [8].

From this viewpoint, a standard finite-state automaton (where all transitions are single-symbol labelled) is a one-block automaton. From now on, we will refer to standard finite-state automata as to one-block automata.

As with one-block automata, a string w has an **accepting computation** in a block automaton A if there is a path from the start state to some final state that spells the string w. The collection of all strings that have an accepting computation in a block automaton A is called the **language of A** and it is denoted by $L(A)$.

A block automaton is **nondeterministic**, and therefore ambiguous, if the same string has more than one accepting computation. As we known, in one-block automata this condition implies there is at least a state that has two outgoing transitions with the same label. For k-block automata the implication is generally weaker: Nondeterminism occurs in a k-block automaton when there is at least a state that has two outgoing transitions whose labels are one prefix of the other. As a consequence, the condition of determinism in a block automaton corresponds to the set of all labels in transitions from a given state being prefix free. Formally, let $A = (Q, \Sigma, \Gamma, E, s, F)$ be a block automaton and, for each $q \in Q$, let $\text{block}(q) \subseteq \Gamma$ be the set of labels in the transitions out of q. A is a **deterministic block automaton** if, for each $q \in Q$, $\text{block}(q)$ is prefix-free.

Deterministic block automata were introduced by Giammarresi and Montalbano [9] when they investigated the minimization of block automata. We will use deterministic block automata to define block-deterministic regular languages.

Observe that block automata can be regarded to as one-block automata when we treat the blocks in the transitions as single symbols—as we do whenever we refer to the elements of a block alphabet. With this assumption, we can apply the usual automata transformations, such as state minimization and determinization, to block automata. Given a block automaton A, we denote its deterministic and minimal deterministic automata by $\mathcal{D}(A)$ and $\mathcal{M}(A)$, respectively, when considering its blocks as single symbols.

We now describe two transformations that are essentially mutual inverses of each other: **state elimination** and **block elimination**. The first one eliminates states from a block automaton creating transitions with longer block labels than the original ones; the second transformation eliminates block-labelled transitions creating states whose transitions have single-symbol labels.

Let A be a block automaton and q be a state of A such that q is not the start state, it is not a final state and it has no self-loops. We define the **state elimination of q in A** as follows: We first remove state q and all transitions into and out of q from A. Second, for every pair (r, u, q) and (q, v, s) of transitions that were in A, we add a new transition (r, uv, s) to A. We denote the resulting automaton by $\mathcal{S}(A, q)$. It is easy to verify that $\mathcal{S}(A, q)$ is indeed a block automaton equivalent to A. We can also extend state elimination to a set $S \subset Q$ of states. Giammarresi and Montalbano [9] prove that if S does not contain the start state and any final state, and the subgraph induced by S is acyclic, then we can construct a unique block automaton $\mathcal{S}(A, S)$ by eliminating the states in S in any order. In this case we say that the set $S \subseteq Q$ of states satisfies the

state-elimination precondition for A. Notice that when S satisfies the above precondition and A is a k-block automaton, the length of blocks in $\mathcal{S}(A, S)$ can increase to at most $|S|k$. Finally, if we apply state elimination to any state of a deterministic finite-state automaton that satisfies the precondition, then we obtain a deterministic block automaton.

Let A be a block automaton and $e = (p, a_1a_2 \cdots a_k, q)$ be a transition in A, where $k \geq 2$. We define the **block elimination of e in A** as follows: We first remove the transition e from A. Second, we introduce new states $p_1, \ldots, p_{k-1}$ and new transitions $(p, a_1, p_1), (p_1, a_2, p_2), \ldots, (p_{k-1}, a_k, q)$. We denote the resulting block automaton by $\mathcal{B}(A, e)$. It is easy to verify that $\mathcal{B}(A, e)$ is indeed a block automaton equivalent to A. Clearly, given a block automaton A, it can be transformad into a one-block automaton by applying $\mathcal{B}(A, B)$, where B denotes the set of all block-labelled transitions in A. Observe that, when A is a deterministic block automaton, the resulting one-block automaton need not be deterministic.

3 Block-Deterministic Regular Expressions

We define block-marked regular expressions, and block-deterministic regular expressions and languages. Then, we characterize block-deterministic regular languages as those languages defined by deterministic block automata.

Let E be a regular expression over an alphabet Σ. We define a **block** of E to be a subexpression of E containing only concatenation operations. For example, given the expression

$$E = (a \cdot a)^* \cdot (a \cdot b \cdot b + b \cdot a) \cdot b^*,$$

then a, aa, ab, abb, b, ba and bb are all possible blocks in E, whereas aab and bbb are not blocks of E although they are factors of words in $L(E)$. We can **partition** the dotted subexpressions in a regular expression E into disjoint blocks. We can partition the running-example expression in more than two ways; for example, we obtain six blocks with the partition

$$([a][a])^*([ab][b] + [ba])([b])^*,$$

where we use square brackets [and] to enclose blocks. There is the minimum partition of a regular expression that treats each maximal dotted subexpression as a block; for example,

$$([aa])^*([abb] + [ba])([b])^*$$

has four blocks. There is also the maximum partition that treats each single symbol as a block; for example,

$$([a][a])^*([a][b][b] + [b][a])([b])^*,$$

has eight blocks. An expression that is partioned into blocks is called a **blocked expression.**

We define a **block marking** of an expression using a blocked version of the expression. A *block marking* of an expression E is obtained by partitioning E into blocks and uniquely marking each block with an integer subscript. When we wish to identify the maximum length, k, of the blocks in a block marking, we call it a **k-block marking.** We denote a block-marked version of an expression E by E'. We denote by block(E') the set of all marked blocks of E'. Thus, a block-marked regular expression E' is a marked regular expression over the alphabet $\Gamma = \text{block}(E')$. For example, one block marking for the running example E is

$$E' = ([a]_1[a]_2)^*([ab]_3[b]_4 + [ba]_5)([b]_6)^*,$$

in which case: $\text{block}(E') = \{[a]_1, [a]_2, [ab]_3, [b]_4, [ba]_5, [b]_6\}$.

The **block unmarking** of a block-marked expression removes all subscripts and the square brackets. If E' is a block-marked expression, then $(E')^\natural$ is the corresponding unmarked and unblocked expression.

Block marking and unmarking of regular expressions can be extended in an obvious way to block marking and unmarking of words and languages. Notice that block marking generalizes the notion of marked expressions [11,14,2,4] that corresponds to one-block marking.

Now, given a block-marked regular expression E', we can extend to E' the functions first, last and follow introduced by Glushkov, McNaughton and Yamada [11,14]. In this case, $\text{first}(E')$, $\text{last}(E')$ and $\text{follow}(x, E')$ are subsets of $\text{block}(E') = \Gamma$. Using these sets, we give a formal definition of block-deterministic regular expressions.

A block-marking E' of E is a **deterministic block-marking** if the following two conditions hold:

1. For all $x, y \in \text{first}(E')$, $x \neq y$ implies that $x^\natural$ and $y^\natural$ are not one prefix of the other.
2. For all $z \in \text{block}(E')$ and for all $x, y \in \text{follow}(z, E')$, $x \neq y$ implies that $x^\natural$ is not a prefix of $y^\natural$.

A regular expression E is **block-deterministic** if there exists a deterministic block-marking E' of E.

If we restrict the block length to one, then one-block-deterministic expressions coincide with one-unambiguous expressions as defined by Brüggemann-Klein and Wood [4]. In general, a deterministic block marking for a given block-deterministic regular expression E is not unique. This observation holds even when the maximal length k of the blocks is specified. As an example, consider the running example expression $E = (aa)^*(abb + ba)(b)^*$. There are two different deterministic two-block markings for E:

$$E_1' = ([aa]_1)^*([ab]_2[b]_3 + [b]_4[a]_5)([b]_6)^*$$

$$E_2' = ([aa]_1)^*([ab]_2[b]_3 + [ba]_4)([b]_5)^*.$$

From now on, we will refer to Γ when the set of blocks will be treated as atomic symbols and we will refer to $\text{block}(E')$, when the set of blocks will be treated as strings.

Given a block-marked expression E' of E, the Glushkov automaton for E', denoted by $G^k(E')$, is defined in the classical way considering as alphabet the block set block(E') and it is called **Glushkov block automaton**. Observe that the Glushkov block automaton for a given regular expression E depends on the block marking of E. We use Glushkov block automata to characterize block-deterministic regular expressions. This characterization generalizes the one of Brüggemann-Klein and Wood [4] to the case of k-block markings for $k > 1$.

Lemma 1. *A k-block marking E' is deterministic if and only if the corresponding Glushkov block automaton $G^k(E')$ is deterministic.*

Proof. From definition of deterministic block marking, and from construction of Glushkov automaton, it follows that for each state q in $G^k(E')$, $block(q)$ is a prefix-free set, that is $G^k(E')$ is a deterministic block automaton.

Since by definition E is block deterministic iff there exists a deterministic block marked expression E' such that $(E')^\natural = E$, then we get the following

Corollary 1. *A regular expression E is block deterministic if and only if it admits a deterministic Glushkov block automaton.*

If we want to emphasize the maximal length k of the blocks, we write **k-block deterministic.**

We now consider the problem of deciding whether a given regular expression E is block-deterministic. Corollary 1 suggest one simple method: To guess a $k \geq 1$ and a k-block marking E' and then construct the corresponding Glushkov k-block automaton $G^k(E')$. If $G^k(E')$ is deterministic, then E is k-block-deterministic.

Note that, for the case $k = 1$, the problem is easy to solve since there is a unique one-block marking and the corresponding Glushkov block automaton is the Glushkov one-block automaton G_E. In this case, if G_E is deterministic, then E is one-block deterministic. We now consider the case when G_E is not deterministic and describe a procedure to determine whether there is a deterministic k-block marking for E, for some $k \geq 2$. More precisely, such a k-block marking will be one with the minimum k.

Lemma 2. *If a regular expression E is k-block-deterministic, then its corresponding Glushkov one-block automaton G_E can be transformed into a deterministic k-block automaton by a sequence of state eliminations.*

Proof. Let E' be a deterministic k-block marking for E and let $G^k(E')$ be the corresponding Glushkov deterministic block automaton. We will prove that $G^k(E')$ is the requested automaton of the statement.

By applying a sequence of block eliminations to all appropriate transitions in $G^k(E')$, we transform $G^k(E')$ into a one-block automaton $\mathcal{B}(A, B) = G^\natural$ (see Section 2 for the definition). Observe that any breaking of a block-labelled transition into a sequence of symbol-labelled transitions corresponds to the block unmarking of one block of E', and to a new subscription of each of its symbols, since all the states introduced by block eliminations correspond to the positions of the symbols in the blocks of E' that are not the last symbols of the blocks. Then, it follows that $G^{\natural} = G_E$. By applying state eliminations of all the new states of $G^{\natural} = G_E$ we obtain $G^k(E')$.

As a consequence of Lemma 2, the Glushkov automaton G_E of a k-block-deterministic expression E can be determinized (as block automaton) without using subset construction: All the states that are responsible for its nondeterminism can be eliminated by a finite sequence of state eliminations to give a deterministic k-block automaton. Moreover we obtain a Glushkov deterministic k-block automaton.

We now show how to get a Glushkov deterministic k-block automaton equivalent to G_E, if it exists, starting by G_E itself. First, we identify the set of states of G_E to be eliminated.

Let A be a (block) automaton and let q_1 and q_2 be two different states of A. Then, q_1 and q_2 are **duplicates** if the following condition holds:

$$\exists p \in Q \text{ and } x \in \Sigma^*\text{: } (p, x, q_1) \text{ and } (p, x, q_2) \text{ are paths in } A.$$

Recall that, given an automaton A, if we apply the subset construction to A we get a deterministic automaton $\mathcal{D}(A)$ whose states are subsets of the original set of states of A. We refer to a state of $\mathcal{D}(A)$ as either a **multiple state** or as a **single state** according to the cardinality of such sets. A state q of A is possibly included in several states, single and multiple, of $\mathcal{D}(A)$.

Observe that the duplicate states of a given automaton A are those that are in multiple states in $\mathcal{D}(A)$. Therefore, an automaton is deterministic if and only if it does not have any duplicate states.

Lemma 3. *Let E be a regular expression and let G_E be a corresponding Glushkov automaton. Let Q_{dup} be the set of all duplicate states of G_E. If Q_{dup} satisfies the state-elimination precondition and $\mathcal{S}(G_E, Q_{dup})$ is a Glushkov block automaton then E is block deterministic.*

Proof. By hypothesis Q_{dup} satisfies the state-elimination precondition (see Section 2) and $G'_E = \mathcal{S}(G_E, Q_{dup})$ is a Glushkov block automaton. Let E' be the block-marked regular expression obtained from G'_E (characterization of Caron and Ziadi show how to obtain it). Note that E' is a block marking of E (in fact, the state eliminations in G_E correspond to combining some one blocks in the standard marking of E), so that to prove block-determinism of E can be reduced to prove that G'_E is deterministic.

Let $\mathcal{D}(G_E)$ be the deterministic automaton obtained by applying the subset construction to G_E, and let Q' denote the set of states in $\mathcal{D}(G_E)$ that are images (under determinization) of all states in Q_{dup}. We claim that Q' satisfies the state-elimination precondition for $\mathcal{D}(G_E)$ and that $\mathcal{S}(\mathcal{D}(G_E), Q') = G'_E$. As a consequence, since transformation $\mathcal{S}$ preserves determinism, G'_E is a deterministic block automaton.

Lemmas 2 and 3 suggest the following theorem.

Theorem 1. *Let E be a regular expression and let G_E be a corresponding Glushkov automaton. Then, E is k-block deterministic, for some $k \geq 2$, if and only if G_E can be transformed into a k-deterministic Glushkov automaton by eliminating all of its duplicate states. Moreover, this Glushkov automaton defines a deterministic k-block marking of E.*

In the sequel, if E is a k-block-deterministic regular expression, we denote by G_E^k the deterministic Glushkov k-block automaton obtained by applying state elimination to all duplicate states in G_E. Moreover, we will refer to the block marking induced by G_E^k as the **standard block marking** of E.

From Lemma 3, we obtain the following algorithm to determine whether a given regular expression E is block deterministic. First compute the Glushkov automaton G_E and identify the set Q_{dup} of its duplicate states. If Q_{dup} satisfies the state-elimination precondition (it does not contain the start state or a final state and it induces an acyclic subgraph), then compute $G'_E = \mathcal{S}(G_E, Q_{dup})$. Second, determine whether G'_E is a Glushkov automaton for the block alphabet using, for example, the characterization of Caron and Ziadi [5]. If it is, then $G'_E = G_E^k$ defines a deterministic block marking of E.

Consider the running example expression E and its Glushkov automaton in Fig. 1(a). It contains two duplicate states; that is, the states in $Q_{dup} = \{1, 3\}$ satisfy the state-elimination precondition. By eliminating these states we obtain $G^k(E')$ that is a deterministic Glushkov automaton for the alphabet $\{a, aa, ab, b\}$.

We conclude this section by mentioning that the application of subset construction to the Glushkov automaton G_E of a block-deterministic regular expression E does not increase the size of the automaton ([6,7]) whereas, in the worst case, subset construction produces exponential blow-up. Indeed, from the proof of Lemma 3 we infer that the number of states of $\mathcal{D}(G_E)$ is at most the number of states of G_E since the set of duplicate states does not induce cycles.

4 Block-Deterministic Regular Languages

A regular language L is **block deterministic** if there is a block-deterministic regular expression E such that $L = L(E)$. We now demonstrate that there are regular languages that are not block deterministic.

We first consider the problem of deciding whether a given regular language is block deterministic. The basic idea is to use the characterization established by Brüggemann-Klein and Wood [4] for unambiguous regular languages (one-block-deterministic regular languages in our terminology). Now, a regular expression is one-unambiguous if and only if its Glushkov automaton is deterministic. Brüggemann-Klein and Wood show that if a Glushkov automaton is deterministic, then it has some properties that are preserved under minimization. Therefore, such properties can be checked on the minimal finite-state automaton M for the given language. Moreover, if these properties hold for some minimal automaton,

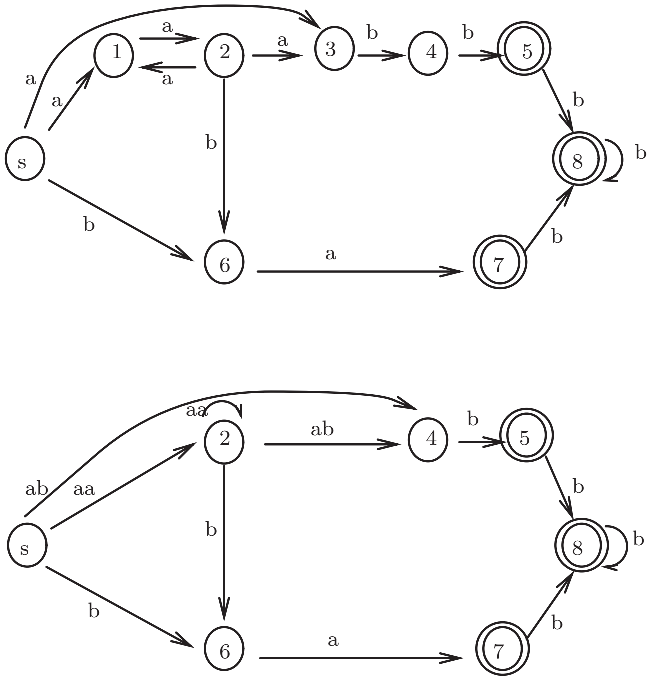


Fig. 1. Two Glushkov automata for the running example expression E. a. The Glushkov automaton G_E for E. b. The deterministic Glushkov block automaton $G^2(E')$ for a two-block marking of E obtained by state elimination in G_E

they prove that the corresponding regular language is one-unambiguous. Thus, they are able to give an algorithm that determines whether a given language is one-block deterministic and, if it is, they are able to construct a one-block-deterministic expression for it. We refer to this characterization as the **BW test for one-block-deterministic languages.** Suppose we want to test whether a given language $L \subset \Sigma^*$ is k-block deterministic for some fixed k. Let M be the minimal automaton for L. We apply state elimination to M to get a k-block automaton N^k. Let N be the same automaton as N^k considered as a minimal automaton on its block alphabet Γ. We can then apply the BW test to N. If L, considered to be over Γ, is one-block-deterministic, then there is a deterministic Glushkov automaton on Γ that reduces to N under minimization. Such a Glushkov automaton gives a k-block-deterministic regular expression together with a deterministic k-block marking for the original L ($L \subseteq \Sigma^*$).

On the other hand, if we consider all possible k-block automata that we can get from M by state elimination and none of them pass the BW test (when considered on the corresponding block alphabet), then we can conclude that L

is not k-block deterministic for any k. This procedure always terminates. Given an automaton A, the number of all possible block automata obtained from A by state elimination is finite.

Notice that the preceding algorithm works only under the assumption that, given a block alphabet Γ, the minimal automaton N for L, when considered to be over Γ, can be obtained by applying state elimination to the minimal finite-state automaton M (the minimal automaton for L when considered to be over Σ). We show that this assumption is valid. If q is a state of a given automaton A, we let L_q denote the language recognized by A using q as the start state. The proof of the following result will be given in the full version.

Lemma 4. *Let L be a block-deterministic regular language. Then, there is a block-deterministic regular expression $E^\natural$ for L with the property that if p and q are two states of $\mathcal{D}(G_{E^\natural})$, then $L_p = L_q$ implies that either p and q are sets of duplicate states of $G_{E^\natural}$ or p and q are (single) non-duplicate states of $G_{E^\natural}$.*

Given a k-deterministic regular expression E, we let G_E and G_E^k be its corresponding Glushkov and k-block Glushkov automata, respectively. We consider the following two automata

$$M = \mathcal{M}(\mathcal{D}(G_E)) \qquad \text{and} \qquad M^k = \mathcal{M}(G_E^k) = \mathcal{M}(\mathcal{S}(G_E, Q_{dup})),$$

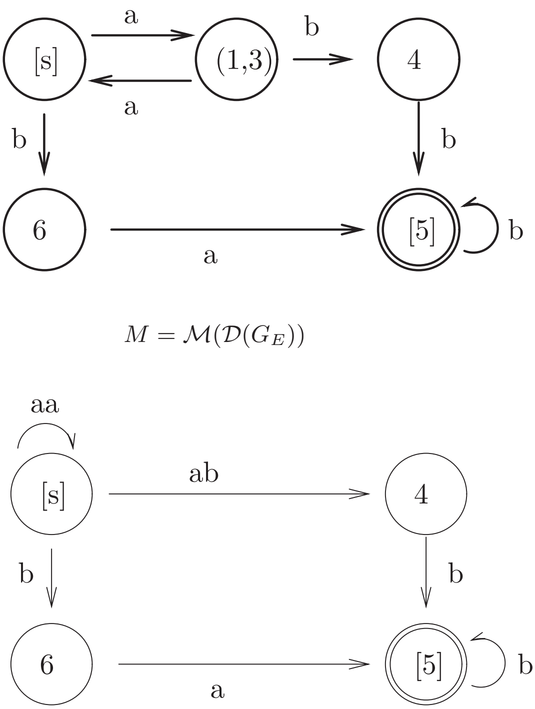
where M is obtained from G_E by applying first subset construction and then minimization whereas M^k is obtained from G_E^k by applying minimization. (Equivalently, M^k is obtained from G_E by first applying state elimination of all duplicate states and then applying minimization.)

Lemma 5. *Let L be a k-deterministic language. Then, there is a block-deterministic regular expression E for L such that M can be transformed into M^k by state elimination.*

Proof. Let Q_M and Q_{M^k} be the sets of states of the automata M and M_k, respectively. By Lemma 4, Q_{M^k} is a proper subset of Q_M (or, more precisely, Q_M contains an isomorphic copy of Q_{M^k}). Moreover, all the states in $Q_M \setminus Q_{M^k}$ are classes of duplicate states of G_E and their corresponding transitions define an acyclic subgraph of M (the set of all such states satisfies the state-elimination precondition).

Let us consider once again the running example expression E; that is, consider the language $L = L(E)$ on the alphabet $\Sigma = \{a, b\}$. The minimal finite-state automaton M for L in Fig. 2(a) is obtained by determinizing G_E of Fig. 1(a) and then minimizing it. When we apply the BW test to M, we see that L is not one-block deterministic. We then eliminate state $(1, 3)$ from M and obtain the automaton N_k of Fig. 2(b).

N_k, considered as an automaton on the block alphabet $\Gamma = \{a, b, aa, ab\}$, can be obtained minimizing the deterministic Glushkov block automaton of Fig. 1(b), where states s and 2 are equivalent, and states 5, 7 and 8 are equivalent. These



$M = \mathcal{M}(\mathcal{D}(G_E))$

$N_k = \mathcal{S}(M, \{1,3\})$

Fig. 2. A minimal finite-state automaton and state elimination. a. The minimal finite-state automaton M for the running example expression E. b. The result of eliminating state $(1,3)$ in M

observations imply that L is a one-block-deterministic automaton on Γ and a two-block-deterministic automaton on Σ.

Using the same approach, we can exhibit languages that are not k-block deterministic, for any k; therefore, they are not k-deterministic. One example language is $L = \{a + b\}^*\{a\{a + b\}^n\}$. Brüggemann-Klein and Wood [4] prove that L is not one-block deterministic. Moreover, we can verify that it does not pass the BW test after the state elimination of all states that satisfy the state-elimination precondition.

References

1. A. V. Aho and J. D. Ullman. *The Theory of Parsing, Translation, and Compiling, Vol. I: Parsing*. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1972. 184
2. G. Berry and R. Sethi. From regular expressions to deterministic automata. *Theoretical Computer Science*, 48:117–126, 1986. 189

3. R. V. Book, S. Even, S. A. Greibach, and G. Ott. Ambiguity in graphs and expressions. *IEEE Transactions on Electronic Computers*, C-20:149–153, 1971. 184, 185, 186
4. A. Brüggemann-Klein and D. Wood. One-unambiguous regular languages. *Information and Computation*, 140:229–253, 1998. 184, 186, 189, 190, 192, 195
5. P. Caron and D. Ziadi. Characterization of Glushkov automata. *Theoretical Computer Science*, 233(1-2):75–90, 2000. 186, 192
6. J. M. Champarnaud. Subset Construction Complexity for Homogeneous Automata, Position Automata and ZPC-Structures. *Theoretical Computer Science*, 269(1-2), to appear. 192
7. J. M. Champarnaud, D. Maurel, D. Ziadi. Determinization of Glushkov automata. Proc. *WIA'98*, LNCS, 1660:57–68, 1999. 192
8. S. Eilenberg. *Automata, Languages, and Machines*, volume A. Academic Press, New York, NY, 1974. 185, 186
9. D. Giammarresi and R. Montalbano. Deterministic generalized automata. *Theoretical Computer Science*, vol 215, 1-2, 191–208, Elsevier 1998. 187
10. D. Giammarresi, J.-L. Ponty, and D. Wood. The Glushkov and Thompson constructions: A synthesis. Unpublished manuscript, July 1998. 186
11. V. M. Glushkov. On a synthesis algorithm for abstract automata. *Ukr. atem. hurnal*, 12(2):147–156, 1960. In Russian. 186, 189
12. V. M. Glushkov. The abstract theory of automata. *Russian Mathematical Surveys*, 16:1–53, 1961. 186
13. ISO 8879: Information processing—Text and office systems—Standard Generalized Markup Language (SGML), October 1986. Int. Organization for Standardization. 184, 186
14. R. McNaughton and H. Yamada. Regular expressions and state graphs for automata. *IEEE Transactions on Electronic Computers*, 9:39–47, 1960. 186, 189

Constructing Finite Maximal Codes from Schützenberger Conjecture*

Marcella Anselmo

Dip. di Informatica ed Appl., Università di Salerno
I-84081 Baronissi (SA) Italy
anselmo@dia.unisa.it

Abstract. Schützenberger Conjecture claims that any finite maximal code C is factorizing, i.e. $SC^*P = A^*$ in a non-ambiguous way, for some S, P. Let us suppose that Schützenberger Conjecture holds. Two problems arise: the construction of all (S, P) and the construction of C starting from (S, P). Regarding the first problem we consider two families of possible languages S: S prefix-closed and S s.t. $S \setminus \{1\}$ is a code. For the second problem we present a method of constructing C from (S, P), that is relied on the construction of right- and left-factors of a language. Results are based on a combinatorial characterization of right- and left-factorizing languages.

1 Introduction

The theory of codes takes its origin in information theory, devised by Shannon in the 1950s. The codes were considered as communication tools. Then, in the 1960s, M. P. Schützenberger pointed out the close relations between codes theory and classical algebra (free monoids, groups, and so on). M. P. Schützenberger and his school investigate codes inside the theory of formal languages, using an algebraic, analytical or combinatorial approach. The aim is to give a structural description of the codes in a way that allows their *construction*. Remark that indeed no systematic method is known even for constructing all *finite codes*. Algorithms exist for some sub-classes: prefix codes, suffix codes, biprefix codes and n-codes, with $n \leq 3$ (see [17] and references inside). Further, starting from a factorizing code C one can construct an infinite family of factorizing codes C', applying *composition* ([6]) or *substitution*, introduced in [4,5]. Another way of constructing factorizing codes starting from a related class is given in [19].

Regarding the problem of constructing codes, there is the famous conjecture due to Schützenberger ([29,6]), that characterizes finite maximal codes as *factorizing codes*. A finite code $C \subseteq A^*$ is factorizing if there exist languages S, P s.t. $SC^*P = A^*$, by non-ambiguous operations; the couple (S, P) is called a *factorization* of C. Remark that non-ambiguity of operations on languages can be expressed, in an elegant and concise way, by introducing formal

* This work was partially supported by 60% Project: ”Linguaggi formali e modelli di calcolo”

A. Restivo, S. Ronchi Della Rocca, L. Roversi (Eds.): ICTCS 2001, LNCS 2202, pp. 197–214, 2001.

power series, and characteristic series in particular. A finite code is factorizing if there exist S, P s.t. $\underline{S}\ \underline{C}^*\underline{P} = \underline{A}^*$. The *Schützenberger Factorization Conjecture* is open after more than 30 years. Despite a lot of researchers worked on it ([10,11,15,16,18,19,20,26,27]), only partial results exist ([17,27,30]). Remark that if a finite code is factorizing then it can have several factorizations, as is the case of biprefix codes for example, or a unique factorization ([9]). On the other hand, by means of the substitution operation, if S appears in some equation $\underline{S}\ \underline{C}^*\underline{P} = \underline{A}^*$ then an infinite family of couples (C', P') can be constructed s.t. $\underline{S}\ \underline{C'}^*\underline{P'} = \underline{A}^*$ ([4,5]).

In this paper we study the construction of (finite maximal) codes, assuming that Schützenberger Factorization Conjecture holds. Indeed two problems arise:

1. the characterization/construction of all the couples (S, P) that can be a factorization of some finite code
2. the construction of a finite code C starting from its factorization (S, P).

Problem 1 was firstly singled out in [15], where it is pointed out that the couple (S, P) can be a factorization of a finite code iff $P(A-1)S + 1 \geq 0$. We survey all the triplets (S, C, P) we know to be related by $\underline{S}\ \underline{C}^*\underline{P} = \underline{A}^*$. Then we consider two more possible families of languages S: S prefix-closed and S s.t. $S \setminus \{1\}$ is a code and $|S| \leq 4$. We establish conditions under which they can appear in a factorization $\underline{S}\ \underline{C}^*\underline{P} = \underline{A}^*$ (Corollary 2 and Proposition 2).

Considering Problem 2, we propose and compare two possible ways of constructing a finite factorizing code C, once its factorization (S, P) is given. The first method uses the definition and comes out to be less efficient than the second one. The second method is based on a result (Proposition 3) that allows to express C in terms of the *right-factor* of S and the *left-factor* of P (see Section 4 for the definitions). It is more efficient than the first method, as far as an efficient construction of right-factors and left-factors of finite languages is available. We propose a new construction of right-factors and left-factors of a finite language, that is more efficient than the one already given in [1]. Remark that this construction, as well as Proposition 2, is based on a combinatorial characterization of the right- (left-, resp.) factor of a language (Theorem 1), involving the factorizations of a word and an *alternating property.*

The paper starts with a section devoted to some background. Section 3 contains a survey on all the triplets (S, C, P) we know to be related by $\underline{S}\ \underline{C}^*\underline{P} = \underline{A}^*$. Section 4 contains our characterization of right- (left-, resp.) factorizing languages. Section 5 contains our contributions to the problem stated in item 1. In Section 6 we consider two methods for constructing C, once its factorization (S, P) is given. Some conclusions are given in the last section.

2 Background and Notations

For definitions about formal languages and automata, see for example [23]. We note here that, given a finite alphabet A, $< A^*, \cdot, 1 >$ denotes the free monoid generated by A and a *language* is any $S \subseteq A^*$. We will denote by $A^{<n} =$

$\{w \in A^* \quad w| < n\}$ and by $x^{-1}w$ the word y s.t. $w = xy$, if any. Moreover the *reverse* of word $w = a_1 \cdots a_n$ is $w^R = a_n \cdots a_1$ and the *reverse* of language S is $S^R = \{w^R \quad \in S\}$. A word $x \in A^*$ is a *prefix* of $w \in A^*$, and we write $x \leq w$, if $w = xy$, with $y \in A^*$; it is a *proper prefix* if $x \neq w$ and we write $x < w$. $Pref(S)$, ($Suff(S)$, $Fact(S)$, resp.) denotes the set of proper prefixes (suffixes, factors, resp.) of words in S. S is *prefix- (suffix-, factor-*, resp.) *closed* if $Pref(S) \subseteq S$ ($Suff(S) \subseteq S$, $Fact(S) \subseteq S$, resp.). The union $X \cup Y$ is non-ambiguous when $X \cap Y = \emptyset$; the product XY is non-ambiguous when $w = xy = x'y'$ with $x, x' \in X$, $y, y' \in Y$ implies $x = x'$ and $y = y'$ and the star X^* is non-ambiguous when all unions and products in its definition are non-ambiguous. For the sake of simplicity, we will sometimes write 1 instead of $\{1\}$.

We now recall some notations about formal power series; for more details see [7,25,28]. Given a finite alphabet A and a semi-ring K, the class $K \ll A \gg$ of *formal power series* (briefly *series*) with non-commuting variables in A and coefficients in K is the set of functions $s : A^* \to K$. As usual, the value of s on $w \in A^*$ is denoted by (s, w) and the power series is written as a formal sum $s = \sum_{w \in A^*} (s, w)w$. The *image* of the series s is the set $Im(s) = \{(s, w)| \quad \in A^*\}$. The *support* of s is the set $supp(s) = \{w \in A^*| \quad s, w) \neq 0\}$. The *characteristic series* of a language $X \subset A^*$, denoted $\underline{X}$, is defined by $(\underline{X}, w) = 1$ if $w \in X$ and $(\underline{X}, w) = 0$ if $w \notin X$. By this formalism, we have that $X \cup Y$ is non-ambiguous iff $\underline{X \cup Y} = \underline{X} + \underline{Y}$; XY is non-ambiguous iff $\underline{XY} = \underline{X} \cdot \underline{Y}$; X^* is non-ambiguous iff $\underline{X^*} = (\underline{X})^*$.

3 Factorizing Codes and Their Factorizations

In this section we consider factorizing codes and present the Schützenberger's Factorization Conjecture. Then we survey some results related to it, in the aim of Problem 1 in Section 1. Our main reference for codes is [6]. See also [11,12] for some open problems in the field.

A subset C of A^* is a *code* if for any $c_1, \ldots, c_h, c'_1, \ldots, c'_k \in C$, the equality $c_1 \cdots c_h = c'_1 \cdots c'_k$ implies $h = k$ and for every $i \in \{1, \ldots, h\}$, $c_i = c'_i$. In the terminology of series, C is a code iff $\underline{C^*} = (\underline{C})^*$. A *prefix* (*suffix*, resp.) code is a language such that no word is a prefix (suffix, resp.) of another one in the language. A code C is *maximal* over A if for any code C' over A then $C \subseteq C'$ implies that $C = C'$.

A finite code C is *factorizing* (over A) if there exist two finite subsets S, P of A^* such that $\underline{S}\ \underline{C}^* \underline{P} = \underline{A}^*$. The couple (S, P) is called a *factorization* of C. A finite language $S \subset A^*$ such that $\underline{S}\ \underline{C}^* \underline{P} = \underline{A}^*$ for finite languages $C, P \subseteq A^*$, is called a *polynomial having solutions* in [15] and *strong factorizing* in [3]. The first terminology is motivated by the remark there exists C s.t. $\underline{S}\ \underline{C}^* \underline{P} = \underline{A}^*$ iff $\underline{P}(\underline{A} - 1)\underline{S} \geq 0$. Further we have that if $\underline{S}\ \underline{C}^* \underline{P} = \underline{A}^*$ then S, P are finite iff C is a finite and maximal code ([29,6]).

The most important conjecture on theory of codes is *Schützenberger Factorization Conjecture* ([29,6]). It claims that *any finite maximal code is factorizing*. In this paper we study the construction of (finite maximal) codes, assuming that Schützenberger Factorization Conjecture holds. A problem that arises is the characterization/construction of couples (S, P) that are factorizations of some code. Let us now survey the results we know about.

Firstly in [4,5] it is proved that it is decidable whether a finite language S is strong factorizing and a construction for related couples (C, P) is given, if any. We will say that a language S is *right-context-closed* if $s = s't$ with $s, s' \in S$ implies $t \in S$. For example, any suffix-closed language is right-context-closed. Any factor-closed language is suffix-closed and hence right-context-closed.

Let us now enumerate some couples (S, P) that appear in some equation $\underline{S}\ \underline{C}^*\underline{P} = \underline{A}^*$. Remark that 8 is a particular case of 5; 9 is a particular case of 11, 12 and 10 is a particular case of 11.

1. $S = 1$, C prefix code, $P = A^* \setminus CA^*$
2. $S = 1$, C maximal prefix code, $P = Pref(C)$ (P is prefix-closed)
3. $S = A^* \setminus A^*C$, C is a suffix code, $P = 1$
4. $S = Suff(C)$ (S is suffix-closed), C maximal suffix code, $P = 1$
5. $S = 1 \cup X$ with X prefix code, $P = Pref(X)$ ([3]) (all possible P's are characterized in [15])
6. $S = 1 \cup X$ with X suffix code iff X is biprefix ([15])
7. $S = 1 \cup X$ with X code, $|S| \leq 3$ iff X is prefix ([3])
8. $S = \{1, v\}$ (all possible P's are characterized in [15])
9. $S = \{1, a^2\}$, $P = \{1, a, aba^2, aba^3, aba^2b\}$,
   $C = \{a^4, ab, aba^6, aba^3b, aba^3ba^2, aba^2ba, aba^2ba^3, aba^2b^2, aba^2b^2a^2, b, ba^2\}$
   (C is an example of a factorizing code that is neither prefix nor suffix)
10. $|S| = 3$ and $S \setminus 1$ not a code iff $S = \{1, v, v^2\}$, $P = Pref(\{v\})$ ([3]) (all possible P's are characterized in [21])
11. $S \subseteq w^*$ iff $S = w^I$ and (I, J) a Krasner factorization, (possible P's are studied in [21])
12. $A = \{a, b\}$ $S \subseteq a^*$ iff $S = a^I$ and (I, J) a Krasner factorization, (all possible P's are characterized in [17])
13. S right-context-closed (suffix-closed, factor-closed), $P = Pref(S \setminus 1) \setminus (S \setminus 1)$ A^+ (P is prefix-closed) ([15]).

Note that from the above list we can obtain an infinite list using: *duality*, *composition*, *substitution* or *extension of the alphabet*, as follows.

If (S, C, P) are s.t. $\underline{S}\ \underline{C}^*\underline{P} = \underline{A}^*$ then (P^R, C^R, S^R) are s.t. $\underline{P^R}\ \underline{C^{R^*}}\ \underline{S^R} = \underline{A}^*$ (see Remark 1). Therefore if (S, C, P) is a triplet in the above list, then its *dual* (P^R, C^R, S^R) can also appear in the list. For example the triplet in point 3 is the dual of the one in point 1. The composition is a well-known operation on codes ([6]). It holds that the composition of two factorizing codes is a factorizing code ([8]). Substitution is an operation on languages introduced in [4,5] that allows to construct from a factorizing code C with factorization (S, P) an infinity family of factorizing codes C' with factorization (S, P').

Further, as an extension of a remark in ([2]), we have that if $\underline{S} \cdot \underline{Y} = \underline{A^*}$ and B is an alphabet s.t. $B \supseteq A$ then $\underline{S} \cdot \underline{Y'} = \underline{B^*}$ with $Y' = Y \cup Y(B \setminus A)B^*$. Moreover one can easily show the following lemma.

Lemma 1. *Let $S \subset A^*$ and B an alphabet s.t. $B \supseteq A$. If $\underline{S}\ \underline{C}^*\underline{P} = \underline{A}^*$ then $\underline{S}\ \underline{C_B}^*\underline{P} = \underline{B}^*$ where $C_B = C \cup P(B \setminus A)S$.*

In Section 5, we will give some contributions to the above list. In view of results 5, 6, 7 we will conjecture that $S = 1 \cup X$ with X code is strong factorizing iff X is prefix. We will prove this result for $|S| = 4$. Then, in order to complete item 13, we will study the case S is prefix-closed and show that S prefix-closed is strong factorizing iff it is factor-closed.

4 A Characterization of Right- and Left-Factorizing Languages

We introduce right- and left- factorizing languages. Right-factorizing languages were firstly defined in [1], where they were simply called factorizing. We characterize them, in terms of some combinatorial properties based on the *factorizations* of a word. This characterization will be used in Section 5 to decide whether some languages are strong factorizing and in Section 6 to construct an automaton recognizing the right-factor of a finite language.

Definition 1. *A language $S \subset A^*$ is* right-factorizing (left-factorizing, *resp.) if there exists $Y \subset A^*$ such that $\underline{S}\ \underline{Y} = \underline{A^*}$ ($\underline{Y}\ \underline{S} = \underline{A^*}$, resp.). In this case, Y is called the* right factor (left factor, *resp.) of S and denoted by $RF(S)$ ($LF(S)$, resp.).*

For the sake of simplicity, we will sometimes write "r-" ("l-", resp.) instead of "right-" ("left-", resp.).

Remark 1. If $\underline{X}\ \underline{Y} = \underline{A^*}$ then $\underline{Y^R}\ \underline{X^R} = \underline{A^*}$. Therefore X is r-factorizing iff X^R is l-factorizing. We emphasize that hence any property on r-factorizing languages yields a *dual* property on l-factorizing languages, just moving to the reverse of languages. For instance, in the dual property "prefix" will be replaced by "suffix".

Remark 2. Let S a right-factorizing language. It can be easily shown that $S \cap RF(S) = \{1\}$; $A^* \setminus RF(S) = (S \setminus 1)RF(S)$; and $A^* \setminus (S \setminus 1)A^* \subseteq RF(S)$. Further, if s is a word of minimal length in $S \setminus 1$, then $A^{<|s|} \subseteq RF(S)$.

Observe that if $\underline{S}\ \underline{C}^*\underline{P} = \underline{A}^*$ then S, SC^* are right-factorizing with $RF(S) = C^*P$ and $RF(SC^*) = P$. Similarly P, C^*P are left-factorizing with $LF(P) = SC^*$ and $LF(C^*P) = S$. Therefore any strong factorizing language is r- factorizing too. An example of a language that is r-factorizing, but not strong factorizing is $S = \{1, a, a^2b\}$ ([3]). The language $S = \{1, a, ab\}$ is not right-factorizing ([1]). Right-factorizing languages with at most three words are completely characterized in [1,2].

In [1,2], given a language $S \subseteq A^*$ with $1 \in S$, the series $r_S = (\underline{S})^{-1}\underline{A}^*$ is considered. It is shown that $(\underline{S})^{-1} = (\underline{S \setminus 1} \cdot \underline{S \setminus 1})^* - (\underline{S \setminus 1} \cdot \underline{S \setminus 1})^*\underline{S \setminus 1}$ and that S containing 1 is right-factorizing iff r_S is a characteristic series; in this case $RF(S) = supp(r_S)$. A combinatorial characterization is then presented by means of the following definitions. A *prefix sequence of w with respect to S* is a sequence $(x_1, x_2, \ldots, x_n)$ such that either $n = 1$ and $x_1 = 1$ or $x_i \in (S \setminus 1)$ for $i \in \{1, \ldots, n\}$ and $x_1 \cdots x_n \leq w$. It is said an *even prefix sequence* if n is even or it is (1) and an *odd prefix sequence* if n is odd. Furthermore a language $S \subseteq A^*$ is right-factorizing iff for any word $w \in A^*$, the difference between the number of its even prefix sequences w.r.t. S and the number of its odd prefix sequences w.r.t. S is always either 0 or 1. Finally, if S is right-factorizing then $RF(S)$ is the set of words for which this difference is 1.

We introduce here $l_S = \underline{A}^*(\underline{S})^{-1}$ and observe that S is left-factorizing iff l_S is a characteristic series and in this case $LF(S) = supp(l_S)$.

Example 1. Let $S = \{1, a, a^2\}$. It can be shown that S is factorizing and $RF(S) = (a^3)^*$ since $(r_S, a^i) = 1$ if $i = 3k$ for some $k \geq 0$ and 0 otherwise ([1,2]). As an example:
for $k = 0$ the unique even prefix sequence is (1) and there are no odd prefix sequences, yielding $(r_S, 1) = 1$;
for $k = 1$ the unique even prefix sequence is (1) and the unique odd prefix sequence is (a), yielding $(r_S, a) = 0$;
for $k = 2$ the even prefix sequences are $(1), (a, a)$ and the odd prefix sequences are $(a), (a^2)$, yielding $(r_S, a^2) = 0$;
for $k = 3$ the even prefix sequences are $(1), (a, a), (a, a^2), (a^2, a)$ and the odd prefix sequences are $(a), (a^2), (a, a, a)$, yielding $(r_S, a^3) = 1$.

Let us give a combinatorial characterization of right- (left-, resp.) factorizing languages and their right- (left-, resp.) factors, starting from the above considerations. Indeed we relate (r_S, wa) with (r_S, w), for $w \in A^*, a \in A$, thus obtaining a sort of recursive way of expressing r_S.

Definition 2. *Let $S \subseteq A^*$ with $1 \in S$. The formal power series f_S is defined as $f_S = (\underline{S})^{-1}$.*

Definition 3. *A* factorization of w with respect to S *is a sequence $(x_1,x_2,\ldots,x_n)$ such that either $n = 1$ and $x_1 = w = 1$ or $x_i \in (S \setminus 1)$ for all $i \in \{1, \ldots, n\}$ and $x_1 \cdots x_n = w$. It is said an* even factorization *if n is even or it is (1) and an* odd factorization *if n is odd.*

Remark 3. As above mentioned, the series f_S equals $f_S = (\underline{S \setminus 1} \cdot \underline{S \setminus 1})^* - (\underline{S \setminus 1} \cdot \underline{S \setminus 1})^* \underline{S \setminus 1}$. Therefore for any $w \in A^*$, the value (f_S, w) is the difference between the number of even factorizations of w and the number of its odd factorizations.

Factorizations of a word are indeed prefix sequences ending at the right-end of the word. Counting factorizations instead of all prefix sequences is thus a gain. Next lemmas show that counting factorizations (by f_S), instead of prefix sequences (by r_S) is sufficient to establish whether a language is r-factorizing and distinguish words in its right factor.

Lemma 2. *Given a language $S \subseteq A^*$ with $1 \in S$, $a \in A$, $w \in A^*$, $z \in A^+$, we have that:*
$(r_S, 1) = (l_S, 1) = (f_S, 1) = 1$;
$(r_S, wa) = (r_S, w) + (f_S, wa)$ *and* $(l_S, aw) = (l_S, w) + (f_S, aw)$;
$(r_S, z) = (r_S, x) + (f_S, z)$, *where x is the longest proper prefix of z s.t.* $(f_S, x) \neq 0$
$(l_S, z) = (l_S, y) + (f_S, z)$, *where y is the longest proper suffix of z s.t.* $(f_S, y) \neq 0$.

From Lemma 2 and using induction on $|w|$, we obtain the following.

Lemma 3. *Given a language $S \subseteq A^*$ with $1 \in S$, if r_S (l_S, resp.) is a characteristic series then*

1. $Im(f_S) \subseteq \{-1, 0, 1\}$
2. *for any* $w \in A^*$,
   if $(f_S, w) = 1$ *then* $(r_S, w) = 1$ ($(l_S, w) = 1$, *resp.*);
   if $(f_S, w) = -1$ *then* $(r_S, w) = 0$ ($(l_S, w) = 0$, *resp.*);
   if $(f_S, w) = 0$ *then* $(r_S, w) = (r_S, x)$ ($(l_S, w) = (l_S, x)$, *resp.*) *where x is the longest prefix (suffix, resp.) of w s.t.* $(f_S, x) \neq 0$.

Corollary 1. *Let $S \subseteq A^*$ and $S_i = \{w \in A^*$ s.t. $(f_S, w) = i\}$, for $i = -1, 0, 1$. If S is a right-factorizing language then*

1. *$RF(S)$ is the set of words whose longest prefix in $S^* \setminus S_0$ belongs to S_1*
2. *$A^* \setminus RF(S)$ is the set of words whose longest prefix in $S^* \setminus S_0$ belongs to S_{-1}.*

Example 2. (continued) Let $S = \{1, a, a^2\}$. S is factorizing and $RF(S) = (a^3)^*$ ([1,2]). As an example:
for $k = 0$ the unique even factorization is (1) and there are no odd factorizations, yielding $(f_S, 1) = 1$;
for $k = 1$ the unique odd factorization is (a), yielding $(f_S, a) = -1$;
for $k = 2$ the unique even factorization is (a, a) and the unique odd factorization is (a^2), yielding $(f_S, a^2) = 0$;
for $k = 3$ the even factorizations are $(a, a^2), (a^2, a)$ and the unique odd factorization is (a, a, a), yielding $(f_S, a^3) = 1$.

Figure 1 shows the values of f_S and r_S on a^k for any $k = 0, \cdots, 8$.

The condition 1 of Lemma 3 is not sufficient, as shown by this example.

Example 3. Let $S = \{1, a, ab\}$. Since $S \setminus 1$ is a code, then every word $w \in A^*$ has one factorization at most and thus $Im(f_S) \subseteq \{-1, 0, 1\}$. On the other hand S is not right-factorizing. Consider for example the word ab. We have $(f_S, 1) = 1$; $(f_S, a) = -1$ and $(r_S, a) = 0$; $(f_S, ab) = -1$ and thus (following Lemma 2) $(r_S, ab) = (r_S, a) + (f_S, ab) = -1$, showing that r_S is not a characteristic series.

A characterization of right-factorizing languages is indeed given by the following theorem.

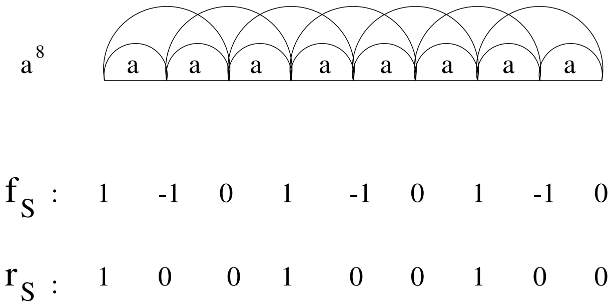


Fig. 1. The values of f_S, r_S on $a^0, a, a^2, \cdots, a^8$ for $S = \{1, a, a^2\}$

Theorem 1. *Let $S \subseteq A^*$ with $1 \in S$ and $f_S = \underline{S}^{-1}$.*
S is a right- (left-, resp.) factorizing language iff A^* can be partitioned as $A^* = S_{-1} + S_0 + S_1$ where $S_i = \{w \in A^*$ s.t. $(f_S, w) = i\}$, for $i = -1, 0, 1$ and the following alternating property *holds:*

1. $\forall \ \in S_1 \setminus 1$ its longest proper prefix (suffix, resp.) in $S^* \setminus S_0$ belongs to S_{-1}
2. $\forall \ \in S_{-1}$ its longest proper prefix (suffix, resp.) in $S^* \setminus S_0$ belongs to S_1.

Proof. We prove the theorem for S right-factorizing. The proof for left-factorizing is the dual one. Recall that ([1,2]) S is r-factorizing iff r_S is a characteristic series.

Let us suppose that S is right-factorizing. Then r_S is a characteristic series ([1,2]) and $Im(f_S) \subseteq \{-1, 0, 1\}$, by Lemma 3. It follows that A^* can be partitioned as $A^* = S_{-1} + S_0 + S_1$.

Now let us suppose that the alternating property does not hold. Let $w \in S_1 \setminus 1$ s.t. its longest proper prefix x in $S^* \setminus S_0$ belongs to S_1. By Lemma 2, $(r_S, w) = (r_S, x) + (f_S, w) = (r_S, x) + 1$. Moreover $(r_S, x) = 1$ because $(f_S, x) = 1$ and Lemma 3 holds. Therefore we had $(r_S, w) = 1 + 1 = 2$ against r_S is a characteristic series. In an analogous way, if $w \in S_{-1}$ and its longest proper prefix x in $S^* \setminus S_0$ belongs to S_{-1} then we had $(r_S, w) = 0 - 1 = -1$.

For the vice versa let us suppose that $Im(f_S) \subseteq \{-1, 0, 1\}$ and that the alternating property holds. We are going to show that r_S is a characteristic series. Indeed we are going to show by induction on $|w|$, that for any $w \in A^*$ and x longest proper prefix of w s.t. $(f_S, x) \neq 0$, we have $(r_S, w) \in \{0, 1\}$ and if $(f_S, w) = 1$ then $(r_S, w) = 1$, if $(f_S, w) = -1$ then $(r_S, w) = 0$, if $(f_S, w) = 0$ then $(r_S, w) = (r_S, x)$.

If $|w| = 0$ then $(r_S, 1) = (f_S, 1) = 1$, by Lemma 2.

Let w s.t. $|w| > 1$. From $Im(f_S) \subseteq \{-1, 0, 1\}$, three cases are possible. If $(f_S, w) = 1$ then $(f_S, x) = -1$ by the alternating property, $(r_S, x) = 0$ by the inductive hypothesis, and then $(r_S, w) = (r_S, x) + (f_S, w) = 0 + 1 = 1$ by Lemma 2. In an analogous way, if $(f_S, w) = -1$ then $(f_S, x) = 1$, $(r_S, x) = 1$, and then $(r_S, w) = (r_S, x) + (f_S, w) = 1 - 1 = 0$. If $(f_S, w) = 0$ then $(r_S, w) = (r_S, x) \in \{0, 1\}$ by Lemma 2 and the inductive hypothesis. □

Let us now express Theorem 1 and Corollary 1, in a more descriptive way. Following the scheme used in Example 2 (see Fig. 1), for any word we point out the values of f_S on its prefixes, from the shortest one by increasing length. Given $S \subseteq A^*$, $w = a_1 \cdots a_n \in A^*$, let us define:

$\rho_S(w) = (f_S, 1)(f_S, a_1) \cdots (f_S, a_1 \cdots a_n)$ and

$\lambda_S(w) = (f_S, a_1 \cdots a_n)(f_S, a_1 \cdots a_{n-1}) \cdots (f_S, 1)$.

Remark that $\rho_S(w)$ also shows the values of ρ_S on every prefix of w.

Example 4. (continued) Let $S = \{1, a, a^2\}$. For any $k \geq 0$ we have $\rho_S(a^{3k}) = (1(-1)0)^k$, $\rho_S(a^{3k+1}) = (1(-1)0)^k 1$, $\rho_S(a^{3k+2}) = (1(-1)0)^k 1(-1)$ (cfr. Fig. 1).

Example 5. If $S = \{1, a, a^2b\}$ then $\rho_S(a^2ba) = 1(-1)1(-1)1$.

Example 6. Let $S = \{1, a^3ba, a, b, a^2, ab, ba, a^3, a^2b, aba, a^3b, a^2ba\}$. Remark that S is a factor-closed language: it contains $1, w = a^3ba$ and every factor of w. Using Lemma 4, it can be shown that $\rho_S(a^3ba) = 1(-1)0000$.

Theorem 1 and Corollary 1 can be restated as follows, keeping in mind that for any language S, $(f_S, 1) = 1$ and S is factorizing iff the values of f_S can be only $0, 1, -1$ and in such a way that non-zero values $1, -1$ alternates in a reading of the word from the left end to its right end.

Theorem 2. *Let $S \subseteq A^*$, $U_+ = 1(0^*(-1)0^*10^*)^*$, $U_- = 1(0^*(-1)0^*10^*)^*(-1)0^*$ and $U = U_+ \cup U_-$.*

The language S is right-(left-, resp.) factorizing iff for any $w \in A^*$, $\rho(w) \in U$ ($\lambda(w) \in U$, resp.). Moreover if S is right- factorizing then $RF(S) = \{w$ s.t. $\rho(w) \in U_+\}$ and $A^* \setminus RF(S) = \{w$ s.t. $\rho(w) \in U_-\}$; if S is left-factorizing then $LF(S) = \{w$ s.t. $\lambda(w) \in U_+\}$ and $A^* \setminus LF(S) = \{w$ s.t. $\lambda(w) \in U_-\}$.

Finally let us show how to compute the value of f_S on a word, once the values of f_S on its proper prefixes are known.

Lemma 4. *Let S a language, $w = a_1 \cdots a_n \in A^+$ and $H_S(w) = \{h \mid h = |x_1 \cdots x_{m-1}|$ and $(x_1, \cdots, x_m)$ is a factorization of $w\}$. Then*

$$(f_S, w) = - \sum_{h \in H_S(w)} (f_S, a_1 \cdots a_h)$$

where $a_1 \cdots a_h = 1$ if $h = 0$.

Proof. Remark that for any factorization $(x_1, \cdots, x_m)$ of w, $m \geq 2$, m is even iff $m-1$ is odd and that if $m = 1$ then $0 \in H_S(w)$. Moreover denote $fe_S = (\underline{S \setminus 1} \cdot \underline{S \setminus 1})^*$, $fo_S = (\underline{S \setminus 1} \cdot \underline{S \setminus 1})^* \underline{S \setminus 1}$ and observe that $(fe_S, 1) = 1$, $(fo_S, 1) = 0$. Consider now $w \in A^+$. Then $(fe_S, w) = \sum_{h \in H_S(w)} (fo_S, a_1 \cdots a_h)$ and $(fo_S, w) = \sum_{h \in H_S(w)} (fe_S, a_1 \cdots a_h)$. Finally, $(f_S, w) = (fe_S, w) - (fo_S, w) =$

$-[\sum_{h \in H_S(w)} (fe_S, a_1 \cdots a_h) - \sum_{h \in H_S(w)} (fo_S, a_1 \cdots a_h)] =$

$-\sum_{h \in H_S(w)} (f_S, a_1 \cdots a_h)$. □

5 Some Classes of Languages S

In this section, we give some contributions to the characterization of finite languages S, C, P s.t. $\underline{S}\ \underline{C}^*\underline{P} = \underline{A}^*$, using results of previous sections. Motivated by the state of art, as presented in Section 3, we consider two more cases of possibly strong factorizing languages: the case S is prefix-closed and the case $S \setminus 1$ is a code. Remark that these two cases are at the opposite sides, regarding the number of factorizations on S of a word. We show that a finite prefix-closed language is strong factorizing iff it is factor-closed. Then we obtain that a language S s.t. $|S| \leq 4$ and $S \setminus 1$ is a code, is strong factorizing iff $S \setminus 1$ is prefix and we conjecture that the result holds for any value of the cardinality of S. Proposition 1 and its proof are partially outlined in [14].

Proposition 1. *A prefix-closed language is right-factorizing iff it is factor-closed.*

Proof. Any factor-closed language is trivially prefix-closed. Moreover it is (strong) r-factorizing, as shown in [15].

For the converse, let $S \subseteq A^*$ be a prefix-closed and r-factorizing language. We are going to show that for any $w \in S$, all of its factors are in S, by induction on the length of w.

If $|w| = 1$ then the only factor of w is w itself and thus the goal.

Let $w = a_1 \cdots a_n$, $n \geq 2$. By the inductive hypothesis, all the factors of w that are factors of $a_1 \cdots a_{n-1}$ belong to S. Using induction on h and Lemma 4 it can be shown that $(f_S, 1) = 1$, $(f_S, a_1) = -1$ and $(f_S, a_1 \cdots a_h) = 0$ for any $2 \leq h \leq a_{n-1}$ (cfr. Example 6).

Let us now consider $(f_S, a_1 \cdots a_n)$. From $a_1 \cdots a_n \in S \setminus 1 \subseteq A^* \setminus RF(S)$ we have that $(f_S, a_1 \cdots a_n)$ equals either 0 or -1. But $(f_S, a_1 \cdots a_n) = -1$ contradicts the alternating property of Theorem 1. Hence $(f_S, a_1 \cdots a_n) = 0$. Using Lemma 4, the unique possibility comes out to be $a_2 \cdots a_n \in S$. The inductive hypothesis applied to $a_2 \cdots a_n$ achieves the proof. □

Corollary 2. *A finite prefix-closed language is strong factorizing iff it is factor-closed.*

Let us now consider the case $S \setminus 1$ is a code. Motivated by results 5, 6, 7 of Section 3, we present the following conjecture and we prove it in the particular case $|S| \leq 4$.

Conjecture 1. A language $S = 1 \cup X$ with X code is strong factorizing iff X is prefix.

The proofs of the following results are very technical and based on combinatorics on words. They use results in Section 4 and are based on the following considerations. Firstly observe that, when studying right-factorizing languages S, the case where $S \setminus 1$ is a code is a very special case. If $S \setminus 1$ is a code then $S^* = S_1 + S_{-1}$, $S^* \cap RF(S) = S_1 = ((S \setminus 1)^2)^*$, and $S^* \setminus RF(S) =$

$S_{-1} = ((S \setminus 1)^2)^*(S \setminus 1)$. Further in the considered case ($S \setminus 1$ code, $|S| = 4$) it is possible to single out some words that are forbidden to be in $RF(S)$ (namely $((S \setminus 1)^2)^*(S \setminus 1)$) and some words that are forced to be in C^+ (namely s_1^{2k} in Lemma 6, $(s_i s_j)^k$ or $s_2 s_j^h l$ in Proposition 2). Such words cannot be arbitrarily concatenated. It is a trivial (but very crucial in the sequel) remark, that if $\underline{S}\ \underline{C}^* \underline{P} = \underline{A}^*$ then $RF(S) = C^*P$, $C \cdot RF(S) \subseteq RF(S)$ and equivalently, $w = ct$ with $w \notin RF(S)$, $c \in C$ implies $t \notin RF(S)$.

Let us now present some results (Lemmas 5, 6, 7) on shortest words of a language S s.t. $S \setminus 1$ is a code.

Lemma 5. *Let S be a right-factorizing language. If $S \setminus 1$ is a non-prefix code and s is a shortest word of $S \setminus 1$ having a proper prefix in $S \setminus 1$ then $s = s's''y$ where $s', s'' \in S \setminus 1$ and $y \in RF(S)$.*

Proof. Let s' be the proper prefix of s in $S \setminus 1$ (it is unique because of minimality of s). Let $s = a_1 \cdots a_n$, $s' = a_1 \cdots a_h$, $1 \leq h < n$. We have $\rho(s') \in 10^*(-1)$ and $(f_S, a_1 \cdots a_n) = -1$. From the alternating property of Theorem 1, there exists $h < k < n$ s.t. $(f_S, a_1 \cdots a_k) = 1$; let k be the smallest one. The minimality of s implies that the even factorization of $a_1 \cdots a_k$ is $(a_1 \cdots a_h, a_{h+1} \cdots a_k)$. Let $s'' = a_{h+1} \cdots a_k \in S \setminus 1$ and $y = a_{k+1} \cdots a_n$. We show that $y \in RF(S)$, by showing that its longest prefix x in $S^* \setminus S_0$ belongs to S_1 (cfr. Corollary 1). Suppose by the contrary that $x \in S_{-1}$. We had that $s's''x, s \in S_{-1}$ and $s's''x \neq s$ since $S \setminus 1$ is a code. Moreover no $z \in S_1$ could exist s.t. $s's''x < z < s$ because of the minimality of s. This is a contradiction to the alternating property. □

Lemma 6. *Let S be a strong factorizing language. If $S \setminus 1$ is a code and s_1 is a shortest word of $S \setminus 1$ then there exists $k > 0$ s.t. $s_1^{2k} \in C^+$.*

Proof. Firstly $(s_1^2)^* \subseteq S_1 \subseteq RF(S)$ and $RF(S) = C^*P$ for some C, P, finite languages. Because of finiteness of P, $(s_1^2)^* \subseteq P$ cannot hold. Let us now suppose that $\nexists k$ s.t. $s_1^{2k} \in C^+$. This means that $\exists h$ (indeed $\exists h_0$ s.t. $\forall h \geq h_0$) s.t. $s_1^{2h} = cp$ where $c \in C^+, p \in P \setminus 1$ and $s_1^{2m} < c < s_1^{2m+2}$ for some $m < h$. Two cases are possible: either $s_1^{2m} < c < s_1^{2m+1}$ or $s_1^{2m+1} \leq c < s_1^{2m+2}$.

If $s_1^{2m} < c < s_1^{2m+1}$ then $s_1^{2m+1} = ct$, $t \neq 1$ and $t \in A^{<|s_1|} \subseteq RF(S)$ from Remark 2. Therefore we had $s_1^{2m+1} \in C^+RF(S) \subseteq RF(S)$ against $s_1^{2m+1} \in S_{-1} \subseteq A^* \setminus RF(S)$.

Let us suppose now $s_1^{2m+1} \leq c < s_1^{2m+2}$. Because $c \in C^+ \subseteq RF(S)$, the longest prefix x of c in $S^* \setminus S_0$, $x \in S_1$ and then $s_1^{2m+1} < x \leq c$. From $x, s_1^{2m+2} \in S_1$ and the alternating property it follows that $\exists z \in S_{-1}, x < c < z < s_1^{2m+2}$. Let $z = ct'$. Then $t' \in A^{<|s_1|} \subseteq RF(S)$ by Lemma 2 and then $z \in C^+RF(S) \subseteq RF(S)$.

□

Lemma 7. *Let S be a strong factorizing language. If $S \setminus 1$ is a non-prefix code, s_1 is a shortest word of $S \setminus 1$ and s_2 is a shortest word in $S \setminus \{1, s_1\}$ then s_1 is not a prefix of s_2.*

Proof. If $|s_2| = |s_1|$ then s_1 cannot be a prefix of s_2. Otherwise (see Fig. 2), let us suppose that $s_1 < s_2$. From Lemma 5 and since there is no s'' s.t. $|s_1| < |s''| < |s_2|$, $s_2 = s_1 s_1 y$ with $y \in RF(S)$. Let k be s.t. $s_1^{2k} \in C^+$, as in Lemma 6. A contradiction holds for the word $w = s_1^{2k} y$. Indeed $w \in C^+ RF(S) \subseteq RF(S)$ against $w = (s_1)^{2(k-1)} s_2 \in S_{-1} \subseteq A^* \setminus RF(S)$. □

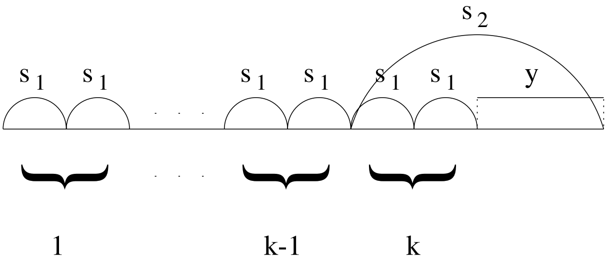


Fig. 2. Proving Lemma 4

Our goal is now to prove that if S is a strong factorizing language with $S \setminus 1$ a code and $|S| = 4$, then $S \setminus 1$ is prefix. In view of Lemma 7, we only have to show that the longest word in S cannot have a proper prefix in $S \setminus 1$. This is achieved by using an (not trivial) extension of the proofs of Lemmas 6 and 7, that we do not give here in details, since it uses similar techniques, but with more cases and sub-cases to handle.

Proposition 2. *Let $S \subseteq A^*$ s.t. $S \setminus 1$ is a code and $|S| = 4$. Then S is strong factorizing iff $S \setminus 1$ is prefix.*

Proof. (Sketch) If $S \setminus 1$ is prefix then it is strong factorizing, as shown in [3].

Let us suppose that S is a strong factorizing language with $\underline{S}\ \underline{C}^* \underline{P} = \underline{A}^*$ and $S \setminus 1$ is a non-prefix code. Let $S = 1 + s_1 + s_2 + s_3$ with $|s_1| \leq |s_2| \leq |s_3|$.

By Lemmas 5 and 7 we have $s_3 = s_i s_j y$ for some $i, j \in \{1, 2\}$ and $y \in RF(S)$. Observe that $(s_i s_j)^* \subseteq S_1 \subseteq RF(S) = C^* P$. Thus for any large enough $k > 0$, if $(s_i s_j)^k \notin C^+$ then $(s_i s_j)^k = cp$ with $c \in C^+, p \in P \setminus 1$. Two cases are possible: either $c = (s_i s_j)^h s_i l$, for some $1 < l < s_j$ (case 1) or $c = (s_i s_j)^h l$, $1 < l < s_i$ (case 2). It can be shown that case 1 is not possible and that case 2 implies that $s_i = s_2$ and $l^{-1} s_i = s_1^d$, d odd. Therefore, for any $i, j \in \{1, 2\}$ then either $\exists k$ s.t. $(s_i s_j)^k \in C^+$ or $s_i = s_2$ and $\exists h$ st. $(s_i s_j)^h l \in C^+$, for $1 < l < s_i$ and $l^{-1} s_i = s_1^d$, d odd.

If $(s_i s_j)^k \in C^+$ then $(s_i s_j)^k y = (s_i s_j)^{k-1} s_3$ would belong both to $C^+ RF(S) \subseteq RF(S)$ and to $S_{-1} \subseteq A^* \setminus RF(S)$.

Otherwise $s_3 = s_2 s_j y$, $y \in RF(S)$. Further, $s_1^d s_j \in RF(S)$, using Corollary 1. Moreover let $(y_1, \cdots, y_n), n$ even, the factorization of the longest prefix of y in $S^* \setminus S_0$, x the longest prefix of $s_1^d s_j y$ in $S^* \setminus S_0$ and $(x_1, \cdots, x_m)$ its factorization. We have that $x_1 = \cdots = x_d = s_1, x_{d+1} = s_j$ and $\forall 1 \leq j \leq n, x_{d+1+j} = y_j$, since $s_1 \not\leq s_2$ and $|s_1^d s_j y| < |s_3|$. Consider then $(s_2 s_j)^h s_3$. We have $(s_2 s_j)^h s_3 = c s_1^d s_j y \in C^+ RF(S) \subseteq RF(S)$ against $(s_2 s_j)^h s_3 \in S_{-1} \subseteq A^+ \setminus RF(S)$. □

6 Constructing C from (S, P)

In this section we consider the problem of constructing finite maximal codes. Observe that any code is a subset of some maximal code. We assume that Schützenberger Conjecture holds, i.e. for any finite maximal code C there exist finite languages S, P s.t. $\underline{S}\ \underline{C}^* \underline{P} = \underline{A}^*$. We focus on the problem of constructing C, once S, P are given, and propose two methods of solution.

The first method uses the definition. From $\underline{S}\ \underline{C}^* \underline{P} = \underline{A}^*$, using properties of formal power series, we obtain that $\underline{C} - 1 = \underline{P}(\underline{A} - 1)\underline{S}$ and $C = supp(\underline{P}(\underline{A} - 1)\underline{S} + 1)$. Starting from an automaton recognizing S and automaton recognizing P, we can easily obtain an automaton recognizing the series $\underline{P}(\underline{A} - 1)\underline{S} + 1$. Let n be the number of its states. Its support (C indeed) is regular because of some results by Schützenberger and Sontag (cfr. [7]). Applying a construction contained in [7] we can obtain an automaton recognizing C with a number of states 2^{n^2}. Remark that $n = \Omega(|Q_S^{min}| + |Q_P^{min}|)$, where Q_X^{min} is the set of states of the minimal automaton for a language X.

The second method we propose for constructing C, given a couple (S, P) of finite languages, is based on the following result.

Proposition 3. *Let S, C, P languages s.t. $\underline{S}\ \underline{C}^* \underline{P} = \underline{A}^*$. Then:*
S is a right-factorizing language and $P \subseteq RF(S)$,
P is a left-factorizing language and $S \subseteq LF(P)$,
$C^* = RF(S) \cap LF(P)$ and $C = C^+ \setminus C^+ C^+$.

Proof. $RF(S) = C^* P$, $LF(P) = SC^*$. Moreover, since $1 \in S$ and $1 \in P$, then $C^* \subseteq RF(S) \cap LF(P)$. Vice versa, let $w \in RF(S) \cap LF(P)$. If $w \notin C^*$ then $w \in (S \setminus 1)C^* \cap C^*(P \setminus 1)$, against the non-ambiguity of SC^*P. Finally $C = C^+ \setminus C^+ C^+$ since C is the basis of C^* (see [6], e.g.). □

For the sake of completeness let us summarize in the next corollary, what Proposition 3 and the characterization of right- and left- factorizing languages of Section 4 yield to languages S, C^*, P. Recall that the values $(r_S, w), (l_S, w)$ are related to the number of prefix- and suffix- sequences of w.

Corollary 3. *Let S, C, P languages s.t. $\underline{S}\ \underline{C}^* \underline{P} = \underline{A}^*$. Then:*
$S \setminus 1 \subseteq \{w \ \ r_S, w) = 0, (l_S, w) = 1\} = (S \setminus 1)C^*$,
$C^* = \{w \ \ r_S, w) = (l_S, w) = 1\}$,
$P \setminus 1 \subseteq \{w \ \ r_S, w) = 1, (l_S, w) = 0\} = C^*(P \setminus 1)$.

Using Proposition 3, an automaton recognizing C can be obtained in the following way. First construct an automaton recognizing C^* as intersection of an automaton recognizing $RF(S)$ and an automaton recognizing $LF(P)$; then obtain an automaton recognizing C using the formula $C = C^+ \backslash C^+C^+$, that is by completion, product and intersection of automata. Therefore the second method strongly depends on the construction we use to recognize $RF(S)$ and $LF(P)$. The rest of the section is devoted to a construction of a deterministic automaton recognizing $RF(S)$, starting from a deterministic automaton recognizing S. The case $LF(P)$ is clearly the symmetric one. We will find that when considering a finite r-factorizing language S, such a construction gives rise to an automaton with a number of states $|Q| \leq 2 \cdot (2|Q_S^{min}|)^{|Q_S|}$. Moreover the completion of an automaton can be obtained by adding 1 state at most to the states of the given automaton; the intersection and the product of two given deterministic automata can be obtained with a number of states equal to the product of the number of states of the two given automata ([23]). Therefore this second method is asymptotically more efficient than the first one.

Let us now construct a deterministic automaton $\mathcal{A}_{f_S}$ that recognizes the right-factor of a regular right-factorizing language S. The construction is based on Theorem 2. Indeed the state of $\mathcal{A}_{f_S}$ reached from the initial state reading a word w, shows the value of f_S on w.

Let $S \subseteq A^*$ a regular language and $\mathcal{A_S}= (Q_S, q_0, \delta_S, F_S)$ a deterministic trim automaton recognizing S. Consider the automaton $(Q_S, q_0, \delta'_S, \{q_0\})$ where $\delta'_S(q,a) = \{\delta_S(q,a)\}$ if $\delta_S(q,a) \notin F_S$ and $\delta'_S(q,a) = \{\delta_S(q,a)\} \cup \{q_0\}$ if $\delta_S(q,a) \in F_S$. Let $\mathcal{A}_{S^*} = (Q_{S^*}, q_0, \delta_{S^*}, \{q_0\})$ be the flower automaton (cf. [6]) of $\mathcal{A_S}$ obtained by removing from $(Q_S, q_0, \delta'_S, \{q_0\})$ every not coaccessible state and all transitions involving it. Let $Q_{S^*} = \{q_0, q_1, \cdots, q_{n-1}\}$. Remark that $\mathcal{A}_{\mathcal{S}^*}$ is no more deterministic, but the number of its transitions is at most twice.

Let us define the function $sign : Z \setminus \{0\} \to \{+, -\}$, as $sign(z) = +$ if $z > 0$ and $sign(z) = -$ if $z < 0$.

The automaton $\mathcal{A}_{f_S} = (Q, 1, \delta, F)$ is the following. A state $q \in Q$ is $q = (l_0, \cdots, l_{n-1}, \sigma)$ with $l_i \in Z \cup \{\infty\}$, $\sigma \in \{+, -\}$; $1 = (1, \infty, \cdots, \infty, +)$ and $F = \{(l_0, \cdots, l_{n-1}, \sigma) | \sigma = +\}$. Further $\delta((l_0, \cdots, l_{n-1}, \sigma), a) = (m_0, \cdots, m_{n-1}, \tau)$ where:

$$m_0 = \begin{cases} -\sum_{\delta_{S^*}(q_j,a)=q_0 \ nd \ _j \neq \infty} l_j & \text{if } \exists j \text{ s.t. } \delta_{S^*}(q_j, a) = q_0 \\ \infty & \text{otherwise} \end{cases}$$

$$m_i = \begin{cases} \sum_{\delta_{S^*}(q_j,a)=q_i \ nd \ _j \neq \infty} l_j & \text{if } \exists j \text{ s.t. } \delta_{S^*}(q_j, a) = q_i \\ \infty & \text{otherwise} \end{cases}$$

$$\tau = \begin{cases} sign(m_0), & \text{if } m_0 \neq 0 \\ \sigma & \text{otherwise.} \end{cases}$$

Remark that the automaton $\mathcal{A}_{f_S}$ is deterministic. Moreover it can have an infinite number of states, since $l_i \in Z \cup \{\infty\}$.

Proposition 4. *Let S a finite right-factorizing language, $\mathcal{A}_{f_S} = (Q, 1, \delta, F)$ constructed as above and $L(\mathcal{A}_{f_S})$ the language recognized by $\mathcal{A}_{f_S}$.*

Then $\mathcal{A}_{f_S}$ is finite and $L(\mathcal{A}_{f_S}) = RF(S)$.

Proof. The proof is based on Theorem 1 and on the following observations that can be proved by induction on the length of w. Let $w \in A^*$ and $q(w) = (l_0, \cdots, l_{n-1}, \sigma)$ the state of $\mathcal{A}_{f_S}$ reached from 1 reading w. We have that:

if $l_i = \infty$ then there is no path in $\mathcal{A}_{S^*}$ from q_0 to q_i labelled w;

if $l_i \neq \infty$ and $i = 0$ then $w \in S^*$, $(f_S, w) = l_0$ and $(r_S, w) > 0$ iff $\sigma = +$;

if $l_i \neq \infty$ and $i \neq 0$ then $w = u_1v_1 = \cdots = u_hv_h$ where for any $j = 1, \cdots, h$, $u_j \in S^*$, $\sum_{j=1,\cdots,h}(f_S, u_j) = l_i$ and there is some path in $\mathcal{A}_S$ from q_0 to q_i labelled v_j and that does not pass through q_0 .

Observe that $\mathcal{A}_{f_S}$ is finite. Indeed if S is factorizing then $(f_S, u_j) \in \{0, 1, -1\}$ for any $u_j \in A^*$. Moreover $l_i \in \{-k, \cdots, 0, \cdots, k, \infty\}$, where:
$|k| = max_{w \in A^*} \sum_{j=1,\cdots,h}(f_S, u_j) \leq max_{w \in A^*}(max_{j=1,\cdots h} v_j) \leq max_{s \in S}|s| - 1.$ □

Remark that $|Q| \leq 2 \cdot (2k+1)^{|Q_S|}$ and $k = O(|Q_S^{min}|)$. The exponential blow-up is due to the deterministic visit of paths in the non-deterministic automaton $\mathcal{A}_{S^*}$. Nevertheless this construction in the case of a finite language S is asymptotically more efficient than the one contained in [1,2]. The number of states of the automaton recognizing $RF(S)$ as constructed in [1,2], is 2^{n^2}, where n is the number of states of an automaton recognizing r_S. Since $n = \Omega(|Q_S^{min}|)$, the number of states of that automaton is $2^{\Omega(|Q_S^{min}|^2)}$. Moreover, the construction presented in this section is simpler: the transitions are obtained by some tests and sums of integers, while in [1,2] they were obtained by some products of matrices $n \times n$ with entries in Z_2.

Example 7. Let $A = \{a, b\}$ and $S = \{1, a, a^2b\}$. Let $\mathcal{A}_S = (Q_S, q_o, \delta_S, F_S)$, where $Q_S = \{q_0, q_1, q_2, q_3\}$, $F_S = \{q_0, q_1, q_3\}$ and the only transitions are $\delta_S(q_0, a) = q_1$, $\delta_S(q_1, a) = q_2$ and $\delta_S(q_2, b) = q_3$. We find $\mathcal{A}_{S^*} = (Q_{S^*}, q_o, \delta_{S^*}, \{q_0\})$, where $Q_S = \{q_0, q_1, q_2\}$, and the only transitions are $\delta_{S^*}(q_0, a) = \{q_0, q_1\}$, $\delta_{S^*}(q_1, a) = q_2$ and $\delta_S(q_2, b) = q_0$.

Using the above construction, $\mathcal{A}_{f_S} = (Q, 1, \delta, F)$ is given as follows. The states are $Q = \{1, 2, 3, 4, 5, 6, 7\}$, where $1 = (1, \infty, \infty, +)$, $2 = (-1, 1, \infty, -)$, $3 = (1, -1, 1, +)$, $4 = (-1, 1, -1, -)$, $5 = (1, \infty, \infty, +)$, $6 = (-1, \infty, \infty, -)$, $7 = (1, -1, \infty, +)$. The final states are $F = \{1, 3, 5, 7\}$. The transitions are as given in Fig. 3.
In Fig. 4 we compare the path in $\mathcal{A}_{f_S}$ from 1 labelled $w = a^4ba$ with the factorizations of w and its prefixes.

7 Conclusions

The problem of constructing codes, is still far away from a solution.

We have singled out two problems to explore, assuming that Schützenberger Conjecture holds: the construction of all the possible factorizations (S, P) of some code C and the construction of a code C starting from its factorization (S, P). Regarding the first problem we have presented the state of art, adding

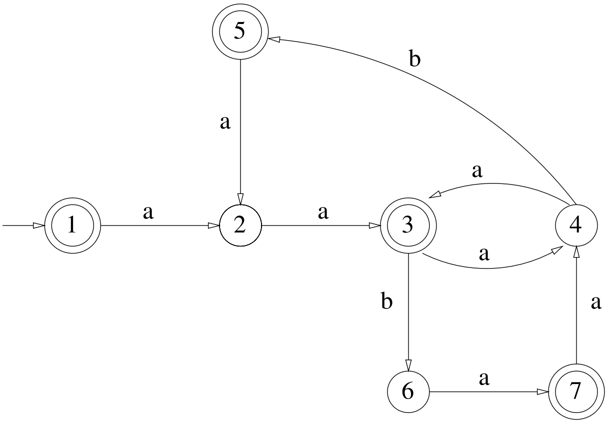


Fig. 3. The automaton $\mathcal{A}_{f_S}$ for $S = \{1, a, a^2b\}$

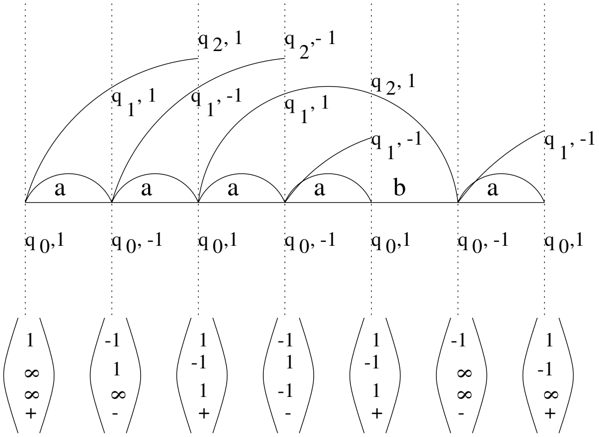


Fig. 4. The path labelled a^4ba in $\mathcal{A}_{f_S}$

some contributions, but a lot of them can be still given. Then we have presented a solution to the second problem. Finally: Schützenberger Factorization Conjecture is indeed still open!

References

1. M. Anselmo, A. Restivo: Factorizing Languages, *Procs. 13th World Computer Congress IFIP 94* **1**, B. Pehrson and I. Simon eds., Elsevier Sc. B. V. North Holland (1994) 445-450 198, 201, 202, 203, 204, 211
2. M. Anselmo, A. Restivo: On Languages Factorizing the free Monoid, *Internat. J. of Algebra and Computation* **6, n.4** (1996) 413-427 200, 201, 202, 203, 204, 211
3. M. Anselmo, C. Defelice, A. Restivo: On some factorization problems, *Bulletin of the Belgian Mathematical Society* **4** (1997) 25-43 199, 200, 201, 208
4. M. Anselmo: A Non-ambiguous Language Factorization Problem *Procs. DLT 99* G. Rozenberg, W. Thomas eds., World Scientific (2000) 141-152 197, 198, 200
5. M. Anselmo. A Non-ambiguous Decomposition of Regular Languages and factorizing Codes, *in revision for a journal* 197, 198, 200
6. J. Berstel, D. Perrin: *Theory of Codes*, Academic Press (1985) 197, 199, 200, 209, 210
7. J. Berstel, Ch. Reutenauer: *Rational Series and their Languages*, EATCS Monographs **12**, Springer Verlag (1988) 199, 209
8. J. M. Boë: Sur les codes factorisants, in: *Théorie des codes- Actes de la 7 école de printemps d'inf. théor.*, D. Perrin ed., LITP (Paris) (1979) 1-8 200
9. J. M. Boë: Sur les codes synchronisants coupants, in: *Non-commutative Structures in Algebra and geometric combinatorics*, A. de Luca, ed., *Quaderni della Ric. Sc. del CNR* **109**, (1981) 7-10 198
10. J. M. Boë: Factorisation par excés du monoïde libre, LIRMM *report* **94-005**, (1994) 198
11. V. Bruyère. Research Topics in the Theory of Codes, *Bull. EATCS* **48** (1992) 412-424 198, 199
12. V. Bruyère, M. Latteux: Variable-length Maximal Codes, *Procs. ICALP 1996* LLNCS Springer-Verlag (1996) 199
13. J. Brzozowsky: *Open Problems about Regular Languages* in *R. V. Book*, ed. Formal Language Theory, Perspectives and Open Problems, Academic Press, London, New York (1980)
14. M. Cariello: *Fattorizzazioni non ambigue di linguaggi formali fattoriali e prefissiali* Tesi di laurea, Univ. di Salerno, anno accademico 1996/97 206
15. C. De Felice: *Construction et complétion de codes finis*, Thèse de 3ème cycle, *Rapport LITP* **85-3**, (1985) 198, 199, 200, 206
16. C. De Felice: Construction of a family of finite maximal codes, *Theor. Comp. Science* **63** (1989) 157-184 198
17. C. De Felice: A partial result about the factorization conjecture for finite variable-length codes, *Discrete Math.*, **122** (1993) 137-152 197, 198, 200
18. C. Defelice: On a property of the factorizing codes, *Int. J. of Algebra and Computation* **9** Nos. 3 & 4 (1999) 325-345 198
19. C. De Felice: Factorizing Codes and Schützenberger Conjectures, in *Proc. MFCS 2000*, Lecture Notes in Comput. Sci. **1893** (2000) 295–303 197, 198, 213
20. C. De Felice: On some Schützenberger Conjectures, *Information and Computation*, to appear (2000), journal version of [19] 198

21. C. De Felice, R. Zizza: Factorizing codes and Krasner factorizations, in *Sixth Italian Conference on Theoretical Computer Science*, World Scientific (1998) 347–358 200
22. S. Eilenberg: *Automata, Languages, and Machines* vol. A, Academic Press, New York (1974)
23. J. E. Hopcroft, J. D. Ullman: *Introduction to Automata Theory, Languages, and Computation* Addison-Wesley New York (1971) 198, 210
24. D. Krob: Codes limités et factorisations finies du monoïde libre, *RAIRO Inform. Théor.* **21** (1987) 437-467
25. W. Kuich, A. Salomaa: *Semirings, Automata, Languages*, EATCS Monographs on Theor. Comp. Sc. **5** Springer Verlag, Berlin, (1986) 199
26. D. Perrin, M. P. Schützenberger: Un problème élémentaire de la théorie de l'information, *"Théorie de l'Information"*, Colloques Internat. CNRS **276**, Cachan (1977) 249-260 198
27. C. Reutenauer: Non commutative factorization of variable-length codes, *J. Pure and Applied Algebra* **36** (1985) 167-186 198
28. A. Salomaa, M. Soittola: *Automata-Theoretic Aspects of Formal Power Series*, Springer, Berlin (1978) 199
29. M. P. Schützenberger: Sur certains sous-monoïdes libres, *Bull. Soc. Math. France* **93** (1965) 209-223 197, 199
30. L. Zang, C. K. Gu: Two classes of factorizing codes - (p,p)-codes and (4,4)-codes, in M.Ito, H. Juergensen, eds., *Words, languages and Combinatorics II*, World Scientific (1994) 477-483 198

An Effective Translation of *Fickle* into Java*

(Extended Abstract)

Davide Ancona[1], Christopher Anderson[2], Ferruccio Damiani[3],
Sophia Drossopoulou[2], Paola Giannini[4], and Elena Zucca[1]

[1] DISI - Università di Genova
[2] Imperial College - London
[3] Dipartimento di Informatica - Università di Torino
[4] DISTA - Università del Piemonte Orientale

Abstract. We present a translation from *Fickle* (a Java-like language allowing dynamic object re-classification, that is, objects that can change their class at run-time) into plain Java. The translation is proved to preserve static and dynamic semantics; moreover, it is shown to be *effective*, in the sense that the translation of a *Fickle* class does not depend on the implementation of used classes, hence can be done in a *separate* way, that is, without having their sources, exactly as it happens for Java compilation. The aim is to demonstrate that an extension of Java supporting dynamic object re-classification could be fully compatible with the existing Java environment.

1 Introduction

Dynamic object re-classification is a feature which allows an object to change its class membership at run-time while retaining its identity. Thus, the object's behavior can change in fundamental ways (*e.g.*, non-empty lists becoming empty, iconified windows being expanded, *etc.*) through re-classification, rather than replacing objects of the old class by objects of the new class. Lack of re-classification primitives has long been recognized as a practical limitation of object-oriented programming. *Fickle* [4] is a Java-like language supporting dynamic object re-classification, aimed at illustrating features for object re-classification which could extend an imperative, typed, class-based, object-oriented language.

Other approaches have been attempted [3,6,7]; however, *Fickle* is more within the main stream of the object oriented approach, and moreover it is type-safe, in the sense that any type correct program (in terms of the *Fickle* type system) is guaranteed never to access non-existing fields or methods.

A further problem is how to construct, starting from the *Fickle* design, a working extension with dynamic object re-classification of a real object-oriented language. Java is the first natural candidate to be considered, since *Fickle* can be

* Partially supported by Murst Cofin'99 - TOSCA Project, CNR-GNSAGA, and the EPSRC (Grant Ref: GR/L 76709).

A. Restivo, S. Ronchi Della Rocca, L. Roversi (Eds.): ICTCS 2001, LNCS 2202, pp. 215–234, 2001.

considered a small subset of Java (with only non-abstract classes, instance fields and methods, integer and boolean types and a minimal set of statements and expressions) enriched with features for dynamic object re-classification. Thus, in particular, a *Fickle* class which does not use these features is a Java class.

In this paper, we provide a first important step towards the solution, that is, we show that a Java environment could be easily and naturally extended in such a way to handle standard Java and *Fickle* classes together.

In order to show that, we define a translation from *Fickle* into plain Java. The translation is proved to preserve static and dynamic semantics (that is, well-formed *Fickle* programs are translated into well-formed Java programs which behave "in the same way"). Moreover, the translation is *effective*, in the sense that it gives the basis for an effective extension of a Java compiler. This is ensured by the fact that the translation of a *Fickle* class does not depend on the implementation of used classes, hence can be done in a *separate* way, that is, without having their sources, exactly as it happens for Java compilation. This is so because type information needed by the translation can be retrieved from type information stored in binary files.

Hence, an extension of Java supporting dynamic object re-classification could be fully compatible with the existing Java environment.

The problems we had to solve in order to define a translation that were both manageable from the theoretical and implementative point of view were not trivial. The main issues we had to face were the following:

1. to find an appropriate encoding for re-classifiable objects;
2. to deal with the fact that a standard Java class c can be extended by a re-classifiable class, possibly after c is translated (*i.e.*, compiled);
3. to make the translation as simple as possible, neglecting efficiency in favor of clearer proofs of correctness;
4. to make the translation effective, in the sense that it truly supports separate compilation as in Java.

Concerning point 1), the basic idea of the translation is to represent each re-classifiable *Fickle* object o through a pair $<w, i>$ of Java objects. Roughly speaking, w is a *wrapper* object providing the (non-mutable) *identity* of o, whereas i is an *implementor* object providing the (mutable) *behavior* of o. A re-classification of o changes i but not w, and method invocations are resolved by i.

To solve problems 2), 3) and 4), even non-re-classifiable objects are represented through such a pair $<o, o>$, where o plays both roles. This greatly simplifies the translation, and allows the same treatment for re-classifiable classes (*i.e.*, *state classes* in *Fickle* terminology), and non-re-classifiable classes.

The work presented in this paper comes out of a collaboration among different research groups and is based on their previous experience in the design and implementation of Java extensions [1,4].

The paper is organized as follows: In Section 2 we introduce *Fickle* informally using an example. In Section 3 we give an informal overview of the translation, while in Section 4 we give the formal description. In Section 5 we state the formal properties of the translation (preservation of static and dynamic semantics) and illustrate the compatibility of the translation with Java separate compilation. In the Conclusion we summarize the relevance of this work and describe further research directions.

A prototype implementation largely based on the translation described in this paper has already been developed [2].

2 *Fickle*: A Brief Presentation

In this section we introduce *Fickle* informally using an example. However, this section is not intended to be a complete presentation of *Fickle*. We refer to [4] for a complete definition of the language.

For readability, in the examples we allow a slightly more liberal syntax than that used in the formal description of the translation (given in Section 4).

The (extended) *Fickle* program in Fig. 1 defines a class `Stack`, with subclasses `EmptyStack` and `NonEmptyStack`. A stack has a capacity (field `int capacity`) that is, the maximum number of integers it can contain, and the usual operators `isEmpty`, `top`, `push`, and `pop`.

In *Fickle* class definitions may be preceded by the keyword `state` or `root` with the following meaning:

- *state classes* are meant to describe the properties of an object while it satisfies some conditions; when it does not satisfy these conditions any more, it must be explicitly re-classified to another state class. For example, `NonEmptyStack` describes non-empty stacks; if these become empty, then they are re-classified to `EmptyStack`.

  We require state classes to extend either root classes or state classes.
- *root classes* abstract over state classes. Objects of a state class `C1` may be re-classified to a class `C2` only if `C2` is a subclass of the uniquely defined root superclass of `C1`. For example, `Stack` abstracts over `EmptyStack` and `NonEmptyStack`; objects of class `EmptyStack` may be re-classified to `NonEmptyStack`, and vice versa.

  We require root classes to extend only non-root and non-state classes.

Objects of a non-state, non-root class `C` behave like regular Java objects, that is, are never re-classified. However, since state classes can be subclasses of non-state, non-root classes, objects bound to a variable `x` of type `C` *may* be re-classified. Namely, if `C` had two state subclasses `C1` and `C2` and `x` referred to an object *o* of class `C1`, then *o* may be re-classified to `C2`.

Objects of an either state or root class `C` are created in the usual way by the expression `new C()`.

```
class StackException extends Exception{
 StackException (String str) {} {super(str);}}
abstract root class Stack{
 int capacity;  // maximum number of elements
 abstract boolean isEmpty()  {};
 abstract int top() {} throws StackException;
 abstract void push(int i) {Stack} throws StackException;
 abstract void pop() {Stack} throws StackException;}
state class EmptyStack extends Stack{
 EmptyStack(int n){} {capacity=n;}
 boolean isEmpty() {} {return true;}
 int top() {} throws StackException {
  throw new StackException("StackUnderflow");}
 void push(int i) {Stack} {
  this!!NonEmptyStack; a=new int[capacity]; t=0; a[0]=i;}
 void pop() {} throws StackException {
  throw new StackException("StackUnderflow");}}
state class NonEmptyStack extends Stack{
 int[] a; // array of elements
 int t;   // index of top element
 NonEmptyStack(int n, int i) {} {capacity=n; a=new int[n]; t=0; a[0]=i;}
 boolean isEmpty() {} {return false;}
 int top() {} {return a[t];}
 void push(int i) {} throw StackException{ t++;
  if (t==capacity) throw new StackException("StackOverflow");
  else a[t]=i; }
 void pop() {Stack} {if (t==0) this!!EmptyStack; else t--;}}
public class StackTest{
 static void main(String[] args) {Stack} throws StackException{
  Stack s=new EmptyStack(100); s.push(3); s.push(5);
  System.out.println(s.isEmpty());
  Stack s1=new NonEmptyStack(100,3); Stack s2=s1; s1.pop();
  System.out.println(s2.isEmpty());}}
```

Fig. 1. Program `StackTest` - stacks with re-classifications

Re-classification statement, `this!!C`, sets the class of `this` to `C`, where `C` must be a state class with the same root class of the static type of `this`. The re-classification operation preserves the types and the values of the fields defined in the root class, removes the other fields, and adds the fields of `C` that are not defined in the root class, initializing them in the usual way. Re-classifications may be caused by re-classification statements, like `this!!NonEmptyStack` in body of method `push` of class `EmptyStack`, or, indirectly, by method calls, like `s.push(3)` in body of `main`. At the start of method `push` of class `EmptyStack` the receiver is an object of class `EmptyStack`, therefore it has the field `capacity`, while it does not have the fields `a` and `t`. After execution of `this!!NonEmptyStack` the receiver is of class NonEmptyStack, the field capacity retains its value while the fields a and t are now available.

Fields, parameters, and values returned by methods (for simplicity, *Fickle* does not have local variables) must have declared types which are not state classes; we call these types *non-state types*. Thus, fields and parameters may denote objects which do change class, but these changes do *not* affect their type. Instead, the type of this *may be a state class* and *may change*.

Annotations like {} and {Stack} before throws clauses and method bodies are called *effects*. Similarly to what happens for exceptions in throws clauses, effects list the root classes of all objects that may be re-classified by execution of that method. Methods annotated by the empty effect {}, like isEmpty, do not cause any re-classification. Methods annotated by non-empty effects, like pop and push by {Stack}, may re-classify objects of (a subclass of) a class in their effect (in the example, of Stack).

A method annotated with effects can be overridden only by methods annotated with the same or less effects[1].

By relying on effects annotations, the type and effect system of *Fickle* ensures that re-classifications will not cause accesses to fields or methods that are not defined for the object.

Note that effects are explicitly declared by the programmer rather then inferred by the compiler. Even though effects inference could be implemented in practice, more flexibility in method overriding can be achieved by allowing the programmer to annotate methods with more effects than those that would be inferred (similarly to what happens for exceptions).

3 An Informal Overview of the Translation

3.1 Encoding *Fickle* Objects

The translation is based on the idea that each object o of a state class c can be encoded in Java by a pair $<w, i>$ of objects; we call w the *wrapper object of* i and i the *implementor object of* w. Roughly speaking, w provides the identity and i the behavior of o, so that any re-classification of o changes i but not w and method invocations are resolved by i.

The class of w is called a *wrapper class* and is obtained by translating the root class of c, whereas the class of i is called an *implementor class* and is obtained by translating c. For any pair $<w, i>$ encoding an object of a state class, the class of i is always a proper subclass of the class of w.

An object o which is not an instance of a state class does not need to be encoded in principle; however, the same kind of encoding proposed above can be adopted also in this case, since o can always be encoded by the pair $<o, o>$, where both the wrapper and the implementor are the object o itself (in other words, if c is not a state class, then it may seen as wrapper class of itself). Even

[1] This means that adding a new effect in a method of a class c does not require any change to the subclasses of c, but may require some changes to its superclasses.

though at first sight this may seem inefficient and unnecessary, it allows for a simpler and more effective translation, as explained in the sequel.

The translation of classes follows the following two rules:

- each *Fickle* class is translated into exactly one Java class (including `Object`);
- the translation preserves the inheritance hierarchy.

Throughout the paper we adopt the following terminology:

- the translation of a non-state, non-root class is called a *non-implementor, non-wrapper class*;
- the translation of a root class is called a *wrapper class*;
- the translation of a state class is called an *implementor class.*

We illustrate the above in terms of the example in Fig.1. After the instruction

```
s=new NonEmptyStack(100,3);
```

where `s` has static type `Stack`, the object stored in `s` is encoded in the translation as sketched in Fig.2.

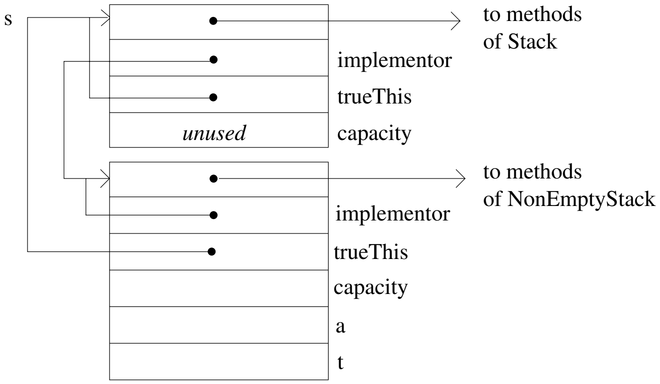


Fig. 2. Encoding of the object stored in `s`

The variable `s` contains an object o of dynamic type `Stack` with three fields: `capacity` is declared in `Stack`, whereas `implementor` and `trueThis` are inherited from class `FickleObject`, have type `FickleObject` and are used in the translation for recovering the implementor and the wrapper of a re-classifiable object, respectively. In this case the field `implementor` points to an object of the implementor class obtained by translating `NonEmptyStack`, whereas `trueThis` points to the object itself. Note that here the field `capacity` is redundant, since its actual value is stored in `implementor.capacity`.

The implementor object contains all fields declared in `NonEmptyStack` (`a` and `t`), and also the field `capacity`, since the implementor class `NonEmptyStack` is a subclass of the wrapper class `Stack`. The field `implementor` points to itself, even though is never used. The field `trueThis` is inherited from class `FickleObject`, has type `FickleObject` and is used to recover the wrapper object of the implementor, which is essential for correctly handling re-classification of `this`.

3.2 Translation of Classes

In this section we introduce some examples in order to explain how classes and expressions are translated.

Example 1. Consider the following class declaration in (extended) *Fickle*:

```
class C{
 int x;
 int m1(){}{m2(); return m2();}
 int m2(){R}{x=0; return x;}
}
```

Our translation maps the declaration of `C` in the following Java class[2]

```
class C extends FickleObject{
 int x;
 int m1(){
  ((C) trueThis.implementor).m2();
  return ((C) trueThis.implementor).m2();}
 int m2(){
  ((C) trueThis.implementor).x=0;
  return ((C) trueThis.implementor).x;}
 C(){}
 C(FickleObject oldImp){
  super(oldImp);
  x=((C) oldImp).x;}
}
```

`FickleObject` is the common ancestor of the Java classes obtained by translating *Fickle* classes, and, in fact, corresponds to the translation of the *Fickle* predefined class `Object`:

```
class FickleObject extends Object{
 FickleObject implementor;
 FickleObject trueThis;
 FickleObject(){
  implementor=this;
  trueThis=this;}
```

[2] The translation examples in this paper do not completely agree with the formal definition given in Sect.4, since some optimization has been performed in order to keep the code simpler.

```
 FickleObject(FickleObject oldImp){ // re-classifies objects
  implementor=this;
  trueThis=oldImp.trueThis;
  trueThis.implementor=this;}
}
```

The fields implementor and trueThis are declared in this top level class for correctly dealing with the encoding of objects which are not instances of state classes, as already explained in 3.1; constructor FickleObject() initializes fields implementor and trueThis to the new instance o so that its encoding is $<o, o>$. This constructor is invoked whenever either a new instance of a non-state class or a new implementor of a state class is created.

On the other hand, constructor FickleObject(FickleObject oldImp) is invoked whenever an object is re-classified and is placed in FickleObject just for avoiding code duplication. An object o which needs to be re-classified to a state class C (recall that in the translation every class is subclass of FickleObject) and which is encoded by the pair $<w, i>$, is transformed into $<w, i'>$, where i' denotes the new implementor of class C (provided by a proper constructor of C; see Example 3 below). The argument of the constructor denotes the old implementor i, from which the wrapper w can be recovered as well (recall that $w.$implementor $= i.$trueThis must hold), whereas i' is denoted by this. Fields are initialized so that wrapper w and the new implementor i' point to each other. The assignment implementor=this could be omitted, since in principle field implementor of implementors should never be used.

Two interesting parts of C translation concern invocations of method m2 in m1 and access of field x in m2.

Method m2 must be invoked on implementor because it could be overridden by some state subclass of C, whereas this must be translated in trueThis because method m2 could be inherited by some subclass of C (hence, this could contain a possibly wrong implementor rather than a wrapper). Downcasting is needed since implementor has type FickleObject.

The same explanations apply also for selection of field x.

Constructor C(FickleObject oldImp) invokes the corresponding constructor in class FickleObject which is used for re-classifying objects, as already explained. However, during re-classification all fields of the new implementor i' which are inherited from non-state classes (like x in the example) must be initialized with the values of the corresponding fields of the old implementor i (x=((C) oldImp).x).

Finally, note that the translation of C is totally independent of any possible existing subclass or client class of C; this property, which is satisfied by our translation for any kind of class, is crucial for obtaining a translation which truly reflects Java separate compilation (see also the related comments in Example 3).

Example 2. Assume now to add to the declaration of Example 1 the following class declaration:

```
root class R extends C{
}
```

This *Fickle* class declaration is translated in the following Java class declaration:

```
class R extends C{
 R(){}
 R(FickleObject oldImp){super(oldImp);}
 R(R imp){
  trueThis=this;
  implementor=imp;
  imp.trueThis=this;}
}
```

In the translation, root classes declare three constructors.

Constructor `R()` is used for creating instances of `R` and simply invokes the corresponding constructor of the direct superclass `C`.

Constructor `R(FickleObject oldImp)` is used for re-classifying objects and simply invokes the corresponding constructor of the direct superclass `C`, since in this case `R` does not declare any field.

Constructor `R(R imp)` is used by state subclasses of `R` for creating new instances. The argument represents the implementor of the object which has been properly created by the constructor of a state subclass of `R`, while the wrapper object is created by the constructor itself. Fields are initialized so that wrapper and implementor point to each other. The assignment `trueThis=this` could be omitted, since field `trueThis` of wrappers will never be used.

Example 3. Consider now the following state classes:

```
state class S1 extends R{
 int m2(){R}{this!!S2;x=1;return x;}
 static void main(String[] args)
  {System.out.println(new S1().m1());}}
state class S2 extends R{
 int y;
 int m2(){R}{y=1;return x+y;}
}
```

They are translated in Java as follows:

```
class S1 extends R{
 int m2(){
  new S2(trueThis.implementor);
  ((S2) trueThis.implementor).x=1;
  return ((S2) trueThis.implementor).x;}
 static void main(String[] args){
  System.out.println(
   ((S1) new R(new S1()).implementor).m1());}
 S1(){}
 S1(FickleObject oldImp){super(oldImp);}
}
```

```
class S2 extends R{
 int y;
 int m2(){
  ((S2) trueThis.implementor).y=1;
  return ((S2) trueThis.implementor).x+
         ((S2) trueThis.implementor).y;}
 S2(){}
 S2(FickleObject oldImp){super(oldImp);}
}
```

In the translation, state classes declare two constructors.

In class `S2`, for instance, constructor `S2()` is used for creating the implementor component of a new instance of `S2`, while constructor `S2(FickleObject oldImp)` is used for re-classifying objects; note that, differently to what happens for non-state classes, no extra-code is added in the body for any field declared in the class (like `y`).

Let us now focus on the translation of object re-classification `this!!S2` (in the body of method `m2` of class `S1`) and on instance creation of class `S1` (in the body of method `main` of class `S1`).

As already explained, for re-classifying an object to class `S2`, the proper constructor of `S2` must be invoked, passing as parameter the current (and soon obsolete) implementor i, denoted by `trueThis.implementor`; then, the constructor creates a new implementor i' (belonging to `S2`), initializes and updates fields so that the wrapper w and the new implementor i' point to each other (recall that the wrapper can be recovered from the old implementor i) and properly initializes all fields inherited from non-state superclasses (like `x`). This last step is performed by invoking all the corresponding constructors of superclasses up to `FickleObject`.

Creation of an instance of `S1` is achieved by invoking the proper constructor of the root class `R` of `S1`; a new implementor, created by invoking the default constructor of `S1`, is passed as parameter to the constructor.

We now consider issues related to the effectiveness of the translation. As already pointed out in Example 1, the translation of a *Fickle* class `C` does not depend on any possible subclass or client of `C`, as happens for Java separate compilation. On the other hand, the translation of class `S1`, for instance, depends on classes `R` and `S2` inherited and used, respectively, by `S1`; for instance, all type casts in the body of `S1` are determined by type-checking `S1` and this process requires to retrieve type information about classes `R` and `S2` (that is, the signature of methods and the inheritance hierarchy). However, the translation of `S1` is clearly independent of the specific bodies of methods of `R` and `S2`.

As a consequence, dependencies computed by our translation process are exactly the same as those computed by the Java compiler. Furthermore, the translation of classes depends only on the inheritance hierarchy and on method signatures; therefore a class c depending on classes $c_1, \ldots, c_n$ could be successfully translated in a context where only the binary files of $c_1, \ldots, c_n$ are available, as happens for Java.

$p ::= \mathit{class}^*$
$\mathit{class} ::= [\texttt{root} \mid \texttt{state}]\ \texttt{class}\ c\ \texttt{extends}\ c'\{\mathit{field}^*\ \mathit{meth}^*\}$
$\mathit{field} ::= t\ f$
$\mathit{meth} ::= t\ m(t'\ x)\phi\{\mathit{sl}\ \texttt{return}\ e;\}$
$t ::= \texttt{boolean} \mid \texttt{int} \mid c$
$\phi ::= \{c^*\}$
$\mathit{sl} ::= s^*$
$s ::= \{\mathit{sl}\} \mid \texttt{if}\ (e)\ s_1\ \texttt{else}\ s_2 \mid \mathit{se}; \mid \texttt{this!!}c;$
$\mathit{se} ::= \mathit{var} = e \mid e_1.m(e_2) \mid \texttt{new}\ c()$
$e ::= \mathit{sval} \mid \mathit{var} \mid \texttt{this} \mid \mathit{se}$
$\mathit{var} ::= \texttt{x} \mid e.f$
$\mathit{sval} ::= \texttt{true} \mid \texttt{false} \mid \texttt{null} \mid n$

Fig. 3. Syntax of *Fickle*

4 Formal Description of the Translation

In this section we give a formal description of the translation. The syntax of the source language is specified in Fig.3. We refer to [4] for the definition of the static semantics of *Fickle* (the type system of *Fickle* can be easily adapted to the subset of Java serving as target for the translation) and of some auxiliary functions used in the sequel.

4.1 Programs

The translation of a *Fickle* program p consists of the translation of all classes declared in p. The classes are translated w.r.t the program p, needed because the translation of expressions depends on their types (in particular, for method invocation and field selection) and on the names of root classes (in particular, constructor invocation and `this`).

$$[\![p]\!]_{\mathit{prog}} \overset{\Delta}{=} [\![\mathit{class}_1]\!]_{\mathit{class}}(p) \ldots [\![\mathit{class}_n]\!]_{\mathit{class}}(p) \qquad \text{where} \ \ p = \mathit{class}_1 \ldots \mathit{class}_n.$$

4.2 Classes

As already explained, each *Fickle* class c is translated into a single Java class containing the translation of all field and method declarations of c and a number of constructors, used for creating instances and for re-classifying objects.

The translation of fields and methods is independent of the kind of class.

However, translation of non-state non-root classes, root classes and state classes leads to the declaration of different constructors. That is why for each kind of class we give a different translation clause.

Class `Object`*:* This class is translated in `FickleObject` which is the common superclass of all translated classes, already defined in Sect.3.2.

Non-state, non-root classes: These classes are translated by translating all their methods, and by adding two constructors: $c()$ is used for the creation of new instances of c and $c(\texttt{FickleObject oldImp})$ is used for the creation of new implementors when objects of subclasses are re-classified. In this last case all fields of the old implementor `oldImp` which are declared in class c must be copied into the corresponding new implementor created by the constructor (see Example 1 in Sect.3.2). The additional parameter c for the translation of methods is needed to determine the class of `this` inside the bodies.

$$
\begin{array}{ll}
\multicolumn{2}{l}{[\![\texttt{class}\ c\ \texttt{extends}\ c'\{t_1\ f_1; \ldots t_m\ f_m;\ \mathit{meth}_1 \ldots \mathit{meth}_n\}]\!]_{\mathit{class}}(p) \overset{\Delta}{=}} \\
\texttt{class}\ c\ \texttt{extends}\ \mathit{name}(c')\{ & [\![t_1\ f_1;]\!]_{\mathit{field}}(c) \ldots [\![t_m\ f_m;]\!]_{\mathit{field}}(c) \\
 & [\![\mathit{meth}_1]\!]_{\mathit{meth}}(p, c) \ldots [\![\mathit{meth}_n]\!]_{\mathit{meth}}(p, c) \\
 & c()\{\} \\
 & c(c\ \texttt{oldImp})\{ \\
 & \quad \texttt{super}(\texttt{oldImp}); \\
 & \quad f_1 = \texttt{oldImp}.f_1; \\
 & \quad \ldots \\
 & \quad f_m = \texttt{oldImp}.f_m; \} \\
 & \}
\end{array}
$$

The auxiliary function *name* is defined as follows:

$$
\mathit{name}(c) = \begin{cases} \texttt{FickleObject} & \text{if } c = \texttt{Object} \\ c & \text{otherwise} \end{cases}
$$

Root classes: The translation of this kind of classes produces three constructors: $c()$ creates instances of c, $c(\texttt{FickleObject oldImp})$ deals with object re-classification, and $c(c\ \texttt{imp})$ creates wrappers of instances of state classes:

$$
\begin{array}{ll}
\multicolumn{2}{l}{[\![\texttt{root class}\ c\ \texttt{extends}\ c'\{t_1\ f_1; \ldots t_m\ f_m;\ \mathit{meth}_1 \ldots \mathit{meth}_n\}]\!]_{\mathit{class}}(p) \overset{\Delta}{=}} \\
\texttt{class}\ c\ \texttt{extends}\ \mathit{name}(c')\{ & [\![t_1\ f_1;]\!]_{\mathit{field}}(c) \ldots\ [\![t_m\ f_m;]\!]_{\mathit{field}}(c) \\
 & [\![\mathit{meth}_1]\!]_{\mathit{meth}}(p, c) \ldots [\![\mathit{meth}_n]\!]_{\mathit{meth}}(p, c) \\
 & c()\{\} \\
 & c(c\ \texttt{oldImp})\{ \\
 & \quad \texttt{super}(\texttt{oldImp}); \\
 & \quad f_1 = \texttt{oldImp}.f_1; \\
 & \quad \ldots \\
 & \quad f_m = \texttt{oldImp}.f_m; \} \\
 & c(c\ \texttt{imp})\{ \\
 & \quad \texttt{trueThis} = \texttt{this}; \\
 & \quad \texttt{implementor} = \texttt{imp}; \\
 & \quad \texttt{imp.trueThis} = \texttt{this}; \} \\
 & \}
\end{array}
$$

State classes: The translation of this kind of classes produces two constructors: the former (with no arguments) for creating new implementors for new instances of class c, the latter for dealing with object re-classification to c:

$$
\begin{array}{l}
[\![\texttt{state class}\ c\ \texttt{extends}\ c'\{\mathit{field}_1\ldots\mathit{field}_m\ \ \mathit{meth}_1\ldots\mathit{meth}_n\}]\!]_{\mathit{class}}(p) \triangleq \\
\texttt{class}\ c\ \texttt{extends}\ \mathit{name}(c')\{\ [\![\mathit{field}_1]\!]_{\mathit{field}}(c)\ldots[\![\mathit{field}_m]\!]_{\mathit{field}}(c) \\
\qquad\qquad\qquad\qquad\qquad\quad [\![\mathit{meth}_1]\!]_{\mathit{meth}}(p,c)\ldots[\![\mathit{meth}_n]\!]_{\mathit{meth}}(p,c) \\
\qquad\qquad\qquad\qquad\qquad\quad c()\{\} \\
\qquad\qquad\qquad\qquad\qquad\quad c(\texttt{FickleObject oldImp})\{\texttt{super(oldImp)}\} \\
\qquad\qquad\qquad\qquad\qquad\quad \}
\end{array}
$$

Note that here $\mathit{name}(c') = c'$, since a state class cannot extend class `Object`.

4.3 Fields

Translation of each field f comes equipped with a static method `to`f used for translating assigments of value v to field f of object `tT` (see the paragraph on expressions translation below), since the implementor of the object `tT` can be correctly selected only after evaluating v.

$$
\begin{array}{l}
[\![t\ f;]\!]_{\mathit{field}}(c) \triangleq \\
\ t\ f; \\
\ \texttt{static}\ t\ \texttt{to}f(\texttt{FickleObject tT}, t\ v)\{\texttt{return}\ ((c)\ \texttt{tT.implementor}) = v;\}
\end{array}
$$

4.4 Methods

Translating methods consists of translating their bodies. Effects are omitted, whereas the signatures remain the same. Since the translation of statements and expressions depends on their types, the program p and the environment γ must be passed as parameters to the corresponding translation functions.

Note that the environment γ' used for translating the returned expression e may be different from γ, since execution of sl could re-classify `this`. Furthermore, translation of each method m comes equipped with a static method `call`m used for translating invocations of m on receiver `tT` and with argument x (see the paragraph on expressions translation below); indeed, the implementor of `tT` can be correctly selected only after evaluating the argument x.

The judgment $p,\gamma \vdash sl : \texttt{void} \parallel c' \parallel \phi'$ is valid (see [4] for the typing rules) whenever sl has type `void` w.r.t. program p and environment γ; c' denotes the type of `this` after evaluating sl, whereas ϕ conservatively estimates the re-classification effect of the evaluation of sl on objects (this last information is never used by our translation). The environment γ defines the type of the parameters and of `this`.

$$
\begin{array}{l}
[\![t\ m(t'\ \texttt{x})\phi\{sl\ \texttt{return}\ e;\}]\!]_{\mathit{meth}}(p,c) \triangleq \\
\ t\ m(t'\ \texttt{x})\{[\![sl]\!]_{\mathit{stmts}}(p,\gamma)\ \texttt{return}\ [\![e]\!]_{\mathit{expr}}(p,\gamma');\} \\
\ \texttt{static}\ t\ \texttt{call}m(\texttt{FickleObject tT}, t'\ \texttt{x})\{ \\
\quad \texttt{return}\ ((c)\ \texttt{tT.implementor}).m(\texttt{x});\}
\end{array}
$$

$$
\text{where } \gamma = t'\ \texttt{x}, c\ \texttt{this},\quad \gamma' = t'\ \texttt{x}, c'\ \texttt{this},\quad \text{and}\ \ p,\gamma \vdash sl : \texttt{void} \parallel c' \parallel \phi'
$$

4.5 Statements

Except for object re-classification, all statements are translated by translating their constituent statements or subexpressions. The notation $\gamma[c\ \texttt{this}]$ denotes the environment obtained by updating γ so that it maps `this` to c.

$$\llbracket s\ sl \rrbracket_{stmts}(p,\gamma) \triangleq \llbracket s \rrbracket_{stmt}(p,\gamma)\ \llbracket sl \rrbracket_{stmts}(p,\gamma')$$
$$\text{where } p,\gamma \vdash s : \texttt{void} \parallel c \parallel \phi \text{ and } \gamma' = \gamma[c\ \texttt{this}]$$

$$\llbracket \{sl\} \rrbracket_{stmt}(p,\gamma) \triangleq \{\llbracket sl \rrbracket_{stmts}(p,\gamma)\}$$

$$\llbracket \texttt{if}\ (e)\ s_1\ \texttt{else}\ s_2 \rrbracket_{stmt}(p,\gamma) \triangleq$$
$$\texttt{if}\ (\llbracket e \rrbracket_{expr}(p,\gamma))\ \llbracket s_1 \rrbracket_{stmt}(p,\gamma')\ \texttt{else}\ \llbracket s_2 \rrbracket_{stmt}(p,\gamma')$$
$$\text{where}\quad p,\gamma \vdash e : \texttt{boolean} \parallel c_1 \parallel \phi_1, \quad \gamma' = \gamma[c_1\ \texttt{this}]$$

$$\llbracket se; \rrbracket_{stmt}(p,\gamma) \triangleq \llbracket se \rrbracket_{expr}(p,\gamma);$$

The translation of re-classification to class c consists of the call to the appropriate constructor of class c. The current implementor (`trueThis.implementor`) is passed as parameter to the constructor in order to correctly initialize the fields of the new implementor.

$$\llbracket \texttt{this}!!c; \rrbracket_{stmt}(p,\gamma) \triangleq \texttt{new c(trueThis.implementor);}$$

4.6 Expressions

Types of expressions are preserved under the translation, up to state classes: more precisely, if a *Fickle* expression e has type t and t is not a state class, then its type is preserved; otherwise, the type of the translation of e is the root superclass of t. This is formalized and proven in Sect.5.

Simple cases: Values, variables and variables assignment: The translation is straightforward.

$$\llbracket sval \rrbracket_{expr}(p,\gamma) \triangleq sval$$
$$\llbracket x \rrbracket_{expr}(p,\gamma) \triangleq x$$
$$\llbracket \texttt{x} = e \rrbracket_{expr}(p,\gamma) \triangleq \texttt{x} = \llbracket e \rrbracket_{expr}(p,\gamma)$$

Field selection: as already explained in Sect.3.1, in the encoding $<w,i>$ of an object o of class c, the fields of o are stored in the implementor object i (belonging to the class obtained by translating c). Therefore, fields can be accessed only through w.`implementor` on object[3] w. Downcasting is needed because field `implementor` has type `FickleObject`.

$$\llbracket e.f \rrbracket_{expr}(p,\gamma) \triangleq ((c)\ \llbracket e \rrbracket_{expr}(p,\gamma).\texttt{implementor}).f$$
$$\text{where } p,\gamma \vdash e : c \parallel c' \parallel \phi$$

Field assignment: Field f of the wrapper object w denoted by the translation of e_1 is accessed through the implementor of w; however, e_2 could re-classify w, therefore selection w.`implementor` is correct only after evaluating the translation of e_2. This is achieved by invoking the auxiliary static method `to`f.

$$\llbracket e_1.f = e_2 \rrbracket_{expr}(p,\gamma) \triangleq c.\texttt{to}f(\llbracket e_1 \rrbracket_{expr}(p,\gamma), \llbracket e_2 \rrbracket_{expr}(p,\gamma'))$$
$$\text{where } p,\gamma \vdash e_1 : c \parallel c' \parallel \phi, \text{ and } \gamma' = \gamma[c'\ \texttt{this}]$$

[3] Note that this is necessary only when c is a state class, while in the other cases selection could be performed directly on the object o itself, since $w = i = o$ holds. However, to keep the mapping simpler, we do not make this distinction.

Method invocation: The same considerations as for field assignment apply in this case: method call is performed by calling the auxiliary static method `call`m, so that `implementor` field of the receiver is selected only after evaluating the translation of e_2.

$$[\![e_1.m(e_2)]\!]_{\mathit{expr}}(p,\gamma) \overset{\Delta}{=} c.\mathtt{call}m([\![e_1]\!]_{\mathit{expr}}(p,\gamma), [\![e_2]\!]_{\mathit{expr}}(p,\gamma'))$$
$$\text{where } p,\gamma \vdash e_1 : c \parallel c' \parallel \phi, \text{ and } \gamma' = \gamma[c' \ \mathtt{this}]$$

Object creation: Creation of instances of a non-state class c only requires invocation of the default constructor of c. If c is a state class, then two objects must be created: the implementor i (created by invoking the default constructor of c), and the wrapper w (created by invoking the proper constructor of class $\mathcal{R}(p,c)$, that is, the wrapper class of c). The implementor is passed as parameter to the constructor of the wrapper so that fields of w and i can be properly initialized to satisfy the equations $w.\mathtt{implementor} = i$ and $i.\mathtt{trueThis} = w$. The term $\mathcal{R}(p,c)$ denotes the least superclass of c which is not a state class: If c is a state class, then $\mathcal{R}(p,c)$ is its unique root superclass, otherwise $\mathcal{R}(p,c) = c$.

$$[\![\mathtt{new}\ c()]\!]_{\mathit{expr}}(p,\gamma) \overset{\Delta}{=} \begin{cases} \mathtt{new}\ \mathcal{R}(p,c)(\mathtt{new}\ c()) & \text{if } p \vdash c \diamond_s \\ \mathtt{new}\ c() & \text{otherwise} \end{cases}$$

This: The expression `this` is translated into `trueThis` because `this` could denote the implementor object i, rather than the wrapper w. Furthermore, the actual implementor of w may have changed because of re-classification, therefore `this` may denote an obsolete implementor. Because `trueThis` has static type `FickleObject`, in order to preserve types, the translation also needs to downcast to the root superclass of the type of `this`[4]. Note that since a state class c cannot be used as a type, the translation is statically correct also when `this` is passed as a parameter or assigned to a field.

$$[\![\mathtt{this}]\!]_{\mathit{expr}}(p,\gamma) \overset{\Delta}{=} (\mathcal{R}(p,\gamma(\mathtt{this})))\ \mathtt{trueThis}$$

5 Properties of the Translation

In this section we formalize the properties of the translation previously mentioned. For lack of space we only sketch some proofs which will be detailed in a future extended version of this paper.

Preservation of Static Correctness

Theorem 1. *For any $\mathcal{F}$ickle program p, if p is well-typed (in $\mathcal{F}$ickle), then $[\![p]\!]_{\mathit{prog}}$ is well-typed (in Java).*

[4] Note that this downcasting is only necessary when `this` is used for parameter passing or assignments, and is unnecessary when `this` is used in method calls or field selection. This is so because in the latter cases field `implementor` of the object denoted by `trueThis` must be selected and `implementor` is declared in the type of `trueThis`. But, as already stated, we do not consider such optimization issues.

In order to be proved, the claim of the theorem must be extended to all subterms of p and, hence, to all typing judgments. The strengthened claim can be proved by induction on the typing rules. The claim concerning judgment for expressions is the most interesting, hence is stated below.

The translation preserves types up to state classes, in the following sense: if a $\mathcal{F}ickle$ expression e has type t w.r.t. a program p and an environment γ, and e is translated into a Java expression e' that has type t' w.r.t. $[\![p]\!]$ and γ, then $t = t'$, when t is not a state class, and t' is the root superclass of t, when t is a state class. For the Java fragment obtained from the translation we can use the $\mathcal{F}ickle$ type system, so that for any well-typed Java expression e we can derive judgments of the form $p, \gamma \vdash e \;:\; t \parallel \gamma(\texttt{this}) \parallel \emptyset$, where t is the type of e. The fact that the type of `this` remains the same, and the set of effects is empty indicates that e contains no re-classifications.

The claim for expressions can be formalized as follows:

Lemma 1. *For any $\mathcal{F}ickle$ expression e, program p, environment γ, if*

- $p, \gamma \vdash e \;:\; t \parallel c \parallel \phi$, *and*
- $[\![e]\!]_{expr}(p, \gamma) = e'$, *and*
- $[\![p]\!]_{prog} = p'$,

then

- $p', \gamma \vdash e' \;:\; \mathcal{R}(p, t) \parallel \gamma(\texttt{this}) \parallel \emptyset$.

Preservation of Dynamic Semantics We now show that the semantics of expressions is preserved by the translation. The semantics of the language $\mathcal{F}ickle$ we consider is the one introduced in [4]. Such semantics rewrites pairs of expressions and stores into pairs of values (or the exception `nullPntrExc`, indicating a reference to a null object), and stores. *Values*, denoted by v, are either booleans, or integers, or addresses, denoted by ι. Stores map the unique parameter[5] `x` and the receiver `this` to values and addresses to objects. *Objects* are mappings between fields and values tagged by the class they belong to: $[\![\mathtt{f}_1 : \mathsf{v}_1, \ldots, \mathtt{f}_r : \mathsf{v}_r]\!]^{c}$. We use o as a metavariable for objects, and if `f` is a field of o, $o(\mathtt{f})$ is the value associated to `f` in o.

The rewriting, defined in the context of a given program p that provides the definition for the classes used in the expression, is defined by the judgment $e, \sigma \underset{p}{\leadsto} \mathsf{v}, \sigma'$. The syntax of $\mathcal{F}ickle$ and the one of the Java fragment considered here are slightly different from the language of [4]. In particular there is a distinction between statements and expressions and classes have constructors. However, the definition of the semantics in [4] can be easily adapted to deal with these features. Note that the Java fragment contains also casting. However, we do not need rules for casting, since well-typing will insure that casting is applied to objects that already have the target type.

[5] Recall that, for simplicity, we assume that in $\mathcal{F}ickle$ syntax each method definition has a unique parameter denoted by `x`.

To state the semantic correctness result we introduce a relation between stores $p \vdash \sigma \approx \sigma'$ that expresses the fact that store σ' is the "translation" of store σ. That is, an object o of class c in σ corresponds univocally to an object o' in σ' that is an instance of the translation of the class c. Both the store σ and the store σ' are assumed to agree with the relative environments and programs. That is, they contain values which agree, w.r.t. typing, with their definitions (see [4] for the formal definition of $p, \gamma \vdash \sigma \diamond$).

Definition 1. *Let* $p, \gamma \vdash \sigma \diamond$ *and* $[\![p]\!], \gamma \vdash \sigma' \diamond$. *We say that* v' *in* σ' corresponds to v *in* σ *w.r.t.* p, *and write* $p, \sigma, \sigma' \vdash \mathsf{v} \approx \mathsf{v}'$, *if either of the following conditions hold:*

- $\mathsf{v} = \mathsf{v}' = \mathtt{true}$, *or* $\mathsf{v} = \mathsf{v}' = \mathtt{false}$, *or* $\mathsf{v} = \mathsf{v}' = n$ *(for some integer* n*), or* $\mathsf{v} = \mathsf{v}' = \mathtt{null}$, *or*
- $\mathsf{v} = \iota$, $\mathsf{v}' = \iota'$, $\sigma(\iota) = [\![\mathtt{f}_1 : \mathsf{v}_1, \ldots, \mathtt{f}_r : \mathsf{v}_r]\!]^{c}$,
$$\sigma'(\iota') = [\![\mathtt{f}_1 : \mathsf{v}'_1, \ldots, \mathtt{f}_q : \mathsf{v}'_q, \mathtt{impl} : \iota'', \mathtt{trueThis} : \iota']\!]^{\mathcal{R}(p,c)},$$
($q \leq r$*) and*
$$\sigma'(\iota'') = [\![\mathtt{f}_1 : \mathsf{v}''_1, \ldots, \mathtt{f}_r : \mathsf{v}''_r, \mathtt{impl} : \iota'', \mathtt{trueThis} : \iota']\!]^{c},$$
and
for all i, $1 \leq i \leq r$, $p, \sigma, \sigma' \vdash \mathsf{v}_i \approx \mathsf{v}''_i$, *and*
if c *is not a state class, then* $\iota' = \iota''$.

Note that if c is not a state class, then $\mathcal{R}(p, c) = c$, and so $q = r$. With this notion of correspondence between values we can define a correspondence between stores.

Definition 2. *Let* $p, \gamma \vdash \sigma \diamond$ *and* $[\![p]\!], \gamma \vdash \sigma' \diamond$. *We say that store* σ' corresponds to σ *w.r.t.* p, *and write* $p \vdash \sigma \approx \sigma'$, *if*

1. $p, \sigma, \sigma' \vdash \sigma(\mathtt{x}) \approx \sigma'(\mathtt{x})$,
2. $p, \sigma, \sigma' \vdash \sigma(\mathtt{this}) \approx (\sigma'(\mathtt{this}))(\mathtt{trueThis})$, *and*
3. *for all* ι *if* $\sigma(\iota)$ *is defined there is a unique* ι' *such that* $p, \sigma, \sigma' \vdash \iota \approx \iota'$, *and*
4. *for all* ι' *if* $\sigma'(\iota')$ *is defined there is a unique* ι *such that* $p, \sigma, \sigma' \vdash \iota \approx (\sigma'(\iota'))(\mathtt{trueThis})$.

The last two conditions of the previous definition assert that there is an injection between the set of addresses defined in σ and the set of addresses defined in σ'.

Theorem 2. *For a well-typed expression* e, *stores* σ_0 *and* σ_1 *such that* $p, \gamma \vdash \sigma_0 \diamond$, $[\![p]\!], \gamma \vdash \sigma_1 \diamond$ *and* $p \vdash \sigma_0 \approx \sigma_1$,

$$e, \sigma_0 \underset{p}{\rightsquigarrow} \mathsf{v}, \sigma'_0 \qquad \textit{if and only if} \qquad [\![e]\!], \sigma_1 \underset{[\![p]\!]}{\rightsquigarrow} \mathsf{v}', \sigma'_1$$

where $p \vdash \sigma'_0 \approx \sigma'_1$ *and* $p, \sigma, \sigma' \vdash \mathsf{v} \approx \mathsf{v}'$

The proof is by induction on the derivation of $e, \sigma \underset{p}{\rightsquigarrow} \mathsf{v}, \sigma'$. The proof that, in case of field selection and method call, the right method is selected relies on the following fact. If $p \vdash \sigma \approx \sigma'$, then: for all ι and c, $\sigma(\iota) = [\![\cdots]\!]^{c}$ implies $\sigma'(\sigma'(\iota')(\mathtt{impl})) = [\![\cdots]\!]^{c}$, where $p, \sigma, \sigma' \vdash \iota \approx \iota'$.

Support for Separate Compilation For any *Fickle* program p, let $\mathit{classes}(p)$ denote the set of all classes defined in p, and, for each class c in $\mathit{classes}(p)$, $\mathit{dep}_p(c)$ the set of all superclasses of c and of all classes (either directly or indirectly) used by c (for reasons of space we omit the formal definitions). The following claim states that a *Fickle* class declaration can be successfully translated in a *Fickle* program p whenever the set of dependencies of c is contained in p, exactly as happens for Java compilation.

Theorem 3. *For any well-formed Fickle program p and class declaration cld in p, if $\mathit{dep}_p(\mathit{class}(\mathit{cld})) \subseteq \mathit{classes}(p)$, then $[\![\mathit{cld}]\!]_{\mathit{cld}}(p)$ is well-defined.*

Let *strip* be the function on *Fickle* programs defined as follows:

$$\begin{array}{l}
\mathit{strip}(\mathit{cld}_1 \ldots \mathit{cld}_n) = \mathit{strip}(\mathit{cld}_1) \ldots \mathit{strip}(\mathit{cld}_n)\\
\mathit{strip}([\mathtt{root} \mid \mathtt{state}]\ \mathtt{class}\ c\ \mathtt{extends}\ c'\{\mathit{field}^*\ \ \mathit{meth}^*\}) =\\
\qquad [\mathtt{root} \mid \mathtt{state}]\ \mathtt{class}\ c\ \mathtt{extends}\ c'\{\mathit{field}^*\ \ \mathit{strip}(\mathit{meth}^*)\}\\
\mathit{strip}(\mathit{meth}_1 \ldots \mathit{meth}_n) = \mathit{strip}(\mathit{meth}_1) \ldots \mathit{strip}(\mathit{meth}_n)\\
\mathit{strip}(t\ m(t'\ x)\phi\{\mathit{sl}\ \mathtt{return}\ e;\}) = t\ m(t'\ x)\phi\{\mathtt{return}\ \mathsf{v}(t);\}\\
\mathsf{v}(t) = \begin{cases} \mathtt{false} & \text{if } t = \mathtt{boolean}\\ 0 & \text{if } t = \mathtt{int}\\ \mathtt{null} & \text{otherwise} \end{cases}
\end{array}$$

The following theorem states that translation of a *Fickle* class c depends only on the body of c and the type information of all other classes, namely, class kind, parent class, method headers and field declarations. This information is stored in a regular Java class file[6], therefore the translation of c can be successfully carried out also when only the binary files of the other classes are available[7].

Theorem 4. *For any Fickle program p and Fickle class declaration cld_1, if $[\![\mathit{cld}_1]\!]_{\mathit{cld}}(p) = \mathit{cld}_2$, then $[\![\mathit{cld}_1]\!]_{\mathit{cld}}(\mathit{strip}(p)) = \mathit{cld}_2$.*

6 Conclusion

We have defined a translation from *Fickle* (a Java-like language supporting dynamic object re-classification) into plain Java, and proved that this translation well-behaves in the sense that it preserves static and dynamic semantics. This is a nice theoretical result, strengthened by the fact that, in order to ensure these properties, we were able to identify some invariants which turned out to be a very useful guide to the translation.

Our concerns are not only theoretical, but we are interested in investigating the possibility of implementing an extension of Java with re-classification. From this point of view, our translation is a good basis since it exhibits the following additional properties:

[6] Except for the kinds `root` and `state`, but class files format can be easily extended for storing this new piece of information.

[7] Note that this property does not depend on Java support for reflection.

- it is fully compatible with Java separate compilation, since each *Fickle* class can be translated without having other class bodies, hence in principle only having other classes in binary form;
- dependencies among classes are exactly those of standard Java compilation, in the sense that a *Fickle* class can be translated only if type information on all the ancestor and used classes is available.

Our translation is similar both in the structure of classes and in their behavior to the state pattern, see [5]. The wrapper class corresponds to the context class (of the pattern) and the implementation to the state class. Access to members require a level of indirection, as in the state pattern. So from the point of view of efficiency our implementation of reclassification performs as well as the state pattern. On the other side our translation maintains the structure of the original hierarchy, whereas the state pattern does not.

A prototype implementation largely based on the translation described in this paper has already been developed [2].[8] However, the work presented here is only a first step towards a working extension of Java with dynamic object re-classification. On one side, an extension of full Java should take into account other Java features (like constructors, access modifiers, abstract classes, interfaces, overloading and casting) which, though in principle orthogonal to re-classification, should be carefully analyzed in order to be sure that the interaction behaves correctly. On the other side, as mentioned above, an extended compiler should be able to work even in a context where only binary files are available, while our prototype implementation works on source files.

Finally, an alternative direction for the implementation of *Fickle* (or, more generally, of an object-oriented language supporting dynamic re-classification of objects) could be in a direct way, through manipulation of the object layout or the object look-up tables.

References

1. D. Ancona, G. Lagorio, and E. Zucca. Jam - a smooth extension of Java with mixins. In *ECOOP'00*, volume 1850 of *LNCS*, pages 154–178. Springer, 2000. 216
2. Christopher Anderson. Implementing Fickle, Imperial College, final year thesis - to appear, June 2001. 217, 233
3. C. Chambers. Predicate Classes. In *ECOOP'93*, volume 707 of *LNCS*, pages 268–296. Springer, 1993. 215
4. S. Drossopoulou, F. Damiani, M. Dezani-Ciancaglini, and P. Giannini. Fickle: Dynamic object re-classification. In J. L. Knudsen, editor, *ECOOP'01*, number 2072 in LNCS, pages 130–149. Springer, 2001. Also available in: Electronic proceedings of FOOL8 (http://www.cs.williams.edu/ kim/FOOL/). 215, 216, 217, 225, 227, 230, 231
5. R. Johnson E. Gamma, R. Elm and J. Vlissides. *Design Patterns*. Addison-Wesley, 1994. 233

[8] The prototype is written in Java. Future releases might be written in (extended) *Fickle*.

6. M. D. Ernst, C. Kaplan, and C. Chambers. Predicate Dispatching: A Unified Theory of Dispatch. In *ECOOP'98*, volume 1445 of *LNCS*, pages 186–211. Springer, 1998. 215
7. M. Serrano. Wide Classes. In *ECOOP'99*, volume 1628 of *LNCS*, pages 391–415. Springer, 1999. 215

Subtyping and Matching for Mobile Objects*

Michele Bugliesi[1], Giuseppe Castagna[2], and Silvia Crafa[1,2]

[1] Dipartimento di Informatica, Univ. "Ca' Foscari"
Venezia, Italy
[2] Département d'Informatique, École Normale Supérieure
Paris, France

Abstract. In [BCC00], we presented a general framework for extending calculi of mobile agents with object-oriented features, and we studied a typed instance of that model based on Cardelli and Gordon's Mobile Ambients. Here, we refine our earlier work and define a new calculus which is based on *Remote Procedure Call* as the underlying protocol for method invocation, and on a different typing technique for method bodies. The new type system is equipped with a subtyping and a matching relation: the combination of matching with subtyping provides new insight into the relationship between ambient opening in the new calculus and method overriding in object-oriented calculi.

1 Introduction

Calculi of mobile agents are receiving increasing interest in the programming language community as advances in computer communications and hardware enhance the development of large-scale distributed programming. *Agents* are effective entities that perform computation and interact with other agents: the term "mobile" implies that agents are bound to *locations* and that this binding may vary over time; agent interaction, in turn, is achieved using resources such as communication channels.

Independently of the new trends in communication technology, object-oriented programming has established itself as the de-facto standard for a principled design of complex software systems.

Drawing on our earlier work [BC00, BCC00], in this paper we study a formal calculus that integrates object-oriented constructs into calculi of mobile agents. The resulting calculus provides foundations for a computation model for distributed applications, where conventional client-server technology —based on remote exchange of messages between static sites— and mobile agents coexist in a uniform way.

The model results from extending the structure of *named* agents in the style of Mobile Ambients [CG98] with method definitions and primitive constructs for *self* denotation and message passing. The extension has interesting payoffs, as

* Work partially supported by MURST Project 9901403824_003, by CNRS Program *Telecommunications*: "Collaborative, distributed, and secure programming for Internet", and by Galileo Action n. 02841UD

A. Restivo, S. Ronchi Della Rocca, L. Roversi (Eds.): ICTCS 2001, LNCS 2202, pp. 235–255, 2001.

it leads to a principled approach to structuring agents: specifically, introducing methods and message passing as primitive, rather than encoding them on top of the underlying calculus of agents leads to a rich and precise notion of agent interface and type. Furthermore, it opens the way to reusing the advances in type system of object-oriented programming and static analysis.

With respect to our earlier work [BC00, BCC00] this paper brings two main contributions to the calculus. For the operational semantics, we study a new model of message passing and method invocation based on Remote Procedure Call (RPC)[1]. For the type system, we discuss a non-trivial blend of matching and subtyping relations. Method invocation based on RPC fits nicely the design of a typed distributed calculus as it allows method bodies to be type-checked locally, in the object where they are defined, independently of the caller. As a consequence, the choice of RPC as the underlying semantics of method invocation yields a notion of interface-type for our mobile objects that is substantially simpler and more tractable than the corresponding notion defined in [BCC00]. The combination of subtyping and matching, in turn, conveys new insight into the relationship between method overriding in object-oriented calculi and the open capability in our mobile objects. As we show, matching is necessary in the type system to ensure type soundness for object opening in the presence of subtping[2].

Plan of the paper In Section 2 we introduce the calculus of mobile objects, named MA^{++} , based on the calculus of Mobile Ambients by [Car99, CG98]. Section 3 illustrates the expressive power of the calculus with several, diversified, examples. In Section 4 we study the type theory of our calculus, and state relevant properties. Related work is discussed in Section 5. Final remarks in Section 6 conclude our presentation with a discussion on current and future work.

2 MA^{++}

The syntax of the calculus is essentially the same as that originally defined in [BCC00], and results from generalizing the structure of ambients to include method definitions, or *interfaces*, as in $a[\![\,\mathrm{I}\,;\,P\,]\!]$, where P is a process and I is a list of method definitions, defined by the following productions:

$$\begin{array}{llcll}
\text{Processes} & P & ::= & \mathbf{0} & \text{inactivity} \\
 & & \mid & P \mid P & \text{parallel composition} \\
 & & \mid & a[\![\,\mathrm{I}\,;\,P\,]\!] & \text{ambient} \\
 & & \mid & (\boldsymbol{\nu} x)P & \text{restriction} \\
 & & \mid & M.P & \text{action}
\end{array}$$

[1] RPC is often referred to as *Remote Method Invocation* (RMI) in this context.

[2] The new version of the type system also rectifies a flaw of the type system we presented in [BCC00].

$$\begin{array}{llll}
\text{Interfaces} & \mathrm{I} ::= & \ell(\boldsymbol{x}) \triangleright \varsigma(z)P & \text{method} \\
 & \quad\;\; \mid & \mathrm{I} :: \mathrm{J} & \text{sequence} \\
 & \quad\;\; \mid & \varepsilon & \text{empty interface}
\end{array}$$

$$\begin{array}{llll}
\text{Patterns} & \boldsymbol{x} ::= & x & \text{variable} \\
 & \quad\;\; \mid & (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n) & \text{tuple } (n \geqslant 1)
\end{array}$$

The syntax of processes is a generalization of the combinatorial kernel of the Ambient Calculus: $\mathbf{0}$ denotes the inactive process, $P \mid Q$ the parallel composition of two processes P and Q, $a[\![\mathrm{I}\,;\,P]\!]$ denotes the object named a with interface I and enclosed process P, $(\boldsymbol{\nu} x)P$ restricts the name x to P, and finally $M.P$ performs the action described by the term M and then continues as P.

Interfaces are lists of labels with associated processes: the syntactic form $\ell(\boldsymbol{x}) \triangleright \varsigma(z)P$ denotes a method labeled ℓ whose associated body is the process P where the ς-bounded variable z is the *self* parameter distinctive of object calculi, representing the method's host object. Finally, the pattern $\boldsymbol{x}$ is the tuple of input parameters for P.

$$\begin{array}{llll}
\text{Terms} & M, N ::= & a, b, \ldots, x, y \ldots & \text{name/variable} \\
 & \quad\;\; \mid & (M_1, \ldots, M_n) & \text{tuple } (n \geqslant 0) \\
 & \quad\;\; \mid & M.M & \text{path} \\
 & \quad\;\; \mid & \varepsilon & \text{empty path} \\
 & \quad\;\; \mid & \mathsf{in}\ a & \text{enter } a \\
 & \quad\;\; \mid & \mathsf{out}\ a & \text{exit } a \\
 & \quad\;\; \mid & \mathsf{open}\ a & \text{open } a \\
 & \quad\;\; \mid & a\ \mathsf{send}\ \ell\langle M\rangle & \text{remote invocation}
\end{array}$$

Terms include the capabilities distinctive of Mobile Ambients. In addition, our ambients are equipped with a capability for remote method invocation: the expression a send $\ell\langle M\rangle$ invokes the method labeled ℓ residing on the object denoted by a with arguments M.

In the following we let $P, Q, R, \ldots$ range over processes, I, J over (possibly empty) interfaces, and use lower case letters to denote generic names, reserving $a, b, \ldots$ for ambient names, and $x, y, \ldots$ for parameters, whenever possible. Method names, denoted ℓ, range over a disjoint alphabet and have a different status: they are fixed labels that may not be restricted, abstracted upon, nor passed as values (they are similar to field labels in record-based calculi). We omit trailing or isolated $\mathbf{0}$ processes and empty interfaces, using M, $a[\![\mathrm{I}]\!]$, $a[\![P]\!]$, and $a[\![\ ]\!]$ as shorthands for, respectively, $M.\mathbf{0}$, $a[\![\mathrm{I}\,;\,\mathbf{0}]\!]$, $a[\![\varnothing\,;\,P]\!]$, and $a[\![\varnothing\,;\,\mathbf{0}]\!]$. Throughout, we use the terms "ambient" and "object" interchangeably.

2.1 Operational Semantics

We define the operational semantics of the calculus by means of a structural congruence and a reduction relation. As usual, the former is used to rearrange a term in order to apply the latter.

Structural Congruence Structural congruence for processes is defined in terms of an auxiliary equivalence relation $\equiv_{\mathrm{I}}$ over interfaces, given in Figure 1. This relation allows method definitions be reordered without affecting the behavior of the enclosing object: reordering of methods, in turn, is used to define the reduction of method invocation.

(Eq Meth Assoc)	$(\mathrm{I} :: \mathrm{J}) :: \mathrm{L}$	$\equiv_{\mathrm{I}} \mathrm{I} :: (\mathrm{J} :: \mathrm{L})$	
(Eq Meth Comm)	$\mathrm{I} :: m(\boldsymbol{x}_m) \triangleright P; \ell(\boldsymbol{y}_\ell) \triangleright Q$	$\equiv_{\mathrm{I}} \mathrm{I} :: \ell(\boldsymbol{y}_\ell) \triangleright Q :: m(\boldsymbol{x}_m) \triangleright P$	$\ell \neq m$
(Eq Meth Over)	$\mathrm{I} :: \ell(\boldsymbol{x}) \triangleright P :: \ell(\boldsymbol{x}) \triangleright Q :: \mathrm{I}$	$\equiv_{\mathrm{I}} \mathrm{I} :: \ell(\boldsymbol{x}) \triangleright Q$	

Fig. 1. Equivalence for methods

Definitions for methods with different name and/or arity may freely be permuted (Eq Meth Comm); instead, if the same method has multiple definitions, then the right-most definition overrides the remaining ones (Eq Meth Over). Similar notions of equivalence can be found in the literature on objects: in fact, our definition is directly inspired by the bookkeeping relation introduced in [FHM94].

Structural congruence of processes is defined as the smallest congruence that forms a commutative monoid with product $\mid$ and unit $\mathbf{0}$, and is closed under the rules in Figure 2, where the set *fn* of *free names* is defined by a standard extension of the definition in [Car99].

(Struct Res Dead)	$(\boldsymbol{\nu} x)\mathbf{0} \equiv \mathbf{0}$	
(Struct Res Res)	$(\boldsymbol{\nu} x)(\boldsymbol{\nu} y)P \equiv (\boldsymbol{\nu} y)(\boldsymbol{\nu} x)P$	$x \neq y$
(Struct Res Par)	$(\boldsymbol{\nu} x)(P \mid Q) \equiv P \mid (\boldsymbol{\nu} x)Q$	$x \notin \mathit{fn}(P)$
(Struct Res Amb)	$(\boldsymbol{\nu} p)a\llbracket \mathrm{I}\,;\, P \rrbracket \equiv a\llbracket \mathrm{I}\,;\, (\boldsymbol{\nu} p)P \rrbracket$	$p \notin \mathit{fn}(\mathrm{I}) \cup \{a\}$
(Struct Path Assoc)	$(M.M').P \equiv M.M'.P$	
(Struct Empty Path)	$\varepsilon.P \equiv P$	
(Struct Cong Amb Meth)	$\mathrm{I} \equiv_{\mathrm{I}} \mathrm{J} \Rightarrow a\llbracket \mathrm{I}\,;\, P \rrbracket \equiv a\llbracket \mathrm{J}\,;\, P \rrbracket$	

Fig. 2. Structural congruence for processes

The first block of clauses are the rules of the π-calculus. The rule (Struct Path Assoc) is a structural equivalence rule for the Ambient Calculus, while the rule (Struct Res Amb) modifies the rule for ambients in the Ambient calculus to account for the presence of methods. Rule (Struct Cong Amb Meth) establishes ambient equivalence up to reordering of method suites. In addition, we identify processes up to renaming of bound names: $(\boldsymbol{\nu} p)P = (\boldsymbol{\nu} q)P\{p := q\}$ if $q \notin \mathit{fn}(P)$.

Reduction Relation The reduction semantics of the calculus is given by the context rules in Figure 3, plus the notions of reduction collected in Figure 4, that we comment below.

$$\begin{array}{ll}
P' \equiv P,\ P \rightarrow Q,\ Q \equiv Q' \Rightarrow P' \rightarrow Q' & P \rightarrow Q \Rightarrow a\llbracket \mathrm{I}\,;\, P \rrbracket \rightarrow a\llbracket \mathrm{I}\,;\, Q \rrbracket \\
P \rightarrow Q \Rightarrow (\boldsymbol{\nu} x)P \rightarrow (\boldsymbol{\nu} x)Q & P \rightarrow Q \Rightarrow P \mid R \rightarrow Q \mid R
\end{array}$$

Fig. 3. Structural rules for reduction

$$\begin{array}{ll}
(\mathit{in}) & b\llbracket \mathrm{I}\,;\ \mathsf{in}\ \ a.P \mid Q \rrbracket \mid a\llbracket \mathrm{J}\,;\, R \rrbracket \ \rightarrow\ a\llbracket \mathrm{I}\,;\, R \mid b\llbracket \mathrm{J}\,;\, P \mid Q \rrbracket\, \rrbracket \\
(\mathit{out}) & a\llbracket \mathrm{I}\,;\, b\llbracket \mathrm{J}\,;\, \mathsf{out}\ \ a.P \mid Q \rrbracket \mid R \rrbracket \ \rightarrow\ b\llbracket \mathrm{J}\,;\, P \mid Q \rrbracket \mid a\llbracket \mathrm{I}\,;\, R \rrbracket \\
(\mathit{open}) & \mathsf{open}\ \ a.P \mid a\llbracket Q \rrbracket \ \rightarrow\ P \mid Q \\
(\mathit{update}) & b\llbracket \mathrm{I}\,;\, \mathsf{open}\ \ a.P \mid a\llbracket \mathrm{J}\,;\, Q \rrbracket \mid R \rrbracket \ \rightarrow\ b\llbracket \mathrm{I} :: \mathrm{J}\,;\, P \mid Q \mid R \rrbracket \qquad \text{for } \mathrm{J} \neq \varepsilon \\
(\mathit{send}) & b\llbracket \mathrm{I}\,;\, a\ \mathsf{send}\ \ell\langle M\rangle.P \mid Q \rrbracket \mid a\llbracket \mathrm{J} :: \ell(\boldsymbol{x}) \triangleright \varsigma(z)R\,;\, S \rrbracket \\
 & \qquad \rightarrow\ b\llbracket \mathrm{I}\,;\, P \mid Q \rrbracket \mid a\llbracket \mathrm{J} :: \ell(\boldsymbol{x}) \triangleright \varsigma(z)R\,;\, R\{z, \boldsymbol{x} := a, M\} \mid S \rrbracket
\end{array}$$

Fig. 4. MA^{++} reduction rules

The first three rules are exactly the same as the corresponding rules for Mobile Ambients. Rule (*update*) is a direct generalization of the open rule that handles the case when the opened ambient contains a non-empty set of method definitions. If a is one such ambient, $\mathsf{open}\ a$ may only be reduced within an enclosing ambient: as a result of reduction, the process local to a is unleashed within the opening ambient, and the interfaces of the opening and the opened ambients are merged as shown by the definition of the rule. The rule (*send*) handles the new syntactic construct for method invocation, implementing the RPC model. The notation $R\{z, \boldsymbol{x} := a, M\}$ indicates the simultaneous substitution in R of a for z and of M for $\boldsymbol{x}$. Informally, the result of the ambient b sending message ℓ to its sibling a, with argument M, is the activation of the process associated with ℓ on the receiver a, with M substituted for the input pattern $\boldsymbol{x}$ and the *self* parameter dynamically bound to the name of the receiver.

3 Expressive Power

We discuss a number of constructs that can be expressed in our calculus, including constructs for method overriding distinctive of object calculi, various forms of process communication, as well as different primitives of method invocation. Some of these examples have been already presented in our earlier work [BCC00] where, however, they were defined in terms of a different semantics for method invocation based on *Code On Demand*. Throughout this section, we use the terms "protocol" and "encoding" as synonyms: technically, this is an abuse of terminology, as we don't claim the protocols to really be encodings, i.e. interference-free simulations of the constructs in question.

3.1 Parent-Child and Local Communications

As a first example, we look at alternative models for method invocation. Having having chosen RPC as our primitive semantics, we now discuss other models, such as those described in Figure 5, for sending messages from an ambient to its parent or children, or to its own methods.

$$\begin{array}{lll}
(\mathit{downsend}) & a\ \mathsf{downsend}\ \ell\langle M\rangle.P \mid a\llbracket \mathrm{I} :: \ell(\boldsymbol{x}) \triangleright \varsigma(z)Q\,;\ R \rrbracket & \\
 & \qquad\qquad \rightarrow\ P \mid a\llbracket \mathrm{I} :: \ell(\boldsymbol{x}) \triangleright \varsigma(z)Q\,;\ R \mid Q\{z := a, \boldsymbol{x} := M\} \rrbracket & \\
(\mathit{upsend}) & a\llbracket \mathrm{I} :: \ell(\boldsymbol{x}) \triangleright \varsigma(z)Q\,;\ R \mid b\llbracket \mathrm{J}\,;\ a\ \mathsf{upsend}\ \ell\langle M\rangle.P \rrbracket \rrbracket & \\
 & \qquad\qquad \rightarrow\ a\llbracket \mathrm{I} :: \ell(\boldsymbol{x}) \triangleright \varsigma(z)Q\,;\ R \mid Q\{z := a, \boldsymbol{x} := M\} \mid b\llbracket \mathrm{J}\,;\ P \rrbracket \rrbracket & \\
(\mathit{local}) & a\llbracket \mathrm{I} :: \ell(\boldsymbol{x}) \triangleright \varsigma(z)Q\,;\ a\ \mathsf{local}\ \ell\langle M\rangle.P_1 \mid P_2 \rrbracket & \\
 & \qquad\qquad \rightarrow\ a\llbracket \mathrm{I} :: \ell(\boldsymbol{x}) \triangleright \varsigma(z)Q\,;\ Q\{z, \boldsymbol{x} := M, a\} \mid P_1 \mid P_2 \rrbracket &
\end{array}$$

Fig. 5. Other constructs for method invocation

Parent-to-child invocation. The intended behavior for this form of method invocation can be obtained by defining the construct for downward method invocation as follows, where $p, q \notin \mathit{fn}(M) \cup \mathit{fn}(P)$):

$$a\ \mathsf{downsend}\ \ell\langle M\rangle.P \ \stackrel{\triangle}{=}\ (\boldsymbol{\nu} p, q)\,(p\llbracket a\ \mathsf{send}\ \ell\langle M\rangle.q\llbracket \mathsf{out}\ p \rrbracket \rrbracket \mid \mathsf{open}\ q.\mathsf{open}\ p.P)$$

Informally, we temporarily create a new ambient p that becomes a sibling of the receiver a on which it invokes the method; the ambient q is used for synchronization, to guarantee that the ambient p be destroyed only after the receiver has served the invocation. It is a routine check to verify that the desired effect of the invocation is achieved by a sequence of reduction steps. To ease the notation, we give the reduction steps in the simplified case of a method which does not have parameters and does not depend on *self* (neither of the two simplifications affects the protocol):

$$
\begin{aligned}
&a\ \mathsf{downsend}\ \ell.P \mid a[\,\ell \triangleright Q\,;\,R\,]\\
&\qquad\equiv (\boldsymbol{\nu} p,q)\Big(p[\,\underline{a\ \mathsf{send}\ \ell\langle M\rangle}.q[\,\mathsf{out}\ p\,]\,] \mid \mathsf{open}\ q.\mathsf{open}\ p.P\Big) \mid a[\,\ell \triangleright Q\,;\,R\,]\\
&\qquad\rightarrow (\boldsymbol{\nu} p,q)\,\big(p[\,q[\,\underline{\mathsf{out}\ p}\,]\,] \mid \mathsf{open}\ q.\mathsf{open}\ p.P\big) \mid a[\,\ell \triangleright Q\,;\,R \mid Q\,]\\
&\qquad\rightarrow (\boldsymbol{\nu} p,q)\,\big(p[\,] \mid q[\,] \mid \underline{\mathsf{open}\ q.\mathsf{open}\ p}.P\big) \mid a[\,\ell \triangleright Q\,;\,R \mid Q\,]\\
&\qquad\rightarrow^{*} P \mid a[\,\ell \triangleright Q\,;\,R \mid Q\,]
\end{aligned}
$$

Local and Self Invocation. Local method invocation within an ambient a is encoded similarly to the previous case. Choosing $p, q \notin fn(M) \cup fn(P)$, one defines:

$$a\ \mathsf{local}\,\ell\langle M\rangle.P \triangleq (\nu p,q)\,(p[\,\mathsf{out}\ a.a\ \mathsf{send}\ \ell\langle M\rangle.\mathsf{in}\ a.q[\,\mathsf{out}\ p\,]\,] \mid \mathsf{open}\ q.\mathsf{open}\ p.P)$$

Relying upon this definition, it is then easy to define self-invocation within method bodies. To exemplify, consider the following process:

$$a[\,\ell_1(x) \triangleright \varsigma(z)z\ \mathsf{local}\ \ell_2\langle x\rangle :: \ell_2(x) \triangleright P\,;\,R\,]$$

Invoking the method ℓ_1 from outside the object a results in the execution of the process P in parallel with R within a.

Child-to-parent. We conclude our survey of alternative models of method invocation with a form of upward invocation, whereby an ambient invokes a method residing in the enclosing ambient. A first definition of the construct is simply

$$a\ \mathsf{upsend}\ \ell\langle M\rangle.P \ \triangleq\ \mathsf{out}\ a.a\ \mathsf{send}\ \ell\langle M\rangle.\mathsf{in}\ a$$

One problem with this definition is that it requires a move of the sender. As an alternative, one may envisage a different protocol that relies on an auxiliary ambient. Assume that the invocation occurs within an object b, and that b is directly enclosed into a:

$$
\begin{aligned}
&a\ \mathsf{upsend}\ \ell\langle M\rangle.P \triangleq\\
&\qquad(\nu\ p,q)\,(p[\,\mathsf{out}\ b.\mathsf{out}\ a.a\ \mathsf{send}\ \ell\langle M\rangle.\mathsf{in}\ a.\mathsf{in}\ b.q[\,\mathsf{out}\ p\,]\,] \mid \mathsf{open}\ q.\mathsf{open}\ p.P)
\end{aligned}
$$

The definition is easily understood by simply looking at the chain of capabilities inside the ambient p. First, the ambient p exits its parent ambient b, then exits the ambient a (that contains the method to be invoked), then performs the message send and is finally destroyed after having opened the locking ambient q. It should be noted that a formal specification of the protocol requires that the definition be given parametrically with respect to the enclosing ambient (b in the definition given above).

3.2 Replication

The behavior of replication in concurrent calculi is typically defined by a structural equivalence rule establishing that $!P \equiv !P \mid P$. In our calculus, we can provide a similar construct by relying upon the implicit form of recursion underlying the reduction of method invocation. Let be $p, q \notin \mathit{fn}(P)$:

$$!P \;\triangleq\; (\boldsymbol{\nu} p, q)\,(p\ \mathsf{downsend}\ !\langle\rangle.\mathsf{open}\ q.P \mid \\ p\boldsymbol{[}\,! \triangleright \varsigma(z)(q\boldsymbol{[}\mathsf{out}\ z.z\ \mathsf{downsend}\ !\langle\rangle.\mathsf{open}\ q.P\boldsymbol{]}\,)\,;\ \boldsymbol{]}\,)$$

The reduction for the encoding of $!P$ is then the following:

$$\begin{aligned} !P \triangleq\ & (\boldsymbol{\nu} p, q)\,\Big(\underline{p\ \mathsf{downsend}\ !\langle\rangle}.\mathsf{open}\ q.P \mid p\boldsymbol{[}\,! \triangleright \varsigma(z)(q\boldsymbol{[}\cdots\boldsymbol{]}\,)\,;\ \boldsymbol{]}\Big) \\ \rightarrow\ & (\boldsymbol{\nu} p, q)\,\big(\mathsf{open}\ q.P \mid p\boldsymbol{[}\,! \triangleright \varsigma(z)(...)\,;\ q\boldsymbol{[}\underline{\mathsf{out}\ p}.p\ \mathsf{downsend}\ !\langle\rangle.\mathsf{open}\ q.P\boldsymbol{]}\ \boldsymbol{]}\big) \\ \rightarrow\ & (\boldsymbol{\nu} p, q)\,\big(\underline{\mathsf{open}\ q}.P \mid q\boldsymbol{[}p\ \mathsf{downsend}\ !\langle\rangle.\mathsf{open}\ q.P\boldsymbol{]} \mid p\boldsymbol{[}\,! \triangleright \varsigma(z)(...)\,;\ \boldsymbol{]}\big) \\ \rightarrow\ & (\boldsymbol{\nu} p, q)\,(P \mid p\ \mathsf{downsend}\ !\langle\rangle.\mathsf{open}\ q.P \mid p\boldsymbol{[}\,! \triangleright \varsigma(z)(...)\,;\ \boldsymbol{]}\,) \\ \equiv\ & P \mid !P \end{aligned}$$

Notice that there is just one capability ready to be exercised at each reduction step. Furthermore, the process P is activated only after the opening of the ambient q, hence it does not interfere with the protocol.

3.3 Code on Demand

We continue our series of examples showing a protocol for method invocation based on Code on Demand (CoD). The behavior CoD can be described as follows: a client c invokes a method ℓ on a server s; the server activates the method and then sends it back to the client for the latter to execute it. Formally this correspond to the following reduction rule:

$$c\boldsymbol{[}\,\mathrm{J}\,;\ s\ \mathsf{send\_cod}\ell\langle M\rangle.R \mid S\,\boldsymbol{]} \mid s\boldsymbol{[}\,\mathrm{I} :: \ell(x) \triangleright \varsigma(z)Q\,;\ P\,\boldsymbol{]} \quad \rightarrow \\ c\boldsymbol{[}\,\mathrm{J}\,;\ Q\{z, x := s, M\} \mid R \mid S\,\boldsymbol{]} \mid s\boldsymbol{[}\,\mathrm{I} :: \ell(x) \triangleright \varsigma(z)Q\,;\ P\,\boldsymbol{]}$$

The intended behavior can be obtained by defining the sender and the receiver ambients as follows:

$$\mathit{server} \triangleq s\boldsymbol{[}\,\mathrm{I} :: \ell(u, v, x) \triangleright \varsigma(z)u\boldsymbol{[}\mathsf{out}\ z.\mathsf{in}\ v.Q\boldsymbol{]}\ ;\ P\,\boldsymbol{]}$$

$$\mathit{client} \triangleq c\boldsymbol{[}\,\mathrm{J}\,;\ (\boldsymbol{\nu} p)s\ \mathsf{send}\ \ell\langle p, c, M\rangle.\mathsf{open}\ p.R \mid S\,\boldsymbol{]}$$

The protocol relies on the agreement between the server and the client upon the name of the ambient that carries the activated process back to the client. This name is decided locally by the client which passes it as an argument of the call together with its own name. Invoking $\ell\langle p, c.M\rangle$ spawns a new process on the server that simply carries the ambient p out of the server and back into the client c: once inside c, the transport ambient p is opened thus unleashing the process Q to be executed on the client.

The protocol can be refined by having the client pass a “return path” rather than just its name. In that case, the client would be in the position to choose where to receive and execute the requested method (e.g. , in one of its subambients).

3.4 Updates

The standard notion of method override in formal object calculi [AC96, FHM94] can be rephrased in our calculus, as follows:

> given the ambient $a[\![\,\mathrm{I}::\ell(\boldsymbol{x}) \triangleright \varsigma(z)P; Q\,]\!]$ replace the current definition P of ℓ by the new definition P' to form the ambient $a[\![\,\mathrm{I} :: \ell(\boldsymbol{x}) \triangleright \varsigma(z)P'; Q\,]\!]$.

Method updates in this form can be expressed in our calculus by means of a protocol that uses an “updater” ambient to carry the new method body inside the ambient to be updated. The updater enters the ambient a to be updated, and the latter has a controlling process that opens the updater thus allowing updates on its own methods. The protocol is defined precisely below in an asynchronous setting, with the update defined as a process term: a similar encoding can be defined for synchronous updates. Moreover, the definition only allows local updates, in that an ambient may only override methods contained in subambients (of course other kind of updates can be expressed similarly)

A method update is denoted by a update $\ell(\boldsymbol{x}) \triangleright \varsigma(z)P$, read “the ℓ method at a gets definition P ”, and is defined as the following process:

$$a \ \mathsf{update}\ \ell(\boldsymbol{x}) \triangleright \varsigma(z)P \ \stackrel{\triangle}{=}\ \textsc{upd}[\![\,\ell(\boldsymbol{x}) \triangleright \varsigma(z)P\,;\ \mathsf{in}\ a\,]\!]$$

The ambient to be updated may now be defined as follows:

$$a^{\star}[\![\,\mathrm{I}\,;\ P\,]\!] \ \stackrel{\triangle}{=}\ a[\![\,\mathrm{I}\,;\ !(\mathsf{open}\ \textsc{upd}) \mid P\,]\!]$$

Now, if we form the composition a update $\ell(\boldsymbol{x}) \triangleright \varsigma(z)P' \mid a^{\star}[\![\,\mathrm{I} :: \ell(\boldsymbol{x}) \triangleright \varsigma(z)P\,;\ Q\,]\!]$, the reduction for open enforces the expected behavior:

$$a\ \mathsf{update}\ \ell(\boldsymbol{x}) \triangleright \varsigma(z)P' \mid a^{\star}[\![\,\mathrm{I} :: \ell(\boldsymbol{x}) \triangleright \varsigma(z)P\,;\ Q\,]\!] \quad \rightarrow^{*} \quad a^{\star}[\![\,\mathrm{I} :: \ell(\boldsymbol{x}) \triangleright \varsigma(z)P'\,;\ Q\,]\!]$$

Multiple updates for the same method may occur in parallel, in which case their relative order is established nondeterministically. The protocol, as defined, relies on the assumption that the name UPD of the updater carrying the new method body is “well known”. A more realistic assumption is that the ambient to be updated and the context agree on the name of the updater prior to start the protocol. This can be accomplished with a different definition of the ambient to be updated, one that assumes that such ambients come with an ad-hoc method that sets the appropriate conditions for the actual update to take place. The *upd* method below serves this purpose.

$$a^{\star}[\![\,\mathrm{I}\,;\ P\,]\!] \ \stackrel{\triangle}{=}\ a[\![\,\mathrm{I} :: \mathit{upd}(u) \triangleright \varsigma(z)\mathsf{open}\ u\,;\ P\,]\!]$$

Now, the protocol comprises two steps. First the ambient to be updated receives the name of the updater, and only then does the update take place:

$$a\ \mathsf{update}\ \ell(\boldsymbol{x}) \triangleright \varsigma(z)P \ \stackrel{\triangle}{=}\ (\boldsymbol{\nu}p)\,(a\ \mathsf{downsend}\ upd\langle p\rangle.p\,\boldsymbol{[}\,\ell(\boldsymbol{x}) \triangleright \varsigma(z)P\,;\ \mathsf{in}\ a\,\boldsymbol{]}\,)$$

3.5 Encoding the π-Calculus

As a final example, we define constructs for synchronous and asynchronous communication between processes (all processes, not just ambients) over named channels. Similar construts for channel-based communication are presented in [CG98], based on the more primitive form of local and *anonymous* communication defined for the Ambient Calculus. Here, instead, we rely on the ability, distinctive of our ambients, to exchange values between methods. We first give a construct for synchronous communication.

A named channel n is represented by an "updatable" ambient n, and three auxiliary ambients n^i, n^o and $\bar{n}$ used for synchronization. The ambient n defines a method ch: a process willing to read from n installs itself as the body of this method, whereas a process willing to write on n invokes ch passing along the argument of the communication.

$$\begin{aligned}
(ch\ n) &\stackrel{\triangle}{=} n^{\star}\boldsymbol{[}\,\mathtt{ch}(x) \triangleright \mathbf{0}\,\boldsymbol{]} \mid n^i\boldsymbol{[}\ \boldsymbol{]}\\
n!\langle y\rangle.Q &\stackrel{\triangle}{=} \mathsf{open}\ n^o.n\ \mathsf{downsend}\ \mathtt{ch}(y).\mathsf{open}\ \bar{n}.(n^i\boldsymbol{[}\ \boldsymbol{]} \mid Q)\\
n?(x).P &\stackrel{\triangle}{=} \mathsf{open}\ n^i.\ n\ \mathsf{update}\ \mathtt{ch}(x) \triangleright (\bar{n}\boldsymbol{[}\,\mathsf{out}\ n.P\,\boldsymbol{]}\,)\ .n^o\boldsymbol{[}\ \boldsymbol{]}
\end{aligned}$$

The steps of the communication protocol are as follows. A process $n?(x).P$ reading from n first grabs the input lock n^i provided by the channel, then installs itself as the body of the ch method in n, and finally releases the output lock n^o. Now the writing process can start its computation: after acquiring the lock n^o, it sends the message $\mathtt{ch}(y)$. The message activates the process $\bar{n}\boldsymbol{[}\,\mathsf{out}\ n.P\{x := y\}\,\boldsymbol{]}$ inside n. One further step brings the ambient $\bar{n}$ outside n where it is opened by the output process: this last step completes the synchronization phase of the protocol, and both processes may continue their computation. The output process releases a new input lock to reset the channel to its initial condition, and the protocol is completed.

Asynchronous communications are obtained directly from the protocol above, by a slight variation of the definition of $n!\langle A\rangle.Q$. We simply need a different way of composing Q with the context:

$$n!\langle y\rangle.Q \ \stackrel{\triangle}{=}\ (\mathsf{open}\ n^o.n\ \mathsf{downsend}\ \mathtt{ch}(y).\mathsf{open}\ \bar{n}.(n^i\boldsymbol{[}\ \boldsymbol{]})) \mid Q$$

Based on this technique, we can encode the synchronous (and similarly, the asynchronous) polyadic π-calculus in ways similar to what is done in [CG99]. Each name n in the π-calculus becomes a quadruple of names in our calculus: the name n of the ambient dedicated to the communication, the names n^i and n^o of the two locks, and the name $\bar{n}$ of the auxiliary ambient. Therefore, communication of a π-calculus name becomes the communication of a quadruple of ambient names.

$$
\begin{aligned}
\langle\!\langle\, (\boldsymbol{\nu} n)P \,\rangle\!\rangle &\triangleq (\boldsymbol{\nu} n, \bar{n}, n^i, n^o)(n^i\llbracket\ \rrbracket \mid n^\star\llbracket \mathtt{ch}(x, \bar{x}, x^i, x^o) \triangleright \mathbf{0}\rrbracket \mid \langle\!\langle P \rangle\!\rangle\,) \quad \bar{n}, n^i, n^o \notin \mathit{fn}(\langle\!\langle P \rangle\!\rangle) \\
\langle\!\langle\, n!\langle y\rangle.Q \,\rangle\!\rangle &\triangleq \mathsf{open}\ n^o.n\ \mathsf{downsend}\ \mathtt{ch}(y, \bar{y}, y^i, y^o).\mathsf{open}\ \bar{n}.(n^i\llbracket\ \rrbracket \mid \langle\!\langle\, Q \,\rangle\!\rangle\,) \\
\langle\!\langle\, n?(x).P \,\rangle\!\rangle &\triangleq \mathsf{open}\ n^i.n\ \mathsf{update}\ \mathtt{ch}(x, \bar{x}, x^i, x^o) \triangleright (\bar{n}\llbracket \mathsf{out}\ n.\langle\!\langle\, P \,\rangle\!\rangle\ \rrbracket\,)\ .n^o\llbracket\ \rrbracket \\
\langle\!\langle\, P \mid Q \,\rangle\!\rangle &\triangleq \langle\!\langle\, P \,\rangle\!\rangle \mid \langle\!\langle\, Q \,\rangle\!\rangle \\
\langle\!\langle\, !P \,\rangle\!\rangle &\triangleq\, !\langle\!\langle\, P \,\rangle\!\rangle \\
\langle\!\langle\, \mathbf{0} \,\rangle\!\rangle &\triangleq \mathbf{0}
\end{aligned}
$$

Fig. 6. Encoding of the synchronous π-calculus

The initialization of the ch method in the ambient that represents the channel n could be safely omitted, without affecting the operational properties of the encoding. However, as given, the definition scales smoothly to the case of a typed encoding, preserving well-typing.

4 Types and Type Systems

The structure of ambient, capability and process types is similar to that of companion type systems for Mobile Ambients: their intended meaning, instead, is different.

$$
\begin{array}{lll}
\text{Signatures} & \Sigma ::= & (\ \ell_i(\mathcal{V}_i)\ )^{i \in I} \\
\text{Ambients} & \mathcal{A} ::= & \mathsf{Amb}[\Sigma] \\
\text{Capabilities} & \mathcal{C} ::= & \mathsf{Cap}[\Sigma] \\
\text{Processes} & \mathcal{P} ::= & \mathsf{Proc}[\Sigma] \\
\text{Values} & \mathcal{V} ::= & \mathcal{A} \mid \mathcal{C} \\
\text{Types} & \mathcal{T} ::= & X \mid \mathcal{A} \mid \mathcal{C} \mid \mathcal{P}
\end{array}
$$

Signatures convey information about the interface of an ambient, by listing the ambient's method names and their input types. The type $\mathsf{Amb}[\Sigma]$ is the type of ambients with methods declared in Σ, while the types $\mathsf{Cap}[\Sigma]$ and $\mathsf{Proc}[\Sigma]$ are the types of capabilities and processes, respectively, whose enclosing ambient (if any) has a signature containing at least the methods included in Σ.

The type $\mathcal{V}$ identifies the type of the expressions that may occur as arguments for method invocation, and defines them to be ambient names and capabilities. The complete syntax of types includes type variables, which are used in the typing rules for the typing of method bodies, as we explain shortly.

4.1 Subtyping and Matching

To enhance the flexibility of ambient typing and mobility, a subtype relationship is introduced over capability and process types, as defined by the two following core rules.

$$\text{(Sub Cap)}\ \frac{\Sigma \subseteq \Sigma'}{\mathsf{Cap}[\Sigma] \leq \mathsf{Cap}[\Sigma']} \qquad \text{(Sub Proc)}\ \frac{\Sigma \subseteq \Sigma'}{\mathsf{Proc}[\Sigma] \leq \mathsf{Proc}[\Sigma']}$$

Informally, the rules state that a capability (resp. process) type $\mathsf{Cap}[\Sigma]$ (resp. $\mathsf{Proc}[\Sigma]$) is a subtype of any capability (resp. process) type whose associated signature (set theoretically) contains Σ. The resulting relation of subtyping is reminiscent of the relation of subtyping in *width* distinctive of type systems for object calculi. Width subtyping is restricted to capability and process types, and does *not* extend to ambient types, as the extension would break type soundness in the presence of ambient opening. The reason is explained, intuitively, as follows: when opening an ambient, one needs *exact* knowledge of the contents of that ambient —specifically, of what exactly is the set its methods and their types— so as to ensure that the possible method overrides resulting from the opening be traced in the types.

As a result of capability and process subtyping, it is nevertheless possible, from within an ambient with interface Σ, to open any enclosed ambient with interface $\Sigma' \subseteq \Sigma$, where the inclusion may be strict. To account for this flexibility, we introduce a relation of *matching* [Bru94] over ambient types to complement the subtype relation over capability and process types. The relation of matching is defined by the following rule:

$$\text{(Match Amb)}\ \frac{\Gamma \vdash \diamond \quad \Sigma' \subseteq \Sigma}{\Gamma \vdash \mathsf{Amb}[\Sigma] \mathrel{<\!\!\#} \mathsf{Amb}[\Sigma']}$$

The complete definition of subtyping and matching includes standard rules for reflexivity and transitivity (not shown). Also, as customary, the subtyping relation is endowed in the type system via a subsumption rule, while matching is not.

A further remark is in order to explain the role of type variables in the syntax of types. As we noted, due to the presence of ambient opening, a method residing in ambient, say a, may be re-installed inside any ambient, say b, that opens a; furthermore, the (sub)typing rules provide guarantees that b has "more methods" than a. Now, in order for the original typing of the methods residing in a to be sound after the methods have been re-installed in b, one must ensure that the bodies of these methods be type-checked under appropriate assumptions for the type of *self*: specifically, this type should be so defined as to represent the type of all ambients where the methods may eventually be re-installed, via opening. This is accomplished in the type system by typing method bodies in type environments that assume the so-called *MyType* [Bru94] typing for the *self* variable, i.e. a match-bounded type variable $X \mathrel{<\!\!\#} \mathcal{A}$, where $\mathcal{A}$ is the type of the ambient where the methods are initially installed.

Our relation of matching, and the technique of *MyType* typing of methods we just outlined are simplified versions of the corresponding relation and technique originally introduced in [Bru94]. The simplifications result from the syntax of types, and specifically from our ambient types being *simple*, i.e. not containing occurrences of type variables (neither free, nor bound). As a consequence, the type system does not support *MyType* method specialization [Bru94, FHM94], the OO-typing technique that allows method-types to be specialized when methods are inherited (or, in our context, when they are subsumed in an opening ambient). Instead, in our calculus a method body has always the same type (the one declared in Σ), independently of the dynamic binding of its *self* variable. This is not surprising, as our method bodies are processes with no return value, hence they are dealt with essentially as methods with return type `unit` in imperative object calculi.

4.2 Judgements and Typing Rules

The typed syntax of the calculus is described by the productions below:

Interfaces $\mathrm{I} \ ::= \ell(\boldsymbol{x}) \triangleright \varsigma(z)P \mid \mathrm{I} :: \mathrm{I} \mid \varepsilon$

Processes $P \ ::= \mathbf{0} \mid P|P \mid a[\mathrm{I};\, P] \mid (\boldsymbol{\nu}x{:}\mathcal{A})P \mid M.P$

Expressions $M ::= x \mid (M_1, \ldots, M_n) \mid x\ \mathsf{send}\ \ell\langle M\rangle \mid \mathsf{in}\ x \mid \mathsf{out}\ x \mid \mathsf{open}\ x \mid \varepsilon$

The only type annotations in the syntax are those introduced by the restriction operator: the types for all the other variables are directly inferred from the existing annotations. Also note that we take method names to be fixed *labels* that may not be passed as values, nor restricted. The first restriction is justified by the fact that method names are part of the structure of ambient (capability and process) types; as a consequence, lifting this restriction would be possible but it would make our types (first-order) dependent types. Instead, lifting the second restriction is possible, and in fact not difficult, even though it complicates the format of the typing rules. For this reason we will disregard this issue in what follows.

Type environments are lists of term and type variable declarations, as defined by the following productions: $\Gamma ::= \varnothing \mid \Gamma, x : \mathcal{W} \mid \Gamma, X \lessdot\!\# \mathcal{A}$. The typing rules derive the following judgement forms, where we let $\mathcal{W}$ range over the set $\{X, \mathcal{A}, \mathcal{C}\}$ of extended value types:

$\Gamma \vdash M : \mathcal{W}$	M has type $\mathcal{W}$
$\Gamma \vdash X \lessdot\!\# \mathcal{A}$	X matches $\mathcal{A}$
$\Gamma \vdash P : \mathcal{P}$	P has type $\mathcal{P}$
$\Gamma \vdash \mathcal{T}$	well-formed type
$\Gamma \vdash \diamond$	well-formed type environment

The complete set of typing rules is presented in Appendix A, the most interesting are discussed below. We start with the rule for typing ambient opening.

(OPEN)

$$\frac{\Gamma \vdash a : \mathsf{Amb}[\Sigma]}{\Gamma \vdash \mathsf{open}\ a : \mathsf{Cap}[\Sigma]}$$

As we noted earlier, opening an ambient requires precise knowledge of the type of the ambient being opened: this is expressed in the rule by fact that the type of the ambient a is an ambient type, not a type variable. Opening a is now legal under the condition that the signature of the opening ambient be equal to (in fact, contain, given the presence of subtyping) the signature of the ambient being opened. This condition is necessary for type soundness, as it guarantees that an ambient may only update existing methods of the opening ambient, preserving their original types.

$$\textsc{(Message)}\quad \frac{\Gamma \vdash a : \mathcal{W} \quad \Gamma \vdash \mathcal{W} \lessdot\!\!\# \, \mathsf{Amb}[\ \ell(\mathcal{V}')\ ] \quad \Gamma \vdash M' : \mathcal{V}'}{\Gamma \vdash a\ \mathsf{send}\ \ell\langle M' \rangle : \mathsf{Cap}[\Sigma]}$$

The rule (Message) states that invoking method ℓ on an ambient a requires the type of a to match an ambient type containing the method ℓ. Note that the type of a may either be an ambient type matching (i.e. "longer" then) $\mathsf{Amb}[\ell(\mathcal{V}')]$, or else an unknown type (i.e. a type variable) occurring match-bounded in the context Γ. Since the body of the invoked method is activated on the receiver (rather than on the sender) no constraint is required on the type of the send capability. Of course, in order for the expression to type check, the message argument and the method parameters must have the same type[3].

$$\textsc{(Amb)}\quad (\Sigma = (\ \ell_i(\mathcal{V}_i)\ )^{i \in I})$$
$$\frac{\Gamma \vdash a : \mathsf{Amb}[\Sigma] \quad \Gamma, Z \lessdot\!\!\# \, \mathsf{Amb}[\Sigma], z{:}Z, x_i{:}\mathcal{V}_i \vdash P_i : \mathsf{Proc}[\Sigma] \quad \Gamma \vdash P : \mathsf{Proc}[\Sigma]}{\Gamma \vdash a[(\ell_i(x_i) \triangleright \varsigma(z)P_i)\ ^{i \in I}\ ;\ P] : \mathsf{Proc}[\Sigma']}$$

The rule (Amb) for typing ambients is similar to the typing rule for objects in the calculus of extensible objects of [BB99]. Each method of the ambient is type-checked under the assumptions that (*i*) the self parameter has a type that matches the type of the enclosing ambient, (*ii*) method parameters have the declared type, and (*iii*) the type of each method body be consistent with the type of the enclosing ambient. As we noted earlier, the use of the match-bound type variable Z as the type of *self* ensures that methods local to ambient a are well-typed also within any other ambient that might eventually open a. On the other hand, the typing rule does not support *MyType* method specialization, as the types of method bodies are independent of the type of *self*.

Also note that the rule requires exact knowledge of the type of the ambient a: a structural rule allowing the name of the ambient to be typed with a match-bounded type variable would break type soundness, since we would not have a precise control of the openings of that ambient (see rule (open)). Finally, no constraint is imposed on the signature Σ', associated with the process type in the conclusion of the rule, as that signature is (a subset of) the signature of the ambient enclosing a (if any).

[3] In fact, since capability types can be subtyped, the type of the arguments can be subtypes of the type of the formal parameters.

4.3 Subject Reduction and Type Soundness

We conclude the description of the basic type system with a result of subject reduction. The proof is rather standard, and only sketched due to lack of space.

Lemma 1 (Substitution).

1. *If* $\Gamma, x : \mathcal{W} \vdash P : \mathcal{P}$ *and* $\Gamma \vdash M : \mathcal{W}$*, then* $\Gamma \vdash P\{x := M\} : \mathcal{P}$.
2. *If* $\Gamma, Z \lessdot\!\# \mathcal{A}, z : Z \vdash P : \mathcal{P}$ *and* $\Gamma \vdash a : \mathcal{A}'$*,* $\Gamma \vdash \mathcal{A}' \lessdot\!\# \mathcal{A}$*,*
   then $\Gamma \vdash P\{z := a\} : \mathcal{P}$.

Proof. By induction on the derivation of the first judgment in hypothesis.

Lemma 2 (Subject Congruence).

1. *If* $\Gamma \vdash P : \mathsf{Proc}[\Sigma]$ *and* $P \equiv Q$ *then* $\Gamma \vdash Q : \mathsf{Proc}[\Sigma]$.
2. *If* $\Gamma \vdash P : \mathsf{Proc}[\Sigma]$ *and* $Q \equiv P$ *then* $\Gamma \vdash Q : \mathsf{Proc}[\Sigma]$.

Proof. By simultaneous induction on the derivations of $P \equiv Q$ *and* $Q \equiv P$.

Lemma 3 (Bounded Weakening).

1. *If* $\Gamma, x : \mathcal{W} \vdash P : \mathcal{P}$ *and* $\Gamma \vdash \mathcal{W}' \leq \mathcal{W}$ *then* $\Gamma, x : \mathcal{W}' \vdash P : \mathcal{P}$.
2. *If* $\Gamma, Z \lessdot\!\# \mathcal{A}, z{:}Z \vdash P : \mathcal{P}$ *and* $\Gamma \vdash \mathcal{A}' \lessdot\!\# \mathcal{A}$ *then* $\Gamma, Z \lessdot\!\# \mathcal{A}', z{:}Z \vdash P : \mathcal{P}$.

Proof. By induction on the derivation of the first judgment in hypothesis.

Theorem 1 (Subject Reduction).
If $\Gamma \vdash P : \mathsf{Proc}[\Sigma]$ *and* $P \rightarrow Q$ *then* $\Gamma \vdash Q : \mathsf{Proc}[\Sigma]$.

Proof. By induction on the derivation of $P \rightarrow Q$*, and a case analysis on the last applied rule.*

Besides being interesting as a meta-theoretical property of the type system, subject reduction may be used to derive a type safety theorem ensuring the absence of run-time (type) errors for well-typed programs. The errors we wish to statically detect are those of the kind "message not understood"distinctive of object calculi. With the current definition of the reduction relation such errors may not arise, as not-understood messages simply block: this is somewhat unrealistic, however, as the result of sending a message to an object (a server) which does not contain a corresponding method should be (and indeed is, in real systems) reported as an error.

To state and formalize type safety, we instrument the reduction relation with an additional error reduction, state as follows:

$$a[\![\mathrm{I}\,;\, P \mid b\ \mathsf{send}\ \ell\langle M\rangle.Q\,]\!] \mid b[\![\mathrm{J}\,;\, R\,]\!] \rightarrow a[\![\mathrm{I}\,;\, P \mid \mathtt{ERR}\,]\!] \mid b[\![\mathrm{J}\,;\, R\,]\!] \qquad (\ell \notin \mathrm{J})$$

where ERR is a distinguished process, with no type. The intuitive reading of the reduction is that a not-understood message causes a local error —for the sender of that message— rather than a global error for the entire system. The rule above is meaningful also in the presence of multiple ambients with equal name, as our type system (like those of [CG99, CGG99, LS00]) ensures that ambients with the same name have also the same type.

It is easy to verify that no system containing an occurrence of `ERR` can be typed in our type system. Type safety, i.e. absence of run-time errors may now be stated follows:

Theorem 2 (Soundness). *Let P be a well-typed MA^{++} process. Then, there exist no context* $\mathbf{C}[-]$ *such that* $P \rightarrow^* \mathbf{C}[\mathtt{ERR}]$.

5 Related Work

In the literature on concurrent object-oriented programming, papers can be classified in two basic categories. The first category includes papers that provide semantics to objects by encoding them into process calculi. Examples of systematic translations of objects into the π-calculus can be found, for instance, in [Wal95, HK96, San98, KS98].

Papers in the second category propose formal calculi where primitive constructs for objects and for concurrent processes coexist. Within this class, one can further distinguish two complementary approaches. In the first, high-level object-oriented constructs are defined on top of name-passing process calculi [Vas94, PT95, FMLR00]. In the second, primitives for concurrency are built on top of imperative object calculi, in ways related to those we have discussed in this paper. Below we present a detailed discussion on papers closest to ours.

Gorgon and Hankin's concς-calculus [GH98]. The *concς*-calculus is a concurrent object calculus that results from Abadi and Cardelli's imperative object calculus by the addition of primitive constructs for parallel composition, restriction and synchronization via mutexes. Type systems for the calculus may be defined by sound extensions of existing type systems for the underlying object calculus to accommodate concurrency.

There are several similarities between *concς* and our calculus. In particular, the semantics of method invocation, based on self-substitution was directly inspired by [GH98]. As in our semantics, in [GH98] objects are explicitly named, and what gets substituted for the *self* variable is the name of the object rather then the object itself.

The fundamental difference between the work of [GH98] and ours is that *concς* does not address process mobility. In [GH98] distribution is completely disregarded, while in our framework objects may move through a hierarchy of nested locations, and communication (method invocation) often requires mobility. Moreover, due to the interplay between the dynamic nesting of ambients and the communication primitives, more method invocation styles can be modeled in our framework. A further difference is that the syntax of *concς* includes sequential composition of expressions that return results. This contrasts with the standard practice in process-based calculi [Vas94, PT95, Wal95, KS98], where the operation of returning a result is translated into sending a message on a result channel. Even though we did not explicitly address the problem of returning a result, it is easy to extend our framework by endowing agent interfaces not only with methods, but also with *fields* whose invocation returns an expression.

A distributed version of *conc*ς is studied in [Jef00], where the syntax of the calculus is enriched with a notion of *location*, and threads are allowed to migrate across locations. A basic difference with our approach is that in [Jef00] the author assumes a flat topology of locations, in which no explicit routing is required for mobility, and locations may *not* be created dynamically. Furthermore, in [Jef00] only a subset of objects (*serializable* objects) can be sent across the network, and only the so-called *located objects* can be accessed via remote threads.

The Ojeblik calculus [NHKM99]. Ojeblik is a concurrent object-based language built on top of Obliq [Car95], Cardelli's lexically scoped distributed programming language. In Ojeblik (and Obliq) object mobility is rendered by means of a migration mechanism that is accomplished by creating a copy of the object at the target site and then modifying the original (local) object such that it forwards future requests to the new (remote) object: The lexical scope rules of Obliq allow the aspects of distribution to safely be disregarded: object migration is then *correct* if the behavior of an object is transparent to whether the object has migrated or not.

Our approach is very different. As in Mobile Ambients, we assume that the process $a[\![\mathrm{I} ; P]\!]$ is an abstraction for both an agent (client) and an object (server). This implies that in our framework mobile objects move without the burden of future obligations at the source location. A client agent willing to invoke a method of a server object, in turn, must approach the server in order to start the communication protocol. In addition, while the work on Ojeblik does not address typing issues, as we do for our calculus.

6 Current and Future Work

We have defined a core calculus for distributed and mobile objects on top of which several extensions can be defined. We conclude our presentation with a discussion on some of these extensions.

Co-capabilities à la Safe Ambients. In [LS00], Levi and Sangiorgi define a variant of Mobile Ambients in which the reduction relation requires actions (i.e. capabilities) to synchronize with corresponding co-actions. To exemplify, consider the ambients $a[\![\mathsf{in}\ b.P]\!] \mid b[\![Q]\!]$. In mobile ambients, the move of a into b is "one sided" as b simply undergoes the action. In Safe Ambients, instead, the move requires mutual agreement between a and b: in order for the move to take place, Q inside b must offer the co-capability $\mathsf{coin}\ b$ to signal that it is willing to be entered. Based on this synchronization mechanisms, Levi and Sangiorgi discuss a suite of type systems for on top of which they develop a rich algebraic theory for their Safe Ambients.

Co-capabilities can be included in our calculus with no fundamental difficulty. In particular, one can include a co-capability $\mathsf{listen}\ a$, the dual of the capability

a send , whose meaning is that the ambient a is ready to serve an invocation to one of its methods. For reasons of space, we do not describe the extension in detail. Nevertheless, it is instructive to point out one of the effects of the extension, showing how it allows us to derive a simple compositional encoding of the π-calculus.

$$\begin{aligned}
\langle\!\langle\, n?(\boldsymbol{x}).P \,\rangle\!\rangle &\triangleq (\boldsymbol{\nu} p)(n\llbracket\, ch(\boldsymbol{x}) \triangleright p\llbracket\, \mathsf{out}\ n.\mathsf{coopen}\ p.\langle\!\langle\, P \,\rangle\!\rangle \,\rrbracket\ ;\ \mathsf{listen}\ n.\mathsf{coout}\ n\,\rrbracket \mid \mathsf{open}\ p) \\
\langle\!\langle\, n!\langle \boldsymbol{x} \rangle \,\rangle\!\rangle &\triangleq n\ \mathsf{downsend}\ ch\langle \boldsymbol{x} \rangle \\
\langle\!\langle\, (\boldsymbol{\nu x})P \,\rangle\!\rangle &\triangleq (\boldsymbol{\nu x})\langle\!\langle\, P \,\rangle\!\rangle \\
\langle\!\langle\, P \mid Q \,\rangle\!\rangle &\triangleq \langle\!\langle\, P \,\rangle\!\rangle \mid \langle\!\langle\, Q \,\rangle\!\rangle \\
\langle\!\langle\, !P \,\rangle\!\rangle &\triangleq !\langle\!\langle\, P \,\rangle\!\rangle \\
\langle\!\langle\, \mathbf{0} \,\rangle\!\rangle &\triangleq \mathbf{0} \\
\langle\!\langle\, n \,\rangle\!\rangle &\triangleq n
\end{aligned}$$

Every input on a channel n generates a new ambient named n, waiting to synchronize with an output on n. Having received input, the transport ambient p carries (the encoding of) P out of n. Once outside n, p is dissolved and the continuation process P unleashed. Notice that the ambient n is left without capabilities after having let the transport p out. As such, after synchronization, n is unavailable for interactions with the context, and thus behaviorally equivalent to the null process (which can be garbage collected). Also, the encoding can be shown to be interference-free, as the use of co-capabilities allows the definition of an interference-free encoding of output construct of the π-calculus, based on downward method invocation.

Other Extensions. Further extensions to the core calculus include the addition of fields and refinements of the type system.

In object calculi, fields are often represented as parameter-less methods, that do not depend on self. This direct representation is not possible in our calculus, as invoking a method spawns a process rather than returning a value, as one would expect from selecting a field. Nevertheless, it is not difficult of explicitly include new syntax for fields, and extend the reduction relation so that selecting a field returns a term rather than triggering a process.

A different extension is to allow method names to be treated as ordinary names. This would allow one to restrict them, thus obtaining private methods, and to communicate them, thus obtaining dynamic messages. This is a straightforward modification in the untyped calculus but it is quite problematic in the typed case since the possibility of communicating method names would naturally give rise to dependent types.

These extensions, together with the study of type-driven security in the calculus are subject of our current and future work.

References

[AC96] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer, 1996. 243

[BB99] V. Bono and M. Bugliesi. Matching for the Lambda Calculus of Objects. *Theoretical Computer Science*, 212(1/2):101–140, Feb. 1999. 248

[BC00] M. Bugliesi and G. Castagna. Mobile objects. In *7th Workshop on Foundation of Object-Oriented Languages*, Boston, 2000. Electronic Proceedings. 235, 236

[BCC00] M. Bugliesi, G. Castagna, and S. Crafa. Typed mobile objects. In *Proceedings of CONCUR 2000 (11th. International Conference on Concurrency Theory)*, number 1877 in Lecture Notes in Computer Science, pages 504–520. Springer, 2000. 235, 236, 240

[Bru94] B. Bruce, K. A paradigmatic object-oriented programming language: Design, static typing and semantics. *Journal of Functional Programming*, 1(4):127–206, 1994. 246, 247

[Car95] L. Cardelli. A language with distributed scope. *Computing Systems*, 8(1):27–59, 1995. 251

[Car99] L. Cardelli. Abstractions for mobile computations. In *Secure Internet Programming*, number 1603 in Lecture Notes in Computer Science, pages 51–94. Springer, 1999. 236, 238

[CG98] L. Cardelli and A. Gordon. Mobile ambients. In *Proceedings of POPL '98*. ACM Press, 1998. 235, 236, 244

[CG99] L. Cardelli and A. Gordon. Types for mobile ambients. In *Proceedings of POPL '99*, pages 79–92. ACM Press, 1999. 244, 250

[CGG99] L. Cardelli, G. Ghelli, and A. Gordon. Mobility types for mobile ambients. In *Proceedings of ICALP '99*, number 1644 in Lecture Notes in Computer Science, pages 230–239. Springer, 1999. 250

[FHM94] K. Fisher, F. Honsell, and J. C. Mitchell. A Lambda Calculus of Objects and Method Specialization. *Nordic Journal of Computing*, 1(1):3–37, 1994. 238, 243, 247

[FMLR00] Cédric Fournet, Luc Maranget, Cosimo Laneve, and Didier Rémy. Inheritance in the Join Calculus. In *Foundations of Software Technology and Theoretical Computer Science*, volume 1974 of *Lecture Notes in Computer Science*. Springer, December 2000. 250

[GH98] A. Gordon and P. D Hankin. A concurrent object calculus: reduction and typing. *In Proceedings HLCL '98, Elsevier ENTC*, 1998. Also Technical Report 457, University of Cambridge Computer Laboratory, February 1999. 250

[HK96] H. Huttel and J. Kleist. Objects as mobile processes. Technical Report Research Series RS-96-38, BRICS, 1996. Presented at MFPS '96. 250

[Jef00] A. Jeffrey. A distributed object calculus. In *7th Workshop on Foundation of Object-Oriented Languages*, Boston, 2000. Electronic Proceedings. 251

[KS98] J. Kleist and D. Sangiorgi. Imperative objects and mobile processes. In *PROCOMET '98 (IFIP Working Conference on Programming Concepts and Methods)*. North-Holland, 1998. 250

[LS00] F. Levi and D. Sangiorgi. Controlling interference in Ambients. In *POPL '00*, pages 352–364. ACM Press, 2000. 250, 251

[NHKM99] U Nestmann, H. Huttel, J. Kleist, and M. Merro. Aliasing models for object migration. In *Proceedings of Euro-Par '99*, number 1685 in Lecture Notes in Computer Science, pages 1353–1368. Springer, 1999. 251

[PT95] B. C. Pierce and D. N. Turner. Concurrent objects in a process calculus. In Takayasu Ito and Akinori Yonezawa, editors, *Theory and Practice of Parallel Programming, Sendai, Japan (Nov. 1994)*, number 907 in Lecture Notes in Computer Science, pages 187–215. Springer, April 1995. 250

[San98] D. Sangiorgi. An interpretation of typed objects into typed π-calculus. *Information and Computation*, 143(1):34–73, 1998. 250

[Vas94] V. T. Vasconcelos. Typed concurrent objects. In M. Tokoro and R. Pareschi, editors, *ECOOP '94*, number 821 in Lecture Notes in Computer Science, pages 100–117. Springer, 1994. 250

[Wal95] D.J Walker. Objects in the π-calculus. *Information and Computation*, 116(2):253–271, 1995. 250

A Typing Rules

Context formation

$$(\textsc{Env-empty})\ \frac{}{\varnothing \vdash \diamond} \qquad (\textsc{Env-}x)\ \frac{\Gamma \vdash \mathcal{W} \quad x \notin \mathit{Dom}(\Gamma)}{\Gamma, x : \mathcal{W} \vdash \diamond} \qquad (\textsc{Env-}X)\ \frac{\Gamma \vdash \diamond \quad X \notin \mathit{Dom}(\Gamma)}{\Gamma, X \mathrel{<\!\!\#} \mathcal{A} \vdash \diamond}$$

Type formation

$$(\textsc{Type X})\ \frac{\Gamma, X \mathrel{<\!\!\#} \mathcal{A}, \Gamma' \vdash \diamond}{\Gamma, X \mathrel{<\!\!\#} \mathcal{A}, \Gamma' \vdash X} \qquad (\textsc{Type Amb})\ \frac{\Gamma \vdash \diamond}{\Gamma \vdash \mathsf{Amb}[\Sigma]} \qquad (\textsc{Type Cap})\ \frac{\Gamma \vdash \diamond}{\Gamma \vdash \mathsf{Cap}[\Sigma]} \qquad (\textsc{Type Proc})\ \frac{\Gamma \vdash \diamond}{\Gamma \vdash \mathsf{Proc}[\Sigma]}$$

Matching : Reflexivity, Transitivity and the following

$$(\textsc{Match X})\ \frac{\Gamma, X \mathrel{<\!\!\#} \mathcal{A}, \Gamma' \vdash \diamond}{\Gamma, X \mathrel{<\!\!\#} \mathcal{A}, \Gamma' \vdash X \mathrel{<\!\!\#} \mathcal{A}} \qquad (\textsc{Match Amb})\ \frac{\Gamma \vdash \diamond}{\Gamma \vdash \mathsf{Amb}[(\ell_i(\mathcal{V}_i))^{i \in 1..n+k}] \mathrel{<\!\!\#} \mathsf{Amb}[(\ell_i(\mathcal{V}_i))^{i \in 1..n}]}$$

Subtyping and subsumption : Reflexivity, Transitivity and the following

$$(\textsc{Sub Cap})\ \frac{\Sigma \subseteq \Sigma'}{\mathsf{Cap}[\Sigma] \leq \mathsf{Cap}[\Sigma']} \qquad (\textsc{Sub Proc})\ \frac{\Sigma \subseteq \Sigma'}{\mathsf{Proc}[\Sigma] \leq \mathsf{Proc}[\Sigma']} \qquad (\textsc{Subsumption})\ \frac{\Gamma \vdash A : \mathcal{T} \quad \mathcal{T} \leq \mathcal{T}'}{\Gamma \vdash A : \mathcal{T}'}$$

Expressions

$$(\textsc{name/var})\ \frac{\Gamma \vdash \diamond}{\Gamma \vdash x : \Gamma(x)} \qquad (\varepsilon)\ \frac{\Gamma \vdash \diamond}{\Gamma \vdash \varepsilon : \mathsf{Cap}[\Sigma]} \qquad (\textsc{path})\ \frac{\Gamma \vdash M_1 : \mathsf{Cap}[\Sigma] \quad \Gamma \vdash M_2 : \mathsf{Cap}[\Sigma]}{\Gamma \vdash M_1.M_2 : \mathsf{Cap}[\Sigma]}$$

$$\text{(OPEN)}\quad \frac{\mathit{\Gamma} \vdash a : \mathsf{Amb}[\mathit{\Sigma}]}{\mathit{\Gamma} \vdash \mathsf{open}\ a : \mathsf{Cap}[\mathit{\Sigma}]} \qquad \text{(INOUT)}\quad \frac{\mathit{\Gamma} \vdash M : \mathcal{W} \quad \mathit{\Gamma} \vdash \mathcal{W} \mathrel{\lessdot\#} \mathsf{Amb}[\mathit{\Sigma}] \quad (M' \in \{\mathsf{in}\ M, \mathsf{out}\ M\})}{\mathit{\Gamma} \vdash M' : \mathsf{Cap}[\mathit{\Sigma}']}$$

$$\text{(Message)}\quad \frac{\mathit{\Gamma} \vdash a : \mathcal{W} \quad \mathit{\Gamma} \vdash \mathcal{W} \mathrel{\lessdot\#} \mathsf{Amb}[\ \ell(\mathcal{V}')\ ] \quad \mathit{\Gamma} \vdash M' : \mathcal{V}'}{\mathit{\Gamma} \vdash a\ \mathsf{send}\ \ell\langle M' \rangle : \mathsf{Cap}[\mathit{\Sigma}]}$$

Processes

$$\text{(PREF)}\quad \frac{\mathit{\Gamma} \vdash M : \mathsf{Cap}[\mathit{\Sigma}] \quad \mathit{\Gamma} \vdash P : \mathsf{Proc}[\mathit{\Sigma}]}{\mathit{\Gamma} \vdash M.P : \mathsf{Proc}[\mathit{\Sigma}]} \qquad \text{(PAR)}\quad \frac{\mathit{\Gamma} \vdash P : \mathsf{Proc}[\mathit{\Sigma}] \quad \mathit{\Gamma} \vdash Q : \mathsf{Proc}[\mathit{\Sigma}]}{\mathit{\Gamma} \vdash P \mid Q : \mathsf{Proc}[\mathit{\Sigma}]}$$

$$\text{(RESTR)}\quad \frac{\mathit{\Gamma}, x{:}\mathcal{A} \vdash P : \mathsf{Proc}[\mathit{\Sigma}]}{\mathit{\Gamma} \vdash (\boldsymbol{\nu} x{:}\mathcal{A})P : \mathsf{Proc}[\mathit{\Sigma}]} \qquad \text{(DEAD)}\quad \frac{\mathit{\Gamma} \vdash \diamond}{\mathit{\Gamma} \vdash \mathbf{0} : \mathsf{Proc}[\mathit{\Sigma}]}$$

$$\text{(Amb)}\qquad (\mathit{\Sigma} = (\ \ell_i(\mathcal{V}_i)\ )^{i \in I})$$

$$\frac{\mathit{\Gamma} \vdash a : \mathsf{Amb}[\mathit{\Sigma}] \quad \mathit{\Gamma}, Z \mathrel{\lessdot\#} \mathsf{Amb}[\mathit{\Sigma}], z{:}Z, x_i{:}\mathcal{V}_i \vdash P_i : \mathsf{Proc}[\mathit{\Sigma}] \quad \mathit{\Gamma} \vdash P : \mathsf{Proc}[\mathit{\Sigma}]}{\mathit{\Gamma} \vdash a[(\ell_i(x_i) \triangleright \varsigma(z)P_i)\ ^{i \in I}\ ;\ P] : \mathsf{Proc}[\mathit{\Sigma}']}$$

On Synchronous and Asynchronous Communication Paradigms

Diletta Cacciagrano and Flavio Corradini

Dipartimento di Matematica Pura ed Applicata, University of L'Aquila
Italy
{cacciagr,flavio}@univaq.it

Abstract. The π-calculus, its asynchronous version and Boudol's mapping from the former language to the latter one are well-known mathematical objects in theoretical computer science. It is also well-known that the mapping is not fully-abstract w.r.t. most of the semantics defined over these two languages.
In this paper we study and fix conditions on the existance of fully-abstract results for Boudol's mapping (and its variants). The testing theories à la De Nicola-Hennessy turned out to be very useful tools for such a purpose.

1 Introduction

Concurrent and distributed systems use communication as a means to exchange information. Communication can be of two kinds: *synchronous* and *asynchronous*. A communication is synchronous when sending and receiving information between a sender and a receiver are simultaneous events. A communication is asynchronous when sending and receiving information between a sender and a receiver do not necessarily happen at the same time instant.

The π-calculus [MPW92] implements a synchronous communication while the asynchronous π-calculus [Bou92] implements an asynchronous one. Since the latter language is essentially a subset of the former one, the natural question is whether or not the π-calculus can be somehow encoded into its asynchronous subset. This would mean that the synchronous communication can be "implemented" via asynchronous communication. Boudol's mapping [Bou92] goes in such a direction. It views the synchronous communication as a sequence of asynchronous communications (a possible "simulation" of the synchronous communication).

If we denote with $\mathcal{P}_s$ the π-calculus, with $\mathcal{P}_a$ the asynchronous π-calculus, with $[\![\_]\!]$ the Boudol's mapping and with $\mathcal{R}$ a generic equivalence generated by a semantic theory, then it is well-known that $[\![\_]\!]$ (typically) does not preserve $\mathcal{R}$; i.e., the following statement

$$\forall P, Q \in \mathcal{P}_s, \quad P \, \mathcal{R} \, Q \text{ \textit{if and only if} } [\![P]\!] \, \mathcal{R} \, [\![Q]\!] \tag{1}$$

does not hold (even for "non-severe" equivalences such as language equivalence). An evidence of this fact can be found in [Bou92], where the author proves

A. Restivo, S. Ronchi Della Rocca, L. Roversi (Eds.): ICTCS 2001, LNCS 2202, pp. 256–268, 2001.

only the adequacy of the mapping (*if* implication) with respect to the Morris' preorder. The other implication does not hold.

This paper is still concerned with Boudol's mapping from the π-calculus to the asynchronous π-calculus. We study and fix conditions on the considered languages and mapping in such a way that the statement (1) holds in both directions. In such a case we will say that $[\![\_]\!]$ preserves $\mathcal{R}$ or $[\![\_]\!]$ is fully-abstract with respect to $\mathcal{R}$.

The testing semantics à la De Nicola-Hennessy [DH84] are particularly suitable to the present study. Of course, as it stated, there are counterexamples for the above statement whichever testing semantics is taken into account. Before going into the details, we briefly recall the main assumptions behind the testing scenario. It resorts on (i) a set $\mathcal{P}$ of processes to be tested (here concentrate on the π-calculus and the asynchronous π-calculus), (ii) a set $\mathcal{O}$ of tests or observers (these are processes that can perform a particular action ω reporting success), (iii) a way to exercise a process on a given test (obtained by letting the process and the observer to run in parallel and by looking at the computations which this embedded process can perform. These computations can be successful or failing, depending on whether or not they allow the execution of action ω) and (iv) a general criterion for interpreting the results of these exercises. Different criteria have been defined which provide $\mathcal{P}$ processes with different semantics. For a given process P and observer o,

- P *may* o if there exists a successful computation between P and o;
- P *must* o if every computation between P and o is successful;
- P *fair* o (proposed in [BRV95, NC95]) if each state of every computation between P and o leads to success after finitely many interactions.

Each criterion above allows the natural definition of a corresponding preorder over $\mathcal{P}$. For any P and Q, $\mathcal{P}$ processes:

- $P \sqsubseteq_{may} Q$ if and only if for each $o \in \mathcal{O}$, P *may* o implies Q *may* o;
- $P \sqsubseteq_{must} Q$ if and only if for each $o \in \mathcal{O}$, P *must* o implies Q *must* o;
- $P \sqsubseteq_{fair} Q$ if and only if for each $o \in \mathcal{O}$, P *fair* o implies Q *fair* o.

As already said, according to these testing theories, counterexamples can be found for the statement in (1). They are reported in full details in Section 5.

We now show (first) how the testing theories can be refined in order to get a fully-abstract result (then we have to operate also at the language level). The key idea is given by the following statement:

$$P \textit{ satisfies } o \text{ iff } [\![P]\!] \textit{ satisfies } [\![o]\!] \tag{2}$$

where *satisfies* can be either *may*, *must* or *fair*. This means that a process P can reach a successful state, when exercised on a test o, if and only if $[\![P]\!]$ can reach a successful state, when exercised on a test $[\![o]\!]$.

Though this idea can appear quite intuitive we show that it is not trivial at all. Indeed, this statement holds for the *may* testing relation and for the *fair* one, while it does not scale to the *must* testing. This is due to the presence of possible divergent computations in the source programs and to the fact that an (atomic) synchronous communication between a sender and a receiver is implemented as a non atomic sequence of asynchronous communications. More in detail, the execution of this sequence may lead the system to states in which a sender process can proceed its execution while the corresponding receiver partner is still involved in simulating the synchronous communication. A phatological situation is when the receiver has the ability to perform the success action, after the completion of the simulation, while the sender can engage in a divergent computation. In such a case, however, the receiver has always the potential to perform the success action after finitely many interactions. This is mainly the reason why the *fair* relation holds (2).

(2) suggests us to consider parameterized versions of the testing theories with respect to sets of observers. Formally, for a given set of observers $O \subseteq \mathcal{O}$,

- $P \sqsubseteq_{\mathit{may}}^{O} Q$ if and only if for each $o \in O$, $P\ \mathit{may}\ o$ implies $Q\ \mathit{may}\ o$;
- $P \sqsubseteq_{\mathit{must}}^{O} Q$ if and only if for each $o \in O$, $P\ \mathit{must}\ o$ implies $Q\ \mathit{must}\ o$;
- $P \sqsubseteq_{\mathit{fair}}^{O} Q$ if and only if for each $o \in O$, $P\ \mathit{fair}\ o$ implies $Q\ \mathit{fair}\ o$.

Of course any parameterized preorder coincides with the original one when O coincides with $\mathcal{O}$ itself. According to these new testing preorders and the observations above, we have the following results:

- $P \sqsubseteq_{\mathit{may}}^{O} Q$ iff $\llbracket P \rrbracket \sqsubseteq_{\mathit{may}}^{\llbracket O \rrbracket} \llbracket Q \rrbracket$;
- $P \sqsubseteq_{\mathit{fair}}^{O} Q$ iff $\llbracket P \rrbracket \sqsubseteq_{\mathit{fair}}^{\llbracket O \rrbracket} \llbracket Q \rrbracket$:
- $P \sqsubseteq_{\mathit{must}}^{O} Q$ if $\llbracket P \rrbracket \sqsubseteq_{\mathit{must}}^{\llbracket O \rrbracket} \llbracket Q \rrbracket$ (but not the other way round).

Apart from this problem with must testing our study gives some insight on the reasons why the statement in (1) cannot hold for the original versions of the testing preorders. The set of processes in the asynchronous π-calculus which are mapping of some process in the π-calculus, indeed, is a strict subset of the whole language. Thus testing a process $\llbracket P \rrbracket$ with respect to a test which is not the coding of any process in the π-calculus means testing $\llbracket P \rrbracket$ over a set of tests which is "more powerful" than that available for testing P.

In order to have a fully-abstract result for the must case, we restrict the source language by considering only those terms which are divergent-free; that is, those terms that can perform only finite internal computations.

The rest of the paper is organized as follows. The next section briefly recalls a few basic notions; namely, the π-calculus and the asynchronous π-calculus. Section 3 presents the testing preorders of De Nicola and Hennessy, as well as their parameterized versions, and Section 4 presents Boudol's mapping from the π-calculus into the asynchronous π-calculus. Section 5, the core of the paper, studies fully-abstract results of the mapping. Section 6 contrasts our work with related ones while Section 7 contains a few concluding remarks and further work.

This paper is an abridged version of [CC00], where the reader can find all the proofs not included in the body of this extended abstract.

2 The π-Calculus and the Asynchronous π-Calculus

Let $\mathcal{N}$ (ranged over by $x, y, z, \ldots$) be a set of names. The set $\mathcal{P}_s$ of π-terms is generated by the following (two level) grammar:

$$\begin{aligned} P &::= \bar{x}y.P \;\Big|\; P \mid P \;\Big|\; (\nu x)P \;\Big|\; !\,P \;\Big|\; G \\ G &::= \;0 \;\Big|\; x(y).P \;\Big|\; \tau.P \;\Big|\; G + G \end{aligned}$$

Terms in $\mathcal{P}_s$ are usually called processes. Input prefix, $y(x).P$, and restriction, $(\nu x)P$, act as name binders for name x in P. Consequently, the notions of free names, $fn(\_)$, bound names, $bn(\_)$, over process terms are as expected. The set of names of process terms, $n(\_)$, is defined as $n(\_) = fn(\_) \cup bn(\_)$.

The operational semantics of processes is given via labelled transition systems. The states of such transition systems are $\mathcal{P}_s$ terms. The labels (ranged over by $\mu, \gamma, \ldots$) correspond to prefixes, input $x\langle y\rangle$, output $\bar{x}y$ and tau τ, and bounded output $\bar{x}\langle y\rangle$ (which models scope extrusion). If $\mu = x\langle y\rangle$ or $\mu = \bar{x}y$ or $\mu = \bar{x}\langle y\rangle$ we let $sub(\mu) = x$ and $obj(\mu) = y$. Functions $fn(\_)$, $bn(\_)$ and $n(\_)$ are extended to cope with labels as follows:

$$\begin{array}{llll} bn(x\langle y\rangle) = \{y\} & bn(\bar{x}\langle y\rangle) = \{y\} & bn(\bar{x}y) = \;\;\emptyset & bn(\tau) = \emptyset \\ fn(x\langle y\rangle) = \{x\} & fn(\bar{x}\langle y\rangle) = \{x\} & fn(\bar{x}y) = \{x, y\} & fn(\tau) = \emptyset \end{array}$$

The transition relation defining the transitional semantics of processes is given in Table 1. $\equiv$, used in Rule Cong, stands for the structural congruence over set $\mathcal{P}_s$ induced by the axioms and inference rules in Table 2.

Notation: $\langle P\rangle$, where $P \in \mathcal{P}_s$, stands for P with some restrictions at the top level; i.e., $\langle P\rangle$ denotes $(\nu x_1)(\nu x_2)\ldots(\nu x_n)P$ for some $x_1, x_2, \ldots, x_n \in \mathcal{N}$ $(n \geq 0)$.

Definition 1. (*Weak Transitions*) Let P and Q be $\mathcal{P}_s$ processes. Then:

- $P \overset{\varepsilon}{\Longrightarrow} Q$ if and only if $P = P_0 \xrightarrow{\tau} P_1 \xrightarrow{\tau} \ldots \xrightarrow{\tau} P_n = Q$ for some $n \geq 0$ and $P_0, P_1, \ldots, P_n \in \mathcal{P}_s$;
- $P \overset{\mu}{\Longrightarrow} Q$ if and only if $P \overset{\varepsilon}{\Longrightarrow} P_1 \xrightarrow{\mu} P_2 \overset{\varepsilon}{\Longrightarrow} Q$ for some P_1, $P_2 \in \mathcal{P}_s$.

Notation: Sometimes we write $P \xrightarrow{\mu}$ $(P \overset{\mu}{\Longrightarrow})$ to mean that there exists P' such that $P \xrightarrow{\mu} P'$ $(P \overset{\mu}{\Longrightarrow} P')$ and write $P \overset{\varepsilon}{\Longrightarrow}\xrightarrow{\mu}$ to mean that there are P' and Q such that $P \overset{\varepsilon}{\Longrightarrow} P'$ and $P' \xrightarrow{\mu} Q$.

The asynchronous π-calculus [HT91, Bou92] is the set $\mathcal{P}_a$ of terms generated by the following grammar:

$$\begin{aligned} P &::= \bar{x}y \;\Big|\; P \mid P \;\Big|\; (\nu x)P \;\Big|\; !\,P \;\Big|\; G \\ G &::= \;0 \;\Big|\; x(y).P \;\Big|\; \tau.P \;\Big|\; G + G \end{aligned}$$

The operational semantics of $\mathcal{P}_a$ is given by the rules in Table 1, when rule Output/Tau is replaced by rules Output and Tau in Table 3. The axioms defining the structural congruence are the same as the ones in Table 2. Similar definitions and notation already given in the synchronous setting are assumed in the asynchronous one. Note that the $\mathcal{P}_a$ calculus is a proper sub-set of $\mathcal{P}_s$ since the output-action process $\bar{x}y$ can be thought as a special case of output prefix $\bar{x}y.0$.

Table 1. Early operational semantics for $\mathcal{P}_s$ terms

$$\text{Input} \quad x(y).P \xrightarrow{x\langle z\rangle} P\{z/y\} \ \text{where}\, x,\ y \in \mathcal{N}$$

$$\text{Output/tau} \quad \alpha.P \xrightarrow{\alpha} P \ \text{where}\, \alpha = \bar{x}y \text{ or } \alpha = \tau$$

$$\text{Open} \quad \frac{P \xrightarrow{\bar{x}y} P'}{(\nu y)P \xrightarrow{\bar{x}\langle y\rangle} P'}\, x \neq y \qquad \text{Res} \quad \frac{P \xrightarrow{\mu} P'}{(\nu y)P \xrightarrow{\mu} (\nu y)P'}\, y \notin nm(\mu)$$

$$\text{Par} \quad \frac{P \xrightarrow{\mu} P'}{P \,|\, Q \xrightarrow{\mu} P' \,|\, Q}\, bn(\mu) \cap fn(Q) = \emptyset$$

$$\text{Com} \quad \frac{P \xrightarrow{x\langle y\rangle} P',\ Q \xrightarrow{\bar{x}y} Q'}{P \,|\, Q \xrightarrow{\tau} P' \,|\, Q'} \qquad \text{Close} \quad \frac{P \xrightarrow{x\langle y\rangle} P',\ Q \xrightarrow{\bar{x}\langle y\rangle} Q'}{P \,|\, Q \xrightarrow{\tau} (\nu y)(P' \,|\, Q')}$$

$$\text{Sum} \quad \frac{P \xrightarrow{\mu} P'}{P + Q \xrightarrow{\mu} P'} \qquad \text{Bang} \quad \frac{P \xrightarrow{\mu} P'}{!P \xrightarrow{\mu} P' \,|\, !P}$$

$$\text{Cong} \quad \frac{P \equiv P' \quad P' \xrightarrow{\mu} Q' \quad Q' \equiv Q}{P \xrightarrow{\mu} Q}$$

3 Testing Preorders

We now briefly summarize the basic definitions behind the testing machinery to the π-calculus and the asynchronous π-calculus. $\mathcal{P}$ denotes either $\mathcal{P}_s$ or $\mathcal{P}_a$.

Definition 2. (*Observers*)

- Let $N' = N \cup \{\omega\}$ be the set of names. By convention we let $fn(\omega) = \omega$, $bn(\omega) = \emptyset$ and $sub(\omega) = \omega$. Action ω is used to report success;
- The set $\mathcal{O}$ (ranged over by $o, o', o'', \ldots$) of observers is defined like $\mathcal{P}$, where the grammar with non terminal P has extended with production $P ::= \omega.P$;
- The operational semantics of $\mathcal{P}$ extends to $\mathcal{O}$ by adding rule: $\omega.o \xrightarrow{\omega} o$.

Definition 3. (*Experiments*) The set of experiments $\mathcal{E}$ is the set $\{\, P \,|\, o \,|\, P \in \mathcal{P}$ and $o \in \mathcal{O}\}$.

Definition 4. (*Maximal Computation*) Given an experiment $P \,|\, o \in \mathcal{E}$, a maximal computation from $P \,|\, o$ is an infinite sequence

Table 2. The structural congruence

a_1)	$P \equiv Q$ iff Q can be obtained from P by alpha-renaming
a_2)	$(\mathcal{P}_s/\equiv, \mid, 0)$ is a commutative monoid
a_3)	$(\mathcal{P}_s/\equiv, +, 0)$ is a commutative monoid
a_4)	$(P + P) \equiv P$
a_5)	$!P \equiv P \mid !P$
a_6)	$((\nu x)P \mid Q) \equiv (\nu x)(P \mid Q),$ if $x \notin fn(Q)$
a_7)	$(\nu x)P \equiv P,$ if $x \notin fn(P)$
a_8)	$(\nu x)(\nu y)P \equiv (\nu y)(\nu x)P$
a_9)	$((\nu x)P + (\nu x)Q) \equiv (\nu x)(P + Q)$

Table 3. The rules for Output and Tau in $\mathcal{P}_a$

Output	$\bar{x}y \xrightarrow{\bar{x}y} 0$	Tau	$\tau.P \xrightarrow{\tau} P$

$$P \mid o = \langle P_0 \mid o_0 \rangle \xrightarrow{\tau} \langle P_1 \mid o_1 \rangle \xrightarrow{\tau} \langle P_2 \mid o_2 \rangle \xrightarrow{\tau} \dots$$

or a finite sequence $P \mid o = \langle P_0 \mid o_0 \rangle \xrightarrow{\tau} \langle P_1 \mid o_1 \rangle \xrightarrow{\tau} \dots \xrightarrow{\tau} \langle P_n \mid o_n \rangle$ such that $n \geq 0$ and $\langle P_n \mid o_n \rangle \not\xrightarrow{\tau}$.

Definition 5. (*May, Must and Fair Relations*) Given a process $P \in \mathcal{P}$ and an observer $o \in \mathcal{O}$, define:

- P *may* o if and only if *there exists* a maximal computation
$$P \mid o = \langle P_0 \mid o_0 \rangle \xrightarrow{\tau} \langle P_1 \mid o_1 \rangle \xrightarrow{\tau} \dots \langle P_i \mid o_i \rangle \xrightarrow{\tau} \dots$$
such that $\langle P_i \mid o_i \rangle \xrightarrow{\omega}$, for some $i \geq 0$;
- P *must* o if and only if *for every* maximal computation
$$P \mid o = \langle P_0 \mid o_0 \rangle \xrightarrow{\tau} \langle P_1 \mid o_1 \rangle \xrightarrow{\tau} \dots \langle P_i \mid o_i \rangle \xrightarrow{\tau} \dots$$
there exists $i \geq 0$ such that $\langle P_i \mid o_i \rangle \xrightarrow{\omega}$;
- P *fair* o if and only if *for every* maximal computation
$$P \mid o = \langle P_0 \mid o_0 \rangle \xrightarrow{\tau} \langle P_1 \mid o_1 \rangle \xrightarrow{\tau} \dots \langle P_i \mid o_i \rangle \xrightarrow{\tau} \dots$$
$\langle P_i \mid o_i \rangle \overset{\omega}{\Longrightarrow}$, for every $i \geq 0$.

Definition 6. (*Testing Preorders*) Given two processes $P, Q \in \mathcal{P}$ and a set of observers $O \subseteq \mathcal{O}$, define:

- $P \sqsubseteq^{O}_{may} Q$ if and only if for every $o \in O$, P *may* o implies Q *may* o;
- $P \sqsubseteq^{O}_{must} Q$ if and only if for every $o \in O$, P *must* o implies Q *must* o;
- $P \sqsubseteq^{O}_{fair} Q$ if and only if for every $o \in O$, P *fair* o implies Q *fair* o.

4 Coding the π-Calculus into the Asynchronous π-Calculus

This section recalls the coding from the π-calculus to the asynchronous π-calculus [Bou92] and states some useful properties.

Definition 7. (*Coding $\mathcal{P}_s$ into $\mathcal{P}_a$*) The mapping $[\![\ ]\!] : \mathcal{P}_s \longmapsto \mathcal{P}_a$ has the following basic clauses:

$$[\![\bar{x}z.P]\!] = (\nu u)(\bar{x}u \mid u(v).(\bar{v}z \mid [\![P]\!])), \text{ where } u, v \notin fn(P)$$
$$[\![x(y).P]\!] = x(u).(\nu v)(\bar{u}v \mid v(y).[\![P]\!]), \text{ where } u, v \notin fn(P)$$

the others are defined extending $[\![]\!]$ homomorphically over all the other operators.

We now state two key properties relating (strong and weak) transitions out of terms in $\mathcal{P}_s$ and those out of their translations. The proof is not conceptually difficult but involved in the details (see [CC00]).

Proposition 1. Let P be a $\mathcal{P}_s$ process. Then

(i) $P \xrightarrow{\mu}$ if and only if $[\![P]\!] \xrightarrow{\gamma}$, where $sub(\mu) = sub(\gamma)$;
(ii) $P \overset{\varepsilon}{\Longrightarrow}\xrightarrow{\mu}$ if and only if $[\![P]\!] \overset{\varepsilon}{\Longrightarrow}\xrightarrow{\gamma}$, where $sub(\mu) = sub(\gamma)$.

5 Fully-Abstract Results of the Coding

Let $\mathcal{P}$ and $\mathcal{P}'$ be two languages and $[\![\_]\!]$ be a coding from the former to the latter language. Let $\mathcal{R}$ be an equivalence generated by a semantic theory. We say that the coding is fully-abstract w.r.t. $\mathcal{R}$ if and only if

$$\forall P, Q \in \mathcal{P}, \quad P\,\mathcal{R}\,Q \text{ if and only if } [\![P]\!]\,\mathcal{R}\,[\![Q]\!]\,.$$

When considering the case $\mathcal{P} = \mathcal{P}_s$, $\mathcal{P}' = \mathcal{P}_a$ and Boudol's coding we have a negative result. Typically only the if implication holds. In order to have fully-abstract results we reduce the expressive power of the languages and refine the considered semantics.

5.1 Full Abstraction of the Coding W.R.T. $\sqsubseteq^{O}_{may}$

We start by considering the $\sqsubseteq^{O}_{may}$ preorder. We prove a fully- abstract result for our coding w.r.t. $\sqsubseteq^{O}_{may}$. In particular we prove that two $\mathcal{P}_s$ processes P and Q are related by $\sqsubseteq^{O}_{\mathrm{may}}$, for a set of tests O, if and only if their translations $[\![P]\!]$ and $[\![Q]\!]$ are related by $\sqsubseteq^{[\![O]\!]}_{\mathrm{may}}$. We first need a preliminary result.

Proposition 2. Let P be a $\mathcal{P}_s$ process and $O \subseteq \mathcal{O}$ be a set of observers. Then, for every $o \in O$, P *may* o if and only if $[\![P]\!]$ *may* $[\![o]\!]$.

Proof. We just prove the *if* implication, since the *only if* one is completely similar. Assume $\llbracket P \rrbracket$ *may* $\llbracket o \rrbracket$. By definition, $\llbracket P \rrbracket$ *may* $\llbracket o \rrbracket$ if and only if there exists a maximal computation

$$\llbracket P \rrbracket \mid \llbracket o \rrbracket = \langle T_0 \mid o_0 \rangle \xrightarrow{\tau} \langle T_1 \mid o_1 \rangle \xrightarrow{\tau} \ldots \langle T_i \mid o_i \rangle \xrightarrow{\tau} \ldots$$

such that $\langle T_i \mid o_i \rangle \xrightarrow{\omega}$, for some $i \geq 0$. By Definition 1 we can also write $\llbracket P \rrbracket \mid \llbracket o \rrbracket \overset{\varepsilon}{\Longrightarrow} \xrightarrow{\omega}$. By Proposition 1, $\llbracket P \rrbracket \mid \llbracket o \rrbracket = \llbracket P \mid o \rrbracket \overset{\varepsilon}{\Longrightarrow} \xrightarrow{\omega}$ if and only if $P \mid o \overset{\varepsilon}{\Longrightarrow} \xrightarrow{\omega}$. Thus P *may* o.

From the previous proposition we have the expected result for may testing.

Theorem 1. (*Full abstraction of the coding w.r.t.* $\sqsubseteq^{O}_{may}$)
Let P and Q be $\mathcal{P}_s$ processes and $O \subseteq \mathcal{O}$ be a set of observers. Then,

$$P \sqsubseteq^{O}_{may} Q \text{ if and only if } \llbracket P \rrbracket \sqsubseteq^{\llbracket O \rrbracket}_{may} \llbracket Q \rrbracket.$$

Remark: The fully-abstract result in Theorem 1 holds when the observers used to test $\llbracket P \rrbracket$ and $\llbracket Q \rrbracket$ in the asynchronous setting are the translations of the observers used to test P and Q in the synchronous one.

The above condition is strictly needed, since if we allow observers not in the set $\llbracket O \rrbracket$ (i.e., consider a more "powerful set" of observers) to test $\llbracket P \rrbracket$ and $\llbracket Q \rrbracket$ then our fully- abstract result does not hold anymore. Intuitively, all that is reasonable: indeed, testing in $\mathcal{P}_a$ the translation of a term, say $\llbracket P \rrbracket$, regarding a generic observer $o' \notin \llbracket O \rrbracket$ (in particular, o' is not the mapping of an observer in O) means testing $\llbracket P \rrbracket$ in $\llbracket \mathcal{P}_s \rrbracket$ with a test which belongs to a more powerful language than $\llbracket \mathcal{P}_s \rrbracket$.

Indeed, consider the pair of $\mathcal{P}_s$ processes $P = \bar{a} \mid \bar{a}$, $Q = \bar{a}.\bar{a}$ and the set of observers $O = \{a.a.\omega\}$. Let $O' = \{a.a.\omega\}$ be the set of observers in $\mathcal{P}_a$. Of course $aa\omega$ is not the translation of any observer in O, though it can perform two a-actions before reporting success exactly as the observer in O. Then it is easy to convince one that $P \sqsubseteq^{O}_{may} Q$ and $O' \neq \llbracket O \rrbracket$.
On the other hand, consider the coding of our terms P and Q

- $\llbracket P \rrbracket = \llbracket \bar{a} \mid \bar{a} \rrbracket = \llbracket \bar{a} \rrbracket \mid \llbracket \bar{a} \rrbracket = (\nu u)(\bar{a}u \mid u(v).(\bar{v} \mid 0)) \mid (\nu t)(\bar{a}t \mid t(h).(\bar{h} \mid 0)) \equiv$
  $(\nu u)(\nu t)(\bar{a}u \mid u(v).(\bar{v} \mid 0) \mid \bar{a}t \mid t(h).(\bar{h} \mid 0))$
- $\llbracket Q \rrbracket = \llbracket \bar{a}.\bar{a} \rrbracket = (\nu u)(\bar{a}u \mid u(v).(\bar{v} \mid \llbracket \bar{a} \rrbracket))$,

and then the corresponding transitions when put in parallel with their observers

- $\llbracket P \rrbracket \mid a.a.\omega \equiv a.a.\omega \mid (\nu u)(\nu t)(\bar{a}u \mid u(v).(\bar{v} \mid 0) \mid \bar{a}t \mid t(h).(\bar{h} \mid 0)) \equiv$
  $(\nu u)(\nu t)(a.a.\omega \mid \bar{a}u \mid u(v).(\bar{v} \mid 0) \mid \bar{a}t \mid t(h).(\bar{h} \mid 0))$
  $\xrightarrow{\tau} (\nu u)(\nu t)(a.\omega \mid 0 \mid u(v).(\bar{v} \mid 0) \mid \bar{a}t \mid t(h).(\bar{h} \mid 0))$
  $\xrightarrow{\tau} (\nu u)(\nu t)(\omega \mid 0 \mid u(v).(\bar{v} \mid 0) \mid 0 \mid t(h).(\bar{h} \mid 0)) = P_1$ and $P_1 \xrightarrow{\omega}$;
- $\llbracket Q \rrbracket \mid a.a.\omega \equiv a.a.\omega \mid (\nu u)(\bar{a}u \mid u(v).(\bar{v} \mid \llbracket \bar{a} \rrbracket)) \equiv (\nu u)(a.a.\omega \mid \bar{a}u \mid u(v).(\bar{v} \mid \llbracket \bar{a} \rrbracket))$
  $\xrightarrow{\tau} (\nu u)(a.\omega \mid 0 \mid u(v).(\bar{v} \mid \llbracket \bar{a} \rrbracket)) = Q_1$ and $Q_1 \not\xrightarrow{\omega}$;

By Definition 6, hence, we have that $P \sqsubseteq^{O}_{may} Q$ but $\llbracket P \rrbracket \not\sqsubseteq^{O'}_{may} \llbracket Q \rrbracket$.

5.2 Full Abstraction of the Coding W.R.T. $\sqsubseteq^{O}_{\mathit{fair}}$

The same fully-abstract result proved for $\sqsubseteq^{O}_{\mathit{may}}$ can be proved for $\sqsubseteq^{O}_{\mathit{fair}}$. Also in this case, to guarantee a fully-abstract coding w.r.t. $\sqsubseteq^{O}_{\mathit{fair}}$, we have to make sure that only the translations of the observers in the synchronous setting are taken into account when comparing translations of $\mathcal{P}_s$ processes. Then the expected result for fair testing follows.

Proposition 3. Let P be a $\mathcal{P}_s$ process and $O \subseteq \mathcal{O}$ be a set of observers. Then, for every $o \in O$, $P\, \mathit{fair}\, o$ if and only if $\llbracket P \rrbracket\, \mathit{fair}\, \llbracket o \rrbracket$.

Theorem 2. (*Full abstraction of the coding w.r.t.* $\sqsubseteq^{O}_{\mathit{fair}}$)
Let P and Q be $\mathcal{P}_s$ processes and $O \subseteq \mathcal{O}$ be a set of observers. Then

$$P \sqsubseteq^{O}_{\mathit{fair}} Q \iff \llbracket P \rrbracket \sqsubseteq^{\llbracket O \rrbracket}_{\mathit{fair}} \llbracket Q \rrbracket.$$

5.3 The Must Preorder

After considering the $\sqsubseteq^{O}_{\mathit{may}}$ and the $\sqsubseteq^{O}_{\mathit{fair}}$ preorders, we tackle the $\sqsubseteq^{O}_{\mathit{must}}$ case. In particular, we look for a proposition similar to Proposition 2 and Proposition 3, since have been central to prove the fully abstract result for the former preorders. Unfortunately, the statement we are looking for does not hold. The following proposition provides a counterexample.

Proposition 4. There exists a $\mathcal{P}_s$ process P and an observer o such that
$P\, \mathit{must}\, o$ but $\llbracket P \rrbracket\, \not{m}\mathit{ust}\, \llbracket o \rrbracket$.

Proof. Consider the $\mathcal{P}_s$ process P defined as $P = \bar{a}.!\tau$, and the observer $o = a.\omega$. The only one maximal computation that $P \mid o$ can perform is

$$P \mid o = \bar{a}.\,!\tau \mid a.\omega \xrightarrow{\tau} !\tau \mid \omega \xrightarrow{\tau} \ldots \xrightarrow{\tau} 0 \mid 0 \mid \ldots \mid !\tau \mid \omega \xrightarrow{\tau} \ldots$$

Of course $P\, \mathit{must} o$. Now, consider $\llbracket P|o \rrbracket = \llbracket P \rrbracket | \llbracket o \rrbracket$ and the maximal computation that this process can perform. Consider the following one:
$\llbracket P \mid o \rrbracket = \llbracket \bar{a}.\,!\tau \rrbracket \mid \llbracket a.\omega \rrbracket =$
$(\nu u)(\bar{a}u \mid u(v).(\bar{v} \mid \llbracket\, !\tau \rrbracket)) \mid a(h).(\nu k)(\bar{h}k \mid k.\llbracket \omega \rrbracket) \equiv$
$(\nu u)(\nu k)(\bar{a}u \mid u(v).(\bar{v} \mid \llbracket\, !\tau \rrbracket) \mid a(h).(\bar{h}k \mid k.\llbracket \omega \rrbracket))$
$\xrightarrow{\tau} (\nu u)(\nu k)(0 \mid u(v).(\bar{v} \mid \llbracket\, !\tau \rrbracket) \mid \bar{u}k \mid k.\llbracket \omega \rrbracket)$
$\xrightarrow{\tau} (\nu u)(\nu k)(0 \mid \bar{k} \mid \llbracket\, !\tau \rrbracket \mid 0 \mid k.\llbracket \omega \rrbracket) \equiv (\nu k)(\bar{k} \mid \llbracket\, !\tau \rrbracket \mid k.\llbracket \omega \rrbracket) = (\nu k)(\bar{k} \mid !\, \llbracket \tau \rrbracket \mid k.\llbracket \omega \rrbracket)$
$\xrightarrow{\tau} (\nu k)(\bar{k} \mid 0 \mid !\, \llbracket \tau \rrbracket \mid k.\llbracket \omega \rrbracket)$
$\xrightarrow{\tau} (\nu k)(\bar{k} \mid 0 \mid 0 \mid !\, \llbracket \tau \rrbracket \mid k.\llbracket \omega \rrbracket)$
$\xrightarrow{\tau} \ldots (\nu k)(\bar{k} \mid 0 \mid 0 \mid \ldots \mid 0 \mid !\, \llbracket \tau \rrbracket \mid k.\llbracket \omega \rrbracket) \ldots$
and note that each intermediate state of the computation cannot perform any ω action. Hence, $\llbracket P \rrbracket\, \not{m}\mathit{ust}\, \llbracket o \rrbracket$.

Thus,

Theorem 3. The coding is not fully-abstract w.r.t. $\sqsubseteq^{O}_{must}$.

Proof. Consider $O = \{a.\omega\}, P = \bar{a}$ and $Q = P \mid \bar{a}.!\tau$. First of all note that $P \sqsubseteq^{O}_{must} Q$. Indeed, $P \,|\, o$ and $Q \,|\, o$ can only perform the following computations $P \mid o = \bar{a} \mid a.\omega \xrightarrow{\tau}\equiv \omega \xrightarrow{\omega}$ and $Q \mid o = P \mid \bar{a}.!\tau \mid a.\omega \xrightarrow{\tau}\equiv \omega \mid \bar{a}.!\tau \xrightarrow{\omega}$ or $Q \mid o = P \mid \bar{a}.!\tau \mid a.\omega \xrightarrow{\tau}\equiv \omega \mid !\tau \xrightarrow{\omega}$, respectively.

Then note that $P \not\sqsubseteq^{[\![O]\!]}_{must} Q$. Indeed, $[\![P]\!] \,|\, [\![o]\!] = [\![\bar{a}]\!] \,|\, [\![a.\omega]\!] \overset{\varepsilon}{\Longrightarrow}\equiv (\nu k)(\bar{k} \,||\, k.[\![\omega]\!]) \xrightarrow{\tau}\equiv \omega \xrightarrow{\omega}$ but there exists a computation from $[\![Q]\!] \mid [\![o]\!]$, namely, $[\![Q]\!] \mid [\![o]\!] = [\![P]\!] \,|\, [\![\bar{a}.!\tau]\!] \,|\, [\![a.\omega]\!] \overset{\varepsilon}{\Longrightarrow}\equiv [\![P]\!] \,|\, (\nu k)(\bar{k} \,|\, ![\![\tau]\!] \,|\, k.[\![\omega]\!]) \xrightarrow{\tau} \ldots [\![P]\!] \,|\, (\nu k)(\bar{k} \,|\, ![\![\tau]\!] \,|\, k.[\![\omega]\!]) \ldots$, where each intermediate state of the computation cannot perform any ω action.

In the following section we state fully-abstract results for the must preorder by restricting the base language.

Fully-Abstract Results for the Must Preorder Let us concentrate on the set of processes, tests and experiments in the π-calculus and its asynchronous version that hold the hereditary convergence predicate meaning that they can perform only finite maximal computations. In the following we will generically use $\mathcal{P}$ to denote either $\mathcal{P}_s$ or $\mathcal{P}_a$.

Definition 8. Let P be a $\mathcal{P}$ process and $O \subseteq \mathcal{O}$. We say that P is *hereditary convergent* w.r.t. O, $P \downarrow_O$, if and only if $\forall\, o \in O$, $(P \,|\, o) \downarrow$; where $(P \,|\, o) \downarrow$, read $P \,|\, o$ is *convergent*, if and only if every maximal computation from $P \,|\, o$ is finite.

If we concentrate on the subset of hereditary convergent processes then we have the following result. We refer the reader to [CC00] for a detailed proof.

Theorem 4. Let P and Q be $\mathcal{P}_s$ processes and $O \subseteq \mathcal{O}$ a set of tests. If $P \downarrow_O$ and $Q \downarrow_O$ then $P \sqsubseteq^{O}_{must} Q$ if and only if $[\![P]\!] \sqsubseteq^{[\![O]\!]}_{must} [\![Q]\!]$.

The hereditary convergence predicate is very severe since processes in parallel with observers are allowed to perform only finite sequences of internal actions. We previously tried with weaker forms than the hereditary convergence predicate. A more generous one is the following. Consider the set *HerConv* of hereditary convergent processes as the largest set of processes P which satisfies:

(i) $P \downarrow$;
(ii) $P \xrightarrow{\mu} Q$ implies $Q \in$ *HerConv*.

Unfortunately, also such a predicate is not enough to obtain a fully-abstract result for the mapping w.r.t. $\sqsubseteq^{O}_{must}$. Indeed, there exists a process P and an observer o which are in *HerConv* and P *must* o but $[\![P]\!]$ $\cancel{must}$ $[\![o]\!]$. As an example, consider $P = !\bar{a}.0$ and $o = !a.\omega$. The parallel composition of their translation can engage an infinite computation of τ actions without showing the presence of ω.

6 Related Works

Mappings from synchronous languages to asynchronous ones (and relative fully-abstract results, when possible) have been considered in many papers. For this reason we report here only a very brief introduction to those papers that are very close to our study.

One of this papers is [QW00]. It also aims at studing the relationships between synchronous and asynchronous mobile processes. The authors consider the polyadic π-calculus and the asynchronous version of the monadic π-calculus as base languages, Boudol's mapping from the former to the latter language and barbed congruence as the semantics to be preserved by the mapping. Some of the ideas exploited in the present paper are also present there. I.e., the restriction of the asynchronous tests, contexts in their setting, to those which are mapping of synchronous tests. However, we have proven that such a condition is necessary but not sufficient to get full abstraction for every semantic theory. Indeed, it can be obtained for may and fair testing but not for must (unless strict restrictions on the base language are considered). In more detail, they provide a type system for processes of the asynchronous monadic π-calculus which characterizes the set of contexts in the asynchronous world. These are all contexts which are mapping of contexts in the synchronous setting. Then prove a fully abstract result for barbed congruence similar to those stated in Theorem 1 and Theorem 2. It is worth of noting that our proof techinque still work when their barbed congruence is considered (and, actually, also when Morris' testing preorder is taken into account).

Another very interesting paper is [Pal97]. Also this paper is concerned with the attempt of solving or, at least, clarify how these two communication mechanisms (synchronous and asynchronous) can be implemented one into the other. The π-calculus and the asynchronous π-calculus are the considered languages together with their own transitional semantics. It has been shown that it is not possible to encode the π-calculus into the asynchronous π-calculus because the "leader election problem" cannot be solved in the latter language while it is still possible in the former one. More in general, it has been shown that it is not possible to map the π-calculus in the asynchronous π-calculus for every possible "uniform" encoding (it is compositional w.r.t. parallel composition and "behaves well" w.r.t. renamings) and for every "reasonable" semantics (it distinguishes two processes P and Q whenever in some computation of P the actions on certain intended channels are different from those of any computation of Q) which one wants to preserve. According to our interpretation of uniform and reasonable, we can say that Boudol's mapping is uniform, may and fair semantics are not reasonable while must is. We cannot, however, exploit Palamidessi's result to justify our negative result with the must preorder. Indeed, her proof technique strongly relies on the presence of mixed choices (input and output prefixes in alternative composition) in the π-calculus while we do not have such choice in our source language. Moreover, such a technique does not hold anymore if separate choice is taken into account as shown in [Nes97]. But, as shown in this paper, the must preorder gives problems anyway.

Another work with similar issues of ours, though following quite different means, is [HT91]. Honda and Tokoro concentrate on the π-calculus without sum and bounded output and provide terms of this algebra with two transitional semantics: one describes processes with a synchronous communication and the other describes processes with an asynchronous communication. The former transitional semantics is that standard while the latter one relies on a new input-prefix rule. It allows any process to perform an input action also when not syntactically specified (this models output as a non blocking action). Then, various observational semantics based on trace, failure and bisimulation, are defined on the top of the considered transitional semantics. The relationships between the synchronous bisimulation and its asynchronous counterpart are investigated. The main result of this study shows that the latter relation is strictly weaker than the former one. Similar results hold for trace and failure-based semantics. To obtain fully abstract results, they introduce the notion of $\mathcal{I}$ *completion.* This is a mapping from a term interpreted asynchronously into a term interpreted synchronously. Any target term is able to mimic all the asynchronous transitions via synchronous transitions. More in detail, the target term is the original one in parallel with the so-called *identity receptors.* These are processes with the ability of performing input actions on suitable channels after which they become themselves in parallel with output actions on the same channels. In this way they simulate the transitions which are in the asynchronous setting but not in the synchronous one. By weakening terms interpreted synchronously in this way, Honda and Tokoro prove that two terms are asynchronously bisimilar (resp. failure, trace) if and only if their mappings are, up to $\mathcal{I}$ *completion*, synchronously bisimilar (resp. failure, trace). They do not mention, however, to fully abstract results for the opposite mapping; i.e., how to implement synchronous communication in terms of the asynchronous one which, instead, is the main purpose of the current work.

7 Further Work

This paper rises several interesting questions to look at. We would like to check whether our results scale up to versions of π-calculus with mixed choice by exploiting the results and the non-uniform encodings in [Nes97].

Another interesting question is related to the negative result for must testing. It is reasonable to ask whether or not there are uniform encodings (also in the case the source language which only has separate choice, as the π-calculus we have considered), that preserve the must testing. We conjecture a negative result for this question. Always regarding the must testing, we would like to investigate on the possibility of proving fully abstract results when some "fair" scheduling assumption is imposed on the execution of the parallel components of a global system. For this questions, instead, we conjecture a positive result. This, of course, would improve the result stated in Section 5.3.

Finally, we intend to import the ideas developed for testing in a bisimulation scenario. At the first glance it seems that [HT91] can provide a valid support to this investigation.

Acknowledgments

We would like to thank Matthew Hennessy for his continuous guidance and encouragement. He is also the source of many ideas reported in this paper. Uwe Nestmann and Catuscia Palamidessi are thanked for their valuable comments that have certainly improved on a previous version of this paper.

References

[BDN95] M. Boreale, R. De Nicola: *Testing Equivalence for Mobile Processes.* Information and Computation, **120**, pp. 279-303, 1995.

[Bou92] G. Boudol: *Asynchrony and the π-calculus.* Technical Report 1702, INRIA, Sophia Antipolis, 1992. 256, 259, 262

[BRV95] E. Brinksma, A. Rensink, W. Vogler: *Fair Testing.* In the Proc. of CONCUR'95, LNCS 962, pp. 313-327, 1995. 257

[CC00] D. Cacciagrano, F. Corradini: *On Synchronous and Asynchronous Communication Paradigms.* Internal Report n. 17/2000, University of L'Aquila, 2000. Available from: http://w3.dm.univaq.it/˜flavio. 258, 262, 265

[ICH98] I. Castellani, M. Hennessy: *Testing Theories for Asynchronous Languages.* In the Proc. of FSTTCS'98, LNCS 1530, pp. 90-101, 1998.

[DH84] R. De Nicola, M. Hennessy: *Testing Equivalence for Processes.* Theoretical Computers Science, **34**, pp. 83-133, 1984. 257

[HT91] K. Honda, M. Tokoro: *An Object calculus for Asynchronous Communication.* In the Proc. of *ECOOP*'91, LNCS, pp. 133-147, 1991. 259, 267, 268

[H88] M. Hennessy: *An Algebraic Theory of Processes.* MIT Press, Cambridge, 1988.

[MPW92] R. Milner, J. Parrow, D. Walker: *A Calculus of Mobile Processes*, Part I and II. Information and Computation, **100**, pp. 1-78, 1992. 256

[NC95] V. Natarajan, R. Cleveland: *Divergence and Fair Testing.* In the Proc. of ICALP'95, LNCS 944, pp. 648-659, 1995. 257

[Nes97] U. Nestmann: *What is a 'Good' Encoding of Guarded Choice?.* In the Proc. of EXPRESS'97, ENTCS 7, pp. 243-264, 1997. 266, 267

[Pal97] C. Palamidessi: *Comparing the Expressive Power of the Synchronous and Asynchronous π-calculus.* In the Proc. of POPL'97, pp. 256-265, 1997. 266

[QW00] P. Quaglia, D. Walker: *On Synchronous and Asynchronous Mobile Processes.* In the Proc. of FOSSACS 2000, LNCS 1784, pp. 283-296, 2000. 266

Complexity of Layered Binary Search Trees with Relaxed Balance

Lars Jacobsen and Kim S. Larsen[*]

Department of Mathematics and Computer Science, University of Southern Denmark
Main Campus: Odense University, Campusvej 55, DK–5230 Odense M, Denmark
{eljay,kslarsen}@imada.sdu.dk
Fax: +45 65 93 26 91

Abstract. When search trees are made relaxed, balance constraints are weakened such that updates can be made without immediate rebalancing. This can lead to a speed-up in some circumstances. However, the weakened balance constraints also make it more challenging to prove complexity results for relaxed structures.

In our opinion, one of the simplest and most intuitive presentations of balanced search trees has been given via layered trees. We show that relaxed layered trees are among the best of the relaxed structures. More precisely, rebalancing is worst-case logarithmic and amortized constant per update, and restructuring is worst-case constant per update.

Introduction

Usually, updating in a balanced search tree is carried out as follows: First, a search is carried out in order to determine the location of the update. Second, the update is performed. Third, local balance constraints are reconsidered. Since balance constraints are usually based on path lengths or subtree sizes, these constraints may have been violated, because most often, an insertion will add at least one node to the tree and a deletion will remove at least one node from the tree. If there is a balance problem, this is fixed completely if possible, and otherwise it is fixed at the cost of introducing a new problem closer to the root. This problem is then handled recursively until it disappears or is moved all the way to the root, where balance problems are normally easily fixed.

The three phases described above are referred to as searching, updating, and rebalancing. Informally, *relaxed balance* is a term used for the following. If a search tree has been equipped with relaxed balance, the searching and updating have been uncoupled from the rebalancing. Thus, it is now possible to search and make an update without performing any rebalancing. For this to be well-defined, the balance constraints must be weakened (relaxed) in such a way that the tree after an update is still in the now broader class of trees. Additionally, the standard tree, which is made relaxed, should belong to the class, and the overall goal of the (presumably generalized and/or expanded) collection of rebalancing

[*] Supported in part by the Danish Natural Sciences Research Council (SNF).

A. Restivo, S. Ronchi Della Rocca, L. Roversi (Eds.): ICTCS 2001, LNCS 2202, pp. 269–284, 2001.

operations is to bring the tree back to fulfilling the constraints of the standard balanced tree.

The benefit of the uncoupling depends on the environment. Discussions of this can be found in many of the papers on the subject, but here is a brief account. In a sequential system, bursts of requests, possibly from an external source, can be served faster if rebalancing is "turned off" during the period. After the burst, rebalancing should gradually bring the tree back in balance, while requests are served at the same time. In a parallel (shared-memory) system, a naïve implementation would lock the root of the tree so frequently that the degree of parallelism would be extremely low. In relaxed structures, it is generally possible to exclusively lock only nodes which will be involved in pointer changes, instead of all nodes which might be involved in pointer changes. This implies that most of the exclusive locking will take place close to the leaves.

The cost of the relaxation is that the guaranteed worst-case bound of logarithmic path lengths is temporarily lost. The options are to trust that this does not become a problem for these short periods of time (maybe the requests are known to be close to uniform), to monitor path lengths and rebalance when some limit is exceeded, to dedicate a fixed minimum amount of rebalancing time to each update (or group of updates), or something else along those lines. The best solution can only be found when the specifics of the concrete scenario are known.

However, to ensure that as much time as possible is dedicated to request processing, it is vital that rebalancing, when it is performed, is performed efficiently. The difficulty in proving the various possible efficiency bounds on the run-time complexity is of course that after the structure has been relaxed, much less is known about its appearance. For instance, if k updates are performed on a standard balanced search tree of size n, usually $(k \log n)$, or fewer, rebalancing operations can easily be shown to completely rebalance the tree. In a relaxed version, path lengths can approach $\log n + k$, so if k is more than a constant, will $(k \log n)$ operations still suffice?

To make relaxed proposals as usefull as possible in the sequential as well as in the parallel setting, it is always required that rebalancing is carried out in local independent steps. However, in the sequential setting, this may not be mandatory.

Finally, relaxed balance is also a topic of theoretical interest. Search trees are some of the most important data structures, and this line of work answers some very fundamental questions concerning whether or not the traditional tight coupling between updating and rebalancing is necessary for the efficient rebalancing results to follow.

We give a very brief summary of the developments; more details can be found in [16], for example. Some of the ideas were initiated in [10,15]. AVL-trees [1,23] were investigated in [17,25,28], red-black trees [3,10,31] in [5,6,7,8,16,26,27], and (a,b)-trees [13,22], B-trees [4], 2-3-trees [2,12] in [18,19,25]. In [20], a general result for balanced trees was developed, and in [9,11,21,24,32], some variations of the standard schemes were investigated. Locking in a parallel setting was discussed in [6,27].

In this paper, we investigate *layered trees* [30]. A relaxed version of layered trees was given in [29]. The primary contribution of this paper is to establish the complexity results which hold for the structure. We give our own presentation of layered trees with and without relaxed balance; partly to make the paper self-contained, but also partly because greater precision in the formulation of rebalancing operations is required in order for a proof of amortized constant rebalancing to be established.

The paper [29] primarily focuses on the design ideas, and on the important issue (not least in a parallel setting) of limiting restructuring. The principal difference between changing a pointer and updating balance information is that searching can proceed simultaneous with the information updating. Thus, if fine-grained locking is an option, limiting restructuring operations is more important. With the set-up in [29], the authors can show that only a constant amount of restructuring is necessary per update.

Layered Trees

It is possible to give a quite general definition of a layered tree [30]. However, to present the ideas in a form as simple as possible, we first give one very specific definition. Later, we discuss the more general alternatives.

A layered tree is a binary search tree. It is leaf-oriented, meaning that all keys are kept in the leaves. Internal nodes contain routers, which are of the same type as the keys and often copies of some of these. However, the only purpose of the routers is to guide the searches to the correct leaves. In a leaf-oriented binary tree, internal nodes always have two children.

Leaf-oriented trees are often the choice in large database-oriented applications because keys often have significant amounts of information attached. It is generally more efficient not to have to encounter this extra information when searching down the tree and when changing internal nodes due to rebalancing.

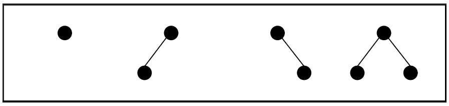


Fig. 1. The four basic configurations

Additionally, when designing relaxed structures, there is no good way of carrying out deletions in a step-wise and local manner if the tree is not leaf-oriented. The problem is that if an internal node with two children should be deleted, the standard method for handling this is to switch keys with its internal predecessor or successor and delete that node instead. However, that node can be located a non-constant distance away.

A leaf-oriented binary search tree is called layered if it can be constructed as described below from the configurations listed in Fig. 1:

1. Select one of the four basic configurations. The top node in the selected configuration will be the root of the whole tree.
2. Add a number of layers. One layer is added as follows: For each node u in the already constructed part of the tree which does not have a left (right) child, select one of the basic configurations and let the top node of the configuration be the left (right) child of u.
3. Construct a final layer of leaves, by adding a leaf everywhere a left or right child is missing.

We refer to the level of leaves as layer 0. The layer on top of that is layer 1 and so on. An edge connecting a node in some layer i with a node in the next layer $i + 1$ is said to *cross the border* between the two layers. In the concrete implementation described in this paper, we assume that borders are explicitly stored in the structure. The most flexible way of doing this is by storing one bit in each node such that the bit is zero if it belongs to an even-numbered layer and one otherwise. The manipulation of this bit in connection with the operations to be discussed is easy, and we will not describe it explicitly. For easy future reference we define the following two subsets of basic configurations: the small configurations $\mathcal{C}_S = \{$, , $\}$ and the large configurations $\mathcal{C}_L = \{$, , $\}$.

Proposition 1. *The height of a layered tree with n leaves is bounded by $2\lfloor \log_2 n \rfloor$.*

Proof. We show by induction in the number of layers that a node in layer i has at least 2^i leaves in its subtree. This is trivial for the base case of a single leaf. For the induction step, we notice that any node u in the configurations from Fig. 1 at any level $i > 0$ has at least two descendants at level $i - 1$. Since each of these, by the hypothesis, have at least 2^{i-1} leaves in their subtrees, u has 2^i leaves in its subtree. Thus, the layer of the root is at most $\lfloor \log_2 n \rfloor$, and so there are at most $\lfloor \log_2 n \rfloor + 1$ layers. Since the height of the highest basic configuration is two, the result follows.

Keys in the search tree come from a totally ordered domain. The keys in the leaves appear in strictly increasing order from left to right. A router in an internal node is greater than or equal to any key in its left subtree and less than any key in its right subtree.

In the light of this and Proposition 1, searching can obviously be performed in logarithmic time. The update operations, insert and delete, can also be performed in logarithmic time, and with at most a constant number of structural changes per update [30]. One way of describing this is as follows.

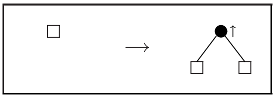
The general idea is to make the update, and register if there is a problem, i.e., if the tree is no longer constructed according to the layered tree definition.

Recursively, we remove the problem if possible, and otherwise move it to the next layer. At the root, any problem can be eliminated.

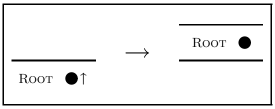
In the following, we describe the updating procedures. Proof of correctness follows later.

Insertions

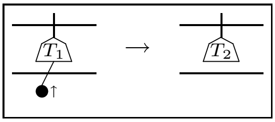
To insert a key, we search for the given key as usual in a search tree, and we end up at a leaf. If that leaf does not already contain the given key, a new leaf is created using operation *New leaf insertion*. The new key and the one already present in the existing leaf are arranged in order, and the key to the left is copied to the new internal node as its router.



New leaf insertion.



Up root.



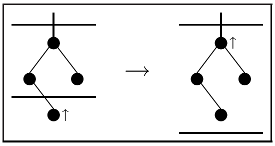
Up finish. $T_1 \in \mathcal{C}_S$. $T_2 \in \mathcal{C}_L$. $|T_2| = |T_1| + 1$.

The new internal node is on layer 0, which is not allowed, and is therefore equipped with a *push-up request* (↑). This push-up request is dealt with recursively as follows. If it reaches the root, the problem is solved using operation *Up root*. Otherwise, if there is room at the next layer, i.e., its parent is part of a configuration consisting of at most two nodes, the problem is solved using operation *Up finish*.

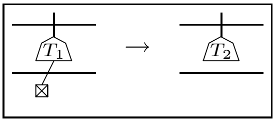
If the parent at the next layer is in a three-node configuration, the problem is moved up one layer using operation *Up push*.

Deletions

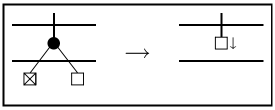
To delete a key, we search for the given key as usual in a search tree, and we end up at a leaf. If that leaf contains the given key, we proceed as follows (the leaf to be deleted is marked with two crossing lines in the figures). If the parent configuration has at least two nodes, using operation *Remove finish*, we can rearrange the nodes such that the leaf and its parent are deleted, while all configurations are still basic configurations.



Up push.



Remove finish. $T_1 \in \mathcal{C}_L$. $T_2 \in \mathcal{C}_S$. $|T_1| = |T_2| + 1$.



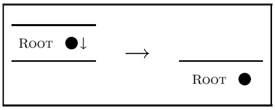
Remove continue.

If the next layer has a one-node configuration, we use operation *Remove continue*. This introduces a leaf at layer 1, which should be moved down to layer 0 before the tree can again be guaranteed to be a layered tree. We register this problem by marking the node with a *pull-down request* (↓).

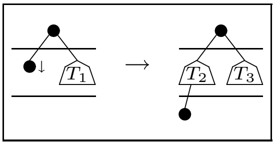
A pull-down request is handled recursively as follows. If it reaches the root, the problem is solved using operation *Down root*. Otherwise, if the sibling and parent configurations have at least three internal nodes together, then there are sufficiently many nodes locally such that the node can be moved down using either operation *Down finish 1* or *Down finish 2*, and at least one-node configurations can be created everywhere.

Finally, if the parent and sibling configurations contain only one node each, the problem is moved up one layer using operation *Down push*.

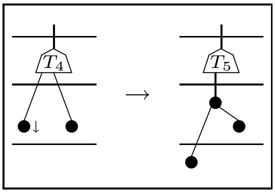
Observe that only operation *Remove continue* and *Down push* create pull-down requests. Since the only nodes which are marked are leaves or internal nodes with exactly one child on the next layer, such requests are created only if the marked node can be pulled down without violating the design criteria for layered trees.



Down root.



Down finish 1. $T_1 \in \mathcal{C}_L$. $T_2, T_3 \in \mathcal{C}_S$. $|T_1| = |T_2| + |T_3|$.



Down finish 2. $T_4 \in \mathcal{C}_L$. $T_5 \in \mathcal{C}_S$. $|T_4| = |T_5| + 1$.

Layered Trees with Relaxed Balance

To make the tree relaxed, we must allow that rebalancing can be interrupted at any time. In particular, its start can be delayed. In addition, the tree must be able to accommodate several updates for which the corresponding rebalancing has not been undertaken.

In addition to the basic configurations, several new configurations are allowed in the tree; any one or two node basic configuration where the bottom-most node is marked by a pull-down request, a zero-node configuration (a layer-crossing edge), and a four node configuration, where the top-most node is marked by a push-up request. The complete set of extra configurations (up to symmetric variants) are depicted in Fig. 2.

When an insertion is made, a leaf is replaced by an internal node with two leaves. If several insertions are made, large trees might be build this way without respecting the design criteria for relaxed layered trees. Such trees are always rooted at an internal node marked with a push-up request at layer 0. This part of the tree is called the *unstructured* part, while the part satisfying the design criteria for relaxed layered trees is called the *structured* part.

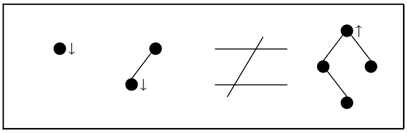


Fig. 2. Relaxed configurations

Rebalancing is now carried out by moving a problem from the unstructured part into the structured part, and recursively towards the root, until the problem is removed.

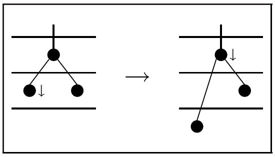
Since we cannot control when a deletion is actually carried out, the leaf to be deleted is marked physically for deletion by the operation *Delete mark*. Observe that the leaf might already be marked with a pull-down request, which is indicated by a parenthesized pull-down request (↓).

A leaf-oriented relaxed layered search tree can be constructed in the following way:

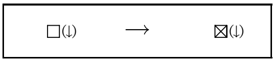
1. Select any configuration,except the layer-crossing edge. The top node of the selected configuration will be the root of the tree.
2. Add a number of layers: For each node u in the already constructed part of the tree, which does not have a left (right) child: if u is not marked with a pull-down request, and u is not on the layer above (a layer-crossing edge) add any of the node-containing configurations as the left (right) child of u. If u is marked with a pull-down request, add any configuration as the left (right) child of u, such that exactly one of the child configurations of a node marked by a pull-down request is a layer-crossing edge.
3. Construct the final layer by adding leaves, leaves marked for deletion, or unstructured trees to every node on the second to final layer that does not have a left (right) child, unless that node is marked by a pull-down request, in which case the node itself is made a leaf or a leaf marked for deletion.

Some operations involve the parent configuration, and some also the sibling configuration. An operation can generally not be carried out if the involved configurations are marked by requests. However, in some situations, we must allow that the sibling and parent configurations contain requests to avoid deadlocks.

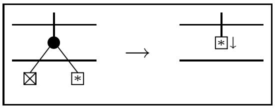
In the case of deletion, the sibling of the deleted leaf in operation *Remove continue*, might be marked for deletion, and is thus of course still marked after the application of the operation. This is indicated by an asterisk in the modified operation *Remove continue*. Analogously, two single-node siblings might both contain pull-down requests. Therefore, operation *Down finish 2* and *Down push* are modified to allow this.



Down push.

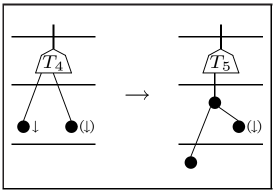


Delete mark.

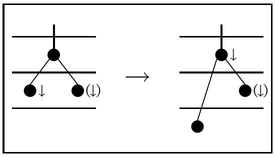


Modified *Remove continue.*

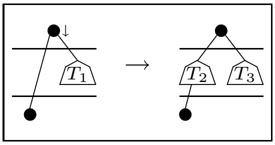
Furthermore, since we cannot control when updates are made, two new operations are needed to handle special cases of insertions. If a leaf is marked for deletion and an insertion is made at the very same leaf, the leaf is recycled as depicted in operation *Insert recycle*. If a leaf is marked with a pull-down request and an insertion is made at the very same leaf, the creation of the new internal node cancel out with the pull-down request; operation *Insert solve*.



Modified *Down finish 2*. $T_4 \in \mathcal{C}_L$. $T_5 \in \mathcal{C}_S$. $|T_4| = |T_5| + 1$.



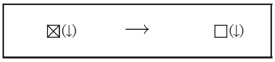
Modified *Down push.*



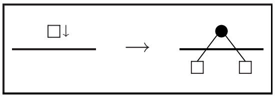
Down cancel. $|T_1| \geq 2$. $|T_2|, |T_3| \geq 1$. $|T_1| = |T_2| + |T_3|$.

Finally, pull-down requests are created if and only if both child configurations and the parent configuration are single nodes. However, when the request is to be resolved, this might not still be the case. One child is always a layer-crossing edge, while the other might be any other configuration. If the other child contains more than one node, these nodes can be rearranged such that it is no longer necessary to pull the marked node down. This is done by operation *Down cancel*. Observe that push-up requests among the rearranged nodes are analogously made obsolete, while pull-down requests must follow their respective layer-crossing child edges. It is an implicit precondition for applying any other operation involving pull-down requests that operation *Down cancel* cannot be applied.

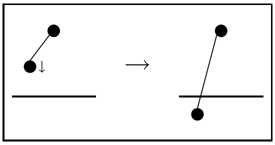
Analogously, the parent (the node marked by a pull-down request) might not be a single node anymore, in which case the node is just pulled down, using operation *Down finish 3*.



Insert recycle.



Insert solve.



Down finish 3.

Correctness and Complexity of Relaxed Balancing

By inspecting the individual operations, one can easily verify that the rebalancing operations satisfy the soundness property; applying any operation turns a relaxed layered tree into a relaxed layered tree.

Now we show that the collection of rebalancing operations is sufficient.

Theorem 1. *Completeness: Let T be a relaxed layered tree. While T contains at least one node marked by a request, some rebalancing operation can be applied.*

Proof. Let $\mathcal{R}$ denote the set of nodes marked by a request or marked for deletion on the top-most layer containing marked nodes.

If the root is in $\mathcal{R}$, then one of the *Root* operation can be applied. Assume that the root is not in $\mathcal{R}$. Assume that $\mathcal{R}$ contains some node u marked with a push-up request. Since u is top-most and non-root, the parent configuration is a basic configuration, and thus either operation *Up finish* or operation *Up push* can be applied. Observe that this is independent of whether or not u is located in the structured or the unstructured part of the tree.

Assume that $\mathcal{R}$ contains no nodes marked by a push-up request. Assume that $\mathcal{R}$ contains nodes marked by a pull-down request, and let u be such a node in a two node configuration, if any such exist. Consider the configurations below u. By the soundness property, one of these configurations is a layer-crossing edge. If the other configuration has at least 2 nodes, then operation *Down cancel* can be applied. Otherwise u can be moved down using operation *Down finish 3*.

Now assume that $\mathcal{R}$ contains nodes marked by a pull-down request, but that all these are single node configurations. Again, if a child which is not a layer-crossing edge contains at least two nodes, operation *Down cancel* can be applied. Otherwise we know that u's sibling configuration is either a single node (possibly marked by a pull-down request) or a basic configuration containing at least two nodes. In the first case, depending on whether the parent configuration of u has more than one node or not, either operation *Down push* or operation *Down finish 2* can be applied (recall that u was a top-most request, which means that u's parent configuration is a basic configuration). In the latter case, operation *Down finish 1* can be applied.

Finally, assume that $\mathcal{R}$ contains only leaves marked for deletion. By this assumption, the parent configuration of such a leaf contains no requests, so either operation *Remove finish* or *Remove continue* can be applied.

Amortized Constant Rebalancing

We use the standard potential function technique [33]. Any update operation creates exactly one problem in the unstructured part. Either a leaf marked for deletion or an internal node. This problem is either removed by a *finishing* rebalancing operation or moved into the structured part as a request which is then in turn moved a number of times using a *non-finishing* operation, until it is removed by a finishing operation.

Theorem 2. *Rebalancing is amortized constant.*

Proof. Assume that we remove every edge which connects two nodes in different layers. This splits the tree up into a collection of small trees with at most four nodes. We let $\mathcal{P}_i(T)$ for $i \in \{1, 2, 3, 4\}$ denote the number of pieces with i nodes resulting from splitting T.

We define the potential $\Phi(T)$ of the tree T as follows:

$$\Phi(T) = \mathcal{P}_1(T) + \mathcal{P}_2(T) + 3\mathcal{P}_3(T)$$

Any update operation, including the operation creating a request in the structured part, and any finishing operation may increase the potential, but it can do so by at most a constant. What remains is to show that every non-finishing operation decreases the potential by at least a constant to cover for its own application. The operations *Up push* and *Down push* are the only non-finishing rebalancing operations.

Operation *Up push* is applied only if the parent configuration of the node marked by the push-up request is a three node basic configuration. Recall that any node marked by a push-up request is the root of a four node configuration. Thus by the application, a three node configuration and a four node configuration is replaced by a four node, a two node, and a one node configuration, which decreases the potential by one.

Operation *Down push* is applied only if the parent and sibling configurations are single nodes. Furthermore, operation *Down push* is applied only if operation *Down cancel* cannot be applied. Thus, the children of the node marked by a pull-down request are a layer crossing edge and a single node, respectively. After the application, the node pulled down forms a two node configuration together with the single node child configuration at the child layer. Thus, four one node configurations are replaced by two one node configurations and a two node configuration, which decreases the potential by one.

Worst-Case Logarithmic Rebalancing

The previous theorem shows that rebalancing is amortized constant, if we start with an initially empty tree. However, if we start with a non-empty layered tree, we cannot use the theorem to guarantee a good complexity immediately. In the following, we show that even if we start with a layered tree, rebalancing is at most logarithmic in the worst-case.

Inspired by [16], we define a *count function* c as follows: If the tree is a standard layered tree, the count function is one on all leaves, and zero for all internal nodes. The *count sum* of a node u is the sum of the count function applied to all nodes in the subtree rooted at u, i.e., $\sum_{v \in T_u} c(v)$.

In a relaxed layered tree, the count function is maintained as follows: When an insertion is made, a leaf ℓ is replaced by an internal node with two leaves. The function value of the internal node is set to $c(\ell) - 1$, while the count function for both leaves is initialized to one.

When a leaf ℓ is actually deleted (not just marked), its parent u is deleted as well. The function value of the node v replacing the parent is then increased by $c(\ell) + c(u)$.

When nodes are rotated, some node is the root of the rotation. The function value of the root is assigned to the new root, while all the remaining function values are reassigned in-order to the remaining nodes involved in the rotation.

Since the count sum of the whole tree is incremented by insertions, but not decremented by deletions, the count sum of the root is always $n + i$ where n is the number of leaves in the tree the last time it was a standard layered tree, and i is the number of insertions.

Note that the values of the count function are always non-negative, and for leaves, they are positive.

We define the *relaxed layer* of a node u to be its layer in a layered tree unless u and u's parent are connected by a layer crossing edge. In this case, we define the relaxed layer to be one higher than its actual layer.

Lemma 1. *For any node u on relaxed layer j: $\sum_{v \in T_u} c(v) \geq 2^j$*

Proof. By induction on the number of operations on the tree since it was last a standard layered tree.

The base case follows by an argument similar to the proof of Proposition 1, since the count sum is exactly the number of leaves in any subtree.

It is easily verified that the result holds for any application of an update operation or an operation bringing a request into the structured part.

If nodes (in the structured part) are rearranged to form basic configurations, i.e., we also consider nodes marked by push-up request which are unmarked as a consequence of the rearrangement, the result follows immediately from the hypothesis since all such nodes have at least two descendants on the next layer.

If a node (marked by a pull-down request) is pulled down, we have two cases: It is either pulled down using operation *Down push*, in which case the relaxed layer is unchanged, or by a finishing operation, in which case the relaxed layer is decreased. In either case, the count sum is unchanged, and the result follows again immediately from the hypothesis.

Observe that any node marked by a push-up request has both its children on the same layer as itself. Thus, such a node is the root of a subtree with twice the count sum it needs, and the result follows from the hypothesis—even when the node is pushed to the next layer.

What remains is to verify that the result holds after the application of operation *Down cancel* when nodes marked by pull-down requests are rearranged. However, this follows from the way relaxed layers are maintained. Since both children of nodes marked by pull-down requests have the same relaxed layer—that of nodes on the next layer—the result follows from the hypothesis.

Theorem 3. *Rebalancing is worst-case logarithmic.*

Proof. Rebalancing after any update involves bringing the problem into the structured part of the tree, applying a number of non-finishing operations, and applying one finishing operation. Hence, if the number of non-finishing operations can be bounded by a logarithmic term, the theorem follows. However, the application of a non-finishing operation moves a problem to a node on a higher relaxed layer, and as, by Lemma 1, the size of a subtree is exponential in the relaxed layer, there can be at most a logarithmic number of such layers.

More precisely, if i insertions (and possibly some deletions) are applied to a tree of size n, we get the bound $n + i = \sum_{v \in T_{\text{Root}}} c(v) \geq 2^{j_{\text{Root}}}$, where j_{Root} is the relaxed layer of the root.

Since at the root, the number of the layer and the relaxed layer must coincide, the root is in layer at most $\lfloor \log_2(n+i) \rfloor$. Including initial, finishing, and non-finishing operations, at most $\lfloor \log_2(n+i) \rfloor + 2$ operations can be applied per update.

Worst-Case Constant Restructuring

The following result is from [29].

Theorem 4. *Restructuring is worst-case constant.*

Proof. As was observed earlier, every finishing rebalancing operation removes at least one request. Hence, at most one finishing rebalancing operation can be applied per update. Since neither of the non-finishing operations make any structural changes, the theorem follows.

Concluding Remarks

The objective of this presentation of relaxed layered trees was twofold. We wanted to give a presentation precise enough that correctness and complexity proofs could be based on it. At the same time, we wanted to keep the presentation simple, in the spirit of the presentation of the standard version. The first objective has been obtained, but, admittedly, some of the simplicity is lost in the transition to a relaxed version. The problem is that the extra configurations, which are allowed in the relaxed setting, multiplies the total number of cases. With the level of precision which is required to establish all the complexity results, there does not seem to be any way to treat the operations at a higher level of abstraction to cut down on the number of cases.

On the positive side, we have shown that relaxed layered trees are among the best relaxed binary search trees. In particular, all the asymptotic complexities of [16] are matched: No update gives rise to more than a logarithmic number of rebalancing operations, of which at most one is restructuring. Additionally, rebalancing is amortized constant per update. It should also be noted that the potential function used in the proof for amortized constant rebalancing can be modified to satisfy the requirements for Theorem 1 in [14]. Thus, rebalancing in relaxed layered trees is exponentially decreasing with respect to the height.

As it is also pointed out in [29], there are many ways of tuning the operations to improve performance. For instance, several rebalancing operations can be redefined or extended such that push-up requests and pull-down requests would cancel out when possible.

There is also a trade-off in the number of legal configurations and the number of rebalancing operations (and their complexity). For example, one could define relaxed layered trees without the two node configuration with the bottom-most node marked by a pull-down request. However, then the set of operations is increased and some operations must be made larger.

References

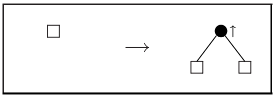
1. G. M. Adel'son-Vel'skiĭ and E. M. Landis. An Algorithm for the Organisation of Information. *Doklady Akadamii Nauk SSSR*, 146:263–266, 1962. In Russian. English translation in *Soviet Math. Doklady*, 3:1259-1263, 1962. 270
2. Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1983. 270
3. R. Bayer. Symmetric Binary B-Trees: Data Structure and Maintenance Algorithms. *Acta Informatica*, 1:290–306, 1972. 270
4. R. Bayer and E. McCreight. Organization and Maintenance of Large Ordered Indexes. *Acta Informatica*, 1(3):97–137, 1972. 270
5. Joan Boyar, Rolf Fagerberg, and Kim S. Larsen. Amortization Results for Chromatic Search Trees, with an Application to Priority Queues. In *Fourth International Workshop on Algorithms and Data Structures*, volume 955 of *Lecture Notes in Computer Science*, pages 270–281. Springer-Verlag, 1995. 270
6. Joan Boyar, Rolf Fagerberg, and Kim S. Larsen. Amortization Results for Chromatic Search Trees, with an Application to Priority Queues. *Journal of Computer and System Sciences*, 55(3):504–521, 1997. 270, 271
7. Joan F. Boyar and Kim S. Larsen. Efficient Rebalancing of Chromatic Search Trees. In *Proceedings of the Third Scandinavian Workshop on Algorithm Theory*, volume 621 of *Lecture Notes in Computer Science*, pages 151–164. Springer-Verlag, 1992. 270
8. Joan F. Boyar and Kim S. Larsen. Efficient Rebalancing of Chromatic Search Trees. *Journal of Computer and System Sciences*, 49(3):667–682, 1994. 270
9. Joaquim Gabarró, Xavier Messeguer, and Daniel Riu. Concurrent Rebalancing on HyperRed-Black Trees. In *Proceedings of the 17th International Conference of the Chilean Computer Science Society*, pages 93–104. IEEE Computer Society Press, 1997. 270
10. Leo J. Guibas and Robert Sedgewick. A Dichromatic Framework for Balanced Trees. In *Proceedings of the 19th Annual IEEE Symposium on the Foundations of Computer Science*, pages 8–21, 1978. 270
11. S. Hanke, Th. Ottmann, and E. Soisalon-Soininen. Relaxed Balanced Red-Black Trees. In *Proc. 3rd Italian Conference on Algorithms and Complexity*, volume 1203 of *Lecture Notes in Computer Science*, pages 193–204. Springer-Verlag, 1997. 270
12. J. E. Hopcroft. Title unknown. Unpublished work on 2-3 trees, 1970. 270
13. S. Huddleston and K. Mehlhorn. A New Data Structure for Representing Sorted Lists. *Acta Informatica*, 17:157–184, 1982. 270
14. Lars Jacobsen and Kim S. Larsen. Exponentially Decreasing Number of Operations in Balanced Trees. In *Seventh Italian Conference on Theoretical Computer Science*, 2001. This volume. 280

15. J. L. W. Kessels. On-the-Fly Optimization of Data Structures. *Communications of the ACM*, 26:895–901, 1983. 270
16. Kim S. Larsen. Amortized Constant Relaxed Rebalancing using Standard Rotations. *Acta Informatica*, 35(10):859–874, 1998. 270, 278, 280
17. Kim S. Larsen. AVL Trees with Relaxed Balance. *Journal of Computer and System Sciences*, 61(3):508–522, 2000. 270
18. Kim S. Larsen and Rolf Fagerberg. B-Trees with Relaxed Balance. In *Proceedings of the 9th International Parallel Processing Symposium*, pages 196–202. IEEE Computer Society Press, 1995. 270
19. Kim S. Larsen and Rolf Fagerberg. Efficient Rebalancing of B-Trees with Relaxed Balance. *International Journal of Foundations of Computer Science*, 7(2):169–186, 1996. 270
20. Kim S. Larsen, Thomas Ottmann, and Eljas Soisalon-Soininen. Relaxed Balance for Search Trees with Local Rebalancing. In *Fifth Annual European Symposium on Algorithms*, volume 1284 of *Lecture Notes in Computer Science*, pages 350–363. Springer-Verlag, 1997. To appear in *Acta Informatica*. 270
21. Kim S. Larsen, Eljas Soisalon-Soininen, and Peter Widmayer. Relaxed Balance through Standard Rotations. In *Fifth International Workshop on Algorithms and Data Structures*, volume 1272 of *Lecture Notes in Computer Science*, pages 450–461. Springer-Verlag, 1997. To appear in *Algorithmica*. 270
22. Kurt Mehlhorn. *Sorting and Searching*, volume 1 of *Data Structures and Algorithms*. Springer-Verlag, 1984. 270
23. Kurt Mehlhorn and Athanasios Tsakalidis. An Amortized Analysis of Insertions into AVL-Trees. *SIAM Journal on Computing*, 15(1), 1986. 270
24. Xavier Messeguer and Borja Valles. HyperChromatic trees: a fine-grained approach to distributed algorithms on RedBlack trees. Tech. Report LSI-98-13-R, Departament de Llenguatges i Sistemes Informàtics, Universitat Politècnica de Catalunya, 1998. 270
25. O. Nurmi, E. Soisalon-Soininen, and D. Wood. Concurrency Control in Database Structures with Relaxed Balance. In *Proceedings of the 6th ACM Symposium on Principles of Database Systems*, pages 170–176, 1987. 270
26. Otto Nurmi and Eljas Soisalon-Soininen. Uncoupling Updating and Rebalancing in Chromatic Binary Search Trees. In *Proceedings of the Tenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 192–198, 1991. 270
27. Otto Nurmi and Eljas Soisalon-Soininen. Chromatic Binary Search Trees—A Structure for Concurrent Rebalancing. *Acta Informatica*, 33(6):547–557, 1996. 270, 271
28. Otto Nurmi, Eljas Soisalon-Soininen, and Derick Wood. Relaxed AVL Trees, Main-Memory Databases and Concurrency. *International Journal of Computer Mathematics*, 62:23–44, 1996. 270
29. Th. Ottmann and E. Soisalon-Soininen. Relaxed Balancing Made Simple. Technical Report 71, Institut für Informatik, Universität Freiburg, 1995. 271, 280
30. Thomas Ottmann and Derick Wood. Updating Binary Trees with Constant Linkage Cost. *International Journal of Foundations of Computer Science*, 3(4):479–501, 1992. 271, 272
31. Neil Sarnak and Robert E. Tarjan. Planar Point Location Using Persistent Search Trees. *Communications of the ACM*, 29:669–679, 1986. 270
32. Eljas Soisalon-Soininen and Peter Widmayer. Relaxed Balancing in Search Trees. In *Advances in Algorithms, Languages, and Complexity*, pages 267–283. Kluwer Academic Publishers, 1997. 270

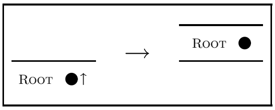
33. Robert Endre Tarjan. Amortized Computational Complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6(2):306–318, 1985. 277

Overview: All the Operations from within the Paper

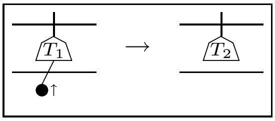
The Sequential Structure



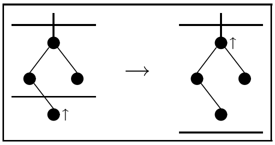
New leaf insertion.



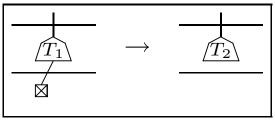
Up root.



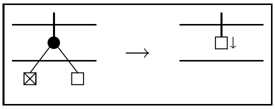
Up finish. $T_1 \in \mathcal{C}_S$. $T_2 \in \mathcal{C}_L$. $|T_2| = |T_1| + 1$.



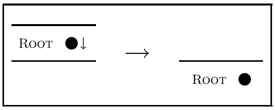
Up push.



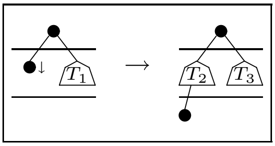
Remove finish. $T_1 \in \mathcal{C}_L$. $T_2 \in \mathcal{C}_S$. $|T_1| = |T_2| + 1$.



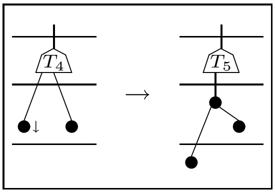
Remove continue.



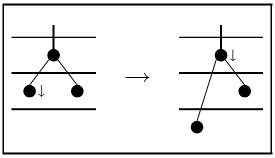
Down root.



Down finish 1. $T_1 \in \mathcal{C}_L$. $T_2, T_3 \in \mathcal{C}_S$. $|T_1| = |T_2| + |T_3|$.

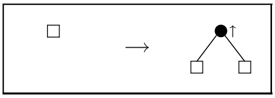


Down finish 2. $T_4 \in \mathcal{C}_L$. $T_5 \in \mathcal{C}_S$. $|T_4| = |T_5| + 1$.

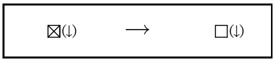


Down push.

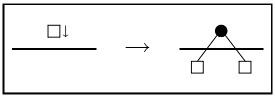
The Relaxed Structure



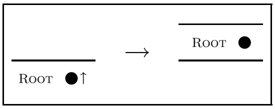
New leaf insertion.



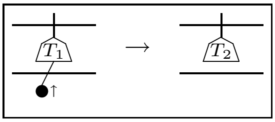
Insert recycle.



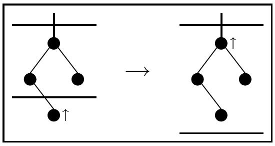
Insert solve.



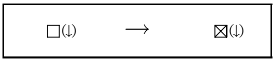
Up root.



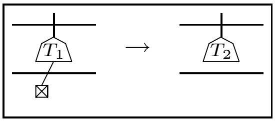
Up finish. $T_1 \in \mathcal{C}_S$. $T_2 \in \mathcal{C}_L$. $|T_2| = |T_1| + 1$.



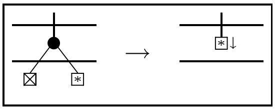
Up push.



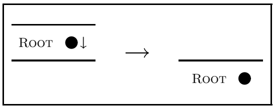
Delete mark.



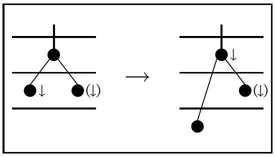
Remove finish. $T_1 \in \mathcal{C}_L$. $T_2 \in \mathcal{C}_S$. $|T_1| = |T_2| + 1$.



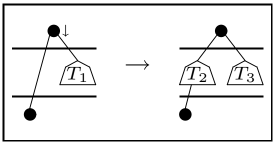
The modified *Remove continue*.



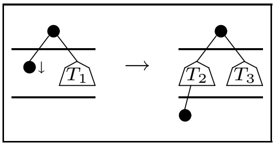
Down root.



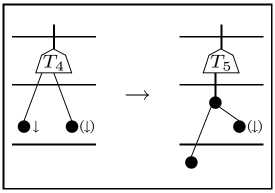
The modified *Down push*.



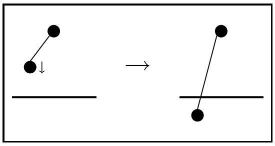
Down cancel. $|T_1| \geq 2$. $|T_2|, |T_3| \geq 1$. $|T_1| = |T_2| + |T_3|$.



Down finish 1. $T_1 \in \mathcal{C}_L$. $T_2, T_3 \in \mathcal{C}_S$. $|T_1| = |T_2| + |T_3|$.



The modified *Down finish 2*. $T_4 \in \mathcal{C}_L$. $T_5 \in \mathcal{C}_S$. $|T_4| = |T_5| + 1$.



Down finish 3.

Distance Constrained Labeling of Precolored Trees*

Jiří Fiala[1,2]**, Jan Kratochvíl[1]***, and Andrzej Proskurowski[3]†

[1] Institute for Theoretical Computer Science and Department of Applied Mathematics, Charles University, Prague
{fiala,honza}@kam.mff.cuni.cz
[2] Christian Albrechts University, Kiel
[3] Department of Computer and Information Science
University of Oregon, Eugene
andrzej@cs.uoregon.edu

Abstract. Graph colorings with distance constraints are motivated by the frequency assignment problem. The so called $\lambda_{(p,q)}$-labeling problem asks for coloring the vertices of a given graph with integers from the range $\{0, 1, \ldots, \lambda\}$ so that labels of adjacent vertices differ by at least p and labels of vertices at distance 2 differ by at least q, where p, q are fixed integers and integer λ is part of the input. It is known that this problem is *NP*-complete for general graphs, even when λ is fixed, i.e., not part of the input , but polynomially solvable for trees for $(p,q)=(2,1)$. It was conjectured that the general case is also polynomial for trees. We consider the precoloring extension version of the problem (i.e., when some vertices of the input tree are already precolored) and show that in this setting the cases $q=1$ and $q > 1$ behave differently: the problem is polynomial for $q=1$ and any p, and it is *NP*-complete for any $p > q > 1$.

1 Introduction

The radio frequency (or channel) assignment problem stems from important practical applications. Its graph theoretical model [9] asks for a labeling of the vertices of an input graph by nonnegative integers so that labels of vertices at distance at most i differ by at least p_i, for every $i \leq k$, where k and $p_1, \ldots, p_k$ are parameters of the problem. As a particular subproblem, Roberts proposed the problem of assigning integers (frequencies) to vertices (transmitters) such that vertices that are “fairly close” to each other (at distance two) receive different

* The authors acknowledge support of joint Czech U.S. grants KONTAKT ME338 and NSF-INT-9802416 during visits of the first two authors to Eugene, OR, and of the third author to Prague.

** Partially supported by EU ARACNE project HPRN-CT-1999-00112. Supported by the Ministry of Education of the Czech Republic as project LN00A056

*** Research partially supported by Czech Research grant GAUK 158/99. Supported by the Ministry of Education of the Czech Republic as project LN00A056

† Supported in part by the grant NSF-ANI-9977524.

A. Restivo, S. Ronchi Della Rocca, L. Roversi (Eds.): ICTCS 2001, LNCS 2202, pp. 285–292, 2001.

labels and vertices that are very close (adjacent) receive labels that are at least two apart. This corresponds to the case of $k = 2$ and $(p_1, p_2) = (2, 1)$, referred to as $(2, 1)$-labelings of graphs [2,4,7,8,10,13,14]. The more general two-parameter problem with $(p_1, p_2) = (p, q)$, $p \geq q > 1$, was considered in [1,4,5].

The minimum λ such that a graph G allows a (p, q)-labeling by integers from the range $\{0, 1, \ldots, \lambda\}$ is denoted by $\lambda_{(p,q)}(G)$. It was shown in [7,14] that determining $\lambda_{(2,1)}(G)$ is an *NP*-complete problem even for graphs G with diameter two. The complexity of deciding $\lambda_{(2,1)}(G) \leq \lambda$ for fixed λ was shown *NP*-complete for every $\lambda \geq 4$ in [4] (answering a problem asked in [14]). It was also shown in [4] that for every $p \geq q \geq 1$, there is a λ (dependent on p, q) such that deciding $\lambda_{(p,q)}(G) \leq \lambda$ is *NP*-complete.

As concerns special graph classes, $\lambda_{(2,1)}(G)$ can be determined efficiently for paths, cycles and wheels [7], and for cographs and trees [2] (thus disproving the conjecture of [7] that the problem is *NP*-complete for trees). D. Welsh suggested [personal communication, 1999] that, by an algorithm similar to Chang and Kuo's, it should be possible to determine $\lambda_{(p,q)}(T)$ for a tree T for arbitrary p, q. We will review the algorithm of [2] in Section 2. The crucial step of the algorithm uses bipartite matchings (or Systems of Distinct Representatives, SDR), and an analogous algorithm for $q > 1$ would need to be able to decide existence of "Systems of *Distant* Representatives". This problem is, however, *NP*-complete in general, as shown in Section 3. It is therefore plausible to conjecture that determining $\lambda_{(p,q)}(T)$ is *NP*-hard for trees, when $q > 1$. Note also that the complexity of determining $\lambda_{(2,1)}(G)$ for graphs of bounded tree-width is not known.

For graph coloring problems, it is natural to consider the *precoloring extension* variants of the problems where some vertices of the input graph are given as already (pre)colored, and the question is if this precoloring can be extended to a proper coloring of the entire graph using a given number of colors. For several results on the complexity of precoloring extension for special graph classes see e.g. [11,12]. The aim of this paper is to consider the precoloring extension variant of (p, q)-labelings of trees. We show that at least in this setting the cases $q = 1$ and $q > 1$ behave quite differently. We will show in Section 2 that Chang and Kuo's algorithm can be easily extended to precolored trees and parameters $(p, 1)$, for any p. On the other hand, as we will show in Section 4, the problem is *NP*-complete for every $p > q > 1$.

2 Algorithm for Trees and $q = 1$

We present a slightly modified version of the algorithm of [2].

The input consists of an integer value λ and a tree T with some vertices precolored by a function $g : U \longrightarrow \{0, 1, \ldots, \lambda\}$, where $U \subseteq V(T)$ is the set of precolored vertices. We first choose a leaf r as a root of T, which defines the parent-child relation between every pair of adjacent vertices. For any edge $xy \in E(T)$ such that x is a child of y (i.e., y is the neighbor of x on the path from x to the root), we denote by T_{xy} the subtree of T rooted in y and containing y, x and all descendants of x. For every such edge and for every pair of colors a, b, we introduce a boolean variable $\phi(x, y; a, b)$ valued true if and only if T_{xy} has a $(p, 1)$-coloring $f : V(T_{xy}) \longrightarrow \{0, 1, \dots, \lambda\}$ which extends the precoloring and such that $f(x) = a, f(y) = b$. We show how to evaluate ϕ in polynomial time (using dynamic programming from the leaves of T towards the root r). This will constitute a polynomial time algorithm for T, since the tree T allows a precoloring extension if and only if there exist colors a, b such that $\phi(r', r; a, b) = \mathsf{true}$, where r' is the only child of r. The evaluation of ϕ goes as folows:

1. Set $\phi(x, y; a, b) = \mathsf{false}$ for every edge xy and any two colors $a, b \in \{0, 1, \dots, \lambda\}$.
2. Consider an edge xy such that x is a leaf. Set $\phi(x, y; a, b) = \mathsf{true}$ for all pairs $a, b \in \{0, 1, \dots, \lambda\}$ such that $|b - a| \geq p$ and, if applicable, such that $a = g(x)$ if $x \in U$ and/or $b = g(y)$ if $y \in U$.
3. Consider an edge xy such that $\phi(x', y'; a, b)$ was already evaluated for all edges $x'y'$ of T_{xy} except xy. Let $z_1, \dots, z_k$ be the children of x.
   For every pair of colors $a, b \in \{0, 1, \dots, \lambda\}$ such that $|b - a| \geq p$, and $a = g(x)$ if $x \in U$ and/or $b = g(y)$ if $y \in U$, define
   $$M_i = \{c| \ \phi(z_i, x; c, a) = \mathsf{true} \text{ and } b \neq c\}, \qquad (*)$$
   and set
   $$\phi(x, y; a, b) = \mathsf{true}$$
   if and only if the set system $M_i, i = 1, 2, \dots, k$, has a System of Distinct Representatives.
4. If there are colors $a, b \in \{0, 1, \dots, \lambda\}$ such that $\phi(r', r; a, b) = \mathsf{true}$ then output "T allows a precoloring extension." else output "T does not allow a $(p, 1)$-labeling extending the precoloring."

The correctness of the algorithm follows by an easy inductive argument. For the time complexity, $\phi(x, y; a, b)$ is evaluated for $n - 1$ edges and for $O(\lambda^2)$ pairs of labels for each edge. The recursive step (assuming x has k children) takes time $O(k\lambda)$ for constructing the sets M_i and then $O((k + \lambda)^2 k\lambda)$ for bipartite matching (SDR). Altogether the running time is $O(n^2\lambda^5)$. Thus we have proved:

Theorem 1. *Existence of a $(p, 1)$-labeling which extends a given precoloring of a given tree can be decided in polynomial time, for any integer p.*

A similar algorithm could be used for the general case of (p, q)-colorings. The only difference is that condition $(*)$ must now read

$$M_i = \{c| \ \phi(z_i, x; c, a) = \mathsf{true} \text{ and } |b - c| \geq q\}$$

and we set $\phi(x, y; a, b) = \mathsf{true}$ if and only if the set system $M_i, i = 1, 2, \dots, k$, allows a System of q-Distant Representatives, i.e., a system of representatives c_i, $i = 1, 2, \dots, k$ such that $c_i \in M_i$ for each i and $|c_i - c_j| \geq q$ for all $i \neq j$. We will show in the next section that this problem is in general *NP*-complete, and hence the algorithm is not straightforwardly polynomial for $q > 1$. It is, however, still possible that the existence of a System of q-Distant Representatives within the algorithm can be decided in polynomial time because of some special properties of the sets M_i. But most likely (unless *P*=*NP*) this can be the case only for the plain (p, q)-labeling problem (when T has no precolored vertices). We will show in Section 4 that for any $p \geq q > 1$, the precoloring extension (p, q)-labeling problem is *NP*-complete.

Note that the hardness results apply only when the bound λ is part of the input:

Theorem 2. *For every fixed p, q and fixed λ, the precoloring extension (p, q)-labeling problem with labels in the range $\{0, 1, \dots, \lambda\}$ is polynomially solvable for trees.*

Proof. Since $\lambda_{(p,q)}(G) \geq q\Delta(G)$ (where $\Delta(G)$ denotes the maximum degree in G), the precoloring may only allow an extension to a (p, q)-labeling if the maximum degree of T is bounded. But then k, the number of children of x in the recursive step is bounded as well and the existence of a System of q-Distant Representatives can be decided in time $O(\lambda^k)$ by brute force.

3 Systems of Distant Representatives

The System of Distinct Representatives is a mapping that assigns to every set of a finite family one of its elements, such that distinct sets are represented by distinct elements. This system can be equivalently described as a matching in bipartite graph.

The theory of Systems of Distinct Representatives is well known and very important for discrete optimization problems. Both for the elegant Hall theorem that describes necessary and sufficient conditions for the existence of an SDR, and for a polynomial time algorithm (augmenting paths or the matching algorithm of Edmonds [3]). Though several generalizations of the concept have been studied, we believe that the concept of *distant* representatives is new.

Definition 1. *Given integers q and m and a family $\mathcal{F} = \{M_i |\ i \in I\}$ of subsets of $X = \{1, 2, \dots, m\}$, a mapping $f : I \longrightarrow X$ is called a* System of q-Distant Representatives *(shortly an Sq-DR) if*

(1) $f(i) \in M_i$ for every $i \in I$,
(2) $|f(i) - f(j)| \geq q$ for every $i, j \in I, i \neq j$.

Note that for $q = 1$ the condition (2) merely says that $f(i) \neq f(j)$, i.e., a System of 1-Distant Representatives is a System of Distinct Representatives. For this case the ordering of the elements of X becomes irrelevant (as long as any two elements of X are at distance at least 1). For $q > 1$, the linear ordering of the elements of X becomes a determining factor. The fact that deciding whether a given family has an Sq-DR is *NP*-complete for $q > 1$ is interesting on its own. For the purpose of λ-colorings we need a somewhat stronger result. We call a set of numbers *t-sparse* if $|x - y| \geq t$ for every two distinct members x, y of the set.

Theorem 3. *For every $q > 1$ and every t, it is NP-complete to decide if a family $\mathcal{F}$ of t-sparse sets of integers has a System of q-Distant Representatives.*

Proof. We reduce from 3-Satisfiability of Boolean formulas in conjunctive normal form. This problem is known to be *NP*-complete even when restricted to formulas whose each clause contains 2 or 3 literals and every variable occurs in exactly 3 clauses – once positive and twice negated [6]. Suppose $\Phi = (V, C)$ is such a formula (where V is its variable set and C its clause set). We assume that the variables are numbered $x_1, x_2, \ldots, x_n$.

Fix a number $s > t + q \geq 2q$ (we may assume without loss of generality that $t \geq q$). For a variable x_i, let clause c contain the positive occurence of x_i and let clauses d, e contain $\neg x_i$. Denote $x_i(c) = (i-1)s + 2$, $x_i(d) = (i-1)s + 1$ and $x_i(e) = (i-1)s + q + 1$. For every clause $c \in C$ create a 3-element set $M_c = \{x_i(c) | x_i \in c \text{ or } \neg x_i \in c\}$.

Observe first that the sets M_c, $c \in C$ are t-sparse. This is because the smallest difference of any two numbers $x_i(c), x_j(d)$, $i \neq j$ is at least $s - q > t$.

We claim that $\mathcal{M} = \{M_c\}_{c \in C}$ contains an Sq-DR if and only if Φ is satisfiable. Suppose first that $\mathcal{M}$ contains an Sq-DR f. We define a truth valuation ϕ of the variables V so that

- $\phi(x_i) = \mathsf{true}$ if $f(c) = x_i(c)$ for some clause c such that $x_i \in c$,
- $\phi(x_i) = \mathsf{false}$ if $f(c) = x_i(c)$ for some clause c such that $\neg x_i \in c$,
- $\phi(x_i)$ is arbitrary if none of the above applies.

Obviously every clause is satisfied by this assignment. We have to show, though, that ϕ is defined correctly. Assume to the contrary that for some variable x_i, $\phi(x_i)$ should be both true and false. That means that $(i-1)s + 2$ and at least one of $(i-1)s+1$ and $(i-1)s+q+1$ are both representatives of some sets. This is, however, impossible, since their difference $(i-1)s + 2 - ((i-1)s + 1) = 1$ or $(i-1)s + q + 1 - ((i-1)s + 2) = q - 1$ would be less than q.

On the other hand, suppose that Φ is satisfied by a truth valuation ϕ. For every clause c, pick a variable (say x_i) which satisfies c and set $f(c) = x_i(c)$. This is clearly a set of representatives of $\mathcal{M}$. To see that the system is q-distant, note that if different clauses c and d are satisfied by different variables then the difference of $f(c)$ and $f(d)$ is at least $s - q > t \geq q$, while if c and d are satisfied by the same variable, say x_i, then this variable must occur negated in both clauses, and hence $f(c) = (i-1)s + 1$ and $f(d) = (i-1)s + q + 1$ (or vice versa), and thus their difference is q.

4 *NP*-Completeness for $q > 1$

We prove the main result in this section:

Theorem 4. *For every fixed $p \geq q > 1$, it is NP-complete to decide if a precoloring of a tree can be extended to a (p, q)-labeling whose labels do not exceed a given λ.*

Proof. We show a reduction from Systems of q-Distant Representatives. Let $t \geq q^2 + 2p$ and let a family $\mathcal{M} = \{M_i\}_{i=1}^n$ of t-sparse sets be given. We may assume that $\min(\bigcup_{i=1}^n M_i) \geq t$ and we set $\lambda = t + \max(\bigcup_{i=1}^n M_i)$.

We build a tree T with a root v_0 and children of the root $v_i, i = 1, 2, \dots, n$ (and another level of nodes $v_{i,j,m}$, as defined later). The root will be precolored by 0, vertices v_i will not be precolored, but each of them will have a certain number of children precolored so that the only admissible colors for v_i will be exactly the elements of M_i. This can be attained as follows.

For a vertex v_i, let $M_i = \{a_{i,1} < a_{i,2} < \dots < a_{i,k_i}\}$. Now consider the interval $[a_{i,j} + 1, a_{i,j+1} - 1]$. Since M_i is t-sparse, $a_{i,j+1} - a_{i,j} \geq q^2 + 2p$ and one can choose numbers $c_{i,j,1} = a_{i,j} + p < c_{i,j,2} < \dots < c_{i,j,l_j} = a_{i,j+1} - p$ so that $q \leq c_{i,j,m+1} - c_{i,j,m} \leq 2p - 1$ for every $m = 1, 2, \dots, l_j - 1$. (In more detail, if $a_{i,j+1} - a_{i,j} - 2p = \alpha q + \beta$ then $\alpha \geq q$ and we set $l_j = \alpha + 1$. Then take $c_{i,j,m+1} = c_{i,j,m} + q + 1$ for $m = 1, 2, \dots, \beta$ and take $c_{i,j,m+1} = c_{i,j,m} + q$ for $m = \beta + 1, \dots, \alpha$. Note that $2p - 1 \geq q + 1 > q$.) Finally for all $m = 1, 2, \dots, l_j$ add leaves $v_{i,j,m}$ precolored by $c_{i,j,m}$ pending on the vertex v_i. In every (p, q)-labeling f, it must be $f(v_i) \notin [a_{i,j} + 1, a_{i,j+1} - 1]$, since any integer from this interval differs by at most $p - 1$ from some $c_{i,j,m}$.

We perform this construction for every vertex v_i and for each interval $[a_{i,j} + 1, a_{i,j+1} - 1]$. Because we have assumed that $a_{i,1} \geq t$ and $\lambda - a_{i,k_i} \geq t$, the initial and terminal intervals $[0, a_{i,1} - 1]$ and $[a_{i,k_i} + 1, \lambda]$ are handled in the same way, with dummy border values $a_{i,0} = -1$ and $a_{i,k_i+1} = \lambda + 1$.

It follows that T allows a (p, q)-labeling which extends the precoloring (and uses labels from $\{0, 1, \dots, \lambda\}$) if and only if $\mathcal{M}$ has a System of q-Distant Representatives.

Suppose that $f : V(T) \longrightarrow \{0, 1, \dots, \lambda\}$ is a (p, q)-labeling which extends the precoloring. For each i, $f(v_i) \in M_i$, since (by the argument above) for every j, $f(v_i) \notin [a_{i,j} + 1, a_{i,j+1} - 1]$. Since for $i \neq i'$, v_i and $v_{i'}$ have a common neighbor (the root v_0), we have $|f(v_i) - f(v_{i'})| \geq q$ and f restricted to $\{v_1, v_2, \dots, v_n\}$ yields a System of q-Distant Representatives for $\mathcal{M}$.

The converse, i.e., showing that T allows a (p, q)-labeling extending the precoloring, provided $\mathcal{M}$ allows a System of q-Distant Representatives, is straightforward. Note here that the precoloring of T constructed from $\mathcal{M}$ itself satisfies the (p, q)-constraints, since for each i the colors used on $v_{i,j,m}$ are at least q apart, they are at least p (and thus at least q away from the label 0 of the root), and they do not interfere with other $v_{i'}$ or $v_{i',j',m'}$ since such vertices are at distance at least three.

5 Conclusion and Open Problems

We have shown a significant difference in the computational complexity of (p, q)-labeling of trees for $q = 1$ and $q > 1$, when some vertices are precolored. This has been achieved by introducing the notion of q-distant representatives for set systems, which generalizes the well known distinct representatives (bipartite matching). We believe that the result that Sq-DR is *NP*-complete for $q > 1$ (while S1-DR, identical with SDR, is polynomial) is interesting on its own.

The following generalization of Systems of Distant Representatives was raised during discussion of the second author and L. Lovász. Let $\mathcal{A}$ be a class of graphs. The input to the S$\mathcal{A}$-IR (*System of $\mathcal{A}$-Independent Representatives*) is a graph $G \in \mathcal{A}$ and a family $\mathcal{F}$ of subsets of the vertex set of G. The question is whether $\mathcal{F}$ allows a System of Distinct Representatives such that the representatives form an independent set in G. Our results show that S$\mathcal{A}$-IR is *NP*-complete if for every n, the class $\mathcal{A}$ contains a graph G with $n \cdot K_{1,2}$ (n disjoint copies of $K_{1,2}$) as an induced subgraph. It would be very interesting to know whether this problem is polynomially solvable for classes of graphs without induced $n \cdot K_{1,2}$ for large n. This would be the first step in the full classification of classes $\mathcal{A}$ for which the question is solvable in polynomial time and for which is *NP*-complete.

Let us remark that S$\mathcal{A}$-IR is at least as difficult as Independent Set restricted to graphs of $\mathcal{A}$: Given a graph $G \in \mathcal{A}$ and a number k, we take G and the multiset of k copies of $V(G)$ as $\mathcal{F}$. Then this family allows a System of Independent Representatives if and only if $\alpha(G) \geq k$. Observe also that S$\mathcal{A}$-IR is solvable in polynomial time if for every graph of $\mathcal{A}$, all its maximal independent sets can be enumerated in polynomial time. (Try the maximal independent sets one by one and solve bipartite matching for each of them.) On the other hand, for $\mathcal{A}$={disjoint unions of complete graphs}, this condition is not fulfilled but S$\mathcal{A}$-IR could be solved in polynomial time by contraction and bipartite matching. Finally, for $\mathcal{A}$={cographs}, the Independent Set problem allows a polynomial-time algorithm, in contrary to the *NP*-completeness of S$\mathcal{A}$-IR (since $n \cdot K_{1,2}$ is a cograph for any n).

For the (p, q)-labeling problem of trees without precolored vertices, the complexity is still open when $q > 1$. It is tempting to try to prove *NP*-completeness along the lines above. One possibility would be to replace precolored vertices by trees that allow only certain labels at the root. Of course one cannot ask for trees that would allow only one possible label, since the set of admissible labels for a particular vertex is always symmetric with respect to the interval $[0, \lambda]$ (if f is a (p, q)-labeling then so is $f' = \lambda - f$). This does not cause problems for the desired reduction since one can show that Systems of q-Distant Representatives are *NP*-complete even when all sets $M_i \in \mathcal{M}$ are symmetric. This observation leads to the following open problems (affirmative solution to the first one would imply *NP*-completeness of (p, q)-labelings of trees):

Problem 1 Does there exist, for any relatively prime $p > q > 1$, any sufficiently large λ and any (q^2+2p)-sparse set M, a construction of trees $T_{x,\lambda}$, $x \in [p, \lambda - p]$, such that

(1) the size of $T_{x,\lambda}$ is polynomial in λ,

(2) $T_{x,\lambda}$ allows a (p, q)-labeling in which the root is labeled by x and all labels of children of the root are at distance at least q from the set M,

(3) in any (p, q)-labeling of $T_{x,\lambda}$, the root is labeled either by x or by $\lambda - x$?

The condition (1) is imposed to guarantee polynomiality of the desired reduction. However, we do not even know of the existence of any $T_{x,\lambda}$ satisfying at least (2-3), and therefore we deem the following problem interesting on its own:

Problem 2 Prove that for any relatively prime $p > q > 1$, any large enough λ and any $x \in [p, \lambda - p]$, there exists a tree $T_{x,\lambda}$ such that in any (p, q)-labeling of $T_{x,\lambda}$ the root is labeled either by x or by $\lambda - x$, and $T_{x,\lambda}$ has such a labeling.

References

1. Battiti, R., A. A. Bertossi, and M. A. Bonuccelli, Assigning codes in wireless networks: Bounds and scaling properties. *Wireless Networks* **5**, 3, (1999), pp. 195–209. 286
2. Chang, G. J. and D. Kuo, The $L(2,1)$-labeling problem on graphs, *SIAM J. Disc. Math.* **9**, (1996), pp. 309–316. 286
3. Edmonds, J., Paths, trees and flowers, *Can. J. Math.* **17**, (1965), pp. 449–467. 288
4. Fiala, J., T. Kloks, and J. Kratochvíl, Fixed-parameter complexity of λ-colorings, In: Graph Theoretic Concepts in Computer Science, WG'99, LNCS 1665, (1999), pp. 350–363. 286
5. Fotakis, D., G. Pantziou, G. Pentaris, and P. Spirakis, Frequency assignment in mobile and radio networks, In: Networks in distributed computing. DIMACS workshop, Ser. Discrete Math. Theor. Comput. Sci. 45, (1997), pp. 73–90. 286
6. Garey, M. R. and D. S. Johnson, *Computers and Intractability*, W. H. Freeman, 1979. 289
7. Griggs, J. R. and R. K. Yeh, Labelling graphs with a condition at distance 2, *SIAM J. Disc. Math.* **5**, (1992), pp. 586–595. 286
8. Georges, J. P. and D. W. Mauro, On the size of graphs labeled with a condition at distance two, *Journal of Graph Theory* **22**, (1996), pp. 47–57. 286
9. van den Heuvel, J., R. A. Leese, and M. A. Shepherd, Graph labeling and radio channel assignment, *Journal of Graph Theory*, **29**, (1998), pp. 263–283. 285
10. Jonas, K., *Graph colorings analogues with a condition at distance two: $L(2,1)$-labelings and list λ-labelings*, Ph.D. thesis, University of South Carolina, 1993. 286
11. Hujter, M. and Zs. Tuza, Precoloring extension III. Classes of perfect graphs, *Combin. Probab. Comput.* **5**, (1996), pp. 35–56. 286
12. Kratochvíl, J. and A. Sebő, Coloring precolored perfect graphs, *J. Graph. Th.* **25**, (1997), pp. 207–215. 286
13. Liu, D. D.-F. and R. K. Yeh, On distance two labellings of graphs, *ARS Combinatorica* **47**, (1997), pp. 13–22. 286
14. Yeh, K.-Ch., *Labeling graphs with a condition at distance two*, Ph.D. Thesis, University of South Carolina, 1990. 286

Exponentially Decreasing Number of Operations in Balanced Trees*

Lars Jacobsen and Kim S. Larsen

University of Southern Denmark, Department of Mathematics and Computer Science
Main Campus: Odense University, Campusvej 55, DK–5230 Odense M, Denmark
{eljay,kslarsen}@imada.sdu.dk

Abstract. While many tree-like structures have been proven to support amortized constant number of operations after updates, considerably fewer structures have been proven to support the more general exponentially decreasing number of operations with respect to distance from the update. In addition, all existing proofs of exponentially decreasing operations are tailor-made for specific structures. We provide the first formalization of conditions under which amortized constant number of operations imply exponentially decreasing number of operations. Since our proof is constructive, we obtain the constants involved immediately. Moreover, we develop a number of techniques to improve these constants.

1 Introduction

When asynchronous processes work on a shared tree-like structure, operations which are carried out by one process near the root are likely to interfere with and slow down other processes. In contrast, if the tree structure is large, then operations near the leaves will most likely not disturb others (this of course is application-dependent). This scenario is one motivation for considering analyses of where operations are carried out.

Many tree-like structures have been proven to have amortized constant time operations (see for instance [5,6,11,12,14]). If operations are initiated at the leaves of trees and move towards the root by local operations, this gives some hope that operations will not often be carried out close to the root. In particular, this hope is justified if some balance constraints on the trees guarantee that all leaves have some minimum non-constant distance to the root. A typical example of such a scenario is bottom-up rebalancing in balanced search trees. Tailor-made proofs of exponentially decreasing operations exist for (a, b)-trees [6] and insertions into AVL-trees [12].

The main contribution of this paper is a formalization of these concepts of balance and locality, and a proof that under these conditions, amortized constant time operations imply that the number of rebalancing operations carried out at a certain level in the tree decreases exponentially in the distance from the leaves.

* Supported by the IST Programme of the EU (ALCOM-FT) and the Danish SNF.

A. Restivo, S. Ronchi Della Rocca, L. Roversi (Eds.): ICTCS 2001, LNCS 2202, pp. 293–311, 2001.

We have focused on formulating sufficient conditions that are as weak as possible such that our theorem is as broadly applicable as possible. This means that the many structures on which operations have been shown to be amortized constant can now, with very few extra arguments, claim to provide the stronger and more directly useful exponentially decreasing operations.

However, our proof is constructive, meaning that once exponentially descreasing operations have been established, constants can also be derived. More precisely, we obtain constants c_1 and c_2 such that the theorem guarantees that at most $c_1 \frac{u}{{c_2}^i}$ operations are carried out at a distance i from the leaves in response to u initiations of amortized constant time operations from the leaves. The constant c_2 is of course particularly interesting; the larger it is, the better.

The rest of this paper is organized as follows: In Section 2 we state and prove the main theorem, and give an example of its use. In Section 3 we show how to improve the constants obtained from the theorem for certain structures. In Section 4 we demonstrate a technique to make non-local structures, which are not naturally covered by the theorem, satisfy the theorem after all. Finally, in Section 5 we show how to make the theorem applicable to the large class of search trees with relaxed balance.

2 The Main Theorem

We now address the statement and proof of the main theorem. We first give a few definitions.

Let T be a tree. A *configuration* is a constant number of connected nodes. A function $\mathcal{L} : T \to \mathbb{N}_0$ is a *layer function* on T if for nodes $u, v \in T$, where v is a proper ancestor of u, $\mathcal{L}(v) \geq \mathcal{L}(u)$, and there exists a constant k such that for all nodes $u, v \in \mathcal{C}$ for any configuration $\mathcal{C}$, $|\mathcal{L}(u) - \mathcal{L}(v)| \leq k$. This naturally defines a *layer* $\ell_i(T) = \{v \in T \mid \mathcal{L}(v) = i\}$ as a subset of T.

A *local potential function* Φ on T is a potential function defined on each node of the tree, such that the potential of a node is a function of a surrounding configuration, and the potential of any subset $S \subseteq T$ of the tree is exactly the sum of the potentials of the nodes in S. In addition, no node has non-constant or negative potential, i.e., there exists some constant c such that $0 \leq \Phi(u) \leq c$ for all nodes $u \in T$.

A *local rule* is a transformation on T describing *before* and *after* configurations. If a configuration in the tree matches the before configuration, it may be transformed to the after configuration. Moreover, there exists some configuration $\mathcal{C}$ containing the before configuration such that for all nodes $u \in T \setminus \mathcal{C}$, neither u, $\Phi(u)$ nor $\mathcal{L}(u)$ are changed by the tranformation.

We distinguish between transformations controlled by external events (from here on referred to as *updates*) and transformations where the before configuration matches a configuration in the tree created by another transformation (from here on referred to as *post-processing operations* or just *operations*).

Theorem 1. *Let T be a tree. Let $\mathcal{L}$ be a layer function on T. Let Φ be a local potential function on T such that every update on T is made at most a constant* *number of layers from layer* 0 *and increases the potential by at most a constant, and every operation decreases the potential by at least a constant. Then there exist constants* c_1, c_2*, satisfying* $c_2 > 1$ *such that the number of operations performed on layer* i *is bounded by* $c_1 \dfrac{\#\text{upd}}{c_2^i}$*, when starting with a tree with potential zero.*$\diamond$

First observe that the restriction on the potential changes by the transformations ensures that post-processing can be done in $O(1)$ amortized time.

In the following, let $\mathcal{U}$ denote the set of updates and let $\mathcal{R}$ denote the set of operations.

When an operation $t \in \mathcal{R}$ is performed, the potential of the entire tree T is decreased. However, since the potential of a node is based on its surrounding configuration, and operations are defined in terms of local rules on configurations, an operation can only affect the potential on a constant number of consecutive layers. This change can be expressed as:

$$\Delta\Phi_t(T) = \sum_{j=j_0}^{j_0+k} \Delta\Phi_t(\ell_j(T))$$

for some j_0 and constant k such that $\Delta\Phi_t(\ell_{j_0}(T))$ and $\Delta\Phi_t(\ell_{j_0+k}(T))$ are both non-zero.

For all operations $t \in \mathcal{R}$, we say that t *spans* k layers and denote this k by k_t. We now define the layer of an operation. For $j_0 \leq i \leq j_0 + k_t$, let

$$\mathcal{N}_t^i = \sum_{j=j_0}^{i} \Delta\Phi_t(\ell_j(T)) \quad \text{, and} \quad \mathcal{P}_t^i = \sum_{j=i+1}^{j_0+k_t} \Delta\Phi_t(\ell_j(T)).$$

Define the set: $\mathcal{I} = \left\{ j_0 \leq i \leq j_0 + k_t \;\middle|\; \mathcal{N}_t^i < 0 \text{ and } \mathcal{P}_t^i \geq 0 \right\}$. Note that this set is non-empty since every operation decreases the potential of the tree, and thus $j_0 + k_t \in \mathcal{I}$. Let $i_t = \min\left\{ i \in \mathcal{I} \;\middle|\; i \text{ minimizes } \mathcal{N}_t^i \right\}$. We say that the operation t is performed on layer i_t.

If we consider an update $t \in \mathcal{U}$, the total potential increase can also be expressed as a sum of changes on a constant number of consecutive layers $j_0, j_0 + 1, \ldots, j_0 + k$ as above. Letting $j_0 = 0$, we say t spans $k_t = k + 1$ layers. Observe that $\Delta\Phi_t(\ell_{j_0}(T))$ might be zero. By definition, the layer of an update is 0.

In the following, we use the shorthand $\Delta\Phi_t^i$ for $\Delta\Phi_t(\ell_i(T))$.

Proof (of Theorem 1). To show the exponentially decreasing post-processing, we want to bound the amount of potential that is moved upwards in the tree, and we want to bound the number of layers this potential is moved.

Let $t \in \mathcal{R}$ be an operation on T. Define:

$$\mathcal{N}'_t = -\sum_{\Delta\Phi_t^i < 0} \Delta\Phi_t^i \quad , \quad \mathcal{P}'_t = \sum_{\Delta\Phi_t^i \geq 0} \Delta\Phi_t^i \quad \text{, and} \quad c_t = \frac{\mathcal{P}'_t}{\mathcal{N}'_t} < 1.$$

Now c_t denotes the largest possible fraction of potential that is moved upwards by t, and this potential is moved at most k_t layers.

Let $k_r = \max_{t \in \mathcal{R}} \{k_t\}$, $k_u = \max_{t \in \mathcal{U}} \{k_t\} - 1$, $c = \max_{t \in \mathcal{R}} \{c_t\}$, and $c_u = \max_{t \in \mathcal{U}} \{\Delta\Phi_t(T)\}$.

We can now bound the amount of potential that is (eventually) created on layer i due to some fixed update $t \in \mathcal{U}$. We denote this by $\Delta\Phi^t_\infty(\ell_i(T))$.

Claim: $\Delta\Phi^t_\infty(\ell_i(T)) \leq c_u \cdot c^{\left\lceil \frac{i-k_u}{k_r} \right\rceil}$.

This is clearly true for $i \in [0, \ldots, k_u]$ since the update might create this much potential on layer i. What remains is to show that for all $j \geq 1$: $\Delta\Phi^t_\infty(\ell_i(T)) \leq c_u \cdot c^j$, for $i \in [k_u + (j-1) \cdot k_r + 1, \ldots, k_u + j \cdot k_r]$

However, this follows from the fact that no potential can reach any layer within this range unless it has been moved at least j times by an operation. Since any operation moves at most the fraction $c < 1$ of the potential, $c_u \cdot c^j$ is an upper bound on the potential that might reach layer i.

By straightforward calculations, we obtain:

$$c_u \cdot c^{\left\lceil \frac{i-k_u}{k_r} \right\rceil} \leq c_u \cdot \sqrt[k_r]{c}^{\,(i-k_u)} = \frac{c_u}{\sqrt[k_r]{c}^{\,k_u}} \sqrt[k_r]{c}^{\,i}.$$

If we consider this when #upd updates are made, we have

$$\Delta\Phi_\infty(\ell_i(T)) \leq \frac{c_u}{\sqrt[k_r]{c}^{\,k_u}} \sqrt[k_r]{c}^{\,i} \cdot \#\text{upd}.$$

Now consider the application of an operation t. By the definition of i_t, we must have that $\Delta\Phi^{i_t}_t < 0$, i.e., whenever an operation is performed on layer i, the potential of layer i is decreased. Let $c_r = \min_{t \in \mathcal{R}}\{-\Delta\Phi^{i_t}_t\} > 0$, then, in total, the potential on layer i is decreased by at least $c_r \cdot \#\text{ops}_i$, where $\#\text{ops}_i$ denotes the total number of operations performed on layer i. Combining these inequalities, we get

$$0 \leq \Phi(\ell_i(T)) \leq \frac{c_u}{\sqrt[k_r]{c}^{\,k_u}} \sqrt[k_r]{c}^{\,i} \cdot \#\text{upd} - c_r \cdot \#\text{ops}_i$$
$$\#\text{ops}_i \leq \frac{c_u}{c_r \cdot \sqrt[k_r]{c}^{\,k_u}} \sqrt[k_r]{c}^{\,i} \cdot \#\text{upd}.$$

If we let $c_1 = \frac{c_u}{c_r \cdot \sqrt[k_r]{c}^{\,k_u}}$ and $c_2 = \left(\sqrt[k_r]{c}\right)^{-1}$, the theorem follows.

Observe that the layer function should define a non-constant number of layers in order for the theorem to provide new information. Moreover, if h is the length of the shortest path of T, then the layer of the top-most node in the tree must be in $O(h)$, since otherwise we would have that some configuration spans a nonconstant number of layers, or updates are made a non-constant distance from layer 0.

Since two applicable layer functions may differ by an additive constant, and the exact layer on which an operation is carried out can be defined in several reasonable ways, the proper interpretation of Theorem 1 is that $\#\text{ops}_i \in O\left(\frac{\#\text{upd}}{c_2^i}\right)$, i.e., the constant in front of the expression is of minor interest.

If the initial tree has a non-zero potential of Φ_0, then this potential can create at most a linear amount of extra work on each layer. Thus,

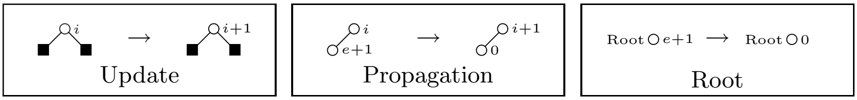


Fig. 1. Transformations for pebble games

$\#\text{ops}_i \in O\left(\Phi_0 + \frac{\#\text{upd}}{c_2^i}\right)$. In [9], the existence of trees with arbitrary sizes and constant potential is discussed.

2.1 Partially Persistent Binary Trees

We consider partial persistence as in [5]. A study of partial persistence through 'pebble games' has been carried out in [4], and we use a similar presentation here.

We consider the following game played on a complete binary tree of height h. Each internal node has, besides pointers to its children, room for e pebbles, for some constant e. Updates consists of updating a leaf, and putting a pebble on the parent (this is the equivalent of copying a leaf and using an extra-pointer at the parent in the node-copying model [5]). Post-processing consists of making sure no internal node is storing more than e pebbles. The possible operations are to remove all pebbles from a node and put one on the parent (this is the equivalent of copying an internal node and using an extra-pointer at the parent in the node-copying model), or removing all pebbles from the root.

Since all transformations are described by local rules, we assume that nodes are capable of storing $e + 1$ pebbles to represent the intermediate state between two consecutive operations.

The transformations[1] are depicted in Figure 1. The number to the right of a node u is the number of pebbles on this node, denoted $\pi(u)$. We then define a layer function $\mathcal{L}(u) = h(u)$, where $h(u)$ denotes the height of the node u (leaves at height 0) and a local potential function $\Phi(u) = \pi(u)$. Using these transformations and the functions $\mathcal{L}$ and Φ, we have proven the following corollary.

Corollary 1. *The number of persistence operations in a balanced binary tree is exponentially decreasing with respect to the distance from the leaves.* ◇

We now turn to use the constructive part of the proof of Theorem 1 to derive constants for the exponential expression. Since an update puts a pebble on an internal node on layer 1, we immediately have: $c_u = 1$ and $k_u = 1$.

To derive the remaining constants, we analyze each operation one by one. A table such as Table 1 can be constructed for the potential changes by each operation. From the table we derive: $k_r = 1$, $c_r = e + 1$, and $c = \frac{1}{e+1}$. Thus, the

[1] The set of transformations showed in this example (and any of the following) is only complete up to symmetric variants of the transformations.

Table 1. Potential changes by operations for the pebble game

Operation t	$\Delta\Phi_t^{i_t}$	$\Delta\Phi_t^{i_t+1}$	c_t
Propagation	$-(e+1)$	1	$\frac{1}{e+1}$
Root	$-(e+1)$	0	0

number of persistence operations at height i, denoted $\#\mathrm{ops}_i$, is bounded by:

$$\#\mathrm{ops}_i \leq \frac{1}{(e+1)\cdot\frac{1}{e+1}} \cdot \frac{\#\mathrm{upd}}{(e+1)^i} = \frac{\#\mathrm{upd}}{(e+1)^i},$$

where #upd denote the number of updates to the tree.

Observe that the constant e is completely independent of the size of the tree, so the exponential expression in the denominator is independent of the size of the subtree. Moreover, this bound is tight, which can easily be verified by considering a sequence of updates to the same leaf.

3 Splitting Large Transformations

In this section, we present a construction to reduce the span of any transformation to one, by replacing the transformation by a sequence of transformations.

Let t be a transformation on T. We use the notation $B \overset{t}{\longmapsto} A$ to denote that applying t to the before configuration B yields the after configuration A.

Lemma 1. *Let T be a tree satisfying the conditions in Theorem 1 with layer function $\mathcal{L}$ and potential function Φ. Let $t \in \mathcal{U} \cup \mathcal{R}$ be a transformation on T with $k_t \geq 1$. Then there exists a sequence of configurations $C_2, C_3, \ldots, C_{k_t-1}$, a sequence of transformations $t_1, t_2, \ldots, t_{k_t-1}$, satisfying for all i, $1 \leq i \leq k_t - 1$: $k_{t_i} = 1$ and $1 > c_{t_i} \geq c_t$, a layer function $\mathcal{L}'$, and a potential function Φ' on T, such that $B \overset{t_1}{\longmapsto} C_2 \overset{t_2}{\longmapsto} \ldots \overset{t_{k_t-1}}{\longmapsto} A$ and T satisfies Theorem 1 using $\mathcal{L}'$ and Φ'. Moreover, no transformation on T other than t_i can be applied to the configuration C_i for $i \geq 2$.*

Proof. The proof is by induction on k_t. The base case ($k_t = 1$) is trivial. Assume that the lemma is true for all $k_t < N$. Consider the case $k_t = N$. We show how to replace a transformation $B \overset{t}{\longmapsto} A$ by two transformations t_1 and t' such that $B \overset{t_1}{\longmapsto} C \overset{t'}{\longmapsto} A$ for some configuration such that $k_{t_1} = 1$ and $k_{t'} = k_t - 1$. The lemma will follow by application of the induction hypothesis on t'.

Observe that by the definition of the local potential function, the potential on layer i cannot change, unless some node is present on this layer. Therefore, to control the potential on every layer from j_0 to $j_0 + k_t$ we must ensure that a node is present on each layer. However, since $\Delta\Phi^t_t j_0 + k_t$ is non-zero, a node is

Table 2. Potential changes by t_1, t' and t

Operation τ	$\Delta\Phi^{\tau}_{t} j_0$	$\Delta\Phi^{\tau}_{t} j_0 + 1$	$\Delta\Phi^{\tau}_{t} j_0 + 2 \cdots$	$\Delta\Phi^{\tau}_{t} j_0 + k_t$
t_1	$\Delta\Phi^{t}_{t} j_0$	$-x \cdot \Delta\Phi^{t}_{t} j_0$	0	0
t'	0	$\Delta\Phi^{t}_{t} j_0 + 1 + x \cdot \Delta\Phi^{t}_{t} j_0$	$\Delta\Phi^{t}_{t} j_0 + 2$	$\Delta\Phi^{t}_{t} j_0 + k_t$
t	$\Delta\Phi^{t}_{t} j_0$	$\Delta\Phi^{t}_{t} j_0 + 1$	$\Delta\Phi^{t}_{t} j_0 + 2$	$\Delta\Phi^{t}_{t} j_0 + k_t$

present on this layer in either B or A (or both). We will only consider the first case here, the latter is analogous.

Let $v \in \ell_{j_0+k_t}(T) \cap B$ before t is applied. We now add two new types of nodes to T. A *neutral* node w^{t_1} having at most one child and a *marker* node for t_1 denoted v^{t_1}. Let v^{t_1} be a copy of v except that the degree of v^{t_1} is one higher than the degree of v.

Define a new layer function:

$$\mathcal{L}'(u) = \begin{cases} \mathcal{L}(v) & \text{, if } u = v^{t_1} \\ \mathcal{L}(u) & \text{, if } u \notin \{w^{t_1}, v^{t_1}\} \\ \mathcal{L}'(u.p) - 1 & \text{, otherwise} \end{cases}$$

Where $u.p$ ($u.p^i$) denotes the (ith transitive) ancestor of u.

The transformation t_1 now replaces the node v by the marker node v^{t_1} and adds a chain of k_t w^{t_1}'s as the extra child of v^{t_1}. We now have at least one node on every layer. The transformation t' removes the w^{t_1}'s again, replaces the marker v^{t_1} by v, and applies t. By weigthing the neutral nodes in the new potential function according to their distance to the marker, the desired properties of t_1 and t' can be obtained. See also Table2 for potential changes on the different layers.

$$\Phi'(u) = \begin{cases} \Phi(v) & \text{, if } u = v^{t_1} \\ \Phi(u) & \text{, if } u \notin \{w^{t_1}, v^{t_1}\} \\ \Delta\Phi^{t}_{t} j_0 & \text{, if } u = w^{t_1} \text{ and } u.p^{k_t} = v^{t_1} \\ -x \cdot \Delta\Phi^{t}_{t} j_0 & \text{, if } u = w^{t_1} \text{ and } u.p^{k_t - 1} = v^{t_1} \\ 0 & \text{, otherwise} \end{cases}$$

What remains is to choose x. Observe that $c_{t_1} = x$ if $\Delta\Phi^{t}_{t} j_0$ is negative, and $c_{t_1} = \frac{1}{x}$ otherwise. So x should simply be chosen such that $0 > \Delta\Phi_{t_1}(T) > \Delta\Phi_{t}(T)$ and $1 > c_{t_1} \geq c_t$.

In the case where $\mathcal{I} = \{j_0, j_0 + 1, \ldots, j_0 + k_t\}$ for the transformation t (the potential is moved upwards with respect to any layer), the optimal $c_{t_1} = c_{t_2} = \cdots = c_{t_{k_t - 1}} = x$, where x is the solution to:

$$0 = -\Delta\Phi_t^{j_0} \cdot x^{k_t} - \Delta\Phi_t^{j_0+1} \cdot x^{k_t - 1} - \cdots - \Delta\Phi_t^{j_0 + k_t} \cdot x^0.$$

This is optimal because then the potential decrease (as a fraction) is balanced over all $k_t - 1$ transformations.

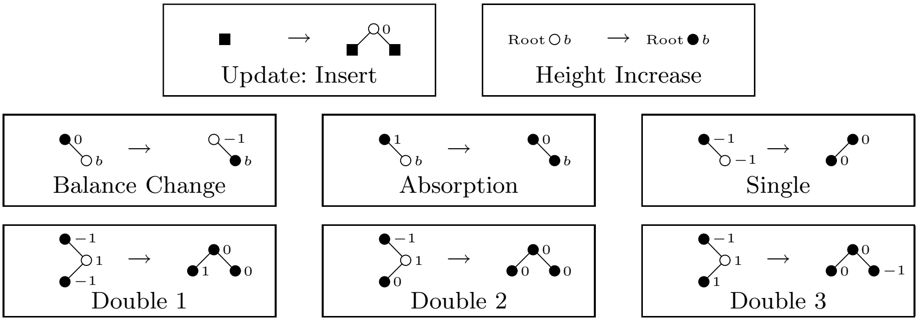


Fig. 2. Insert and rebalancing operations on AVL-trees. For all operations: $b \in \{-1, 0, 1\}$

Observe that when updates are split using the lemma above, all resulting transformations, except t_1, become post-processing operations.

4 Designing Local Rules for Non-local Structures

In the following, #upd denotes the number of updates. The number of rebalancing operations on layer i, with respect to some layer function, is denoted $\#\text{ops}_i$. Moreover, the search trees considered in the following are all leaf-oriented. This means that all keys stored in internal nodes are routers used only to guide the search, and all elements are stored in the leaves. This is to satisfy the requirement that updates are always made near layer 0.

4.1 Semi-Dynamic AVL-Trees: Insertions

The complexity of sequences of insertions into an AVL-tree [1] is treated in [12]. In [12], it is shown that the number of post-processing (rebalancing) operations is exponentially decreasing in the height. In this section, we prove matching bounds using our construction.

Again, as noted in the previous section, transformations on the tree must be described by local rules, i.e., no transformation (including updates) can make changes to more than a constant number of nodes. In particular, we cannot immediately update the balance of all nodes on the search path of an update.

To satisfy this requirement, we introduce a special node that indicates that the height of the tree below has been increased by one. Rebalancing then consists of propagating this node upwards, and it ceases when the *critical node* [12] or the root is reached. The critical node is the bottommost node on the search path with non-zero balance. The set of operations in terms of local rules is depicted in Figure 2. The special node described above is the white node.

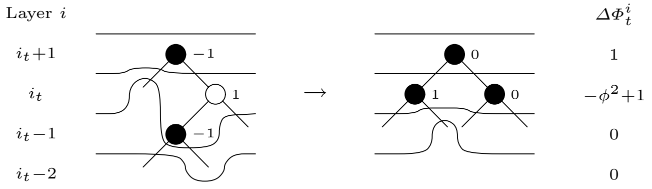


Fig. 3. Example of analysis of the *Double 1* rebalancing operation. The lines indicate the layer boundaries

We define the height of a node in a partially rebalanced AVL-tree as follows:

$$H(u) = \begin{cases} 0 & \text{, if } u \text{ is a leaf} \\ \max\{H(u.l), H(u.r)\} & \text{, if } u \text{ is white} \\ \max\{H(u.l), H(u.r)\} + 1 & \text{, otherwise} \end{cases}$$

where $u.l$ ($u.r$) denote the left (right) child of the node u. The *balance* of a node $b(u)$ is defined as $b(u) = H(u.l) - H(u.r)$. The number to the right of each node in Figure 2 is its balance. A balanced AVL-tree contains no white nodes and satisfies $b(u) \in \{-1, 0, 1\}$ for all nodes u.

The layer of a node is simply its height, $\mathcal{L}(u) = H(u)$. Observe that as a consequence of the definition of $\mathcal{L}(u)$, the layer of the topmost node of any operation (except *Height Increase*) is not changed by the operation.

Finally, we define a local potential function[2]:

$$\Phi(u) = \mathbf{1}_{b(u)=0} + \phi^2 \cdot \mathbf{1}_{u \text{ is white}},$$

where ϕ denotes $\frac{1+\sqrt{5}}{2}$, the golden ratio.

With the layer and potential functions we now have, by Theorem 1, that rebalancing is exponentially decreasing in AVL-trees with insertions. To determine the exact bound, we analyze each transformation on the tree. By analyzing each rebalancing operation one by one (see Figure 3), Table 3 (top) can be constructed.

By examining *Insert*, we find $c_u = \Delta\Phi(T) = \phi^2 + 1$ and $k_u = 2$. From the table we have $c = \frac{1}{\phi}$, $k_r = 2$, and $c_r = \phi$. However, this will not yield the best possible bound. For *Absorption* and *Insertion*, both having $k_t > 1$, we apply Lemma 1. The potential changes of the new transformations are depicted schematically in Table 3 (bottom). Observe that *Insert*$_2$ is now a rebalancing operation.

This way, k_r is reduced to 1 and k_u to 0, while c remains $\frac{1}{\phi}$ and c_u becomes $2\phi^2 - 1$. The value of $c_r = \phi$ is unchanged.

Finally, we find that the number of rebalancing operations at height i is: $\#\text{ops}_i \leq \frac{2\phi^2-1}{\phi\cdot\sqrt[1]{\frac{1}{\phi}}^{0}} \cdot \frac{\#\text{upd}}{\sqrt[1]{\phi}^{i}} = (\phi+1)\cdot\frac{\#\text{upd}}{\phi^i}$, matching the result of [12].

[2] $\mathbf{1}_p$ is the indicator function, i.e., $\mathbf{1}_p = 1$ if p is true and $\mathbf{1}_p = 0$ otherwise.

Table 3. Top: Potential changes by rebalancing operations on AVL-trees. Bottom: Potential changes by resulting transformations after splitting *Absorption* and *Insert*

Operation t	$\Delta\Phi_t^{i_t}$	$\Delta\Phi_t^{i_t+1}$	$\Delta\Phi_t^{i_t+2}$	c_t
Height Increase	$-\phi^2$	0	0	0
Balance Change	$-\phi^2$	ϕ^2-1	0	$\frac{1}{\phi}$
Absorption	$-\phi^2$	0	1	$\frac{1}{\phi^2}$
Single, Double 1–3	$-\phi^2+1$	1	0	$\frac{1}{\phi}$

Operation t	$\Delta\Phi_t^{i_t}$	$\Delta\Phi_t^{i_t+1}$	$\Delta\Phi_t^{i_t+2}$	c_t
Absorption$_1$	$-\phi^2$	ϕ	0	$\frac{1}{\phi}$
Absorption$_2$	0	$-\phi$	1	$\frac{1}{\phi}$
Absorption	$-\phi^2$	0	1	$\frac{1}{\phi^2}$

Transformation t	$\Delta\Phi_t^0$	$\Delta\Phi_t^1$	c_t
Insert$_1$	$2\phi^2-1$	0	
Insert$_2$	$-\phi^2+1$	1	$\frac{1}{\phi}$
Insert	ϕ^2	1	

4.2 Semi-dynamic AVL-Trees: Deletions

The amortized complexity of sequences of deletions with rebalancing in AVL-trees is treated in [14]. However, unlike for sequences of insertions, to our knowledge, rebalancing has not been shown to be exponentially decreasing. We show that in this section, and show that the constants obtained using Theorem 1 and the technique described in the previous section are tight.

Again, as in the previous section, we use a special node (this time indicating that the height of the subtree below has decreased by one) to capture the notion of a partially rebalanced tree. Again rebalancing is done by propagating this node upwards, and it ceases when the node is removed by some transformation. The set of transformations in terms of local rules is depicted in Figure 4. The special node described above is the white node.

We define the height $H(u)$ and the *corrected height* $CH(u)$ of a node u as follows:

$$H(u) = \begin{cases} 0 & \text{, if } u \text{ is a leaf} \\ \max\{CH(u.l), CH(u.r)\} + 1 & \text{, otherwise} \end{cases}$$

$$CH(u) = H(u) + \mathbf{1}_{u \text{ is white}}$$

Here $u.l$ ($u.r$) denotes the left (right) child of the node u. The *balance* of a node $b(u)$ is defined as $b(u) = H(u.l) - H(u.r)$. The number to the right of each node in Figure 4 is its balance. Again, a balanced AVL-tree contains no white nodes and satisfies $b(u) \in \{-1, 0, 1\}$ for all nodes u.

The layer function $\mathcal{L}(u)$ is defined to be the corrected height, i.e, $\mathcal{L}(u) = CH(u)$. Again, as a consequence of this definition, the layer of the top-most node is not changed by any operation, except for *Height Decrease*.

Finally, the local potential function:

$$\Phi(u) = x \cdot \mathbf{1}_{|b(u)|=1} + y \cdot \mathbf{1}_{u \text{ is white}},$$

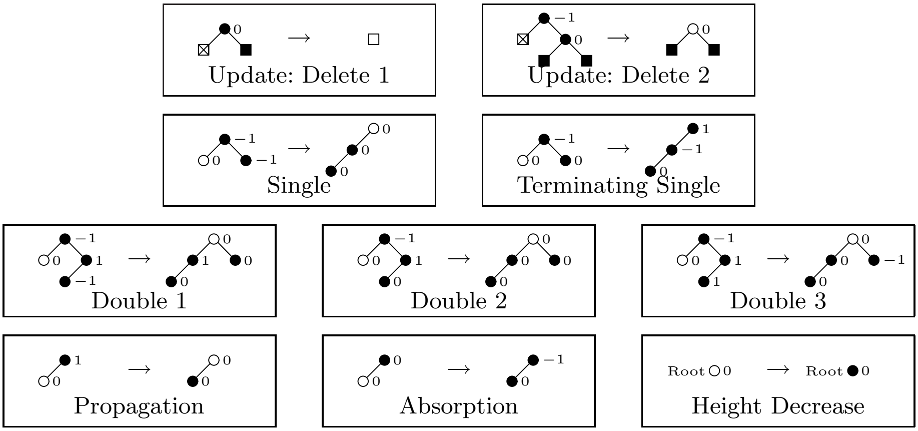


Fig. 4. Operations on AVL-trees to handle sequences of deletions

for positive constants x, y such that $y > x$. One can easily verify that this suffices to show amortized constant rebalancing, wherefore Theorem 1 applies. To determine the constants, we analyze each operation one by one, as in the previous section, and obtain Table 4 (top).

From the table, we find that $k_r = k_u = 2$. However, this will again not yield the best c_2. For *Single*, *Double 1–3*, and *Delete 2* having $k_t > 1$, we apply Lemma 1 to reduce k_t. See Table 4 (bottom) for the new potential changes.

We now find from the tables that $k_r = 1$ and $c = \max\left\{\frac{x}{y}, \frac{y-x}{y}\right\} = \frac{1}{2}$ when $y = 2x$ and $c_r = y$. By examination of the *Delete* operations, we now find that $k_u = 1$ and $c_u = y$. Hence,

$$\#\text{ops}_i \le \frac{y}{y \cdot \sqrt[1]{\frac{1}{2}}^{-1}} \cdot \frac{\#\text{upd}}{\sqrt[1]{2}^{i}} = 2 \cdot \frac{\#\text{upd}}{2^i}.$$

Moreover, this bound is tight. Consider a perfectly balanced binary tree of height h. That is, all internal nodes have balance 0 and the tree has zero potential. Delete every second element. In this way, every remaining internal node has had its balance changed to 1 (or -1) by an *Absorption* operation once, and has been the 'white' node of a *Propagation* (or *Height Decrease*) operation once. Only the latter counts as an operation on the layer of the node (which is one higher than where it ends), i.e., the number of rebalancing operations on any layer is exactly the number of internal nodes on this layer, which is $\frac{2^h}{2^i} = 2 \cdot \frac{\#\text{upd}}{2^i}$ on layer i.

4.3 (a, b)-Trees

In this section, we consider (a, b)-trees [2,6], i.e., multi-way search trees satisfying that the number of children of each node (except maybe the root) is between a

Table 4. Top: Potential changes by rebalancing operations on AVL-trees. Bottom: Potential changes by resulting transformations after splitting *Single*, *Double 1–3*, and *Delete 2*. The layer i_t refers to the layer of *Single* (and *Double 1–3*)

Operation t	$\Delta\Phi_t(T)$	$\Delta\Phi_t^{i_t-1}$	$\Delta\Phi_t^{i_t}$	$\Delta\Phi_t^{i_t+1}$	c_t
Term. Single	$x-y$	0	$-y$	x	$\frac{x}{y}$
Absorption	$x-y$	0	$-y$	x	$\frac{x}{y}$
Propagation	$-x$	0	$-y$	$y-x$	$\frac{y-x}{y}$
Height Decrease	$-y$	0	$-y$	0	0
Single, Double 1–3	$-2x$	$-y$	$-x$	$y-x$	$\frac{y-x}{y+x}$

Operation t	$\Delta\Phi_t^{i_t-1}$	$\Delta\Phi_t^{i_t}$	$\Delta\Phi_t^{i_t+1}$	c_t
Single$_1$, Double 1_1–3_1	$-y$	$y-x$	0	$\frac{y-x}{y}$
Single$_2$, Double 1_2–3_2	0	$-y$	$y-x$	$\frac{y-x}{y}$
Single, Double 1–3	$-y$	$-x$	$y-x$	$\frac{y-x}{y+x}$

Operation t	$\Delta\Phi_t^1$	$\Delta\Phi_t^2$	c_t
Delete 2_1	y	0	$\frac{y-x}{y}$
Delete 2_2	$-y$	$y-x$	
Delete 2	0	$y-x$	

and b for $b \geq 2a$. In [6,11], it is shown that rebalancing is exponentially decreasing in (a,b)-trees. Here, we demonstrate how to obtain comparable constants to [6] using Theorem 1 and refining the potential function.

To represent intermediate states between operations, we assume that nodes are capable of storing $b+1$ pointers. We define the set of local rules depicted in Figure 5. The layer of a node is its height, $\mathcal{L}(u) = h(u)$, with the leaves at height 0.

In the following, we assume that $b > 2a$. We only consider potential functions of the form $\phi(u) = \psi(\rho(u))$, i.e., the potential of a node u depends only on the number of children $\rho(u)$ of this node. In this way, k_r remains one, and so $c_2 = c^{-1}$. Moreover, we assume that ψ is a piecewise linear function; see Figure 6.

Assume that we want to achieve $c^{-1} = C$ for some $C > 1$. The function ψ should then meet two objectives; $-\Delta\Phi_t^i \geq C \cdot \Delta\Phi_t^{i+1}$ and $\Delta\Phi_t(T) < 0$, for all rebalancing operations t—the latter to ensure that rebalancing is amortized constant.

Based on the five rebalancing operations, ψ should satisfy the following inequalities, where $\psi(+1)$ ($\psi(-1)$) is the maximum increase of ψ caused by adding (removing) a pointer:

$$\begin{array}{ll}
\text{Root 1:} & C \cdot \psi(a) \leq \psi(b+1) - \left(\psi\left(\left\lfloor \frac{b+1}{2} \right\rfloor\right) + \psi\left(\left\lceil \frac{b+1}{2} \right\rceil\right)\right) \\
\text{Root 2:} & \qquad\quad 0 < \psi(a-1) \\
\text{Split} & : C \cdot \psi(+1) \leq \psi(b+1) - \left(\psi\left(\left\lfloor \frac{b+1}{2} \right\rfloor\right) + \psi\left(\left\lceil \frac{b+1}{2} \right\rceil\right)\right) \\
\text{Fuse} & : C \cdot \psi(-1) \leq \psi(a-1) + \psi(a) - \psi(2a-1) \\
\text{Share} & : \qquad\quad 0 < \psi(a-1) + \psi(d) - \left(\psi\left(\left\lfloor \frac{d+a-1}{2} \right\rfloor\right) + \psi\left(\left\lceil \frac{d+a-1}{2} \right\rceil\right)\right)
\end{array}$$

Observe that the last inequality is always satisfied if the slope of the leftmost segment is strictly less than -1, and the second is always satisfied by the same argument and the fact that $\psi(\rho) \geq 0$ for all ρ, since $\psi(a-1) = \psi(a) + 2$.

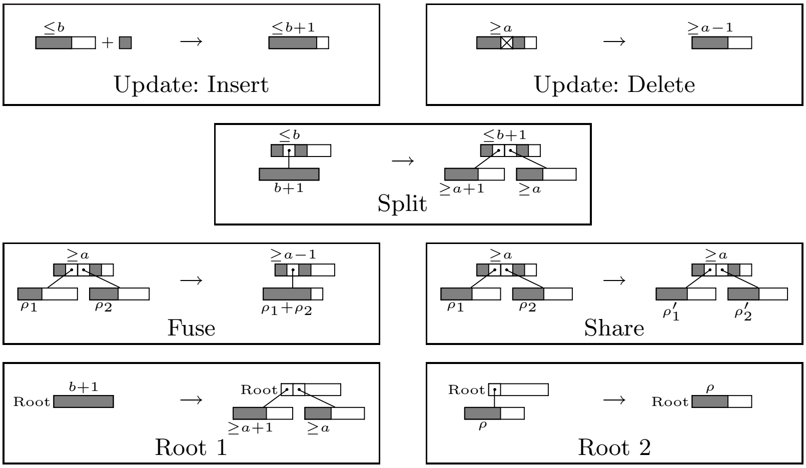


Fig. 5. Operations on (a, b)-trees. Conditions: Fuse: $\rho_1 < a$ or $\rho_2 < a$ and $\rho_1 + \rho_2 < 2a$. Share: $\rho_1 < a$ or $\rho_2 < a$ and $\rho_1 + \rho_2 \geq 2a$

Moreover, the best C for this set of inequalities is achieved when $\psi(a) = \psi(+1)$, reducing the first and third inequality to just one.

Depending on a and b, we consider two cases (recall that $b > 2a$):

Case 1 $2a - 1 > \lceil \frac{b+1}{2} \rceil$: Assume that in Figure 6, $\rho_1 = \lceil \frac{b+1}{2} \rceil$ and $\rho_2 = 2a - 1$. Let $\psi(2a-1) = \psi(\lceil \frac{b+1}{2} \rceil) = k$. Then $\psi(a) = k + (\lceil \frac{b+1}{2} \rceil - a)$ and $\psi(b+1) = k + (b+1-(2a-1)) \cdot \psi(a)$. We determine C from the two remaining inequalities:

$$\begin{aligned} C \cdot \psi(a) &\leq k + (b+1-(2a-1)) \cdot \psi(a) - (2k+1) \\ C \cdot 2 &\leq 2(k + \lceil \tfrac{b+1}{2} \rceil - a) + 2 - k \end{aligned}$$

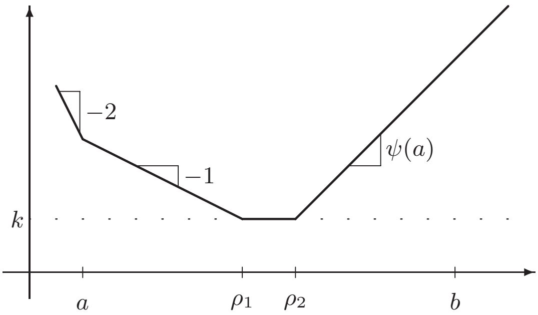


Fig. 6. The function ψ. The slopes of the segments from left to right are: -2, -1, 0, and $\psi(a)$, respectively. For the root node, the horizontal segment is extended to accommodate the additional legal degrees

The second of these inequalities reduces to $C \leq k+2(\lceil \frac{b+1}{2} \rceil - a+1)$. If $2k+1 \leq k+\psi(a) = k+k+(\lceil \frac{b+1}{2} \rceil - a)$, which is satisfied for $b \geq 2a+1$, then, by the first inequality, $C \leq b-2a+1$, and we can simply choose $k \geq 2(C-(\lceil \frac{b+1}{2} \rceil - a+1))$ and the same C satisfies the second inequality.

Case 2 $2a-1 \leq \lceil \frac{b+1}{2} \rceil$: Let $\rho_1 = \rho_2 = \lceil \frac{b+1}{2} \rceil$. Then $\psi(\lceil \frac{b+1}{2} \rceil) = k$ and $\psi(2a-1) = k+\lceil \frac{b+1}{2} \rceil - (2a-1)$. Furthermore, $\psi(a) = k+\lceil \frac{b+1}{2} \rceil - (2a-1)+a-1$. Since $\psi(+1) = \psi(a)$, we get $\psi(b+1) = k + \lfloor \frac{b+1}{2} \rfloor \cdot \psi(a)$, and the two inequalities become:

$$\begin{aligned} C \cdot \psi(a) &\leq k + \lfloor \tfrac{b+1}{2} \rfloor \cdot \psi(a) - (2k+1) \\ C \cdot 2 &\leq 2(k + \lceil \tfrac{b+1}{2} \rceil - (2a-1) + a) - (k + \lceil \tfrac{b+1}{2} \rceil - (2a-1)) \\ &= k + \lceil \tfrac{b+1}{2} \rceil + 1 \end{aligned}$$

Using that $\psi(a) \geq k+1$, we obtain $C \leq \lfloor \frac{b-1}{2} \rfloor$ from the first inequality. Choosing $k \geq 2C - (\lceil \frac{b+1}{2} \rceil + 1)$, this C satisfies the second inequality as well.

We find that $c_1 = \psi(a)$ since $c_u = \psi(a)$, $k_u = 0$, $k_r = 1$, and $c_r = 1$. Using that $C = \frac{3}{2}$ in the case $b = 2a$, we summarize the result above:

Assume that $b \geq 2a$. Then the number of rebalancing operations performed at height i, $\#\text{ops}_i$, in an (a,b)-tree satisfies: $\#\text{ops}_i \leq \psi(a) \cdot \frac{\#\text{upd}}{C^i}$ for $C = \max\left\{\frac{3}{2}, \min\left\{b-2a+1, \lfloor \frac{b-1}{2} \rfloor\right\}\right\}$ and a constant $\psi(a)$, matching the result of [6].

5 Relaxed Balance

A large family of search trees satisfying the criteria for applying the results in this paper is search trees with relaxed balance.

Relaxed balance is a paradigm for rebalancing search trees in small independent steps, such that while still being efficient, rebalancing can at any time be suspended. In particular, relaxed balance has desirable properties in a parallel setting. For brief a survey of the motivation and results on relaxed balance, see [8]. A discussion of the synchronization issues involved in using a relaxed structure in a parallel environment can be found in [3].

In this section we show how to apply our results to an (a,b)-tree with relaxed balance [7,10,13]. For our results to apply, we need to *interpret* the relaxed structure and the transformation on it in a special way. While we study a specific example in this section, the technique is generally applicable to any relaxed balanced search tree satisfying the remaining criteria. The structure considered is the second proposal from [7], which is an extension of the proposal from [10]. For details concerning completeness and correctness of this structure, we refer the reader to [7].

The major difference between a relaxed (a,b)-tree and the (a,b)-tree discussed in the previous section (from here on referred to as a *standard* (a,b)-tree), is that rebalancing is completely uncoupled from updates. In particular, no rebalancing can be assumed to be applied immediately following an update. Analogously, when a rebalancing operation is applied, we cannot assume that

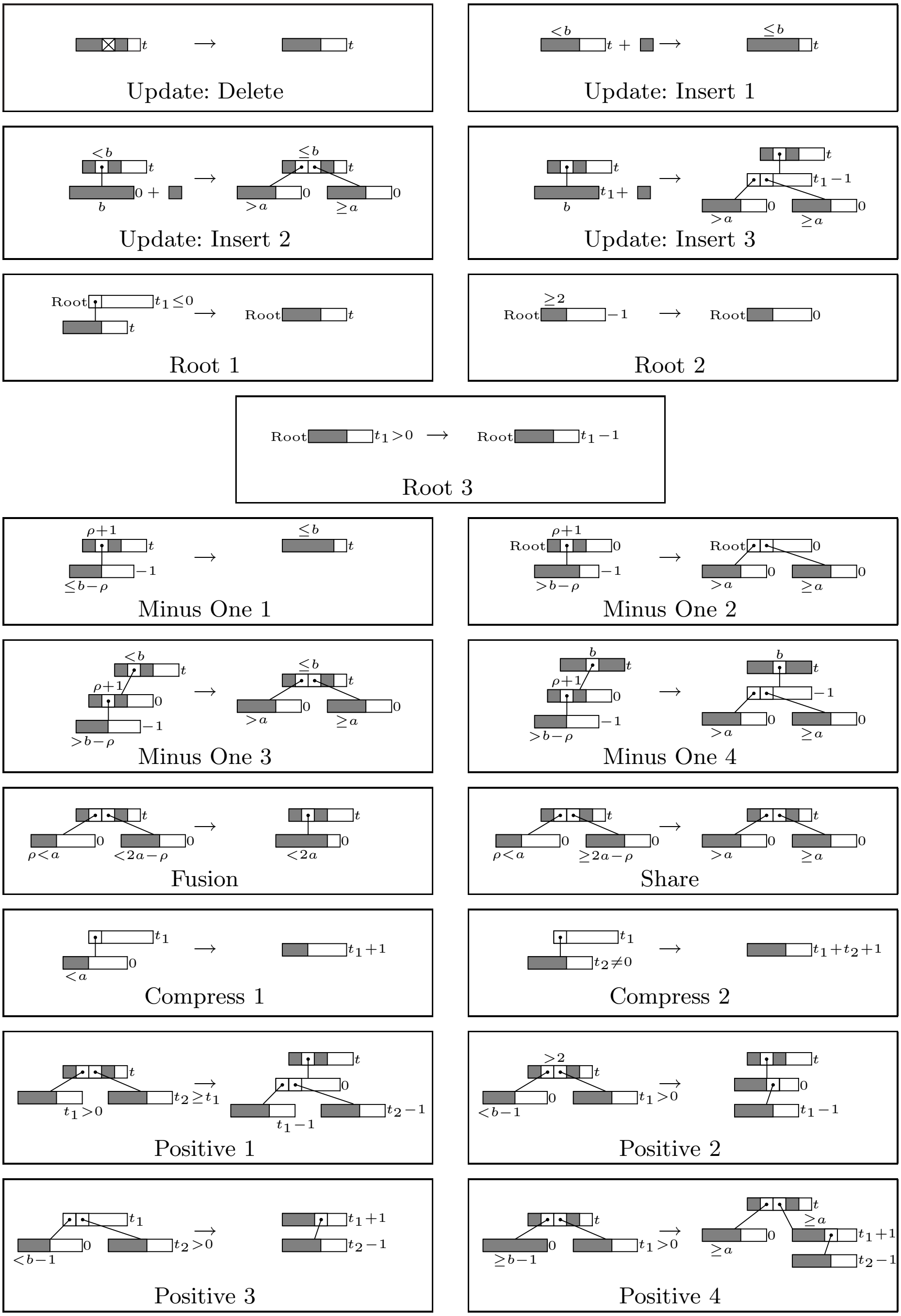


Fig. 7. Operations on relaxed (a, b)-trees. The tag of a node is written to its right. For Insert 3, $\rho = b$ or $t_1 > 0$

rebalancing afterwards immediately run to completion. Hence, we must be able to handle any number of insertions (deletions) in any node before any rebalancing is performed. Yet, once rebalancing has been completed, the tree must be a standard (a, b)-tree.

Each node u in a relaxed (a, b)-tree has an associated integer tag-value $t(u)$. We define the *relaxed height* of a node u as follows:

$$rh(u) = \begin{cases} 0 & \text{, if } u \text{ is a leaf} \\ rh(c(u)) + 1 + t(c(u)) & \text{, if } u \text{ is an internal node with a child } c(u) \end{cases}$$

An invariant for the structure is that the tags are maintained in such a way that for all children of a node, the sum of its tag and its relaxed height is the same. Therefore it suffices to just refer to any child in the definition of relaxed height above.

The tags are used to compensate for postponed rebalancing. If a node contains too many pointers, it is split, and, if possible, the extra pointer is inserted into the parent. If the parent is already completely full, we create an extra node between the split node and its parent with a tag of -1, making it neutral in the definition of relaxed height. In this way, we avoid the ripple effect if the entire path needs to be split. Similarly, if an underfull node is the only child of its parent, we cannot merge it with a sibling. In this case, to speed up searches in the subtree, we remove the parent, while incrementing the tag of the child. This unit of positive tag counts for the parent in the computation of relaxed height.

We allow any node u to have $1 \leq \rho(u) \leq b$, where $\rho(u)$ denotes the number of children. The operations on a relaxed (a, b)-tree are depicted in Figure 7. All operations are identical to those in the second proposal from [7], with the exception of *Positive 1*, where the two bottom nodes in [7] had their tags decremented by $\min\{t_1, t_2\}$ rather than one, and the operations involving the root. As will be discussed shortly, this is to avoid operations of non-constant size.

Since collapsing paths of unary nodes creating positive tags might result in a non-constant number of layers between a node and its parent, we cannot apply Theorem 1 immediately. Therefore we introduce an *interpretation* of the positive tags that allows us to use Theorem 1. By this interpretation, a node u with tag $t(u) > 1$ is a chain with $t(u)$ unary nodes and the node u at the bottom, without its positive tag. Indeed, this interpretation is what the tree would have looked like had we not collapsed the path of unary nodes and introduced positive tags. Decrementing a positive tag is then the equivalent of removing a unary node, while incrementing the positive tag is the equivalent of adding a unary node. Note that this is only an interpretation; the physical representation of the tree remains the same, using the positive tags. See Figure 8 for an illustration of this interpretation. To distinguish the special unary nodes introduced to interprete positive tags, we use round nodes to represent these.

The layer $\mathcal{L}(u)$ of a node u (which is referred to in Figure 8) is based on the interpretation and is defined as:

$$\mathcal{L}(u) = \begin{cases} \mathcal{L}(c(u)) & \text{, if } u \text{ is a round unary node} \\ rh(u) & \text{, otherwise} \end{cases}$$

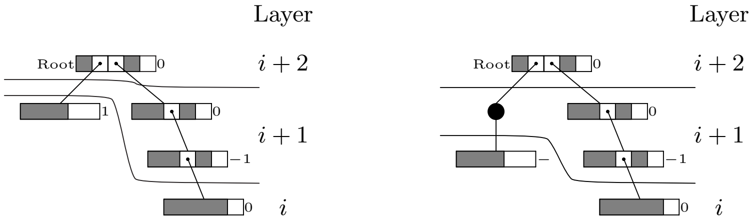


Fig. 8. Illustration of the interpretation of positive tags. The lines indicate layer boundaries. Left: The actual representation of a relaxed (a, b)-tree. Right: The interpretation of the same tree

What now remains to make Theorem 1 applicable is a local potential function. In [7], relaxed (a, b)-trees are shown to have amortized constant behavior using an almost local potential function. Since the potential of nodes with positive tags depends linearly on the value of the tag, the maximum potential is not bounded by a constant. However, this function is easily adapted to the interpretation of positive tags (recall that all non-zero tags are negative):

$$\Phi(u) = \begin{cases} 1 + 2(\rho(u) - 1) & \text{, if } t(u) < 0 \\ 4 & \text{, if } \rho(u) = 1 \text{ and } u \text{ is not round} \\ 3 & \text{, if } 1 < \rho(u) < a_u \text{ or if } u \text{ is round} \\ 1 & \text{, if } \rho(u) = a_u \\ 0 & \text{, if } a < \rho(u) < b \\ 2 & \text{, if } \rho(u) = b \end{cases},$$

where a_u is the minimum legal degree for the node in a standard (a, b)-tree, which is 2 for the root, and a otherwise. One can easily verify that Φ is a local potential satisfying that any update increases the potential by at most a constant and any rebalancing operation decreases the potential by at least a constant, so it follows from Theorem 1 that rebalancing in relaxed (a, b)-trees is exponentially decreasing.

To determine the constants involved, we analyze each operation one by one and obtain Table 5. After applying Lemma 1 to the larger updates, we find that $k_r = 1$, $k_u = 0$, $c_u = 4$, $c_r = 1$ and that the best value of c is $\frac{2}{3}$ for $a > 2$ and $\frac{3}{4}$ for $a = 2$. In all cases, the number of rebalancing operations on layer i is at most:

$$\#\text{ops}_i \leq \frac{4}{1 \cdot \sqrt[1]{\frac{3}{4}}^{0}} \sqrt[1]{\frac{3}{4}}^{i} \cdot \#\text{upd} = \frac{4 \cdot \#\text{upd}}{\left(\frac{4}{3}\right)^i}$$

6 Concluding Remarks

A reasonable question to consider is whether or not the theorem has found its right form. In particular, are all the requirements necessary? Clearly, if opera-

Table 5. Potential changes

Operation t	$\Delta\Phi_t(T)$	$\Delta\Phi_t^{i_t}$	$\Delta\Phi_t^{i_t+1}$	$\Delta\Phi_t^{i_t+2}$	c_t
Insert 1	2	2	0	0	
Insert 2	1	−1	2	0	
Insert 3 ($t_1 > 0$)	−1	−1	0	0	
Insert 3 ($t_1 = 0$)	2	−1	3	0	
Delete	3	3	0	0	
Root 1	−1	−1	0	0	0
Root 2	−2	−2	0	0	0
Root 3	−3	−3	0	0	0
Minus One 1	−1	−1	0	0	0
Minus One 2 & 3	−3	−5	2	0	$\frac{3}{5}$
Minus One 4	−2	−5	3	0	$\frac{2}{5}$
Fusion ($a = 2$)	−1	−4	3	0	$\frac{3}{4}$
Fusion ($a > 2$)	−1	−3	2	0	$\frac{2}{3}$
Share	−1	−1	0	0	0
Compress 1 & 2	−1	−1	0	0	0
Positive 1	−3	−3	0	0	0
Positive 2 & 3	−1	−3	2	0	$\frac{2}{3}$
Positive 4	−1	−1	0	0	0

tions are not local, then every operation could involve the root. Furthermore, requiring that there is a local potential function which assigns at most a constant amount of potential to each node gives a similar type of control over the progress of operations. Without this requirement, one can construct the scenario where every $(\log n)$'th operation progresses all the way to the root, while all other operations finish immediately at the leaves. Then, assuming that the height of the tree (as well as the number of layers) is $\Theta(\log n)$, the operation is clearly amortized constant, while the number of operations cannot decrease exponentially, since the root is accessed too often.

References

1. G. M. Adel'son-Vel'skiĭ and E. M. Landis. An Algorithm for the Organisation of Information. *Doklady Akadamii Nauk SSSR*, 146:263–266, 1962. In Russian. English translation in *Soviet Math. Doklady*, 3:1259-1263, 1962. 300
2. R. Bayer and E. McCreight. Organization and Maintenance of Large Ordered Indexes. *Acta Informatica*, 1:173–189, 1972. 303

3. Joan Boyar, Rolf Fagerberg, and Kim S. Larsen. Amortization Results for Chromatic Search Trees, with an Application to Priority Queues. *Journal of Computer and System Sciences*, 55(3):504–521, 1997. 306
4. Paul F. Dietz and Rajeev Raman. Persistence, Amortization and Randomization. In *Proceedings of the Second Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 78–88, 1991. 297
5. James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. Making Data Structures Persistent. *Journal of Computer and System Sciences*, 38:86–124, 1989. 293, 297
6. Scott Huddleston and Kurt Mehlhorn. A New Data Structure for Representing Sorted Lists. *Acta Informatica*, 17:157–184, 1982. 293, 303, 304, 306
7. Lars Jacobsen and Kim S. Larsen. Variants of (a, b)-Trees with Relaxed Balance. *International Journal of Foundation of Computer Science*. To appear. 306, 308, 309
8. Lars Jacobsen and Kim S. Larsen. Complexity of Layered Binary Search Trees with Relaxed Balance. In *Seventh Italian Conference on Theoretical Computer Science*, 2001. This volume. 306
9. Lars Jacobsen, Kim S. Larsen, and Morten N. Nielsen. On the Existence and Construction of Non-Extreme (a, b)-Trees. Tech. rep. 11, Department of Mathematics and Computer Science, University of Southern Denmark, Odense, 2001. 297
10. Kim S. Larsen and Rolf Fagerberg. Efficient Rebalancing of B-Trees with Relaxed Balance. *International Journal of Foundations of Computer Science*, 7(2):169–186, 1996. 306
11. Kurt Mehlhorn. *Sorting and Searching*, volume 1 of *Data Structures and Algorithms*. Springer-Verlag, 1984. 293, 304
12. Kurt Mehlhorn and Athanasios Tsakalidis. An Amortized Analysis of Insertions into AVL-Trees. *SIAM Journal on Computing*, 15(1), 1986. 293, 300, 301
13. Otto Nurmi, Eljas Soisalon-Soininen, and Derick Wood. Concurrency Control in Database Structures with Relaxed Balance. In *Proceedings of the 6th ACM Symposium on Principles of Database Systems*, pages 170–176, 1987. 306
14. Athanasios K. Tsakalidis. Rebalancing Operations for Deletions in AVL-Trees. *R.A.I.R.O. Informatique Théorique*, 19(4):323–329, 1985. 293, 302

Directed Hypergraphs: Problems, Algorithmic Results, and a Novel Decremental Approach*

Giorgio Ausiello[1], Paolo G. Franciosa[2], and Daniele Frigioni[1]

[1] Dipartimento di Informatica e Sistemistica
Università di Roma "La Sapienza", Italy
{ausiello,frigioni}@dis.uniroma1.it

[2] Dipartimento di Statistica, Probabilità e Statistiche Applicate
Università di Roma "La Sapienza", Italy
paolo.franciosa@uniroma1.it

Abstract. The purpose of this paper is twofold. First, we review several basic combinatorial problems that have been stated in terms of directed hypergraphs and have been studied in the literature in the framework of different application domains. Among them, transitive closure, transitive reduction, flow and cut problems, and minimum weight traversal problems. For such problems we illustrate some of the most important algorithmic results in the context of both static and dynamic applications. Second, we address a specific dynamic problem which finds several interesting applications, especially in the framework of knowledge representation: the maintenance of minimum weight hyperpaths under hyperarc weight increases and hyperarc deletions. For such problem we provide a new efficient algorithm applicable for a wide class of hyperpath weight measures.

Keywords: Directed hypergraph, minimum weight hyperpath, dynamic algorithm, AND/OR graph.

1 Hypergraph Structures in Computer Science

Hypergraph is a common name for various combinatorial structures that generalize graphs. Beside the most known undirected hypergraphs (or simply *hypergraphs* [13,14]), a relevant role is played by *directed hypergraphs*, a generalization of directed graphs, which find applications in several areas of computer science and mathematics for representing implicative structures. In a directed hypergraph we are given a set of nodes N and a set of pairs $\langle T, h\rangle$ (*hyperarcs*) where T is a subset of N and h is a single node in N. The most obvious interpretation of a hyperarc $\langle T, h\rangle$ is that the information associated to h functionally depends on the information associated to nodes in T.

Directed hypergraphs and other strictly related combinatorial structures are widely used in computer science. Notably, they are used in artificial intelligence

* Work partially supported by the IST Programme of the EU under contract number IST-1999-14186 (ALCOM-FT).

A. Restivo, S. Ronchi Della Rocca, L. Roversi (Eds.): ICTCS 2001, LNCS 2202, pp. 312–328, 2001.

for representing problem solving relationships (And-Or graphs [32] and recursive label node hypergraphs [15]), in database theory for representing functional dependencies among attributes (FD-graphs [5] and connections in acyclic hypergraphs [29,36]), in deductive databases [23], in fuzzy logic, for determining the reliability of facts [8], in propositional logic, for satisfiability check (namely in the case of Horn formulæ [9,21,33]), in formal languages (weighted context free grammars [26]), in the theory of concurrency (Petri net paths [37]), in model checking (dependency graphs [28] and boolean graphs [3]), in diagnostics [2]. More applications can be found in [1,4,34].

In all these applications we need to provide a formal representation for a set of many-to-one implications and to study properties of the resulting structures such as assessing reachability, evaluating flows and cuts, determining less expensive implication chains under various cost criteria ('shortest' hyperpaths). In most cases determining hypergraph properties that consist in generalizations of corresponding properties on ordinary graphs is a much more complex (usually NP-hard) task. Therefore, the study of particular polynomial restrictions and of polynomial time approximation algorithms is quite important and constitutes a largely still unexplored field.

The purpose of this paper is twofold. First we want to review several basic combinatorial problems that have been stated in terms of directed hypergraphs and have been studied in the literature in the framework of different application domains. Among them, transitive closure and transitive reduction [6,10], flow and cut problems [16,21], minimum weight traversal problems [12,21,35]. For such problems we illustrate some of the most important algorithmic results in the context of both static and dynamic (mostly incremental) applications.

In second place, we want to address a specific dynamic problem which finds several interesting applications especially in the framework of knowledge representation: the maintenance of minimum weight hyperpaths under hyperarc deletions and weight increases. For such problem we provide a new efficient algorithm applicable for a large variety of hyperpath weight measures.

The paper is organized as follows. In section 2 the basic definitions concerning hypergraphs and hyperpaths are given and various hyperpath weight measures are presented. In Section 3 the problems of transitive closure and transitive reduction are discussed and the main algorithmic results are reviewed. Section 4 is devoted to flows and cuts in hypergraphs. Section 5 is devoted to hypergraph traversal problems with respect to a variety of hyperpath weight measures. Finally, in Section 6 we give a summary of algorithmic results in the dynamic framework and present a new efficient algorithm for maintaining minimum weight hyperpaths under hyperarc eliminations and weight increase operations. In the last section we draw some conclusions and suggest future research directions.

2 Hypergraphs and Hyperpaths

The following definitions concerning directed hypergraphs are from [6] and [12], and are consistent with the more general definitions given in [22].

A *directed hypergraph* $\mathcal{H}$ (see Fig. 1 for an example) is a pair $\langle N, A \rangle$, where N is a non empty set of *nodes* and A is a set of hyperarcs; a *hyperarc* e is an ordered pair $\langle T, h \rangle$, with $T \subseteq N$, $T \neq \emptyset$, and $h \in N \setminus T$; T and h are called the *tail* and the *head* of e, and are denoted by $\mathrm{tail}(e)$ and $\mathrm{head}(e)$, respectively. A set of nodes is called a *source set* if it is the tail of some hyperarc; the *source area* of $\mathcal{H}$ is the sum of the cardinalities of its source sets. The *forward star* of $v \in N$ is the set $\mathrm{fstar}(v) = \{e \in A : v \in \mathrm{tail}(e)\}$, while the *backward star* of v is the set $\mathrm{bstar}(v) = \{e \in A : v = \mathrm{head}(e)\}$.

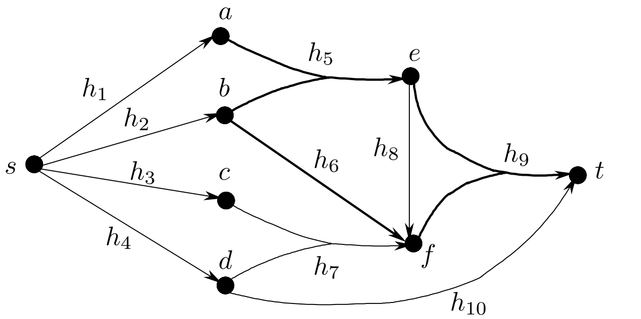


Fig. 1. A directed hypergraph

The *size* of a hypergraph $\mathcal{H}$ is defined as $\mathrm{size}(\mathcal{H}) = \sum_{e \in A} |\mathrm{tail}(e)|$. The following different notion of size has been also used in the literature: a "compact" hypergraph $\mathcal{H}^c$ is derived from $\mathcal{H}$ where a set of p hyperarcs having the same tail T is represented by a single hyperarc from T to a dummy node c, plus p hyperarcs from c to the original heads; the size of this hypergraph is denoted by $\mathrm{minsize}(\mathcal{H}) = \mathrm{size}(\mathcal{H}^c)$. This parameter is called *size* of a hypergraph $\mathcal{H}$ in [12], and is denoted by $|\mathcal{H}|$.

Given a hypergraph $\mathcal{H}$, a *subhypergraph* of $\mathcal{H}$ is a hypergraph $\mathcal{H}' = \langle N', A' \rangle$ with $N' \subseteq N$ and $A' \subseteq A$. A subhypergraph is *proper* if at least one of the inclusions is strict. A *hyperpath* in $\mathcal{H}$ from a set of nodes $S \subset N$, with $S \neq \emptyset$, called *source*, to a target node $t \in N$ is a subhypergraph $\Pi_{S,t} = \left\langle N_{\Pi_{S,t}}, A_{\Pi_{S,t}} \right\rangle$ of $\mathcal{H}$ having the following property: if $t \in S$, then $A_{\Pi_{S,t}} = \emptyset$, otherwise its $k \geq 1$ hyperarcs can be ordered in a sequence $\langle e_1, \ldots, e_k \rangle$ such that:

1. $\forall e_i \in A_{\Pi_{S,t}}, \mathrm{tail}(e_i) \subseteq S \cup \{\mathrm{head}(e_1), \ldots, \mathrm{head}(e_{i-1})\}$
2. $t = \mathrm{head}(e_k)$
3. No proper subhypergraph of $\Pi_{S,t}$ is a hyperpath from S to t in $\mathcal{H}$.

The above definition of hyperpath is consistent with the notion of *folded hyperpath* given in [12], and generalizes the notion of simple path in a directed graph. A node t is said to be *reachable* in $\mathcal{H}$ from source S if there exists a hyperpath $\Pi_{S,t}$ in $\mathcal{H}$. The *port* of a node v on $\Pi_{S,t}$, denoted by $\mathrm{port}(v)$, is the hyperarc in $A_{\Pi_{S,t}}$ having v as head; it is unique by condition 3 above. In Fig. 1, thick hyperarcs represent the hyperpath from source $S = \{a, b\}$ to node t consisting of hyperarcs $\langle h_5, h_6, h_9\rangle$.

The most intuitive and classical measure of the weight of a hyperpath is its *cost*, defined as the sum of the weights of its hyperarcs (see, for example, [11]). A different approach leads to defining the weight of a hyperpath in an inductive manner. First attempts in this direction can be found in [31] and in [11]. This approach has been formalized in [12] in the following terms.

Definition 1. *A* functional hypergraph $\mathcal{H}_F = \langle N, A; F\rangle$ *is a directed hypergraph* $\mathcal{H} = \langle N, A\rangle$ *in which each hyperarc* $e = \langle X, y\rangle \in A$ *is associated to a triple* $F_e = (w_e, \psi_e, f_e)$*, where:*

- w_e *is a real value;*
- ψ_e *is a function from* $|X|$*-tuples of reals to reals;*
- f_e *is a function from pairs of reals to reals.*

Let $\Pi_{S,t}$ be a hyperpath from S to t, and let $\langle Z, t\rangle$ be the last hyperarc in $\Pi_{S,t}$ (i.e., the *port* of t), where $Z = \{z_1, z_2, \ldots, z_k\}$: then[1] $\Pi_{S,t} = \Pi_{S,z_1} \cup \Pi_{S,z_2} \cup \ldots \cup \Pi_{S,z_k} \cup \{\langle Z, t\rangle\}$, where Π_{S,z_i} is the subhyperpath of $\Pi_{S,t}$ going from S to z_i, $1 \le i \le k$. The weight of $\Pi_{S,t}$ depends on $w_{\langle Z,t\rangle}$, that gives the *weight* of hyperarc $\langle Z, t\rangle$, and on $\psi_{\langle Z,t\rangle}$, that takes into account the weights of all hyperpaths Π_{S,z_i}. Function $f_{\langle Z,t\rangle}$ combines these two weights. This is formalized in the following definition.

Definition 2. *Given a functional directed hypergraph* $\mathcal{H}_F = \langle N, A; F\rangle$*, a* weight measure μ *associates a real weight to a hyperpath* $\Pi_{S,t}$ *as follows:*

- *if* $\Pi_{S,t}$ *has no hyperarcs (i.e.,* $t \in S$*), then* $\mu(\Pi_{S,t}) = \mu_0$*, where* μ_0 *is a proper constant;*
- *if* $\Pi_{S,t} = \Pi_{S,z_1} \cup \Pi_{S,z_2} \cup \ldots \cup \Pi_{S,z_k} \cup \{\langle Z, t\rangle\}$*, then* $\mu(\Pi_{S,t}) = f_{\langle Z,t\rangle}(w_{\langle Z,t\rangle}, \psi_{\langle Z,t\rangle}(\mu(\Pi_{S,z_1}), \mu(\Pi_{S,z_2}), \ldots, \mu(\Pi_{S,z_k})))$.

Several weight measures have been introduced in the literature to define the weight of a hyperpath in a functional directed hypergraph, by considering, given a hyperarc e such that $\mathrm{tail}(e) = \{x_1, x_2, \ldots, x_k\}$, different choices for functions ψ_e and f_e:

rank: $f_e(x, y) = x + y$, $\psi_e(x_1, x_2, \ldots, x_k) = \max_{1\le i\le k}\{x_i\}$, and $\mu_0 = 0$;
gap: $f_e(x, y) = x + y$, $\psi_e(x_1, x_2, \ldots, x_k) = \min_{1\le i\le k}\{x_i\}$, and $\mu_0 = 0$;
threshold: $f_e(x, y) = \max\{x, y\}$, $\psi_e(x_1, x_2, \ldots, x_k) = \max_{1\le i\le k}\{x_i\}$, and $\mu_0 = 0$;

[1] By a little abuse of notation, given two hypergraphs $\mathcal{H}_1 = \langle N_1, A_1\rangle$ and $\mathcal{H}_2 = \langle N_2, A_2\rangle$ we denote the hypergraph $\langle N_1 \cup N_2, A_1 \cup A_2\rangle$ by $\mathcal{H}_1 \cup \mathcal{H}_2$.

bottleneck: $f_e(x, y) = \min\{x, y\}$, $\psi_e(x_1, x_2, \dots, x_k) = \min_{1 \le i \le k}\{x_i\}$, and $\mu_0 = +\infty$;
traversal cost: $f_e(x, y) = x + y$, $\psi_e(x_1, x_2, \dots, x_k) = \sum_{1 \le i \le k}\{x_i\}$, when $w_e > 0$ for each $e \in A$, and $\mu_0 = 0$;
closure: $f_e(x, y) = \min\{x, y\}$, $\psi_e(x_1, x_2, \dots, x_k) = \min_{1 \le i \le k}\{x_i\}$, when $w_e = 1$ for each $e \in A$, and $\mu_0 = 0$;

For example, if we assume that all hyperarcs of the hypergraph in Fig. 1 but h_5 have unit weight, while hyperarc h_5 has weight 2, then the thick hyperpath has rank 3, gap 2 and traversal cost 4.

Some of the weight measures listed above have been used by other authors with different names. For example, in [21,34], the *rank* is called *distance*, and the *traversal cost* is called *value*.

3 Transitive Closure and Transitive Reduction

In this section, we summarize the main results achieved in the literature concerning the notions of *transitive closure* and *transitive reduction* in directed hypergraphs, that extend the analogous notions in the case of directed graphs. Both problems find application in the field of databases, where on one side we want to derive transitive functional dependencies, and on the other side we want to find minimal covers of sets of functional dependencies, that is a minimal set of functional dependencies equivalent to the given ones.

Transitive closure.
In [6] the authors introduce the notion of *transitive closure* of a directed hypergraph $\mathcal{H} = \langle N, A \rangle$ as the hypergraph $\mathcal{H}^+ = \langle N, A^+ \rangle$ such that $\langle X, i \rangle$ is a hyperarc in A^+ if and only if one of the following conditions hold:

- $\langle X, i \rangle \in A$
- $i \in X$ *Extended reflexivity*
- there exists a set of nodes $Y = \{n_1, n_2, \dots, n_q\}$ such that for each $1 \le j \le q$, $\langle X, n_j \rangle \in A^+$ and $\langle Y, i \rangle \in A$ *Extended transitivity*

Note that, in our framework such definition is equivalent to say that a hyperarc $\langle X, i \rangle$ is in A^+ if and only if there is a hyperpath in $\mathcal{H}$ from X to i. If X and Y are singletons, then the extended reflexivity and transitivity coincide with the usual notions of reflexivity and transitivity of graphs. Furthermore, note that the size of the closure of a directed hypergraph can be exponential in the number of nodes of the hypergraph, in fact, set X can be any element of 2^N (the power set of N).

In the same paper it is shown that using a suitable representation of $\mathcal{H}$, i.e., its FD-graph and the closure of the FD-graph (see [6] for the definitions), it is possible to compute the transitive closure of a directed hypergraph without falling into the exponential explosion of the hypergraph closure. In fact, the FD-graph closure grows at most quadratically. Using this approach the transitive closure of $\mathcal{H}$ is computed in $O(\text{size}(\mathcal{H}) \cdot n_s)$ worst case time, where n_s is the number of source sets of $\mathcal{H}$.

Transitive reduction.
The problem of finding the *transitive reduction* of a directed hypergraphs, that is finding a minimal hypergraph that has the same closure of a given one, is much more complex than in the case of graphs for two main reasons: *i*) the closure of a hypergraph has an exponential number of hyperarcs; *ii*) in the case of hypergraphs it is possible to define minimality with respect to a variety of parameters, while in the case of graph the only notion of minimality is with respect to the number of arcs. In [6], minimality in hypergraphs is defined with respect to the following criteria:

1. minimum size;
2. minimum number of hyperarcs;
3. minimum number of source sets;
4. minimum source area.

In the same paper, it has been shown that, given a directed hypergraph $\mathcal{H}$, the problems of deciding whether there exists a hypergraph $\mathcal{H}'$ with the same closure of $\mathcal{H}$, such that $\mathcal{H}'$ is minimum with respect to criteria 1, 2 or 4 are NP-hard, while deciding whether there exists a hypergraph $\mathcal{H}'$ with the same closure of $\mathcal{H}$ having the minimum number of source sets is polynomial and can be done in $O(\text{size}(\mathcal{H}) \cdot n_s)$ worst case time. Notice that the transitive reduction of a directed graph, that corresponds to minimality with respect to number of hyperarcs in a directed hypergraph, can be computed in polynomial time.

4 Flows and Cuts in Hypergraphs

Other classical graph problems that have been extended to hypergraphs are the problems of determining flows and cuts. In this section we review the main results achieved in the literature concerning these problems.

Flows in hypergraphs.
Flows in hypergraphs, also known as *hyperflows*, have been recently introduced as generalizations of *flows* in graphs. In particular, in [16] the *minimum cost hyperflow* problem is considered and some analogies with the standard *minimum cost flow* problem in directed graphs are shown.

In what follows, we summarize the results given in [16]. Given a directed hypergraph $\mathcal{H} = \langle N, A \rangle$, an *upper capacity* $u(e)$ and a *cost* $c(e)$ are associated with each $e \in A$, and a positive real *multiplier* $m_v(e)$ is associated with each $v \in \text{tail}(e)$. Moreover, a real *demand* $b(v)$ is associated with each $v \in N$. A *flow* in $\mathcal{H}$ is a function $f : A \to \mathbb{R}$, that satisfies the following *conservation* constraint:

$$\sum_{\text{head}(e)=v} f(e) \quad - \sum_{v \in \text{tail}(e)} m_v(e) f(e) = b(v), \quad \forall v \in N$$

The flow is *feasible* if it satisfies the following *capacity* constraint:

$$0 \leq f(e) \leq u(e), \quad \forall e \in A$$

The *minimum cost hyperflow* problem consists of finding a feasible flow in $\mathcal{H}$ which minimizes the value $\sum_{e \in A} c(e)f(e)$.

An interesting issue is computing the maximum flow in a directed hypergraph. Let us assume that $\mathcal{H}$ has only one node s such that $\text{bstar}(s) = \emptyset$ and only one node t such that $\text{fstar}(t) = \emptyset$; they are called the *source* and the *terminal* of $\mathcal{H}$, respectively (if s and t do not exist, dummy nodes and hyperarcs can be added in a standard way). The *maximum hyperflow* problem consists of finding a feasible flow in $\mathcal{H}$ that maximizes the amount of flow entering t, and satisfies the conservation constraint in the special case that $b(v) = 0$, for each $v \in N$.

In [16] the notion of *spanning hypertree* of a hypergraph is introduced as a generalization of the notion of spanning tree of a graph, and it is shown that there exists a correspondence between *basis matrices* and spanning hypertrees. Based on this correspondence, the authors propose a simplex like algorithm for minimum cost hyperflow computations, and show that most of the computations performed by this algorithm have a direct and elegant hypergraph interpretation.

Cuts in hypergraphs.
The notion of *cut* in a directed hypergraph has been defined in [21] as follows. A *cut* $\mathcal{C}_{st}$ between two nodes s and t of a directed hypergraph $\mathcal{H} = \langle N, A \rangle$ is a partition of N into two subsets N_s and N_t such that $s \in N_s$ and $t \in N_t$. The associated *cutset* is the set of all hyperarcs $e \in A$ such that $\text{tail}(e) \subseteq N_s$ and $\text{head}(e) \in N_t$. The *size* of a cut is the number of hyperarcs in its cutset.

It is well known that the unsatisfiability of a propositional Horn formula corresponds to the existence of a hyperpath in a hypergraph connecting two special nodes s and t, where s corresponds to True and t to False [9,19]. As a consequence, it is clear that the Maximum Horn Sat problem can be reduced to the problem of finding a *minimum cardinality cut* in a directed hypergraph, i.e., a cut whose associated cutset has the minimum number of hyperarcs. In [21], the authors also show that the latter problem is equivalent to the problem of finding the minimum set of hyperarcs of $\mathcal{H}$ such that each hyperpath of $\mathcal{H}$ contains at least one of these hyperarcs. Therefore, the minimum cardinality cut problem can be formulated as the problem of assigning 0/1 weights to the hyperarcs of $\mathcal{H}$ in order to make the weight of each hyperpath larger than or equal to 1, and to minimize the sum of the assigned weights.

Based on this result, the authors propose three different IP formulations for the minimum cardinality cut problem, by using three different measures to assign hyperpath weights, that is: the *cost*, the *distance* (*rank*), and the *value* (*traversal cost*). They also show that these three formulations and the well known ILP formulation of the Max Horn SAT problem are all equivalent.

Finally, they investigate the properties of the relaxations of these IP formulations by showing that they define a hierarchy. The weakest relaxation in the hierarchy, that is the relaxation of the formulation making use of the *value* (*traversal cost*) weight function, is shown to correspond to the *dual* of a hypergraph *max flow* problem with unit capacities [16]. This implies that the well-known max-flow-min-cut theorem for directed graphs (see, e.g., [17]) holds in the case of directed hypergraphs with unit upper capacities. For example, if we assume unit upper capacities for all the hyperarcs in the hypergraph of Fig. 1, and $m_v(e) = 1/|\text{tail}(e)|$ for each $v \in \text{tail}(e)$, then the minimum cut and the maximum flow are both equal to 2.

In the case of non unit upper capacities, the size of a cut is the sum of the upper capacities of the hyperarcs in the cutset. An interesting feature of hypergraphs is that, if hyperarcs have arbitrary upper capacities, then the max-flow-min-cut theorem does not hold any longer, even in the restrictive case in which we fix $m_v(e) = 1/|\text{tail}(e)|$ for each $v \in \text{tail}(e)$. In fact, if in the hypergraph shown in Fig. 1 we assume the following upper capacities: $u(h_1) = u(h_2) = u(h_3) = u(h_4) = 10$, $u(h_5) = 2$, $u(h_6) = 1$, $u(h_7) = 1$, $u(h_8) = 1$, $u(h_9) = 4$, $u(h_{10}) = 1$, then the minimum cut is 3 (partitioning nodes into the sets $\{s, a, b, c, d, f\}$ and $\{e, t\}$), while the maximum flow is 5.

5 Minimum Weight Traversal Problems

One of the most important problems in hypergraphs is finding minimum weight hyperpaths. Such problem finds relevant applications in reachability in Petri nets [1], reliability in fuzzy systems [8], assumption-based truth maintenance systems [18,27], computation of minimum models of Horn formulæ [9], search of optimal strategies in problem solving [30], analysis of dynamic programming [24].

Several results have been proposed in the literature concerning the problem of finding minimum (or maximum) weight hyperpaths in a directed hypergraph. As already mentioned, there are various ways in which a weight can be attached to a hyperpath. Depending on the weight measure used to assign a weight to a hyperpath the problem can be polynomially tractable or NP-hard. For example, in [11] it has been shown that the problem of finding minimum *cost* hyperpaths in a directed hypergraph is NP-hard, where the *cost* of a hyperpath is the sum of the weights of its hyperarcs. Conversely, when we take into consideration inductively defined measures, it is possible to characterize a number of cases in which the problem can be solved in polynomial time. Examples of inductively defined measures that can be computed in polynomial time are the *rank* and the *gap* [25]. Observe that cost, rank and gap are all generalizations of the standard notion of distance in graphs, which can be computed in polynomial time.

The more general approach for finding minimum weight hyperpaths in a functional directed hypergraph is finding a *fixed point* of the following set of equations, known as *generalized Bellman-Ford equations* (see, e.g., [21,35]):

$$L(v) = \begin{cases} 0 & \text{if } v \in S \\ \min\limits_{e \in \text{bstar}(v)} \{f_e(w_e, \psi_e(x_1, x_2, \ldots, x_k))\} & \text{if } v \in N \setminus S \end{cases}$$

where $\text{tail}(e) = \{x_1, x_2, \dots, x_k\}$, and f_e and ψ_e depend on the particular weight measure used. The complexity of this problem strongly depends on the characteristics of functions f_e and ψ_e, since, as we will see in the remainder of this section, in several cases the solution of the general problem can be achieved with a Dijkstra-like computation.

In his definition of grammar problems [26], Knuth introduces the concepts of *superior* and *inferior* function in the framework of context-free grammars, as follows.

Definition 3. *A function $g(x_1, \dots, x_k)$ from $(\mathbb{R}^+)^k$ to $\mathbb{R}^+$ is:*

- superior *(SUP) if it is monotone nondecreasing in each variable and $g(x_1, \dots, x_k) \geq \max(x_1, \dots, x_k)$*
- inferior *(INF) if it is monotone nondecreasing in each variable and $g(x_1, \dots, x_k) \leq \min(x_1, \dots, x_k)$*

Examples of superior functions are $\max_{1 \leq i \leq k}\{x_i\}$, and $\Sigma_{i=1}^{k} x_i$. Examples of inferior functions are: $\min_{1 \leq i \leq k}\{x_i\}$, and $\Pi_{i=1}^{k}\{x_i\}$ when $0 \leq x_i \leq 1$, $i = 1, \dots, k$. Ramalingam and Reps in [35] introduced the following generalizations of superior (inferior) functions.

Definition 4. *A function $g(x_1, \dots, x_k)$ from $(\mathbb{R}^+)^k$ to $\mathbb{R}^+$ is:*

- weakly superior *(WSUP) if it is monotone nondecreasing in each variable and, for $1 \leq i \leq k$, $g(x_1, \dots, x_k) < x_i \Rightarrow g(x_1, \dots, x_i, \dots, x_k) = g(x_1, \dots, \infty, \dots, x_k)$*
- weakly inferior *(WINF) if it is monotone nondecreasing in each variable and, for $1 \leq i \leq k$, $g(x_1, \dots, x_k) > x_i \Rightarrow g(x_1, \dots, x_i, \dots, x_k) = g(x_1, \dots, \infty, \dots, x_k)$*

Examples of weakly superior functions that are not superior are $\min_{1 \leq i \leq k}\{x_i\}$, $\min_{1 \leq i \leq k}\{x_i\} * 2$, and any constant function. Examples of weakly inferior functions that are not inferior functions are: $\max_{1 \leq i \leq k}\{x_i\}$, $\max_{1 \leq i \leq k}\{x_i\}/2$, and any constant function.

Both Knuth [26] and Ramalingam and Reps [35] considered also the classes of *strict superior*, *strict inferior*, *strict weakly superior*, and *strict weakly inferior* functions, as defined below.

Definition 5. *A function $g(x_1, \dots, x_k)$ from $(\mathbb{R}^+)^k$ to $\mathbb{R}^+$ is:*

- strict superior *(SSUP) if it is monotone nondecreasing in each variable and $g(x_1, \dots, x_k) > \max(x_1, \dots, x_k)$*
- strict inferior *(SINF) if it is monotone nondecreasing in each variable and $g(x_1, \dots, x_k) < \min(x_1, \dots, x_k)$.*

Definition 6. *A function $g(x_1, \dots, x_k)$ from $(\mathbb{R}^+)^k$ to $\mathbb{R}^+$ is:*

- strict weakly superior *(SWSUP) if it is monotone nondecreasing in each variable and, for $1 \leq i \leq k$, $g(x_1, \dots, x_k) \leq x_i \Rightarrow g(x_1, \dots, x_i, \dots, x_k) = g(x_1, \dots, \infty, \dots, x_k)$*

- strict weakly inferior *(SWINF) if it is monotone nondecreasing in each variable and, for* $1 \leq i \leq k$, $g(x_1,\ldots,x_k) \geq x_i \;\Rightarrow\; g(x_1,\ldots,x_i,\ldots,x_k) = g(x_1,\ldots,\infty,\ldots,x_k)$

We now relate the above defined classes of functions to hypergraphs. Remember that in a functional hypergraph, each hyperarc $e \in \mathcal{H}$ is associated to a triple (w_e, ψ_e, f_e). Given $e = \langle X, t\rangle$, with $X = \{x_1, x_2, \ldots, x_k\}$, the weight μ of any hyperpath $\Pi_{S,t}$ having e as the last hyperarc is given by $\mu(\Pi_{S,t}) = f_e(w_e, \psi_e(\mu(\Pi_{S,x_1}), \mu(\Pi_{S,x_2}), \ldots, \mu(\Pi_{S,x_k})))$. If functions f_e and ψ_e, for all $e \in \mathcal{H}$, are, say, superior functions, then the overall weight measure μ is superior as well (with respect to arguments w_e, for each hyperarc e in the considered hyperpath). The table in Fig. 2 summarizes the properties of the weight measures known in the literature with respect to the classes of function defined above. Concerning the relationship among the different classes of weight measures, as well as their compositions, we refer to [12].

weight measure μ	$f_e(w_e, \psi_e)$	$\psi_e(\mu_1, \ldots, \mu_k)$	resulting properties
rank $w_e > 0$	$+$ *SSUP*	max *SUP, WINF*	*SSUP*
gap $w_e > 0$	$+$ *SSUP*	min *WSUP, INF*	*SWSUP*
bottleneck $w_e > 0$	min *WSUP, INF*	min *WSUP, INF*	*WSUP, INF*
threshold $w_e > 0$	max *SUP, WINF*	max *SUP, WINF*	*SUP, WINF*
traversal cost $w_e > 0$	$+$ *SSUP*	$\sum$ *SSUP*	*SSUP*
closure $w_e = 1$	min *WSUP, INF*	min *WSUP, INF*	*WSUP, INF*

Fig. 2. Properties of hyperpath weight measures

The generalization of Dijkstra algorithm to directed hypergraphs (or similar structures) was first tackled by Knuth in [26]. For weight measures based on superior functions, it is possible to determine minimum cost derivation trees in a weighted context free grammar (or, equivalently, a minimum weight hyperpath in a functional directed hypergraph) in $O(\text{size}(\mathcal{H}) + m \log n)$ worst case time (or $O(\text{size}(\mathcal{H}) + n \log n)$ by using Fibonacci heaps [20]). This result was subsequently extended in [12,31,35] to weakly superior (inferior) functions, thus including all weight measures in Fig. 2. In [12], it is also shown how the suitability of Dijkstra-based algorithms is related to the properties of weight measures and to the existence of particular types of cycles in hypergraphs.

6 Dynamic Traversal Algorithms

In this section we consider the problem of dynamically maintaining reachability and minimum weight hyperpaths in a directed hypergraph subject to modifications as hyperarc insertions and deletions, and hyperarc weight changes. These problems arise in several application domains as, for example, in the minimum model maintenance in Horn Formulæ [7,8,9,35].

In this setting we do not want to recompute hyperpaths from scratch after each change, but we want to take advantage of the part of the previous solution that is still valid. A common classification of dynamic problems in hypergraphs is among fully-dynamic ones, where insertions and deletions of hyperarcs can be intermixed, and semi-dynamic ones (incremental or decremental), where only insertions or only deletions are allowed, respectively.

The first hypergraph problem stated in a dynamic framework has been the maintenance of the transitive closure of a directed hypergraph $\mathcal{H}$ under hyperarc insertions. In [10], it is shown that this problem can be solved in $O(\text{size}(\mathcal{H}) \cdot n_s)$ worst case time, where n_s is the number of source sets of $\mathcal{H}$, in such a way that checking the existence of a hyperpath takes $O(\log n_s)$ time.

Minimum weight traversal problems have been studied by various authors. For the incremental case, an algorithm is presented in [9], that maintains satisfiability and the minimum model of a Horn formula $\mathcal{F}$ in $O(n \cdot \text{Length}(\mathcal{F}))$ total time over a sequence of clause insertions, where $\text{Length}(\mathcal{F})$ is the sum of the number of literals in the clauses of $\mathcal{F}$. This incremental algorithm is extended in [8] to the case of Horn formulæ with uncertainty. Algorithms are given in [25] for the incremental maintenance of minimum rank and minimum gap hyperpaths under a sequence of hyperarc insertions in overall $O(n \cdot \text{size}(\mathcal{H}))$ time, under the assumption of unit hyperarc weights.

The fully dynamic version of the problem has been considered in [35]. The authors propose a Dijkstra-like procedure, applicable to strict weakly superior functions. The proposed algorithm takes $O(||\delta|| \cdot (\log ||\delta|| + M))$ worst case time per update operation, where each operation may consist of several insertions/deletions of hyperarcs and hyperarc weight changes. Parameter $||\delta||$ represents the number of nodes interested by the operation plus the total number of hyperarcs incident to these nodes, while M is the time needed to compute the weight function. Note that in the worst case $||\delta|| = m$.

In this section, we are interested in the decremental case for two reasons: first, it finds applications in the context of assumption-based truth maintenance systems [18,27], where maintaining minimum hyperpaths under deletions of hyperarcs corresponds to maintaining a set of "small" explanations of all observations under the elimination of either an hypothesis or a clause in the background theory. Second, better complexity bounds can be obtained with respect to the fully dynamic case. For example, in [7] a decremental algorithm is proposed for maintaining the minimum rank hyperpaths under a sequence of hyperarc deletions in overall $O(n \cdot \text{size}(\mathcal{H}))$ time and $O(\text{size}(\mathcal{H}))$ space, under the assumption of unit hyperarc weights. In the case of integer hyperarc weights in $[1, C]$ the algorithm requires $O(C \cdot n \cdot \text{size}(\mathcal{H}))$ total time and $O(C + \text{size}(\mathcal{H}))$ space to handle any sequence of hyperarc deletions and weight increase operations. This improves over the solution in [35] when applied to the decremental maintenance of minimum rank hyperpaths in the case of integer hyperarc weights in $[1, C]$.

In the next subsection, we propose a new decremental algorithm, with respect to those given in [7,35], that applies to all weight function in *SWSUP*. This algorithm is based on a novel approach, since it essentially performs a bfs-like visit, rather than a Dijkstra-like visit.

6.1 A New Decremental Algorithm

Let $\mathcal{H} = \langle N, A; F \rangle$ be a functional directed hypergraph with n nodes and m hyperarcs. Procedure `Weight_Increase` (see Fig. 3), maintains the minimum weight hyperpaths from a fixed source $S \subset N$ to all other nodes of $\mathcal{H}$ under hyperarc weight increases and hyperarc deletions.

In a minimization (maximization) problem, as a consequence of a hyperarc weight increase or hyperarc deletion, the weight of some hyperpath may increase or some hyperpaths may disappear, and the weight of minimum weight hyperpaths can only increase (decrease).

In what follows, we consider the minimization problem in the case that the weight of a hyperarc e is increased by a positive quantity δ, and then discuss how to modify Procedure `Weight_Increase` in order to handle hyperarc deletions. Each f_e is supposed to assume values in an interval of integers $[1, C]$.

We denote by $\mathcal{H}'$ the hypergraph obtained from $\mathcal{H}$ after updating hyperarc $e = \langle X, y \rangle$, and by $\mathrm{w}(v)$ $(\mathrm{w}'(v))$ the weight of the minimum weight hyperpath from S to v in $\mathcal{H}$ $(\mathcal{H}')$. Analogously, $\mathrm{p}(v)$ $(\mathrm{p}'(v))$ denotes the port of v in $\mathcal{H}$ $(\mathcal{H}')$.

First of all, we observe that after updating w_e in $\mathcal{H}$, each hyperpath in $\mathcal{H}$ that does not contain e preserves its weight. Hence, $\mathrm{w}'(v) = \mathrm{w}(v)$ for each node v whose minimum weight hyperpath in $\mathcal{H}$ does not contain e. We thus concentrate on the set of nodes in $\mathcal{H}$ whose current minimum weight hyperpath from S contains e. Given one of these nodes v, either $\mathrm{w}'(v) = \mathrm{w}(v)$, or $\mathrm{w}'(v) > \mathrm{w}(v)$.

Hypergraph $\mathcal{H}$ is represented by associating to each node v two simple lists containing all hyperarcs in $\mathrm{bstar}(v)$ and $\mathrm{fstar}(v)$. Minimum weight hyperpaths are represented as follows. For each node v, we store:

- $\mathrm{weight}(v)$, that coincides with $\mathrm{w}(v)$ $(\mathrm{w}'(v))$ before (after) the update;
- $\mathrm{port}(v)$, that coincides with $\mathrm{p}(v)$ $(\mathrm{p}'(v))$ before (after) the update.

For each hyperarc e, we explicitly store the value $F(e) = f_e(w_e, \psi_e(\mathrm{weight}(x_1), \mathrm{weight}(x_2), \ldots, \mathrm{weight}(x_k)))$, where $\{x_1, x_2, \ldots, x_k\} = \mathrm{tail}(e)$. In order to keep the space occupancy within $O(\mathrm{size}(\mathcal{H}))$, we explicitly represent each hyperarc in $\mathcal{H}$ only once; all the occurrences of hyperarcs in the above data structures are implemented as references.

Procedure `Weight_Increase` explores the set of nodes whose weight and/or port changes under the update of hyperarc $e = \langle X, y \rangle$, and builds the new minimum weight hyperpaths as follows. The hypergraph is examined starting from node y by a bfs-like visit (that is, by increasing minimum weight), that

```
Procedure Weight_Increase($e = \langle X, y\rangle$; $\delta$)
begin
1.    $w_e \leftarrow w_e + \delta$
2.    Update($e$,$v$)
3.    if port($y$) $\neq$ $e$ or $F(e)$ has not changed then EXIT
4.    NewWeightSet(weight($y$)) $\leftarrow$ $\{y\}$
5.    for $i$ $\leftarrow$ weight($y$) to $C$ do
6.        foreach $v \in$ NewWeightSet($i$) do
7.            delete $v$ from NewWeightSet($i$)
8.            search the first $h$ in bstar($v$) such that $F(h) = i$
9.            if $h$ exists then
10.               port($v$) $\leftarrow$ $h$
11.           else
12.               port($v$) $\leftarrow$ NIL
13.               weight($v$) $\leftarrow$ min$\{F(h) \mid h \in$ bstar($v$)$\}$
14.               insert $v$ in NewWeightSet(weight($v$))
15.               foreach $e' \in$ fstar($v$) do
16.                   Update($e'$,$v$)
17.                   if $F(e')$ has changed then
18.                       let $z$ be head($e'$)
19.                       if $e'$=port($z$) and $z \notin$ NewWeightSet(weight($z$)) then
20.                           insert $z$ into NewWeightSet(weight($z$))
21.               endfor
22.       endfor
23.   endfor
end
```

Fig. 3. Procedure `Weight_Increase`

is pruned any time a hyperarc is found that does not belong to any minimum weight hyperpath and/or whose weight does not change.

The i-th iteration of the for loop at Line 5 identifies the set of nodes having weight i (after the update), whose weight and/or port changes due to the hyperarc update, by selecting all nodes v such that $\mathrm{w}(v) = i$, $\mathrm{w}'(v) = i$ and $\mathrm{p}'(v) \neq \mathrm{p}(v)$, and all nodes v such that $\mathrm{w}(v) < i$ and $\mathrm{w}'(v) = i$.

Inspected nodes are temporarily stored in an array of sets of nodes, named NewWeightSet. A node v is put in NewWeightSet(i) if and only if $\mathrm{w}'(v)$ is known to be at least i. Nodes are extracted from set NewWeightSet(i) by increasing i, and for each node v we check whether there is a hyperarc h such that $f_h(w_h, \psi_h(\mathrm{tail}(h))) = i$. If this is the case, the weight of node v is set to i, otherwise v is inserted into some set NewWeightSet(j), where $j > i$, for future inspection.

Procedure `Update`(e,v), called at Lines 2 and 16 of Procedure `Weight_Increase`, updates $F(e)$, by re-evaluating ψ_e and f_e under the increase of weight(v), $v \in \mathrm{tail}(e)$, or w_e.

In order to manage hyperarc deletions, we add a new item NewWeightSet($C+1$), containing all nodes that are found to be unreachable as a consequence of a deletion. The algorithm detects unreachability from the source, by checking whether a node gets a weight greater than C or its backward star is empty. In these cases, unreachable nodes are put in the set NewWeightSet($C+1$) and considered at the end of the algorithm, by setting the port to nil and the weight to $+\infty$.

Provided that the weight functions associated to hyperarcs fulfill the following conditions:

- the composition of functions f_e and ψ_e is in *SWSUP* (*SWINF*) for each hyperarc e. For example, it is sufficient that f_e and ψ_e are both monotone non decreasing (non increasing), and either f_e or ψ_e are in *SWSUP* (*SWINF*);
- $f_e(w_e, \psi_e(x_1, x_2, \ldots, x_k))$ can be maintained under increments (decrements) of x_i within $O(k)$ space and $O(1)$ amortized time;
- $f_e(w_e, \psi_e(x_1, x_2, \ldots, x_k))$ can be maintained under increments (decrements) of w_e within $O(k)$ space and $O(k)$ worst case time.

we can show that Procedure `Weight_Increase` manages a sequence of hyperarc weight increases and deletions in $O((C+n) \cdot \text{size}(\mathcal{H}))$ overall worst case time.

Note that the algorithm in [7] is a special case of the algorithm in this paper, in fact it maintains minimum rank hyperpath of a directed hypergraph within the same time bounds of Procedure `Weight_Increase` and the rank is a special case of *SWSUP* weight functions (see table of Fig. 2). It is also clear that our algorithm improves that of [35] with respect to the amortized time per operation, if only hyperarc deletions and weight increases are allowed.

7 Concluding Remarks

There is a large body of results on implicative structures, that is structures in which we need to provide a formal representation for a set of many-to-one implications and to study the resulting properties, such as determining less expensive implication chains under various cost criteria. Our aim has been to illustrate the most important problems and the main algorithmic results that are available in the literature. Besides we have discussed in particular the problem of maintaining chains of implications under implication elimination. This problem consists of maintaining minimum hyperpaths (under some minimality criteria) under hyperarc deletions, and can be efficiently tackled for several natural hyperpath weight measures.

It is worth noting that the higher complexity of the problems we have discussed in this paper with respect to the complexity of the corresponding problems in graphs, arises from the peculiar combinatorial features of implicative structures (see, e.g., [12,31] for the notions of path, distance, cycle, etc). The comprehension of such aspects can be greatly enhanced by making use of a uniform representation of implicative structures in terms of directed hypergraphs.

Future research directions might address various goals. On one side, it would be desirable to determine hypergraph properties under which hypergraph problems become tractable; on the other side, approximability properties of hypergraph problems should be better explored; finally, since the connection between flows and cuts in hypergraphs is significantly different with respect to graphs, the complexity of the various formulations of flow problems in hypergraphs deserves further investigation.

References

1. P. Alimonti, E. Feuerstein, and U. Nanni. Linear time algorithms for liveness and boundedness in conflict–free petri nets. In *Proceedings of Latin American symposium on Theoretical INformatics (LATIN'92)*, volume 583 of *Lecture Notes in Computer Science*, pages 1–14. Springer-Verlag, 1992. 313, 319
2. C. Alonso, B. Pulido, and G. G. Acosta. On line industrial diagnosis : an attempt to apply artificial intelligence techniques to process control. In *11th International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems (IEA-98-AIE)*, volume 1415 of *Lecture Notes in Artificial Intelligence*, pages 804–813. Springer-Verlag, 1998. 313
3. H. R. Andersen. Model checking and boolean graphs. *Theoretical Computer Science*, 126(1):3–30, 1994. 313
4. J. Aráoz. Forward chaining is simple(x). *Operations Research Letters*, 26:23–26, 2000. 313
5. G. Ausiello, A. D'Atri, and D. Saccà. Graph algorithms for functional dependency manipulation. *Journal of the ACM*, 30(4):752–766, 1983. 313
6. G. Ausiello, A. D'Atri, and D. Saccà. Minimal representation of directed hypergraphs. *SIAM Journal on Computing*, 15(2):418–431, 1986. 313, 314, 316, 317
7. G. Ausiello, P. G. Franciosa, D. Frigioni, and R. Giaccio. Decremental maintenance of minimum rank hyperpaths and minimum models of Horn formulæ. Technical Report ALCOMFT-TR-01-19, IST Programme of the EU IST-1999-14186 (ALCOM-FT), 2001. http://www.brics.dk/cgi-alcomft/db?state=reports. 322, 323, 325
8. G. Ausiello and R. Giaccio. On-line algorithms for satisfiability problems with uncertainty. *Theoretical Computer Science*, 171(1–2):3–24, 1997. 313, 319, 322
9. G. Ausiello and G. F. Italiano. On-line algorithms for polynomially solvable satisfiability problems. *Journal of Logic Programming*, 10(1/2/3–4):69–90, 1991. 313, 318, 319, 322
10. G. Ausiello, G. F. Italiano, and U. Nanni. Dynamic maintenance of directed hypergraphs. *Theoretical Computer Science*, 72(2-3):97–117, 1990. 313, 322
11. G. Ausiello, G. F. Italiano, and U. Nanni. Optimal traversal of directed hypergraphs. Technical Report TR-92-073, International Computer Science Institute, Berkeley, CA, September 1992. 315, 319
12. G. Ausiello, G. F. Italiano, and U. Nanni. Hypergraph traversal revisited: Cost measures and dynamic algorithms. In *Symposium on Mathematical Foundations of Computer Science (MFCS'98)*, volume 1450 of *Lecture Notes in Computer Science*, pages 1–16. Springer-Verlag, 1998. 313, 314, 315, 321, 325
13. C. Berge. *Graphs and Hypergraphs.* North-Holland, Amsterdam, The Netherlands, 1973. 312

14. C. Berge. *Hypergraphs: combinatorics of finite sets*. North-Holland, Amsterdam, The Netherlands, 1989. 312
15. H. Boley. Directed recursive labelnode hypergraphs: A new representation language. *Artificial Intelligence*, 9(1):49–85, 1977. 313
16. R. Cambini, G. Gallo, and M. G. Scutellà. Flows on hypergraphs. *Mathematical Programming*, 78:195–217, 1997. 313, 317, 318, 319
17. T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to algorithms*. MIT Press and McGraw-Hill Book Company, 1992. 319
18. J. de Kleer. An assumption based truth maintenance system. *Artificial Intelligence*, 28:127–162, 1986. 319, 322
19. W. F. Dowling and J. H. Gallier. Linear-time algorithms for testing the satisfiability of propositional horn formulæ. *Journal of Logic Programming*, 1(3):267–284, 1984. 318
20. M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, 1987. 321
21. G. Gallo, C. Gentile, D. Pretolani, and G. Rago. Max horn SAT and the minimum cut problem in directed hypergraphs. *Mathematical Programming*, 80:213–237, 1998. 313, 316, 318, 319
22. G. Gallo, G. Longo, S. Nguyen, and S. Pallottino. Directed hypergraphs and applications. *Discrete Applied Mathematics*, 42:177–201, 1993. 314
23. G. Gallo and G. Rago. A hypergraph approach to logical inference for datalog formulæ. Technical Report TR-28/90, Dipartimento di Informatica, Università di Pisa, Italy, 1990. 313
24. S. Gnesi, U. Montanari, and A. Martelli. Dynamic programming as graph searching: An algebraic approach. *Journal of the ACM*, 28(4):737–751, 1981. 319
25. G. F. Italiano and U. Nanni. On-line maintenance of minimal directed hypergraphs. In *Italian Conference on Theoretical Computer Science*, pages 335–349, 1989. 319, 322
26. D. E. Knuth. A generalization of Dijkstra's algorithm. *Information Processing Letters*, 6(1):1–5, 1977. 313, 320, 321
27. K. Konolige. Abductive theories in artificial intelligence. In *Principles of Knowledge Representation*, pages 129–152. CSLI Publications, 1996. 319, 322
28. X. Liu and S. A. Smolka. Simple linear-time algorithms for minimal fixed points. In *International Colloquium on Automata, Languages and Programming (ICALP'98)*, volume 1443 of *Lecture Notes in Computer Science*, pages 53–65. Springer-Verlag, 1998. 313
29. D. Maier and J. D. Ullman. Connections in acyclic hypergraphs. In *Symposium on Principles of Database Systems (PODS'82)*, pages 34–39. ACM Press, 1982. 313
30. A. Martelli and U. Montanari. Additive and/or graphs. In *3rd International Joint Conference on Artificial Intelligence (IJCAI'73)*, pages 1–11, 1973. 319
31. S. Nguyen and S. Pallottino. Hypergraphs and shortest hyperpaths. In *Combinatorial Optimization*, volume 1403 of *Lecture Notes in Mathematics*, pages 258–271. Springer-Verlag, 1986. 315, 321, 325
32. N. J. Nilsson. *Problem solving methods in Artificial Intelligence*. McGraw-Hill, New York, 1971. 313
33. D. Pretolani. Satisfiability and hypergraphs. Technical Report TD-12/93, Dipartimento di Informatica, Università di Pisa, Italy, 1993. 313
34. D. Pretolani. A directed hypergraph model for random time dependent shortest paths. *European Journal of Operational Research*, 123:315–324, 2000. 313, 316

35. G. Ramalingam and T. Reps. An incremental algorithm for a generalization of the shortest-path problem. *Journal of Algorithms*, 21(2):267–305, 1996. 313, 319, 320, 321, 322, 323, 325

36. J. D. Ullman. *Principles of Database Systems*. Computer Science Press, 1982. 313

37. H. C. Yen. A unified approach for deciding the existence of certain Petri net paths. *Information and Computation*, 96(1):119–137, 1992. 313

Coupon Collectors, *q*-Binomial Coefficients and the Unsatisfiability Threshold

Alexis C. Kaporis[1], Lefteris M. Kirousis[1], Yannis C. Stamatiou[1,2], Malvina Vamvakari[1,2], and Michele Zito[3]*

[1] University of Patras, Department of Computer Engineering and Informatics
Rio 265 00, Patras, Greece
{kaporis,kirousis,stamatiu,mvamv}@ceid.upatras.gr
[2] Computer Technology Institute
Kolokotroni 3, 26321, Patras, Greece
[3] Department of Computer Science, University of Liverpool, UK
michele@csc.liv.ac.uk

Abstract. The problem of determining the unsatisfiability threshold for random 3-SAT formulas consists in determining the clause to variable ratio that marks the (experimentally observed) abrupt change from almost surely satisfiable formulas to almost surely unsatisfiable. Up to now, there have been rigorously established increasingly better lower and upper bounds to the actual threshold value. An upper bound of 4.506 was announced by Dubois et al. in 1999 but, to the best of our knowledge, no complete proof has been made available from the authors yet. We consider the problem of bounding the threshold value from above using methods that, we believe, are of interest on their own right. More specifically, we explain how the method of *local maximum satisfying truth assignments* can be combined with results for coupon collector's probabilities in order to achieve an upper bound for the unsatisfiability threshold less than 4.571. Thus, we improve over the best, with an available complete proof, previous upper bound, which was 4.596. In order to obtain this value, we also establish a bound on the q-binomial coefficients (a generalization of the binomial coefficients) which may be of independent interest.

1 Introduction

Let ϕ be a random 3-SAT formula constructed by selecting uniformly and with replacement m clauses from the set of all possible clauses with three literals of three distinct variables. It has been observed experimentally that as the numbers n, m of variables and clauses respectively tend to infinity, while the ratio m/n tends to a constant r, the random formulas exhibit a *threshold* behaviour: if $r > 4.17$ (approximately) then almost all random formulas are unsatisfiable while the opposite is true if $r < 4.17$. The constant r is called the *density* of the formula. On the theoretical side, Friedgut [10] has proved that there exists

* Research supported by EPSRC grant GR/L/77089.

A. Restivo, S. Ronchi Della Rocca, L. Roversi (Eds.): ICTCS 2001, LNCS 2202, pp. 328–338, 2001.

a sequence γ_n such that for any $\epsilon > 0$, if $m/n \leq \gamma_n - \epsilon$ for sufficiently large n then the probability of a random formula being satisfiable approaches 0 while if $m/n \geq \gamma_n + \epsilon$ for sufficiently large n then this probability approaches 1 although it is not known if the sequence γ_n convergences to some constant value γ. Thus, finding the *exact* value of the threshold point or even proving that a threshold value exists is still a major problem in probability and complexity theory. Up to now, only upper and lower bounds have been rigorously established for the threshold value. The best lower bound has been recently proved by Achlioptas and Sorkin [1] and it is 3.26 while the currently best upper bound has been announced by Dubois et al. [7] and it is 4.506.

In this paper, we address the upper bound question for the unsatisfiability threshold from a new perspective that combines the idea of *local maximum* satisfying truth assignments proposed by Kirousis et al. [14], with the use of sharp estimates on some of the probabilities involved based on results about the so called coupon collector experiment (see for instance [17] and references thereafter). We obtain an upper bound of 4.571 thus improving over the best, with an available complete proof, previous upper bound (4.596 given in [12]). As a by-product of our proof, we also establish an upper bound to the q-binomial coefficients (a generalization of the binomial coefficients). Despite the extensive literature on q-binomial coefficients (see, e.g., [9,11,15]), no such bound was, to the best of our knowledge, known.

2 The Method of Local Maxima

In this section, we will state briefly the methodology followed in [14] and obtain the starting upper bound on the probability that a random formula is satisfiable. Let $\mathcal{S}$ be the class of all truth assignments to n variables and $\mathcal{A}_n$ the (random) class of truth assignments that satisfy a random formula ϕ. For a given $A \in \mathcal{S}$, a *single flip sf* is the change in A of exactly one FALSE value to TRUE and by A^{sf} we denote the truth assignment that results from this change. We define as $\mathcal{A}_n^1 \subseteq \mathcal{A}_n$ the random class of truth assignments with the following two properties:

- $A \models \phi$,
- for every single flip *sf*, it holds $A^{sf} \not\models \phi$.

A partial order can be defined on $\mathcal{S}$: a truth assignment A is smaller than a truth assignment A' iff there exists an i such that both A and A' assign the same value to all variables x_j, for all $j < i$ while A assigns FALSE to x_i and A' assigns TRUE to it. The random class $\mathcal{A}_n^1$ coincides with the set of satisfying truth assigments that are *local maxima* with respect to the partial order defined above among satisfying truth assignments that differ in one bit.

A more restricted random class of truth assignments results from $\mathcal{A}_n^1$ if we extend the scope of locality in obtaining a local maximum. A *double flip* is the change of exactly two variables x_i and x_j (with $i < j$) where x_i is changed from FALSE to TRUE and x_j from TRUE to FALSE. In analogy with single flips, by A^{df}

we denote the truth assignment that results from A if we apply the double flip df. Let $\mathcal{A}_n^{2\sharp}$ be defined as the set truth of assignments A that have the following properties:

- $A \models \phi$,
- for all single flips *sf*, it holds $A^{sf} \not\models \phi$,
- for all double flips *df*, it holds $A^{df} \not\models \phi$.

Our starting point is the following inequality:

Lemma 1. [14]

$$\begin{aligned}\Pr[\phi \text{ is satisfiable}] \leq \mathrm{E}[|\mathcal{A}_n^{2\sharp}|] &= \sum\nolimits_{A\in S} \Pr[\forall df\ A^{df} \not\models \phi, \forall sf\ A^{sf} \not\models \phi, A \models \phi] \\ &= (7/8)^{rn} \sum\nolimits_{A\in\mathcal{S}} \Pr[\forall df\ A^{df} \not\models \phi, \forall sf\ A^{sf} \not\models \phi \mid A \models \phi] \\ &= (7/8)^{rn} \sum\nolimits_{A\in\mathcal{S}} \Pr[\forall sf\ A^{sf} \not\models \phi \mid A \models \phi] \cdot \Pr[\forall df\ A^{df} \not\models \phi \mid A \in \mathcal{A}_n^1]. \qquad (1)\end{aligned}$$

In order to find an upper bound for the unsatisfiability threshold, it suffices to find the smallest possible value for r for which the right-hand side of (1) tends to 0. In the sections to follow, we will describe the sequence of steps that will lead us to the determination of an upper bound on the probabilities that appear in the third line of (1).

3 Coupon Collectors and Single Flips

For notational convenience, we will consider a formula ϕ as a set of clauses. Thus, the expression $\phi \cap \mathcal{A}$, with $\mathcal{A}$ a set of clauses, has the meaning of set intersection with the additional requirement that a clause that appears in the intersection, appears as many times as it appears in ϕ.

Given a truth assignment A, and a variable x such that $A(x) =$ FALSE, the set of *critical clauses* for x in A, $\mathcal{B}(A, x)$, is the set of clauses whose unique TRUE literal is $\neg x$. Note that $|\mathcal{B}(A, x)| = \binom{n-1}{2}$ and $\mathcal{B}(A, x) \cap \mathcal{B}(A, y) = \emptyset$ for $x \neq y$.

Assuming A sets k variables FALSE, the probability $\Pr[\forall sf\ A^{sf} \not\models \phi \mid A \models \phi]$ in (1) is the ratio between a function $N(n, m, k)$, counting the number of ways to build a formula with m clauses out of n variables containing at least one critical clause for each of the k *critical* variables, and the total number of ways to build a formula on m clauses out of n variables which is satisfied by A. Hence

$$\Pr[\forall sf\ A^{sf} \not\models \phi \mid A \models \phi] = \frac{N(n,m,k)}{[7\binom{n}{3}]^m}$$

If ϕ contains $l \in \{k, k+1, \ldots, m\}$ critical clauses, then

$$\Pr[\forall sf\ A^{sf} \not\models \phi \mid A \models \phi] = \sum\nolimits_{l=k}^{m} \frac{C(n,m,k,l)\ \ R(n,m,k,l)}{[7\binom{n}{3}]^m}$$

where $C(n, m, k, l)$ counts the number of ways of choosing l critical clauses so that at least one member of $\mathcal{B}(A, x)$ is chosen for each of the k critical variables and $R(n, m, k, l)$ counts the number of ways of filling up the remainder of ϕ with $m - l$ clauses that are true under A but not critical.

Lemma 2. *For any choice of the parameters* $R(n,m,k,l) = \left(7\binom{n}{3} - k\binom{n-1}{2}\right)^{m-l}$.

Proof. There are $7\binom{n}{3}$ clauses consistent with A. If A forces k variables to be critical there are k disjoint groups of $\binom{n-1}{2}$ critical clauses. □

Lemma 3. *For any choice of the parameters* $C(n,m,k,l) = \binom{m}{l}\left[k\binom{n-1}{2}\right]^l$ $\mathrm{coupon}(l,k)$ *where* $\mathrm{coupon}(l,k)$ *is the probability that a coupon collector picks* k *distinct random coupons over* l *trials.*

Proof. Assume that there are k critical variables associated with a given assignment A. Moreover ϕ contains l critical clauses. There are $\binom{m}{l}$ ways of choosing l positions out of the m available. Also, there are $k\binom{n-1}{2}$ critical clauses. Therefore, if we do not distinguish among the non-critical clauses, there are $\binom{m}{l}\left[k\binom{n-1}{2}\right]^l$ ways of choosing a sequence of m clauses so that exactly l of them are critical. Since $C(n,m,k,l)$ counts the number of these which has at least one occurrence of a critical clause for each of the k critical variables, and since there are equal numbers of possible critical clauses for each variable, the ratio of these terms is the probability $\mathrm{coupon}(l,k)$. □

To be able to state the main result in this section we need to quote a result giving asymptotic approximations to the probabilities $\mathrm{coupon}(l,k)$.

Theorem 1. [4] *Let* $x = l/k$ *with* $l = \Theta(k)$. *For all* $x > 1$ *define* $g_1(x) =_{df} (\mathrm{e}^{r_0} - 1)\left(\frac{x}{\mathrm{e}r_0}\right)^x$ *where* r_0 *is the solution of* $\frac{r\mathrm{e}^r}{\mathrm{e}^r - 1} = x$. *Also let* $g_1(1) = \mathrm{e}^{-1}$. *Then for all sufficiently large integer* k *and all* $x \geq 1$, $\mathrm{coupon}(l,k) \sim g_1(x)^k$.

The proof of the following theorem is entailed by the argument above, the use of the estimate given in Theorem 1 and Stirling's approximation to the various factorials involved. In the following result $F \asymp G$ denotes the fact that $\ln F \sim \ln G$. So for example $\binom{m}{l} \asymp \left[(\frac{rn}{l})^{\frac{l}{rn}}(\frac{rn}{rn-l})^{(1-\frac{l}{rn})}\right]^{rn}$.

Theorem 2. *The probability that a truth assignment* A *with* αn FALSE *values is a local maximum satisfies:*

$$\Pr[\forall sf\ A^{sf} \not\models \phi \mid A \models \phi] \asymp \sum_{l=\alpha n}^{rn} \left(\frac{3\alpha rn}{7l}\right)^l \left(\frac{(7-3\alpha)rn}{7(rn-l)}\right)^{rn-l} g_1\left(\frac{l}{\alpha n}\right)^{\alpha n}. \quad (2)$$

An important remark is that in the expression given in Theorem 2, two *polynomially large* factors have been omitted: one implicit in the relation "∼" used in Theorem 1 and one related to the asymptotics of the binomial coefficients. However, for our goal of making a certain expression that contains (2) converge to 0, such factors are immaterial and what is required is an optimal estimate only for the exponential factors which is guaranteed by Theorem 1 and the asymptotics for the binomial coefficients given above.

4 Probability Models for Random Formulas

A random 3-SAT formula ϕ with $m = rn$ clauses is most commonly formed according to one of the following probability models (Ω is the set of $8\binom{n}{3}$ possible 3-SAT clauses):

1. Select the m clauses of ϕ, drawing each clause uniformly and independently from Ω, with replacement (model G_{mm}).
2. Select the m clauses of ϕ, drawing each clause uniformly and independently from Ω, without replacement (model G_m).
3. With probability $p(n)$ each clause is chosen independendly of the others and with probability $p(n)$ for inclusion in ϕ (model G_p).

The probability that a random formula ϕ generated according to model G_m, G_{mm} or G_p belongs to a set Q defining some property, is denoted by $\Pr_m[\phi \in Q]$, $\Pr_{mm}[\phi \in Q]$ and $\Pr_p[\phi \in Q]$ respectively. Notice that the probabilities in (1) are all in G_{mm} since the model we considered until now allows clause repetitions when forming a formula. We will now outline an argument showing that the second probability in the third line of (1) can be rewritten into the G_p model, in order to take advantage of the computation of this probability in G_p that has already been performed in [14].

Consider again an arbitrary but fixed truth assignment A. As all probabilities which will undergo a change in the probabilistic model are conditional on $A \models \phi$, in the considerations below we assume that the universe of all clauses is restricted to those that are satisfied by A, and consequently that $p(n) = \frac{rn}{7\binom{n}{3}} \sim \frac{6r}{7n^2}.$

First let "NoRep" be the event that ϕ has no two clauses identical and let $\overline{\text{NoRep}}$ its complement. Then, because the order of the number of all possible clauses is $\Theta(n^3)$ and the order of the number of the clauses contained in ϕ is $\Theta(n)$, $\lim_{n\to\infty} \Pr_{mm}[\overline{\text{NoRep}}] = 0$.

Now let Q_1 and Q_2 be two arbitrary events such that the following two conditions, which we call *regularity* conditions hold:

- For some $\epsilon > 0$ and for all n, $\ln(\Pr_{mm}[Q_2|Q_1]) < -\epsilon$, i.e. $\Pr_{mm}[Q_2|Q_1]$ is bounded away from 1.
- $\lim_{n\to\infty} \Pr_{mm}[\overline{\text{NoRep}}|Q_1, Q_2] = \lim_{n\to\infty} \Pr_{mm}[\overline{\text{NoRep}}|Q_1] = 0$.

Notice that the events we consider in this paper have probabilities (conditional or not) that are exponentially small, so the first of the two regularity conditions is satisfied. Also, the second regularity condition is true when Q_1 and Q_2 are the events $A \in \mathcal{A}_n^1$ and $\forall df\ A^{df} \not\models \phi$, respectively. Indeed both these events and their conjunction are negatively correlated with $\overline{\text{NoRep}}$, so $\Pr_{mm}[\overline{\text{NoRep}}|Q_1] \leq \Pr_{mm}[\overline{\text{NoRep}}] \to 0$ and similarly for $\Pr_{mm}[\overline{\text{NoRep}}|Q_1, Q_2]$. To prove the negative correlation claim for, say, Q_1 and $\overline{\text{NoRep}}$, observe that the correlation claim is equivalent to $\Pr_{mm}[Q_1|\text{NoRep}] \geq \Pr_{mm}[Q_1]$, which in turn is equivalent to $\Pr_m[Q_1] \geq \Pr_{mm}[Q_1]$. This last inequality is intuitively obvious under the assumption that $A \models \phi$, because the probability to get at least a critical clause for each critical variable of the satisfying truth assignment A increases when the clauses of the formula are assumed to be different. For a formal proof of this for general increasing and reducible properties (like Q_1 and Q_2), we refer to [13]. Therefore, the second regularity condition is also true for the probabilities we will consider below.

Under the above regularity conditions, we have that:

$$\Pr_m[Q_2|Q_1] \asymp \Pr_{mm}[Q_2|Q_1]. \tag{3}$$

Indeed,

$$\begin{aligned}\Pr_m[Q_2\ |Q_1] = \Pr_{mm}[Q_2\ |Q_1, \text{NoRep}] &= \frac{\Pr_{mm}[Q_2\ |Q_1] - \Pr_{mm}[Q_2 \wedge \overline{\text{NoRep}}\ |Q_1]}{1 - \Pr_{mm}[\overline{\text{NoRep}}\ |Q_1]} \\ &= \Pr_{mm}[Q_2\ |Q_1] \frac{1 - \Pr_{mm}[\overline{\text{NoRep}}\ |Q_2, Q_1]}{1 - \Pr_{mm}[\overline{\text{NoRep}}\ |Q_1]}.\end{aligned}$$

Now first taking logarithms, then dividing both sides with $\ln\left(\Pr_{mm}[Q_2|Q_1]\right)$ and finally letting $n \to \infty$, we get the required inequality (the regularity conditions are needed in the computation of the limits).

On the other hand, it follows easily from Theorem II.2 (iii) in [3] that:

$$\Pr_m[Q_2\ |Q_1] \leq 3m^{1/2}\Pr_p[Q_2\ |Q_1]. \tag{4}$$

The inequality above has been proved by Bollobas for an arbitrary unconditional event. In general, it might not be true for conditional events. However, if both Q_2 and Q_1 are monotone increasing (i.e. $\Pr_{m_2}[Q_i] \geq \Pr_{m_1}[Q_i]$ for any $m_2 \geq m_1$, for both $i = 1, 2$), then it still holds true. For an informal explanation why this is indeed so, first observe that the conditional Q_1, being monotone increasing, "forces more clauses" into the formula in the variable-length G_p model. Since Q_2 is also monotone increasing, this has as a consequence that the conditional probability of Q_2 in G_p deviates even further to the right from the corresponding conditional probability in the fixed-length model G_m. A formal proof of this, based in the four-functions theorem of Ahlswede and Daykin [8] (see [2] for a nice presentation) will be given in the full paper. Finally, notice that the events Q_2 and Q_1, for which we apply this inequality below are trivially monotone increasing.

Now, (3) (4) and (1) imply that $\Pr_{mm}[\phi$ is satisfiable] is at most (ignoring polynomial factors):

$$\left(\frac{7}{8}\right)^{rn} \sum_{A \in \mathcal{S}} \Pr_{mm}[\forall sf\ A^{sf} \not\models \phi \mid A \models \phi] \cdot \Pr_p[\forall df\ A^{df} \not\models \phi \mid A \in \mathcal{A}_n^1]. \tag{5}$$

We are now in a position to use the probability calculations performed earlier in this paper using the coupon collector analogy for the method of local maxima with the probability calculations in [14] in order to derive a value for r that makes the right-hand side of (5) converge to 0. It is perhaps interesting to note that $\Pr_p[A \models \phi] \not\asymp \Pr_{mm}[A \models \phi]$ (the former is larger). Therefore, as both $\Pr_p[A \models \phi]$ and $\Pr_{mm}[A \models \phi]$ are easily computable, it was advantageous to retain in the first factor of the right hand side of (5) the value of $\Pr[A \models \phi](=$

$(7/8)^{rn}$) computed in the model G_{mm} rather than replace it with its value in G_p ($\Pr_p[A \models \phi] \asymp (e^{-(1/8)rn})$. Also, $\Pr_p[\forall sf\ A^{sf} \not\models \phi \mid A \models \phi] \not\asymp \Pr_{mm}[\forall sf\ A^{sf} \not\models \phi \mid A \models \phi]$. The coupon collector analogy helped us exploiting the advantage of the model G_{mm} for the probability of single flips. We were unable to do the same for the computation of $\Pr_{mm}[\forall df\ A^{df} \not\models \phi \mid A \in \mathcal{A}_n^1]$, so we resorted to the "easier", but worse, model G_p (in contrast, by Equation (3), the models G_{mm} and G_m are not asymptotically distinguishable, ignoring polynomial factors).

5 Calculations

Let *sf*(A) denote the number of FALSE values assigned by a truth assignment A. We define the following functions of r:

$$u = e^{-r/7}$$

$$z = -\frac{6u^6 \ln(1/u)}{1-u^3} + \frac{18u^9 \ln^2(1/u)}{(1-u^3)^2}\phi_2\left(\frac{6u^6 \ln(1/u)}{1-u^3}\right) \tag{6}$$

$$X(sf(A)) = \Pr_{mm}[\forall sf\ A^{sf} \not\models \phi \mid A \models \phi] \tag{7}$$

$$Y = 1 + z\frac{1}{n} + o\left(\frac{1}{n}\right), \quad \text{(notice that } z < 0), \tag{8}$$

where $\phi_2(t)$ is the smallest root of $\phi_2(t) = e^{t\phi_2(t)}$ and can also be expressed by means of the Lambert $\mathcal{W}$ function [6]. Although ϕ_2 is defined only in the interval $[0, e^{-1}]$, we have verified that the argument to ϕ_2 in the definition of z in (6) lies in this interval for all $r \geq 0$. In [14], it was proved that

$$\Pr_p[\forall df\ A^{df} \not\models \phi \mid A \in \mathcal{A}_n^1] \leq Y^{df(A)}.$$

Therefore, using Equations (7) and (8), expression (5) may be written as follows:

$$\left(\frac{7}{8}\right)^{rn} \sum\nolimits_{A \in \mathcal{A}_n} X(sf(A)) Y^{df(A)}. \tag{9}$$

Furthermore the following equality can be derived [14] by induction on n:

$$\sum\nolimits_{A \in \mathcal{A}_n} X(sf(A)) Y^{df(A)} = \sum_{k=0}^{n} \binom{n}{k}_Y X(k), \tag{10}$$

where $\binom{n}{k}_q$ denotes the *q-binomial* or *Gaussian* coefficients (see [11]). From expression (9), the definition of $X(sf(A))$ in (7), Equality (10) and Theorem 2, we deduce that $\Pr_{mm}[\phi$ is satisfiable$]$ is at most (omitting a polynomial factor)

$$\left(\frac{7}{8}\right)^{rn} \sum_{k=0}^{n} \sum_{l=k}^{rn} \binom{n}{k}_Y E\left(\frac{k}{n}, \frac{l}{rn}, r, n\right), \tag{11}$$

where

$$E(\alpha, \beta, r, n) = \left[\left(\tfrac{3\alpha}{7\beta}\right)^{\beta r} \left(\tfrac{7-3\alpha}{7(1-\beta)}\right)^{r - r\beta} g_1\left(\tfrac{r\beta}{\alpha}\right)^{\alpha}\right]^n$$

We will now consider an arbitrary term of the double sum that appears in (11) and examine for which values of r it converges to 0. If we find a condition on r that forces all such terms to converge to 0, then the whole sum will converge to 0 since it contains polynomially many terms, all of which vanish exponentially fast. This technique avoids the problem of finding a closed-form upper bound for the sum itself. However, in order to handle an arbitrary term, we need an upper bound for the q-binomial coefficients. To establish such a bound one will need the following standard result:

Lemma 4. [18] *Let $f(z) = \sum_{i=0}^{\infty} f_i z^i$ be the generating function for the sequence f_i, $i \geq 0$. Then if $f(z)$ is analytic in $|z| < R$ and if $f_i \geq 0$ for all $i \geq 0$, then for any t, $0 < t < R$, and any $n \geq 0$, it holds that $f_n \leq t^{-n} f(t)$.*

Using this lemma, we can prove the following:

Theorem 3. *Let $\binom{n}{\alpha n}_q$ denote the q-binomial coefficients for α real in $(0,1)$ and αn an integer. Then the following inequality holds:*

$$\binom{n}{\alpha n}_q \leq 2q^{-\binom{n}{2}} x_0^{-n} e^{\frac{1}{\ln q}[\mathrm{dilog}(1+x_0) - \mathrm{dilog}(1+x_0 q^{n-1})]} \tag{12}$$

where $x_0 = \frac{1-q^{\alpha n}}{q^{\alpha n} - q^{n-1}}$ and $\mathrm{dilog}(x) = \int_1^x \frac{\ln t}{1-t} dt$.

Proof. For the ordinary generating function of $q^{\binom{i}{2}} \binom{n}{i}_q$ it holds [5, p.118]

$$\begin{aligned}\sum_{i=0}^{n} q^{\binom{i}{2}} \binom{n}{i}_q x^n = \prod_{i=1}^{n} (1 + xq^{i-1}) &= e^{\sum_{i=1}^{n} \ln(1+xq^{i-1})} \\ &= (1+x) \cdot e^{\sum_{i=2}^{n} \ln(1+xq^{i-1})}.\end{aligned}$$

Since $\ln(1 + xq^{i-1})$ is decreasing in i,

$$\sum_{i=0}^{n} q^{\binom{i}{2}} \binom{n}{i}_q x^i \leq (1+x) \cdot e^{\int_1^n \ln(1+xq^{i-1}) di} = (1+x) \cdot e^{\frac{1}{\ln q}[\mathrm{dilog}(1+x) - \mathrm{dilog}(1+xq^{n-1})]}.$$

Applying Lemma 4 and using the fact that $x \leq 1$, we obtain the inequality

$$q^{\binom{i}{2}} \binom{n}{i}_q \leq x^{-i} (1+x) \cdot e^{\frac{1}{\ln q}[\mathrm{dilog}(1+x) - \mathrm{dilog}(1+xq^{n-1})]}. \tag{13}$$

The above inequality holds for any value of $x \in (0,1)$. Therefore, we may optimize it by choosing the value $x_0 = \frac{1-q^i}{q^i - q^{n-1}}$ that minimizes the expression on the right-hand side of (13). From this we obtain

$$\binom{n}{i}_q \leq 2q^{-\binom{i}{2}} x_0^{-i} e^{\frac{1}{\ln q}[\mathrm{dilog}(1+x_0) - \mathrm{dilog}(1+x_0 q^{n-1})]}. \tag{14}$$

which gives the required inequality by setting $i = \alpha n$. □

Setting $q = Y = 1 + z/n$ in (12) and using the approximation $\ln(1+z/n) \sim z/n$, as $n \to \infty$, the following can be derived:

$$\binom{n}{\alpha n}_q \leq 2\left[\left(\frac{1}{x_0}\right)^{\alpha} \cdot e^{-\frac{\alpha^2 z}{2}+\frac{1}{z}[\mathrm{dilog}(1+x_0)-\mathrm{dilog}(1+x_0 e^z)]}\right]^n, \tag{15}$$

where $x_0 = \frac{1-e^{\alpha z}}{e^{\alpha z}-e^z}$, which is expedient in the proof of the following:

Theorem 4. *An arbitrary term of the double sum in (11) is asymptotically (ignoring polynomial multiplicative factors) bounded from above by:*

$$T_r(\alpha,\beta)^n = \left[\left(\frac{3\alpha}{7\beta}\right)^{\beta r}\left(\frac{7-3\alpha}{7(1-\beta)}\right)^{r-r\beta} g_1\left(\frac{r\beta}{\alpha}\right)^{\alpha} \frac{e^{-\frac{\alpha^2 z}{2}+\frac{1}{z}[\mathrm{dilog}(1+x_0)-\mathrm{dilog}(1+x_0 e^z)]}}{{x_0}^{\alpha}}\right]^n$$

where, $\alpha = \frac{k}{n}$, $\beta = \frac{l}{rn}$, $x_0 = \frac{1-e^{\alpha z}}{e^{\alpha z}-e^z}$, *$z$ as given in (6).*

An immediate consequence of this result is that the smallest value of r for which $T_r(\alpha,\beta)$ is smaller than 1 for all $\alpha \in (0,1)$ and $\beta \in (\alpha/r, 1)$ is an upper bound for the unsatisfiability threshold.

We finally claim that for any value of r, the expression $\ln T_r(\alpha,\beta)$ is a convex function of α,β over the domain $\mathcal{D} = \{\alpha,\beta \in [0,1] \text{ and } \frac{\beta r}{\alpha} \geq 1\}$. Therefore we will compute its *unique* maximum value for $r = 4.571$ and $(\alpha,\beta) \in \mathcal{D}$. Due to the complexity of the expression for $\ln T_r(\alpha,\beta)$, we maximized it numerically using a Maple [16] implementation of *Downhill Simplex*. This implementation is based on the method and the code described in [19] and it is freely distributed by F.J. Wright in his Web page [21]. Using the plots of $T_r(\alpha,\beta)$ we obtained with Maple, we chose as a starting set of values for the downhill simplex algorithm the values $(\alpha,\beta) = (0.42, 0.21)$ and we set the accuracy and the scale parameters equal to 10^{-50}. In addition, we set the *Digits* parameter of Maple (accuracy of floating point numbers) equal to 100. We ran the algorithm and it returned as the maximum value of $\ln T_r(\alpha,\beta)$ the value -0.0000884. Additionally, we computed all the partial derivatives of $\ln T_r(\alpha,\beta)$ at the point where downhill simplex claims that it has located the maximum and they were found to be numerically equal to 0. Therefore, this provides additional support that at this point the function attains its maximum. As a final check, we generated 30000 random points close to the point at which downhill simplex finds the maximum of $\ln T_r(\alpha,\beta)$ and we confirmed that the value of $\ln T_r(\alpha,\beta)$ is not above the value returned by the method. All these considerations show that -0.0000884 is a global maximum of $\ln T_r(\alpha,\beta)$, which establishes the value $r = 4.571$ as an upper bound to the unsatisfiability threshold.

6 Discussion

We derived an upper bound for the unsatisfiability threshold that improves over all previously proved upper bounds, except the one announced in [7]. We looked at the problem from a new perspective, by combining the method of local maximum satisfying truth assignments proposed in [14] with the sharp estimates on some of the probabilities involved based on the coupon collector experiment. In addition, we gave a relationship between two conditional probability spaces for generating random formulas that allowed us to use probability calculations performed in the easier to handle probability model according to which each of the clause is selected independently of the others with some fixed probability to appear in the formula. As a final ingredient, we proved a tight upper bound for the q-binomial coefficients that may be of interest in its own right. Our approach showed that the unsatisfiability threshold is less than 4.571. This bound improves over the best previous upper bound with a complete proof (4.596, see [12]). Dubois et al. in [7] have announced the value 4.506 but to the best of our knowledge, no complete proof is available yet from the authors. Nevertheless, we believe that our approach contains elements of a separate value and interest that might be useful in another context or in other applications: exact (in the exponential order) computation of the first probability in the last line of (1) using the coupon collector problem, relationships between conditional probability models and an upper bound for the q-binomial coefficients. And even though we used a numerical method for maximizing our function in order to show that for $r = 4.571$ it is strictly below 1 for every legal value of its parameters, our proof that the function is convex and the observation that its derivatives are bounded, renders our proof essentially rigorous since the Downhill Simplex method is certain to find a *global* maximum within *guaranteed* accuracy. We also believe that our approach of using the occupancy problem for the accurate computation of the first probability in (1) can be extented in order to give a sharp estimate also for the second probability in the last line of (1). To this end, we are currently working on extending the coupon collector approach or some similar scheme to model *double flips*. If this is accomplished, then it is conceivable that an analogue of Theorem 2 can be proved that will enable further improvements on the value obtained in this paper. There is also the question of combining this approach with the idea of "typical formulas" proposed in [7], thus obtaining still better bounds, under 4.5.

References

1. D. Achlioptas, G. B. Sorkin, "Optimal Myopic Algorithms for Random 3-SAT" in: *Proc. 41st Annual Symposium on Foundations of Computer Science (FOCS)*, 2000. pp 590–600. 329
2. N. Alon, J. H. Spencer, P. Erdős, *The Probabilistic Method*, John Wiley and Sons, 1992. 333
3. B. Bollobás, *Random Graphs*, Academic Press, London, 1985. 333
4. V. Chvátal, "Almost all graphs with 1.44n edges are 3-colourable," *Random Structures and Algorithms* 2, pp 11–28, 1991. 331
5. L. Comtet, *Advanced Combinatorics; The Art of Finite and Infinite Expansions*. D. Reidel, 1974. 335
6. R. M. Corless, G. H. Gonnet, D. E. G. Hare, D. J. Jeffrey, D. E. Knuth, "On the Lambert W function," manuscript, Computer Science Department, University of Waterloo. 334

7. O. Dubois, Y. Boufkhad, J. Mandler, “Typical random 3-SAT formulae and the satisfiability threshold,” in: *Proc. 11th Symposium on Discrete Algorithms (SODA)*, pp 126–127, 2000. 329, 336, 337
8. C. M. Fortuin, R. W. Kasteleyn, J. Ginibre, Correlation inequalities on some partially ordered sets, *Commun. Math. Physics* 22, pp. 89–103, 1971. 333
9. N. J. Fine, *Basic Hypergeometric Series and Applications*, Mathematical Surveys and Monographs, Number 27, 1980. 329
10. E. Friedgut, appendix by J. Bourgain, “Sharp thresholds of graph properties, and the k-sat problem,” *J. Amer. Math. Soc.* 12, pp 1017–1054, 1999. 328
11. G. Gasper, M. Rahman, *Basic Hypergeometric Series*, Encyclopedia of Mathematics and its Applications, vol 35, Cambridge University Press, Cambridge, 1990. 329, 334
12. S. Janson, Y. C. Stamatiou, M. Vamvakari, “Bounding the unsatisfiability threshold of random 3-SAT,” *Random Structures and Algorithms*, 17, pp 108–116, 2000. 329, 337
13. L. M. Kirousis, Y. C. Stamatiou, *An inequality for reducible, increasing properties of randomly generated words*, Tech. Rep. TR-96.10.34, C. T. I., University of Patras, Greece, 1996. 333
14. L. M. Kirousis, E. Kranakis, D. Krizanc, Y. C. Stamatiou, “Approximating the unsatisfiability threshold of random formulas,” *Random Structures and Algorithms* 12, pp 253–269, 1998. 329, 330, 332, 333, 334, 336
15. D. E. Knuth, *Fundamental Algorithms*, The Art of Computer Programming, 2nd ed., 1973. 329
16. M. B. Monagan, K. O. Geddes, K. M. Heal, G. Labahn, S. M. Vorkoetter, *Programming Guide*, Maple V Release 5, Springer–Verlag, 1998. 336
17. R. Motwani, P. Raghavan. *Randomized Algorithms.* Cambridge University Press, 1995. 329
18. A. M. Odlyzko, “Asymptotic Enumeration Methods,” in: R. L. Graham, M. Grötschel, and L. Lovász, eds. *Handbook of Combinatorics*, Chapter 22, 1063–1229, Elsevier, 1995. 335
19. W. H. Press, S. A. Teukolsky, W. T. Vetterling, B. P. Flannery, *Numerical Recipes in C: The Art of Scientific Computing*, 2nd ed., Cambridge University Press, 1993. 336, 338
20. A. E. Taylor, W.R Mann, *Advanced Calculus*, 3rd ed., John Wiley & Sons, 1983.
21. F. J. Wright: `http://centaur.maths.qmw.ac.uk/Computer_Algebra/`. A Maple (V-5) implementation of *Downhill Simplex* based on code given in [19]. 336
22. M. Zito, Randomised Techniques in Combinatorial Algorithmics, PhD Thesis, Department of Computer Science, University of Warwick, November 1999.

Relating Partial and Complete Solutions and the Complexity of Computing Smallest Solutions*

André Große[1], Jörg Rothe[2], and Gerd Wechsung[1]

[1] Institut für Informatik, Friedrich-Schiller-Universität Jena
07740 Jena, Germany
{grosse,wechsung}@informatik.uni-jena.de
[2] Math. Institut, Heinrich-Heine-Universität Düsseldorf
40225 Düsseldorf, Germany
rothe@cs.uni-duesseldorf.de

Abstract. We prove that computing a single pair of vertices that are mapped onto each other by an isomorphism ϕ between two isomorphic graphs is as hard as computing ϕ itself. This result optimally improves upon a result of Gál et al. We establish a similar, albeit slightly weaker, result about computing complete Hamiltonian cycles of a graph from partial Hamiltonian cycles. We also show that computing the lexicographically first four-coloring for planar graphs is Δ_2^p-hard. This result optimally improves upon a result of Khuller and Vazirani who prove this problem to be NP-hard, and conclude that it is not self-reducible in the sense of Schnorr, assuming $\mathrm{P} \neq \mathrm{NP}$. We discuss this application to non-self-reducibility and provide a general related result.

Keywords: *partial solutions; complexity of smallest solutions; self-reducibility; graph isomorphisms; Hamiltonian cycles; graph colorability*

1 Introduction

Computational complexity theory and, in particular, the theory of NP-completeness [5] traditionally is concerned with the decision versions of problems. For practical purposes, however, to find or to construct a solution of a given NP problem is much more important than merely to know whether or not a solution exists. For example, computing an isomorphism between two isomorphic graphs (that is, solving the search version of the graph isomorphism problem) is much more important for most applications than merely to know that the graphs are isomorphic. Therefore, much effort has been made in the past to relate the complexity of solving the search problem to the complexity of solving the corresponding decision problem. This property is known as "search reducing to decision," see, e.g., [8] and the references cited therein. The decisive property enabling search to reduce to decision for NP problems such as the graph isomorphism problem is their self-reducibility.

* This work was supported in part by grant NSF-INT-9815095/DAAD-315-PPP-gü-ab. The second author was supported in part by a Heisenberg Fellowship of the Deutsche Forschungsgemeinschaft.

A. Restivo, S. Ronchi Della Rocca, L. Roversi (Eds.): ICTCS 2001, LNCS 2202, pp. 339–356, 2001.

The present paper is concerned with both these properties: how to reduce the search problem and whether or not concrete problems are self-reducible.

Regarding the first property, we build on the recent work of Gál, Halevi, Lipton, and Petrank [4] who studied a property that might be dubbed "complete search reducing to partial search." They showed for various NP problems A that, given an input $x \in A$, computing a small fraction of a solution for x is no easier than computing a complete solution for x. For example, given two isomorphic graphs, computing roughly logarithmically many pairs of vertices that are mapped onto each other by a complete isomorphism ϕ between the graphs is as hard as computing ϕ itself.

As Gál et al. note, their results have two possible interpretations. Positively speaking, their results say that to efficiently solve the complete search problem it is enough to come up with an efficient algorithm for computing only a small part of a solution. Negatively speaking, their results say that constructing even a small part of a solution to instances of hard problems also appears to be a very difficult task. The work of Gál et al. [4] also has consequences with regard to fault-tolerant computing (in particular, for recovering the complete problem solution when parts of it are lost during transmission), and for constructing robust proofs of membership.

As regards the self-reducibility property, we build on the work of Khuller and Vazirani [11] who proved an NP-hardness lower bound for computing the lexicographically first solutions of the planar graph four colorability problem, which we denote by `Pl-4-Color`. It follows from their result that, assuming $\mathrm{P} \neq \mathrm{NP}$, the problem `Pl-4-Color` is not self-reducible in the sense of Schnorr [16,17]. Noting that their result appears to be the first such non-self-reducibility result for problems in P, they proposed as an interesting task to find other problems in P that are not self-reducible under some plausible assumption.

The present paper makes the following contributions. Firstly, we improve the above-mentioned result of Gál et al. [4] by showing that computing even a single pair of vertices that are mapped onto each other by a complete isomorphism ϕ between two isomorphic graphs is as hard as computing ϕ itself. This result is a considerable strengthening of the previous result and an optimal improvement. Interestingly, the self-reducibility of the graph isomorphism problem is the key property that makes our stronger result possible.

Secondly, we obtain a similar, albeit somewhat weaker, result about computing complete Hamiltonian cycles of a given graph from accessing to partial information about the graph's Hamiltonian cycles.

Thirdly, we raise Khuller and Vazirani's NP-hardness lower bound for computing the lexicographically smallest four-coloring for planar graphs to Δ_2^p-hardness. Our result is optimal, since this problem belongs to (the function analog of) the class Δ_2^p.

$\Delta_2^p = \mathrm{P}^{\mathrm{NP}}$ is the class of problems solvable in deterministic polynomial time with an NP oracle. Papadimitriou [15] proved that `Unique-Travaling-Optimal-Salesperson` is Δ_2^p-complete, and Krentel [13] and Wagner [21] established many more Δ_2^p-completeness results, including the result that the problem `Odd-Max-SAT` is Δ_2^p-complete.

As mentioned above, if for some problem in P computing the lexicographically smallest solution is hard, then the problem itself cannot be self-reducible in the sense of Schnorr [16,17], unless P = NP. We discuss this application to non-self-reducibility and provide a general related result. In particular, it follows from this result that even a set as simple as Σ^* has representations in which it is not self-reducible in Schnorr's sense, unless P = NP.

2 Computing Complete Graph Isomorphisms from Partial Ones

Gál et al. [4] prove the following result. Suppose there exists a function oracle f that, given any two isomorphic graphs with m vertices each, outputs a part of an isomorphism between the graphs consisting of at least $(3 + \epsilon) \log m$ vertices for some constant $\epsilon > 0$. Then, using the oracle f, one can compute a complete isomorphism between any two isomorphic graphs in polynomial time.

We improve their result by showing the same consequence under the weakest assumption possible: Assuming that we are given a function oracle that provides *only one vertex pair* belonging to an isomorphism between two given isomorphic graphs, one can use this oracle to compute complete isomorphisms between two isomorphic graphs in polynomial time. Thus, our improvement of the previous result by Gál et al. [4] is optimal.

Definition 1. *For any graph G, the vertex set of G is denoted by $V(G)$, and the edge set of G is denoted by $E(G)$.*

Let G and H be undirected and simple graphs, i.e., graphs with no reflexive and multiple edges.

An isomorphism *between G and H is a bijective mapping ϕ from $V(G)$ onto $V(H)$ such that, for all $x, y \in V(G)$,*

$$\{x, y\} \in E(G) \iff \{\phi(x), \phi(y)\} \in E(H).$$

Let $\mathrm{ISO}(G, H)$ *denote the set of isomorphisms between G and H.*

We now state our main result.

Theorem 1. *Suppose there exists a function oracle f that, given any two isomorphic graphs $\hat{G}$ and $\hat{H}$, outputs two vertices $x \in V(\hat{G})$ and $y \in V(\hat{H})$ with $\hat{\phi}(x) = y$, for some isomorphism $\hat{\phi}$ from* $\mathrm{ISO}(\hat{G}, \hat{H})$.

Then, there is a recursive procedure g that, given any two isomorphic graphs G and H, uses the oracle f to construct a complete isomorphism $\phi \in$ $\mathrm{ISO}(G, H)$ *in polynomial time.*

Before proving Theorem 1, we explain the main difference between our proof and the proof of Gál et al. [4]. Crucially, to make their recursive procedure terminate, they ensure in their construction that the (pairs of) graphs they construct are of strictly decreasing size in each loop of the procedure. In contrast, for our algorithm this strong requirement is not necessary to make the procedure terminate.

Let us informally explain why. Our algorithm is inspired by the known self-reducibility algorithm for the graph isomorphism problem. A self-reduction for a problem A is a computational procedure for solving A, where the set A itself may be accessed as an oracle. To prevent this notion from being trivialized, one requires that A cannot be queried about the given input itself; usually, only queries about strings that are "smaller" than the input string are allowed. When formally defining what precisely is meant by "smaller," most self-reducibility notions—including that of Schnorr [16,17], see Definition 5—employ the useful concepts of "polynomially well-founded" and "length-bounded" partial orders, rather than being based simply on the lengths of strings. This approach is useful in order to "obtain full generality and to preserve the concept under polynomially computable isomorphisms" [10, p. 84], see also [14,18]. That means that the strings queried in a self-reduction may be *larger in length* than the input strings as long as they are *predecessors in a polynomially well-founded and length-bounded partial order*. It is this key property that makes our algorithm terminate without having to ensure in the construction that the (pairs of) graphs constructed are of strictly decreasing size in each loop.

Here is an intuitive description of how our algorithm works. Let G and H be the given isomorphic graphs. The function oracle will be invoked in each loop of the procedure to yield any one pair of vertices that are mapped onto each other by some isomorphism between the graphs as yet constructed. However, if we were simply deleting this vertex pair, we would obtain new graphs $\hat{G}$ and $\hat{H}$ such that $\mathrm{ISO}(\hat{G}, \hat{H})$ might contain some isomorphism not compatible with $\mathrm{ISO}(G, H)$, which means it cannot be extended to an isomorphism in $\mathrm{ISO}(G, H)$. That is why our algorithm will attach cliques of appropriate sizes to each vertex to be deleted, and the deletion of this vertex, and of the clique attached to it, will be delayed until some subsequent loop of the procedure. That is, the (pairs of) graphs we construct may increase in size in some of the loops, and yet the procedure is guaranteed to terminate in polynomial time.

We now turn to the formal proof.

Proof of Theorem 1. Let G and H be two given isomorphic graphs with n vertices each. Let f be a function oracle as in the theorem. We describe the recursive procedure g that computes an isomorphism $\phi \in \mathrm{ISO}(G, H)$. Below, we use variables $\hat{G}$ and $\hat{H}$ to denote (encodings of) graphs obtained from G and H according to g, and we refer to the vertices of G and H as the *old* vertices and to the vertices of $\hat{G} - G$ and $\hat{H} - H$ as the *new* vertices.

On input $\langle G, H \rangle$, the algorithm g executes the following steps:

1. Let $\hat{G} = G$ and $\hat{H} = H$, and set i to $n = ||V(G)||$. Let $\phi \subseteq V(G) \times V(H)$ be a set variable that, eventually, gives the isomorphism between G and H to be constructed. Initially, set ϕ to the empty set.

2. Query f about the pair $(\hat{G}, \hat{H})$. Let (x, y) be the vertex pair returned by $f(\hat{G}, \hat{H})$, where $x \in V(\hat{G})$ and $y \in V(\hat{H})$ and $\hat{\phi}(x) = y$ for some isomorphism $\hat{\phi} \in \mathrm{ISO}(\hat{G}, \hat{H})$.
3. Consider the following two cases:
   Case 3.1: $x \in V(G)$ is an old vertex.
   We distinguish the following two cases:
   (a) Set ϕ to $\phi \cup \{(x, y)\}$. Modify the graphs $\hat{G}$ and $\hat{H}$ as follows.
   Delete x, all new neighbors of x, and all edges incident to either of these vertices from $\hat{G}$. Attach to each old neighbor $x' \in V(G)$ of x a copy of a clique $C_{i,x'}$ consisting of $i - 1$ new vertices each of which is connected with x' by an edge; hence, the graph induced by $V(C_{i,x'}) \cup \{x'\}$ forms an i-clique. Make sure that all the new clique vertices are pairwise disjoint and disjoint with (the old) graph $\hat{G}$. Call the resulting graph (the new) $\hat{G}$.
   Modify $\hat{H}$ in the same way: Delete y and all new neighbors of y from $\hat{H}$, and extend each old neighbor $y' \in V(H)$ of y to a clique consisting of the i vertices $V(C_{i,y'}) \cup \{y'\}$.
   (b) Let $\tilde{y} \in V(H)$ be the unique old vertex adjacent to y, i.e., y is a member of the clique $C_{j,\tilde{y}}$ that was previously attached to $\tilde{y}$ in the $(j - n + 1)$th loop, where $n \leq j < i$. Note that the size of the clique $C_{j,\tilde{y}} \cup \{\tilde{y}\}$ equals j. Since $\hat{\phi}(x) = y$, the old vertex x must belong to the clique $C_{j,x} \cup \{x\}$ of size j and, thus, cannot have any old neighbors in $\hat{G}$. It follows that $\tilde{y}$ is also not adjacent to any old vertex in the current graph $\hat{H}$. That is, both the clique $C_{j,x} \cup \{x\}$ and the clique $C_{j,\tilde{y}} \cup \{\tilde{y}\}$ are connecting components of their graphs $\hat{G}$ and $\hat{H}$, respectively.
   Set ϕ to $\phi \cup \{(x, \tilde{y})\}$. Modify the graphs $\hat{G}$ and $\hat{H}$ by deleting the cliques $C_{j,x} \cup \{x\}$ and $C_{j,\tilde{y}} \cup \{\tilde{y}\}$.
   Set i to $i + 1$.
   Case 3.2: $x \notin V(G)$ is a new vertex in $\hat{G}$.
   It follows that x is a member of a clique $C_{j,\tilde{x}}$, where $n \leq j < i$, that was previously attached to some old vertex $\tilde{x} \in V(G)$ in the $(j-n+1)$th loop. Also, by construction, $\tilde{x}$ is the only old vertex adjacent to x. Similarly, it holds that y is a member of a clique $C_{j,\tilde{y}} \cup \{\tilde{y}\}$ in $\hat{H}$ with a uniquely determined old vertex $\tilde{y} \in V(H)$.
   If $y = \tilde{y}$, then this case reduces to Case 3.1(a), with x being replaced by $\tilde{x}$.
   If $y \neq \tilde{y}$, then $\hat{\phi}(x) = y$ implies that $\hat{\phi}(\tilde{x}) = \tilde{y}$ and, thus, that $\tilde{x}$ and $\tilde{y}$ have the same number of old neighbors. Hence, this case also reduces to Case 3.1(a), with x being replaced by $\tilde{x}$ and y being replaced by $\tilde{y}$.
4. If there are no vertices left in $\hat{G}$ and $\hat{H}$, output ϕ, which gives a complete isomorphism between G and H. Otherwise, go to Step 2.

As alluded to in the above informal description of the algorithm, the intuition behind introducing cliques of increasing sizes in the construction is to keep the isomorphisms $\hat{\phi} \in \mathrm{ISO}(\hat{G}, \hat{H})$ compatible with $\phi \in \mathrm{ISO}(G, H)$ when vertices from G and H are deleted. That is, we want to preclude the case that deleting $x \in V(G)$ and $y \in V(H)$ results in reduced graphs $\hat{G}$ and $\hat{H}$ such that there is some $\hat{\phi} \in \mathrm{ISO}(\hat{G}, \hat{H})$—and our oracle f might pick some vertex pair corresponding to such a $\hat{\phi}$—that cannot be extended to $\phi \in \mathrm{ISO}(G, H)$.

The following example illustrates this intuition and shows how the algorithm works.

Example 1. Fig. 1 gives an example of a pair of isomorphic graphs G and H with $\mathrm{ISO}(G, H) = \{\phi_1, \phi_2\}$, where

$$\phi_1 = \begin{pmatrix} 1\,2\,3\,4\,5 \\ 1\,5\,4\,3\,2 \end{pmatrix} \text{ and } \phi_2 = \begin{pmatrix} 1\,2\,3\,4\,5 \\ 5\,1\,4\,3\,2 \end{pmatrix}.$$

Suppose that the function oracle f, when queried about the pair (G, H), returns, e.g., the vertex pair $(5, 2)$. If we were simply deleting the vertex 5 from G and the vertex 2 from H, then we would obtain graphs $\widehat{G}$ and $\widehat{H}$ such that $\mathrm{ISO}(\widehat{G}, \widehat{H})$ contains six isomorphisms only two of which are compatible with the pair $(5, 2)$; see Fig. 2. But then f, when queried about $(\widehat{G}, \widehat{H})$, might pick, e.g., the vertex pair $(4, 5)$, which belongs neither to ϕ_1 nor to ϕ_2.

To preclude cases like this, our algorithm attaches cliques of size 5 to the vertex 4 in G and to the vertex 3 in H; see Fig. 3. Old vertices are represented by full circles and new vertices by empty circles. Note that each $\phi \in \mathrm{ISO}(G_1, H_1)$ is compatible with the vertex pair $(5, 2)$ from $\phi_1, \phi_2 \in \mathrm{ISO}(G, H)$.

Fig. 3 through Fig. 6 show how g, on input (G, H), continues to work for a specific sequence of oracle answers from f. In Fig. 6, the only old vertex left in G_4 is the vertex 4, and the only old vertex left in H_4 is the vertex 3. Hence, whichever vertex pair f when queried about (G_4, H_4) picks, g maps vertex 4 in G_4 to vertex 3 in H_4, which completes the isomorphism

$$\phi_2 = \begin{pmatrix} 1\,2\,3\,4\,5 \\ 5\,1\,4\,3\,2 \end{pmatrix}$$

that is in $\mathrm{ISO}(G, H)$. Finally, both G_4 and H_4 are deleted, and the algorithm terminates. ▮ *End of Example 1*

To prove the correctness of the algorithm, we argue that:

(a) each pair $\langle \hat{G}, \hat{H} \rangle$ constructed in any loop of g is a pair of isomorphic graphs—hence, f can legally be called in each loop of g; and

(b) the mapping ϕ computed by g on input $\langle G, H \rangle$ is in $\mathrm{ISO}(G, H)$.

Proof of (a): This assertion follows immediately from the construction and the assumption that G and H are isomorphic.

Proof of (b): The first call to f yields a valid initial segment (x_1, y_1) of an isomorphism between G and H, since f is queried about the unmodified graphs G and H.

Let $\phi_i = \{(x_1, y_1), (x_2, y_2), \ldots, (x_i, y_i)\}$ be the initial segment of ϕ that consists of i vertex pairs for some i, $1 \leq i \leq n$, where (x_i, y_i) is the pair added in

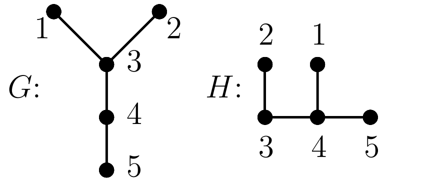


Fig. 1. Two graphs G and H with $\mathrm{ISO}(G,H) = \{\phi_1, \phi_2\}$

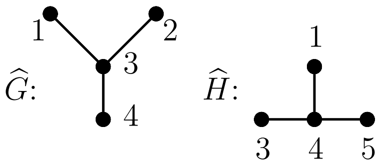


Fig. 2. Two graphs $\widehat{G}$ and $\widehat{H}$ for which $\mathrm{ISO}(\widehat{G},\widehat{H})$ contains isomorphisms not compatible with $(5,2)$

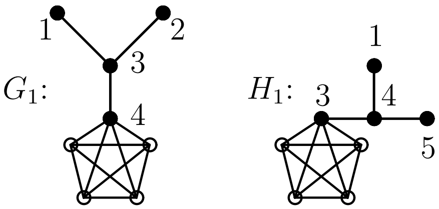


Fig. 3. Two graphs G_1 and H_1 obtained from G and H according to g when $f(G,H)$ returns $(5,2)$

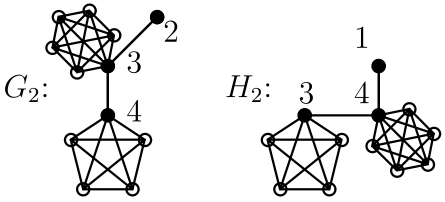


Fig. 4. Two graphs G_2 and H_2 that result from $f(G_1,H_1)$ returning $(1,5)$

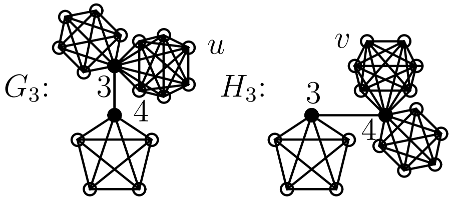


Fig. 5. Two graphs G_3 and H_3 that result from $f(G_2,H_2)$ returning $(2,1)$

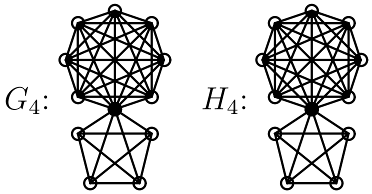


Fig. 6. Two graphs G_4 and H_4 that result from $f(G_3,H_3)$ returning (u,v)

the i loop of g. Let G_i and H_i be the graphs constructed from G and H when loop i is entered; for example, $G_1 = G$ and $H_1 = H$. Fix some i with $1 < i \leq n$. We show that the extension ϕ_i of ϕ_{i-1} (obtained by adding the pair (x_i, y_i) in the ith loop of g) is compatible with ϕ_{i-1}. That is, for each $(x_j, y_j) \in \phi_{i-1}$, it holds that

$$\{x_i, x_j\} \in E(G) \text{ if and only if } \{y_i, y_j\} \in E(H).$$

Assume $\{x_i, x_j\} \in E(G)$. In loop $j < i$, all neighbors of x_j, including x_i, and all neighbors of y_j were extended to a clique of size $n + j - 1$. Note that, in each loop of g, the clique sizes are increased by one, each clique contains exactly one old vertex, and any two cliques in G_i (respectively, in H_i) can overlap only by having their unique old vertex in common. It follows that any isomorphism between G_i and H_i must map cliques of size $n + j - 1$ in G_i onto cliques of size $n + j - 1$ in H_i. Since y_i is chosen in loop i of g, it follows from our construction that the clique C_{n+j-1,x_i} in G_i was mapped onto the clique C_{n+j-1,y_i} in H_i. Hence, y_i is a neighbor of y_j in H, i.e., $\{y_i, y_j\} \in E(H)$.

The converse implication ($\{y_i, y_j\} \in E(H) \implies \{x_i, x_j\} \in E(G)$) follows by a symmetric argument.

Finally, we estimate the time complexity of the algorithm g. Since in each loop of g, a pair of old vertices from $V(G) \times V(H)$ is deleted from the graphs and is added to the isomorphism $\phi \in \mathrm{ISO}(G, H)$, the algorithm terminates after n loops. Within each loop, g makes one oracle call to f, updates ϕ, and modifies the current graphs $\hat{G}$ and $\hat{H}$ by deleting certain vertices and by adding at most $2(n - 1)$ cliques of size at most $2n - 1$. Hence, g runs in cubic time. ∎

3 Computing Complete Hamiltonian Cycles from Partial Ones

Now we turn to the problem of computing complete Hamiltonian cycles in a graph from partial ones. Our construction is easier to describe for multigraphs, i.e., graphs with reflexive and multiple edges allowed. We may do so, as for Hamiltonian cycles it does not matter whether simple graphs or multigraphs are used. We also assume that all graphs are connected.

Let us informally describe how our procedure works. As in the preceding section, suppose we have a function oracle f that, given any multigraph G that contains a Hamiltonian cycle, returns an edge e that is part of a Hamiltonian cycle c of G. We want to reduce G by deleting e and identifying the two vertices incident to e, and then want to recursively apply f to this reduced graph, call it $\hat{G}$. However, this approach would destroy important information about e, namely the "left" and the "right" context of e in G. Thus, in the next recursion loop, the oracle might return an edge contained in a Hamiltonian cycle $\hat{c}$ of $\hat{G}$ that is not compatible with the previously chosen edge e, which means that adding e back to $\hat{G}$ does not necessarily imply that $\hat{c}$ can be extended to a Hamiltonian cycle of G. To preclude cases like this, we require our oracle to return only edges contained in Hamiltonian cycles that are compatible with the left-right-context of the edges previously chosen. This additional requirement regarding f makes Theorem 2 somewhat weaker than Theorem 1.

First, we define what we mean by a left-right-context of (the edges of) G, and what we mean by Hamiltonian cycles being compatible (or consistent) with a left-right-context of G.

Definition 2. *Let $G = (V, E)$ be an undirected multigraph with n vertices.*

- *A* Hamiltonian cycle *of G is a sequence $(v_1, v_2, \dots, v_n)$ of pairwise distinct vertices from V such that $\{v_n, v_1\} \in E$ and $\{v_i, v_{i+1}\} \in E$ for each i with $1 \leq i \leq n-1$.*
- *For any set S, let $\mathfrak{P}(S)$ denote the power set of S. For any $v \in V$, let $E(v)$ denote the set of edges in E incident to v.*
  A left-right-context of *G is a function $\pi : V \to \mathfrak{P}(E) \times \mathfrak{P}(E)$ satisfying that, for every $v \in \mathrm{domain}(\pi)$, there exist sets $L(v)$ and $R(v)$ such that*
  1. *$\pi(v) = (L(v), R(v))$,*
  2. *$L(v) \cup R(v) \subseteq E(v)$, and*
  3. *$L(v) \cap R(v) = \emptyset$.*
- *We say that a Hamiltonian cycle c of G is* consistent with a left-right-context π of G *if and only if for every $v \in \mathrm{domain}(\pi)$, c contains exactly one edge from $L(v)$ and exactly one edge from $R(v)$, where $\pi(v) = (L(v), R(v))$.*

We now state our result.

Theorem 2. *Let $\hat{G}$ be any multigraph, and let π be any left-right-context of $\hat{G}$. Suppose there exists a function oracle f that, given $(\hat{G}, \pi)$, outputs some edge $e \in E(\hat{G})$ such that some Hamiltonian cycle consistent with π contains e (provided $\hat{G}$ has a Hamiltonian cycle consistent with π).*

Then, there is a recursive procedure g that, given any multigraph G that has a Hamiltonian cycle, uses the oracle f to construct a complete Hamiltonian cycle of G in polynomial time.

Proof. Let G be any multigraph with n vertices that contains a Hamiltonian cycle. Let f be a function oracle as in the theorem.

In the procedure described below, whenever we identify two vertices u and v, deleting the edge(s) connecting u and v, we assume by convention that in the resulting graph the vertex $u = v$ has two name tags, namely u and v. This convention simplifies the description of our construction and does no harm.

We now describe the procedure g on input G:

Step 0: Let $G_0 = (V_0, E_0)$ be the given multigraph G, and let π_0 be the nowhere defined function (on the domain V_0). Set C to the empty set. Note that C will, eventually, contain the complete Hamiltonian cycle of G to be constructed.

Step i, $1 \leq i \leq n-1$: Let $G_{i-1} = (V_{i-1}, E_{i-1})$ be the multigraph and let π_{i-1} be the left-right-context of G_{i-1} constructed in the previous step. Compute the edge $e_i = f(G_{i-1}, \pi_{i-1})$ by querying the oracle, and add e_i to C. Let $e_i = \{u_i, v_i\}$. Consider the following three cases.

Case 1: $e_i \cap \mathrm{domain}(\pi_{i-1}) = \emptyset$.

Cancel e_i from G_{i-1}, and identify the vertices u_i and v_i. Call the resulting graph $G_i = (V_i, E_i)$. Define the left-right-context $\pi_i : V_i \to \mathfrak{P}(E_i) \times \mathfrak{P}(E_i)$ by $\mathrm{domain}(\pi_i) = \mathrm{domain}(\pi_{i-1}) \cup \{u_i\}$ and

$$\pi_i(v) = \begin{cases} \pi_{i-1}(v) & \text{if } v \in \mathrm{domain}(\pi_{i-1}) \\ (L_i(u_i), R_i(u_i)) & \text{if } v = u_i, \end{cases}$$

where

- $L_i(u_i) = E_{i-1}(u_i) - \{e_i\}$ and
- $R_i(u_i) = \{\{u_i, z\} \mid \{v_i, z\} \in E_{i-1} \ \wedge \ z \neq u_i\}$.

Case 2: $e_i \cap \mathrm{domain}(\pi_{i-1}) = \{x\}$ for some vertex $x \in V_{i-1}$.

By our assumption that f returns only edges consistent with the given left-right-context, e_i must belong to exactly one of $L_{i-1}(x)$ or $R_{i-1}(x)$. Assume $x = v_i$ and $e_i \in L_{i-1}(x)$; the other cases—such as the case "$x = u_i$ and $e_i \in R_{i-1}(x)$"—can be treated analogously.

Cancel e_i from G_{i-1}, and identify the vertices u_i and v_i, which equals x. Call the resulting graph $G_i = (V_i, E_i)$. Define the left-right-context $\pi_i : V_i \to \mathfrak{P}(E_i) \times \mathfrak{P}(E_i)$ by $\mathrm{domain}(\pi_i) = \mathrm{domain}(\pi_{i-1})$ and

$$\pi_i(v) = \begin{cases} \pi_{i-1}(v) & \text{if } v \neq x \\ (L_i(x), R_i(x)) & \text{if } v = x, \end{cases}$$

where

- $L_i(x) = \{\{x, z\} \mid \{u_i, z\} \in E_{i-1} \ \wedge \ z \neq v_i\}$ and
- $R_i(x) = R_{i-1}(x)$.

Case 3: $e_i \cap \mathrm{domain}(\pi_{i-1}) = \{x, y\}$ for two vertices $x, y \in V_{i-1}$ with $x \neq y$.

It follows that $e_i = \{x, y\}$ in this case. By our assumption that f returns only edges consistent with the given left-right-context, e_i must belong to exactly one of $L_{i-1}(z)$ or $R_{i-1}(z)$, for both $z = x$ and $z = y$. Assume $e_i \in L_{i-1}(x) \cap R_{i-1}(y)$; the other cases can be treated analogously.

Cancel e_i from G_{i-1}, and identify the vertices x and y. Call the resulting graph $G_i = (V_i, E_i)$. Define the left-right-context $\pi_i : V_i \to \mathfrak{P}(E_i) \times \mathfrak{P}(E_i)$ by $\mathrm{domain}(\pi_i) = \mathrm{domain}(\pi_{i-1})$ and

$$\pi_i(v) = \begin{cases} \pi_{i-1}(v) & \text{if } v \neq x = y \\ (L_i(y), R_i(y)) & \text{if } v = x = y, \end{cases}$$

where

- $L_i(y) = L_{i-1}(y)$ and
- $R_i(y) = \{\{y, z\} \mid \{x, z\} \in R_{i-1}(x)\}$.

Step *n*: Since in each of the $n - 1$ previous steps two vertices have been identified and one edge has been added to C, the graph G_{n-1} constructed in the previous step contains only one vertex, say z, having possibly multiple reflexive edges. Also, C contains $n - 1$ elements, and π_{n-1} is either of the form

- $\pi_{n-1} = (\emptyset, R_{n-1}(z))$ or

- $\pi_{n-1} = (L_{n-1}(z), \emptyset)$,

  where any edge in $R_{n-1}(z)$ (respectively, in $L_{n-1}(z)$) can be used to complete the Hamiltonian cycle constructed so far. Thus, we may choose any one edge from $R_{n-1}(z)$ (respectively, from $L_{n-1}(z)$) and add it to C.

This concludes the description of the procedure g. Note that g runs in polynomial time. To prove the correctness of the algorithm, note that, for each $i \in \{1, 2, n\text{–}2\}$, and for each Hamiltonian cycle c of G_i consistent with π_i, it holds that inserting the edge e_i into c yields a Hamiltonian cycle of G_{i-1}, thus ensuring consistency of the overall construction. ∎

4 Self-Reducibility and the Hardness of Computing Smallest Solutions

4.1 Computing the Smallest Four-Coloring is Δ_2^p-Hard

The previous two sections studied the property of "complete search reducing to partial search" for two standard NP problems that, unless P = NP, are computationally hard. The present section shows that even for efficiently solvable problems it can be computationally hard to compute their smallest solutions. As noted by Khuller and Vazirani [11], unless P = NP, such hardness results indicate the *non*-self-reducibility of such problems.

The problem of deciding whether a planar graph can be colored with four colors is well-known to be efficiently solvable, see [1,2]. We show that computing the lexicographically first k-coloring for planar graphs is Δ_2^p-hard for any $k \geq 4$. Since the lexicographically smallest k-coloring of planar graphs certainly can be computed in (the function analog of) Δ_2^p, this result optimally improves upon the previous NP-hardness lower bound for this problem established by Khuller and Vazirani [11].

Definition 3. *Let $k > 1$. A* k-coloring *of a graph $G = (V, E)$ is a mapping $\psi_G : V \rightarrow \{0, 1, \ldots, k-1\}$, where $\{0, 1, \ldots, k-1\}$ represents the set of colors. A k-coloring ψ_G is said to be* legal *if for every edge $\{u, v\} \in E$ it holds that $\psi_G(u) \neq \psi_G(v)$. A graph G is said to be* k-colorable *if there exists a legal k-coloring of G.*

Let `Pl-`k`-Color` denote the planar graph k-colorability problem. `Pl-3-Color` was shown to be NP-complete by Stockmeyer [19], see also [6]. Solving the famous Four Color Conjecture in the affirmative, Appel and Haken [1,2] showed that every planar graph is four-colorable; hence, `Pl-`k`-Color` $\in$ P for $k \geq 4$.

Definition 4 (Khuller and Vazirani [11]).
Let $k > 1$, and let the vertex set of a given graph $G = (V, E)$ be ordered as $V = \{v_1, v_2, \ldots, v_n\}$. Then, every k-coloring ψ_G of G may be represented by the string

$$\psi_G = \psi_G(v_1)\psi_G(v_2)\cdots\psi_G(v_n)$$

from $\{0,1,\ldots,k-1\}^n$.

The lexicographically smallest (legal) k-coloring *of a planar graph G with n vertices is defined by*

$$\mathrm{LF}_{\texttt{Pl-}k\texttt{-Color}}(G) = \begin{cases} \min\left\{\psi_G \,\middle|\, \begin{array}{l}\psi_G \textit{ is a legal}\\ k\textit{-coloring of } G\end{array}\right\} & \textit{if } G \in \texttt{Pl-}k\texttt{-Color}\\ 10^n & \textit{otherwise,}\end{cases}$$

where the minimum is taken with respect to the lexicographic ordering of strings.

Theorem 3. *Computing the lexicographically smallest k-coloring for planar graphs is Δ_2^p-hard for any $k \geq 4$.*

Proof. For simplicity, we show this claim only for $k = 4$. Let ρ_4 be the reduction of Khuller and Vazirani [11, Theorem 3.1]. Recall that ρ_4 maps a given planar graph $G = (V, E)$ whose vertices are ordered as $V = \{v_1, v_2, \ldots, v_m\}$ to the planar graph $H = (U, F)$ defined as follows:

- The vertex set of H is ordered as $U = \{u_1, u_2, \ldots, u_{2m}\}$, where u_i is a new vertex and $u_{m+i} = v_i$ is an old vertex for each i, $1 \leq i \leq m$.
- The edge set of H is defined by $F = E \cup \{\{u_i, u_{m+i}\} \mid 1 \leq i \leq m\}$.

It follows immediately from this construction that

$$(1) \qquad G \in \texttt{Pl-3-Color} \iff \mathrm{LF}_{\texttt{Pl-4-Color}}(\rho_4(G)) \in \{0^m w \mid w \in \{1,2,3\}^m\},$$

that is, "$G \in$ Pl-3-Color?" can be decided by looking at the first m bits of $\mathrm{LF}_{\texttt{Pl-4-Color}}(H)$.

Now, we give a reduction from the problem Odd-Min-SAT, which is defined to be the set of all boolean formulas $F = F(x_1, x_2, \ldots, x_n)$ in conjunctive normal form for which, assuming F *is* satisfiable, the lexicographically smallest satisfying assignment $\alpha : \{x_1, x_2, \ldots, x_n\} \to \{1, 2\}$ is "odd," i.e., for which $\alpha(x_n) = 1$. Here, "1" represents "true" and "2" represents "false." It is well-known that Odd-Min-SAT is Δ_2^p-complete; Krentel [13] and also Wagner [21] proved the corresponding claim for the dual problem Odd-Max-SAT.

Let $F = F(x_1, x_2, \ldots, x_n)$ be any given boolean formula, where without loss of generality we may assume that F is in conjunctive normal form with exactly 3 literals per clause. Assume that F has z clauses. Let σ be the Stockmeyer reduction from 3-SAT to Pl-3-Color, see [19] and also [6]. This reduction σ, on input F, yields a graph $G = (V, E)$ with $m > n$ vertices, where $m = m(F)$ depends on the number n of variables, the number z of clauses, and the structure of F. Note that F's structure induces a certain number of "crossovers" of edges to guarantee the planarity of G; see [6,19] for details.

Now we order the vertex set of G as $V = \{v_1, v_2, \ldots, v_m\}$ such that

(a) for each i, $1 \leq i \leq n$, v_i represents the variable x_i; and
(b) for each i, $n < i \leq m$, v_i represents any other vertex of G.

Note that G is a planar graph such that

(i) F is satisfiable if and only if G is 3-colorable, using the colors 1,2, and 3, and

(ii) each satisfying assignment α of F corresponds to a 3-coloring ψ_α of G such that $\psi_\alpha(v_i) = \alpha(v_i) \in \{1, 2\}$, $1 \leq i \leq n$, and the color 3 is used for the other vertices of G.

Now apply the reduction ρ_4 of Khuller and Vazirani to G and obtain a planar graph $H = \rho_4(G) = \rho_4(\sigma(F))$ that satisfies Equation (1) as described above. It follows immediately from this construction and from Equation (1) that

$$F \in \texttt{Odd-Min-SAT} \iff$$
$$\mathrm{LF}_{\texttt{Pl-4-Color}}(\rho_4(\sigma(F))) \in \{0^m w 1 y \mid w \in \{1,2\}^{n-1} \ \wedge\ y \in \{1,2,3\}^{m-n}\},$$

that is, "$F \in \texttt{Odd-Min-SAT}$?" can be decided by looking at the first m bits and at the $(m+n)$th bit of $\mathrm{LF}_{\texttt{Pl-4-Color}}(H)$.

For $k > 4$, the claim of the theorem follows from an analogous argument that employs in place of ρ_4 the appropriate reduction ρ_k from [11, Thm. 3.2]. ■

4.2 Problems with Hard Smallest Solutions Yielding Non-self-reducible Sets in P

From their NP-hardness lower bound for computing the lexicographically first four-coloring of planar graphs, Khuller and Vazirani [11] conclude that $\texttt{Pl-}k\texttt{-Color}$, $k \geq 4$, is not self-reducible, unless P = NP. The type of (functional) self-reducibility used by Khuller and Vazirani in [11] is due to Schnorr [16,17], see also [3].

Definition 5 (Schnorr [16,17]).

- *Let Σ and Γ be alphabets with at least two symbols; instances of problems are encoded over Σ and solutions of problems are encoded over Γ. For any set $B \subseteq \Sigma^* \times \Gamma^*$ and any polynomial p, the* p-projection *of B is defined to be*

$$\mathrm{proj}_p(B) = \{x \in \Sigma^* \mid (\exists y \in \Gamma^*)\, [|y| \leq p(|x|) \ \wedge\ (x,y) \in B]\}.$$

- *A partial order $\leq$ on Σ^* is* polynomially well-founded and length-bounded *if and only if there exists a polynomial q such that*
  (a) *every $\leq$-decreasing chain with maximum element x has at most $q(|x|)$ elements; and*
  (b) *for all $x, y \in \Sigma^*$, $x < y$ implies $|x| \leq q(|y|)$.*
- *Let $A = \mathrm{proj}_p(B)$ for some set $B \subseteq \Sigma^* \times \Gamma^*$ and some polynomial p. The projection A is said to be* self-reducible with respect to (B, p) *if and only if there exist a polynomial-time computable function g mapping from $\Sigma^* \times \Gamma$ to Σ^* and a polynomially well-founded and length-bounded partial order $\leq$ such that for all $x \in \Sigma^*$, for all $y \in \Gamma^*$, and for all $\gamma \in \Gamma$, it holds that*
  (i) *$g(x, \gamma) < x$, and*

(ii) $(x, \gamma y) \in B \iff (g(x, \gamma), y) \in B$.
If the pair (B, p), for which $A = \mathrm{proj}_p(B)$, is clear from the context, we omit the phrase "with respect to (B, p)."

We mention in passing that various other important types of self-reducibility have been studied, such as the self-reducibility defined by Meyer and Paterson [14] and the disjunctive self-reducibility studied by Selman [18], Ko [12], and many others. We refer the reader to the excellent survey by Joseph and Young [10] for an overview and for pointers to the literature. Note that, in sharp contrast with Schnorr's self-reducibility, every set in P is self-reducible in the sense of Meyer and Paterson [14], Ko [12], and Selman [18].

Definition 6. *Let $\Sigma = \{0, 1\}$. Given any set $A \subseteq \Sigma^*$ in* NP*, there is an associated set $B_A \subseteq \Sigma^* \times \Sigma^*$ in* P *and an associated polynomial p_A such that $A = \mathrm{proj}_{p_A}(B_A)$.*

- *For any $x \in \Sigma^*$, define the* set of solutions for x with respect to B_A and p_A *by*
$$\mathrm{Sol}_{(B_A,p_A)}(x) = \{y \in \Sigma^* \mid |y| \leq p_A(|x|) \ \wedge \ (x, y) \in B_A\}.$$
Note that $x \in A$ if and only if $\mathrm{Sol}_{(B_A,p_A)}(x) \neq \emptyset$.
- *For any $x \in \Sigma^*$, define the* lexicographically first solution with respect to B_A and p_A *by*
$$\mathrm{LF}_{(B_A,p_A)}(x) = \begin{cases} \min \mathrm{Sol}_{(B_A,p_A)}(x) & \text{if } x \in A \\ \mathrm{bin}(2^{p(|x|)}) & \text{otherwise,} \end{cases}$$
where the minimum is taken with respect to the lexicographic ordering of Σ^*, and $\mathrm{bin}(n)$ *denotes the binary representation of the integer n without leading zeroes.*
If the pair (B_A, p_A), for which $A = \mathrm{proj}_{p_A}(B_A)$, is clear from the context, we use $\mathrm{Sol}_A(x)$ and $\mathrm{LF}_A(x)$ as shorthands for respectively $\mathrm{Sol}_{(B_A,p_A)}(x)$ and $\mathrm{LF}_{(B_A,p_A)}(x)$.

It is well-known that if A is self-reducible then LF_A can be computed in polynomial time by prefix search, via suitable queries to the oracle A. Moreover, if A is in P then LF_A can even be computed in polynomial time without any oracle queries. It follows that if computing LF_A is NP-hard then A cannot be self-reducible, assuming P $\neq$ NP.

Khuller and Vazirani [11] propose to prove P problems other than `Pl-4-Color` to be non-self-reducible, assuming P $\neq$ NP. As Theorem 4 below, we provide a general result showing that it is almost trivial to find such problems: For any NP problem A for which LF_A is hard to compute, one can define a P-decidable version D of A such that LF_D is still hard to compute; hence, D is not self-reducible, assuming P $\neq$ NP.

To formulate this result, we now define the functional many-one reducibility that was introduced by Vollmer [20] as a stricter reducibility notion than Krentel's metric reducibility [13]. We also define the function class min $\cdot$P that was introduced by Hempel and Wechsung [9].

Definition 7. *Let f and h be functions from Σ^* to Σ^*.*

- [20] *We say f is* polynomial-time functionally many-one reducible *to h (in symbols, $f \leq_{\mathrm{m}}^{\mathrm{FP}} h$) if and only if there exists a polynomial-time computable function g such that for all $x \in \Sigma^*$, $f(x) = h(g(x))$.*
- *We say h is* $\leq_{\mathrm{m}}^{\mathrm{FP}}$-hard *for a function class $\mathcal{C}$ if and only if for every $f \in \mathcal{C}$, $f \leq_{\mathrm{m}}^{\mathrm{FP}} h$.*
- *We say h is* $\leq_{\mathrm{m}}^{\mathrm{FP}}$-complete *for $\mathcal{C}$ if and only if $h \in \mathcal{C}$ and h is $\leq_{\mathrm{m}}^{\mathrm{FP}}$-hard.*
- [9] *Define the class* $\min \cdot \mathrm{P}$ *to consist of all functions f for which there exist a set $A \in \mathrm{P}$ and a polynomial p such that for all $x \in \Sigma^*$,*

$$f(x) = \min\{y \in \{0,1\}^* \mid |y| \leq p(|x|) \ \wedge \ \langle x, y \rangle \in A\}.$$

If the set over which the minimum is taken is empty, define by convention $f(x) = \mathrm{bin}(2^{p(|x|)})$.

Note that $\mathrm{LF}_A = \mathrm{LF}_{(B,p)}$ is in $\min \cdot \mathrm{P}$ for every NP set A and for every representation of A as a p-projection $A = \mathrm{proj}_p(B)$ of some suitable set $B \in \mathrm{P}$ and polynomial p.

Theorem 4. *Let $A \in \mathrm{NP}$ and $B, D \in \mathrm{P}$ be sets, and let p be a polynomial such that $D \supseteq A = \mathrm{proj}_p(B)$ and LF_A is $\leq_{\mathrm{m}}^{\mathrm{FP}}$-complete for* $\min \cdot \mathrm{P}$.

Then, there exist a set $C \in \mathrm{P}$ and a polynomial q such that $D = \mathrm{proj}_q(C)$ and computing LF_D is Δ_2^p-hard.

Hence, D is not self-reducible with respect to (C, q), assuming $\mathrm{P} \neq \mathrm{NP}$.

Taking Σ^* as the set D of Theorem 4, it is clear that the hypothesis of the theorem can be satisfied by suitably choosing A, B and p. It follows that Σ^*, unconditionally, has representations in which it is not self-reducible in the sense of Schnorr, unless $\mathrm{P} = \mathrm{NP}$.

Proof of Theorem 4. Let A, B, and p be given as in the theorem, where $A \subseteq \Sigma^*$ and $B \subseteq \Sigma^* \times \Sigma^*$ and $\Sigma = \{0, 1\}$. Let D be any set in P with $D \supseteq A$. Define

$$C = B \cup \{(x, \mathrm{bin}(2^{p(|x|)})) \mid x \in D\},$$

and let $q(n) = p(n) + 1$ for all n. Note that $C \in \mathrm{P}$ and $D = \mathrm{proj}_q(C)$. It also follows that $\mathrm{LF}_D(x) = \mathrm{LF}_A(x)$ if $x \in D$, and $\mathrm{LF}_D(x) = 2 \cdot \mathrm{LF}_A(x)$ if $x \notin D$.

We now show that computing LF_D is as hard as deciding the Δ_2^p-complete problem `Odd-Min-SAT`, which was defined in Section 4.1. Since LF_A is $\leq_{\mathrm{m}}^{\mathrm{FP}}$-complete for $\min \cdot \mathrm{P}$, we have $\mathrm{LF}_{\mathtt{SAT}}(F) = \mathrm{LF}_A(t(F))$ for some polynomial-time computable function t. Hence,

$$\begin{aligned} F \in \texttt{Odd-Min-SAT} \iff & \mathrm{LF}_{\mathtt{SAT}}(F) \equiv 1 \bmod 2 \\ \iff & \mathrm{LF}_A(t(F)) \equiv 1 \bmod 2 \\ \iff & \mathrm{LF}_D(t(F)) \equiv 1 \bmod 2. \end{aligned}$$

Thus, one can decide "$F \in$ `Odd-Min-SAT`?" by looking at the last bit of $\mathrm{LF}_D(t(F))$. ∎

5 Conclusions and Future Work

In this paper, we studied two important properties of NP problems: self-reducibility, and how to compute complete solutions from partial solutions. Regarding the latter, we in particular studied the graph isomorphism problem. We showed as Theorem 1 that computing even a single pair of vertices belonging to an isomorphism between two isomorphic graphs is as hard as computing a complete isomorphism between the graphs. Theorem 1 optimally improves upon a result of Gál et al. [4].

We propose to establish analogous results for NP problems other than the graph isomorphism problem. For example, Gál et al. [4] investigated many more hard NP problems, and showed that computing partial solutions for them is as hard as computing complete solutions. However, their results are not known to be optimal, which leaves open the possibility of improvement. Relatedly, what impact does the self-reducibility of such problems have for reducing complete search to partial search?

We obtained as Theorem 2 a similar result about reducing complete search to partial search for the Hamiltonian cycle problem. However, this result appears to be slightly weaker than Theorem 1, since in Theorem 2 we require a stronger hypothesis about the function oracle used. Whether this stronger hypothesis in fact is necessary remains an open question. It would be interesting to know whether, also for the Hamiltonian cycle problem, one can prove a result as strong as Theorem 1. More precisely, is it possible to prove the same conclusion as in Theorem 2 when we are given a function oracle that is merely required to return any one edge of a Hamiltonian cycle of the given graph, without requiring in addition that the edge returned belong to a Hamiltonian cycle consistent with the edge's left-right-context?

In Theorem 3, we strengthened Khuller and Vazirani's [11] lower bound for computing the lexicographically first four-coloring for planar graphs from NP-hardness to Δ_2^p-hardness. The non-self-reducibility of the `Pl-4-Color` problem follows immediately from these lower bounds. Khuller and Vazirani [11] asked whether similar non-self-reducibility results can be proven for problems in P other than `Pl-4-Color`, under plausible assumptions such as $\mathrm{P} \neq \mathrm{NP}$. We established as Theorem 4 a general result showing that it is almost trivial to find such problems.

This general result subsumes a number of results [7] providing concrete, although somewhat artificial, problems in P that are not self-reducible in Schnorr's sense, unless $\mathrm{P} = \mathrm{NP}$. These problems are artificial, since they are P versions of standard NP-complete problems—such as the satisfiability problem, the clique problem, and the knapsack problem—defined by encoding directly into each solvable problem instance a trivial solution to this instance. In contrast, the `Pl-4-Color` problem is a quite natural problem. Can one, under a plausible assumption such as $\mathrm{P} \neq \mathrm{NP}$, prove the non-self-reducibility of other *natural* problems in P?

Acknowledgments. We thank Edith and Lane A. Hemaspaandra for introducing us to this interesting topic and for stimulating discussions and comments. We acknowledge interesting discussions about graph theory with Haiko Müller. We thank the anonymous referees for their helpful and insightful comments on the paper. In particular, we thank the referee who suggested an idea that led to Theorem 4, which subsumes some results from an earlier draft of this paper.

References

1. K. Appel and W. Haken. Every planar map is 4-colorable – 1: Discharging. *Illinois J. Math*, 21:429–490, 1977. 349
2. K. Appel and W. Haken. Every planar map is 4-colorable – 2: Reducibility. *Illinois J. Math*, 21:491–567, 1977. 349
3. A. Borodin and A. Demers. Some comments on functional self-reducibility and the NP hierarchy. Technical Report TR 76-284, Cornell Department of Computer Science, Ithaca, NY, July 1976. 351
4. A. Gál, S. Halevi, R. Lipton, and E. Petrank. Computing from partial solutions. In *Proceedings of the 14th Annual IEEE Conference on Computational Complexity*, pages 34–45. IEEE Computer Society Press, May 1999. 340, 341, 354
5. M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, New York, 1979. 339
6. M. Garey, D. Johnson, and L. Stockmeyer. Some simplified NP-complete graph problems. *Theoretical Computer Science*, 1:237–267, 1976. 349, 350
7. A. Große. Partielle Lösungen NP-vollständiger Probleme. *Diploma thesis*, Friedrich-Schiller-Universität Jena, Institut für Informatik, Jena, Germany, December 1999. In German. 354
8. E. Hemaspaandra, A. Naik, M. Ogihara, and A. Selman. P-selective sets and reducing search to decision vs. self-reducibility. *Journal of Computer and System Sciences*, 53(2):194–209, 1996. 339
9. H. Hempel and G. Wechsung. The operators min and max on the polynomial hierarchy. *International Journal of Foundations of Computer Science*, 11(2):315–342, 2000. 352, 353
10. D. Joseph and P. Young. Self-reducibility: Effects of internal structure on computational complexity. In A. Selman, editor, *Complexity Theory Retrospective*, pages 82–107. Springer-Verlag, 1990. 342, 352
11. S. Khuller and V. Vazirani. Planar graph coloring is not self-reducible, assuming $\mathrm{P} \neq \mathrm{NP}$. *Theoretical Computer Science*, 88(1):183–189, 1991. 340, 349, 350, 351, 352, 354
12. K. Ko. On self-reducibility and weak P-selectivity. *Journal of Computer and System Sciences*, 26(2):209–221, 1983. 352
13. M. Krentel. The complexity of optimization problems. *Journal of Computer and System Sciences*, 36:490–509, 1988. 340, 350, 352
14. A. Meyer and M. Paterson. With what frequency are apparently intractable problems difficult? Technical Report MIT/LCS/TM-126, MIT Laboratory for Computer Science, Cambridge, MA, 1979. 342, 352
15. C. Papadimitriou. On the complexity of unique solutions. *Journal of the ACM*, 31(2):392–400, 1984. 340

16. C. Schnorr. Optimal algorithms for self-reducible problems. In S. Michaelson and R. Milner, editors, *Proceedings of the 3rd International Colloquium on Automata, Languages, and Programming*, pages 322–337, University of Edinburgh, July 1976. Edinburgh University Press. 340, 341, 342, 351
17. C. Schnorr. On self-transformable combinatorial problems, 1979. Presented at *IEEE Symposium on Information Theory*, Udine, and *Symposium über Mathematische Optimierung*, Oberwolfach. 340, 341, 342, 351
18. A. Selman. Natural self-reducible sets. *SIAM Journal on Computing*, 17(5):989–996, 1988. 342, 352
19. L. Stockmeyer. Planar 3-colorability is NP-complete. *SIGACT News*, 5(3):19–25, 1973. 349, 350
20. H. Vollmer. On different reducibility notions for function classes. In *Proceedings of the 11th Annual Symposium on Theoretical Aspects of Computer Science*, pages 449–460. Springer-Verlag *Lecture Notes in Computer Science #775*, February 1994. 352, 353
21. K. Wagner. More complicated questions about maxima and minima, and some closures of NP. *Theoretical Computer Science*, 51:53–80, 1987. 340, 350

On the Distribution of a Key Distribution Center

Paolo D'Arco

Dipartimento di Informatica ed Applicazioni, Università di Salerno
84081 Baronissi (SA), Italy
paodar@dia.unisa.it

Abstract. A Key Distribution Center of a network is a server who *generates* and *distributes* secret keys used by groups of users to securely communicate. A Distributed Key Distribution Center is a set of servers that *jointly* realizes a Key Distribution Center. In this paper we study Distributed Key Distribution Centers, pointing out the advantages of this approach to Key Distribution. We propose an information theoretic model of Distributed Key Distribution Center, and present bounds holding on the model. Moreover, we show that a protocol described in [5], which uses Shamir's secret sharing schemes, meets the bounds and is, hence, *optimal* with respect to information storage, communication complexity, and randomness as well.

Keywords: Key Distribution, Protocols, Distributed Systems.

1 Introduction

Key Distribution is an intriguing and deeply studied problem in Cryptography. Loosely speaking, it can be described as follow: a group of users of a network with insecure channels, can decide to use encryption algorithms to privately communicate. If a public key infrastructure is available, an easy solution is the following: a user recovers the public key of the recipient, from a publicly accessible bulletin board, and encrypts, with the public key encryption algorithm, the message he wishes to send. If he needs to send the same message to n different recipients, he computes n encryptions of the message, using the n different public keys of the n recipients, and then sends the message to each of them. However, public key encryption and decryption are slow operations and, when the communication involves a group of users big in size, the above solution is completely inefficient. On the other hand, symmetric algorithms are more efficient than asymmetric ones and, if the group holds a common secret key, a user has to encrypt a message *just once* before sending it to all the other users of the group. Moreover, if a broadcast channel is available across the network, he can broadcast the encrypted message along this channel, sure that all the "authorized" recipients will receive and decrypt the message. Hence, computational and communication complexities can be improved pursuing this approach. But the group, hereafter referred to as a *conference*, needs a common key, to encrypt

A. Restivo, S. Ronchi Della Rocca, L. Roversi (Eds.): ICTCS 2001, LNCS 2202, pp. 357–369, 2001.

and to decrypt the messages, before starting the communication. Therefore, the problem is *how to design an efficient protocol to give each conference a common key*.

Notice that an improvement on the "trivial" use of public key algorithms can be the *hybrid* approach: a user chooses at random a common key and sends it in encrypted form, using the public keys, to all the other members of the conference. Then, they can communicate using a symmetric algorithm. However, this solution is still not efficient and it is possible to do better.

In traditional models of a network, a frequently used approach is the *Key Distribution Center* (for short, KDC), a server of the network who generates and distributes the secret keys. In this setting each user shares a common key with the center. When the user wants to securely communicate with other users, he sends a request for a conference key. The center "checks for membership" of the user in that conference, and distributes in encrypted form the common key to each member of the conference. Needham and Schroeder [6] began this approach, implemented most notably in the Kerberos System [7].

The KDC is a suitable solution to the key distribution problem, since, apart from the "pure distribution" of the keys to the users, several related key-management aspects (i.e., life time, authentication, usage restrictions and so on) can be easily solved with this third part.

The scheme implemented by the Key Distribution Center to give each conference a key is referred to as a *Key Distribution Scheme* (for short, KDS). A KDS is said to be *unconditionally secure* if its security does not rely on computational assumptions on the power of the adversary who can try to break the scheme.

Several unconditionally secure key distribution schemes have been proposed so far (see [9] for a survey). However, these models and protocols assume the presence of a single server accomplishing the key distribution task.

It is not difficult to see that with this approach the KDC must be *trusted*, since it knows all the conference keys; moreover, the KDC could become a performance bottleneck, since all users have to communicate with it every time they wish to obtain a conference key. Besides, a crash of the KDC stalls the whole system. Hence, if from one hand the KDC is a reasonable solution, on the other hand it could be source of weaknesses and security holes across the network.

Well known and applied solutions to the availability and reliability issues are the *replication* of the KDC in several points of the network and the *partition* of the network in several domains with dedicated KDCs, responsible of the key management for only a fixed local area. However, these solutions are partial and expensive solutions [5].

Instead, a robust and efficient solution can be a *Distributed* KDC [5] (DKDC, for short). A DKDC is a set of n servers of a network that jointly realizes the same function of a KDC. In this setting, a user who need to participate to a conference, sends a key-request message to a fixed-size subset at his choice of the n servers. The contacted servers answer with some information enabling the user to compute the conference key. With this approach, the concentration of secrets and the slow down factor which arise in a network with a single KDC are eliminated. A single server by itself does not know the secret keys, since they are *shared* between the n servers. Moreover, each user can send a key-request in parallel to different servers. Hence, there is no loss in time to compute a conference key with respect to a centralized environment. Besides, the users can obtain the keys they need even if they are unable to contact some of the servers.

In this paper we study DKDCs. We propose an information theoretic model for a *Distributed Key Distribution Scheme* (DKDS, for short), a scheme realizing a DKDC. Then, we analyze the relations between the sizes of the different pieces of information needed to setup and manage a DKDC. We show bounds on the amount of information that each server has to store, on the amount of information that each server has to send to answer to a key-request message, and on the size of the messages that have to be generated in setup phase to initialize the DKDC. Moreover, we quantify the randomness needed to setup a DKDC. The bounds we show are tight, since they are met by a protocol [5] which uses multiple copies of the Shamir's secret sharing scheme, based on polynomial interpolation.

2 The Model

Let $\mathcal{U} = \{U_1, \ldots, U_m\}$ be a set of m users, and let S= $\{S_1, \ldots, S_n\}$ be a set of n servers. Each user has private connections with *all* the servers. A distributed key distribution scheme is divided in three phases: an *initialization phase*, which involves only the servers; a *key request phase*, in which users ask for keys to servers; and a *key computation phase*, in which users retrieve keys from the messages received from the servers contacted during the key request phase. We assume that the initialization phase is done by k servers say, without loss of generality, $S_1, \ldots, S_k$. Each of these servers, using a *private source* of randomness r_i, generates some information that securely distributes to the others. More precisely, for $i = 1, \ldots, k$, server S_i generates/sends to S_j, the value $\gamma_{i,j}$, where $j = 1, \ldots, n$. At the end of the initialization phase, each server S_i stores some secret information $a_i = f(\gamma_{1,i}, \ldots, \gamma_{k,i})$, which can be computed from the information he has received. Assume that a group of users $C_h \subseteq \mathcal{U}$, referred to as a *conference*, wants to securely communicate. Each user U_j in C_h, to compute a key for the conference C_h (we denote such a key with κ_h), contacts k servers at least. Then, server S_i, contacted by user U_j, checks[1] for membership of U_j in C_h; if so, the server S_i computes a value $y^h_{i,j} = F(a_i, j, h)$, which is a function of the private information a_i, j, and the index h of the requested key. Otherwise, the server sets $y^h_{i,j} = \perp$, a special value which does convey no information on the conference key. Finally, the server sends the value $y^h_{i,j}$ to U_j. The users in C_h compute the conference key as a function of the information received by the contacted servers, i.e., each user U_j in C_h computes $\kappa_h = G(y^h_{i_1,j}, \ldots, y^h_{i_k,j})$,

[1] We do not consider the underline authentication mechanism involved in a key request phase.

where $i_1, \ldots, i_k$ are the indices of the servers he has contacted and G is a publicly known function.

We are interested in formalizing, within an information theoretic framework, the notion of a DKDS. To this aim, we need to setup our notation.

- Let $\mathcal{C} \subset 2^{\mathcal{U}}$ be the set of conferences on $\mathcal{U}$ who need to securely communicate, and assume they are indexed by elements of $\mathcal{H} = \{1, 2, \ldots\}$.
- For any coalition $G = \{U_{j_1}, \ldots, U_{j_g}\} \subseteq \mathcal{U}$ of users, denote with $\mathcal{C}_G$ the set of conferences containing some user in G, and with $\mathcal{H}_G$ the set of corresponding indices. In other words, $\mathcal{C}_G = \{C_h \in \mathcal{C} : C_h \cap G \neq \emptyset\}$, and $\mathcal{H}_G = \{h \in \mathcal{H} : C_h \in \mathcal{C}_G\}$.
- For $i = 1, \ldots, k$, let $\mathit{\Gamma}_{i,j}$ be the set of values $\gamma_{i,j}$ that can be sent by server S_i to server S_j, for $j = 1, \ldots, n$, and let $\mathit{\Gamma}_j = \mathit{\Gamma}_{1,j} \times \cdots \times \mathit{\Gamma}_{k,j}$ be the set of values that S_j, for $j = 1, \ldots, n$, can receive during the initialization phase.
- Let K_h be the set of possible values for κ_h, and let A_i be the set of values a_i the server S_i can compute during the initialization phase.
- Finally, let $Y_{i,j}^h$ be the set of values $y_{i,j}^h$ that can be sent by S_i when it receives a key-request message from U_j for the conference C_h.

Given three sets of indices $X = \{i_1, \ldots, i_r\}$, $Y = \{j_1, \ldots, j_s\}$, and $H = \{h_1, \ldots, h_t\}$, and three families of sets $\{T_i\}$, $\{T_{i,j}\}$ and $\{T_{i,j}^h\}$, we will denote with $T_X = T_{i_1} \times \cdots \times T_{i_r}$, $T_{X,Y} = T_{i_1,j_1} \times \cdots \times T_{i_r,j_s}$, and $T_{X,Y}^H = T_{i_1,j_1}^{h_1} \times \cdots \times T_{i_r,j_s}^{h_t}$, the corresponding cartesian products. According to this notation, we will consider the following cartesian products, defined on the sets of our interest

Table 1. Cartesian products

$\mathit{\Gamma}_Y$ Set of values that can be received by server S_j, for $j \in Y$
$\mathit{\Gamma}_{X,j}$ Set of values that can be sent by server S_i to S_j, for $i \in X$
$\mathit{\Gamma}_{X,Y}$ Set of values that can be sent by server S_i to S_j, for $i \in X$ and $j \in Y$
K_X Set of $\lvert X \rvert$-tuple of conference keys
A_X Set of $\lvert X \rvert$-tuple of private information a_i
$Y_{X,j}^h$ Set of values that can be sent by S_i, for $i \in X$, to U_j for the conference C_h
Y_G^h Set of values that can be sent by $S_1, \ldots, S_n$ to U_j, with $j \in G$, for C_h
Y_G^H Set of values that can be sent by $S_1, \ldots, S_n$ to U_j, with $j \in G$, for C_h $\forall h \in H$

We will denote in boldface the random variables $\mathbf{\Gamma}_{i,j}, \mathbf{\Gamma}_j, \ldots, \mathbf{Y}_G^{\mathcal{H}_G \setminus \{h\}}$ assuming values on the sets $\mathit{\Gamma}_{i,j}, \mathit{\Gamma}_j, \ldots, Y_G^{\mathcal{H}_G \setminus \{h\}}$ according to the probability distributions $\mathcal{P}_{\mathbf{\Gamma}_{i,j}}, \mathcal{P}_{\mathbf{\Gamma}_j}, \ldots, \mathcal{P}_{\mathbf{Y}_G^{\mathcal{H}_G \setminus \{h\}}}$.

Roughly speaking, a DKDC must satisfy the following properties:

- **Correct Initialization Phase.** When the initialization phase correctly terminates, each server S_i must be able to compute his private information a_i. On the other hand, if server S_i misses/does-not-receive *just one* message from the servers sending information, then S_i must not gain any information about a_i. We model these two properties by relations 1 and 2 of the formal definition.

- **Consistent Key Computation.** Each user in a conference $C_h \subseteq \mathcal{U}$ must be able to compute *the same* conference key, after interacting with the servers of a subset P at his choice of size at least k. Relations 3 and 4 of the formal definition ensure these properties. More precisely, relation 3 establishes that each server uniquely determines an answer to any key-request message; while, property 4 establishes that each user uniquely computes the same conference key, using the messages received by the subset of servers he has contacted for that conference key.
- **Conference Key Security.** A conference key must be secure against attacks performed by coalitions of servers, coalitions of users, and hybrid coalitions (servers and users). This is the most intriguing and difficult property to formalize. Indeed, the worst case scenario to look after consists of a coalition of users G that run the protocol many times, retrieving several conference keys and, then, with the cooperation of some dishonest servers, try to gain information on a new conference key, they didn't ask before. Notice that, according to our notation, the maximum amount of information the coalition can acquire honestly running the protocol is represented by $\mathbf{Y}_G^{\mathcal{H}_G \setminus \{h\}}$; moreover, dishonest servers, belonging to a subset F of size at most $k-1$, know $\mathbf{\Gamma}_F$ and, maybe, $\mathbf{\Gamma}_{Z,N}$. This random variable takes into account the possibility that some of the dishonest servers send information in the initialization phase (i.e. $Z \subseteq F \cap \{1, \ldots, k\}$). Hence, they know the messages they send out to the other servers in this phase. Relation 5 ensures that such coalitions of adversaries, do not gain information on any new key.

Formally, a Distributed Key Distribution Scheme can be defined as follows:

Definition 1. *Let $\mathcal{U}$ be a set of users, and let $\mathcal{C} \subseteq 2^{\mathcal{U}}$. A $(k, n, \mathcal{C})$-Distributed Key Distribution Scheme (for short, $(k, n, \mathcal{C})$-DKDS) is a protocol which enables each user of $C_h \in \mathcal{C}$ to compute a common key κ_h interacting with at least k of the n servers of the network. More precisely, the following properties are satisfied:*

1. *For each $i = 1, \ldots, n$, it holds that $H(\mathbf{A}_i | \mathbf{\Gamma}_i) = 0$.*
2. *For each $X \subset \{1, \ldots, k\}$, and $i \in \{1, \ldots, n\}$, it holds that $H(\mathbf{A}_i | \mathbf{\Gamma}_{X,i}) = H(\mathbf{A}_i)$.*
3. *For each $C_h \in \mathcal{C}$, for each $U_j \in C_h$, and for each $i = 1, \ldots, n$, it holds that $H(\mathbf{Y}_{i,j}^h | \mathbf{A}_i) = 0$.*
4. *For each $C_h \in \mathcal{C}$, for each $P \subset \{1, \ldots, n\}$ of size at least k, and for each $U_j \in C_h$, it holds that $H(\mathbf{K}_h | \mathbf{Y}_{P,j}^h) = 0$.*
5. *For each $C_h \in \mathcal{C}$, for each G, and for each subset of servers F of size less than k it holds that*

$$H(\mathbf{K}_h | \mathbf{Y}_G^{\mathcal{H}_G \setminus \{h\}} \mathbf{\Gamma}_F \mathbf{\Gamma}_{Z,N}) = H(\mathbf{K}_h)$$

where $Z = F \cap \{1, \ldots, k\}$ and $N = \{1, \ldots, n\}$.

In the following, without loss of generality, we assume that for different $h, h' \in \mathcal{H}$, we have $H(\mathbf{K}_h) = H(\mathbf{K}_{h'})$.

3 Some Technical Lemmas

To show the main properties of our model, we need some technical lemmas whose proofs will appear in the full version of the paper.

Recalling that $\mathcal{C}_G = \{C_h \in \mathcal{C}$ and $C_h \cap G \neq \emptyset\}$, for any $G \subseteq \mathcal{U}$, hereafter we set $\ell_G = |\mathcal{C}_G|$, i.e., ℓ_G denotes the number of conference keys that a coalition of adversaries G can retrieve interacting with the servers.

The following two lemmas are useful to show lower bounds on the amount of information that each of the initializing server $S_1, \ldots, S_k$ has to send to the other servers during the initialization phase, and on the randomness (to be defined later) needed to setup a $(k, n, \mathcal{C})$-DKDS.

The first one essentially establishes a bound on the amount of information server S_i, performing the distribution, has to send out to server S_j.

Lemma 1. *In any $(k, n, \mathcal{C})$-DKDS, for each $i = 1, \ldots, k$, for each $j = 1, \ldots, n$, for each set $Y \subset \{1, \ldots, n\} \setminus \{j\}$ of size at most $k-1$, and for any coalition G of users, it holds that*

$$H(\mathbf{\Gamma}_{i,j} | \mathbf{\Gamma}_Y \mathbf{\Gamma}_{X,j}) \geq \ell_G \cdot H(\mathbf{K}),$$

where $X = \{1, \ldots, k\} \setminus \{i\}$.

The second one is a consequence of the above lemma. It bounds the amount of "residual" information a server of the system still has, given the information received by a subset of at most $k-1$ other servers.

Lemma 2. *In any $(k, n, \mathcal{C})$-DKDS, for each $j = 1, \ldots, n$, for each set $Y \subset \{1, \ldots, n\} \setminus \{j\}$ of size at most $k-1$, and for any coalition G of users, it holds that*

$$H(\mathbf{\Gamma}_j | \mathbf{\Gamma}_Y) \geq k \cdot \ell_G \cdot H(\mathbf{K}).$$

Notice that the intuition behind the above lemmas is that there is an "information gap" between a "forbidden" subset of servers (Y), and an "authorized" one $(Y \cup \{j\}$, assuming $|Y| = k-1)$. The evaluation of this gap, as we will show, enables us to prove in an easy way lower bounds holding on the model.

4 Properties and Bounds

In this section we show some properties of our model.

The following theorem establishes a lower bound both on the amount of information $\gamma_{i,j}$ that each of the initializing servers, $S_1, \ldots, S_k$, has to send during the setup phase to $S_1, \ldots, S_n$, and on the amount of information γ_i that each server must receive in order to be able to compute his private information a_i.

Theorem 1. *In any $(k, n, \mathcal{C})$-DKDS, for each $i = 1, \ldots, k$, and $j = 1, \ldots, n$, the following inequalities are satisfied:*

$$H(\mathbf{\Gamma}_{i,j}) \geq \ell_G \cdot H(\mathbf{K}) \textit{ and } H(\mathbf{\Gamma}_j) \geq k \cdot \ell_G \cdot H(\mathbf{K}).$$

Proof. Notice that, from (3) of Appendix A and from Lemma 1, setting $X = \{1, \ldots, k\} \setminus \{i\}$ and $Y \subset \{1, \ldots, n\} \setminus \{j\}$, we have that

$$H(\mathbf{\Gamma}_{i,j}) \geq H(\mathbf{\Gamma}_{i,j} | \mathbf{\Gamma}_Y \mathbf{\Gamma}_{X,j}) \geq \ell_G \cdot H(\mathbf{K}).$$

Moreover, from (3) of Appendix A and Lemma 2, choosing $Y \subset \{1, \ldots, n\} \setminus \{j\}$, it results

$$H(\mathbf{\Gamma}_j) \geq H(\mathbf{\Gamma}_j | \mathbf{\Gamma}_Y) \geq k \cdot \ell_G \cdot H(\mathbf{K}).$$

Thus, the theorem holds. □

Remark 1. Assume that the keys are uniformly chosen (i.e., $H(\mathbf{K}) = \log |K|$). Since $\log |\Gamma_{i,j}| \geq H(\mathbf{\Gamma}_{i,j})$ (see Appendix A) then, from Theorem 1, we get that $\log |\Gamma_{i,j}| \geq \ell_G \log |K|$. Therefore, each server, performing the initialization phase, has to send to each other server a message consisting of at least $\ell_G \cdot \log |K|$ bits. On the other hand, the inequality $H(\mathbf{\Gamma}_j) \geq k \cdot \ell_G \cdot H(\mathbf{K})$ implies that, when the keys are uniformly chosen, each server, during the initialization phase, will receive messages whose sizes will sum up to at least $k \cdot \ell_G \cdot \log |K|$ bits.

Using some basic properties of the entropy function, it is possible to obtain a lower bound on the amount of information that each server, contacted by a user, has to send upon receiving a key-request message.

Theorem 2. *In any $(k, n, \mathcal{C})$-DKDS, for any $C_s \in \mathcal{C}$, for any $i = 1, \ldots, n$, and for any $U_j \in C_s$, it holds that*

$$H(\mathbf{Y}^s_{i,j}) \geq H(\mathbf{K}).$$

We can also show that each server, to answer the user's key-request messages, has to store some information whose size is lower bounded by $\ell_G \cdot H(\mathbf{K})$.

Theorem 3. *In any $(k, n, \mathcal{C})$-DKDS, for each $i = 1, \ldots, n$, the private information a_i, stored by the server S_i, satisfies*

$$H(\mathbf{A}_i) \geq \ell_G \cdot H(\mathbf{K}).$$

Denoting with ℓ the maximum number of conference keys that a coalition of adversaries G can retrieve interacting with the servers, i.e., $\ell = \max_{G \subseteq \mathcal{U}} \ell_G$, we can establish a lower bound on the communication complexity of a DKDS, and on the total randomness needed to set up such a scheme.

Communication Complexity The Communication Complexity ($\mathcal{CC}$, for short) of a DKDS is measured by the amount of information sent by the servers $S_1, \ldots, S_k$ during the initialization phase. Using Theorem 1, it is simple to check that

$$\mathcal{CC} = \sum_{i=1}^{k} \sum_{j=1}^{n} H(\mathbf{\Gamma}_{i,j}) \geq k \cdot \ell \cdot n \cdot H(\mathbf{K}).$$

Randomness When we want to set up a cryptographic protocol and, in this case, a Distributed Key Distribution Scheme, often we need random bits. This resource is usually referred to as the *randomness* of the scheme. [2] The randomness of a scheme can be measured in different way. Knuth and Yao [4] proposed the following approach: let `Alg` be an algorithm that generates the probability distribution $P = \{p_1, \ldots, p_n\}$, using only independent and unbiased random bits. Denote by $T(\texttt{Alg})$ the average number of random bits used by `Alg` and let $T(P) = min_{\texttt{Alg}} T(\texttt{Alg})$. The value $T(P)$ is a measure of the average number of random bits needed to simulate the source described by the probability distribution P. In [4] it has been shown the following result

Theorem 4. $H(\mathbf{P}) \leq T(\mathbf{P}) < H(\mathbf{P}) + 2.$

Thus, the entropy of a random source is very close to the average number of unbiased random bits necessary to simulate the source. Hence, the entropy of a random source is a natural measure of the randomness.

It is easy to see that the randomness $\mathcal{R}$ of a Distributed Key Distribution Scheme can be lower bounded by $H(\mathbf{\Gamma}_1 \ldots \mathbf{\Gamma}_n)$. The following theorem shows a lower bound on $\mathcal{R}$.

Theorem 5. *In any* $(k, n, \mathcal{C})$*-DKDS the randomness* $\mathcal{R}$ *satisfies*

$$\mathcal{R} \geq k^2 \cdot \ell \cdot H(\mathbf{K}).$$

Proof. Notice that, for each $\{j_1, \ldots, j_k\} \subset \{1, \ldots, n\}$, from Theorem 4, and property (5) of Appendix A, one gets

$$\begin{aligned}
\mathcal{R} \geq H(\mathbf{\Gamma}_1 \ldots \mathbf{\Gamma}_n) &\geq H(\mathbf{\Gamma}_{j_1} \ldots \mathbf{\Gamma}_{j_k}) \\
&= \sum_{r=1}^{k} H(\mathbf{\Gamma}_{j_r} | \mathbf{\Gamma}_{j_1} \ldots \mathbf{\Gamma}_{j_r - 1}) \text{ (applying (2) of Appendix A)} \\
&\geq \sum_{r=1}^{k} H(\mathbf{\Gamma}_{j_r} | \mathbf{\Gamma}_{Y}) \\
&\quad \text{(setting } Y = \{j_1, \ldots, j_k\} \setminus \{j_r\} \text{ and (7) of Appendix A)} \\
&\geq \sum_{r=1}^{k} k \cdot \ell \cdot H(\mathbf{K}) \text{(from Lemma 2)} \\
&= k^2 \cdot \ell \cdot H(\mathbf{K}).
\end{aligned}$$

[2] A detailed analysis of the randomness in distribution protocols can be found in [2].

Hence, the theorem holds. □

The following table summarizes the main bounds obtained by the above analysis assuming the keys are uniformly chosen in a set K.

Table 2. Bounds on DKDS for uniformly chosen keys

Parameters	**Information needed (in bits)**
Randomness ($\mathcal{R}$)	$k^2 \cdot \ell \cdot \log \|K\|$
Memory Server Storage ($H(\mathbf{A}_i)$)	$\ell \cdot \log \|K\|$
Communication Complexity ($\mathcal{CC}$)	$k \cdot n \cdot \ell \cdot \log \|K\|$

5 On the Size of the Coalition of Adversaries

The model described in Section 2 is a generalization of the model proposed in [1]. In that paper we considered a network with m users represented by the set $\mathcal{U} = \{U_1, \ldots, U_m\}$, a set $\mathcal{S} = \{S_1, \ldots, S_n\}$ of n servers, and a set of possible conferences $\mathcal{C}$. Moreover, we required the scheme to be secure against coalitions of up to $k-1$ servers, coalition of users of *any* size, and hybrid coalitions. In that simplified scenario a trusted authority, the *dealer*, realizes the initialization phase distributing the private information a_i's to each of the n servers of the network. It is not difficult to see that, such a model is described by relations 3 and 5 of the current model, assuming that in relation 5 the set G is equal to $\mathcal{U}$ and substituting $\mathbf{\Gamma}_X \mathbf{\Gamma}_{Z,N}$ with $\mathbf{A}_X$.

In the previous sections, the analysis has been done making *no assumptions* on the structure of the coalitions of adversaries that will try to break the scheme. This approach enable us to model multiple scenarios, usually considered in the analysis of unconditionally secure key distribution schemes. For example, it is unrealistic to assume that all the users of a wide area network, like the Internet, can collude to break a scheme. It is more realistic to consider an *upper bound* g on the size of a coalition of adversaries. Such schemes are often referred to as g-resilient scheme. Moreover, to further reduce resources requirements of a $(k, n, \mathcal{C})$-DKDC, we can consider an environment in which the set of conferences $\mathcal{C}$ is composed only by the subsets of users of size up to t. Such schemes are characterized by the two parameters t and g and are referred to as g-resilient t-conference schemes.

g-resilient schemes In such a scenario we fix the size of the possible coalitions of adversaries G to be at most equal to g. Assuming that $\mathcal{C}$ contains all the possible conferences of $\mathcal{U}$, and hence $|\mathcal{C}| = 2^m - m - 1$, we can show by a counting argument, that the maximum number of conference keys that can be recovered by any coalition $G \subseteq \{U_1, \ldots, U_m\}$ is

$$\begin{aligned}\ell &= \sum_{j=2}^{g}\binom{g}{j} + \sum_{j=1}^{g}\binom{g}{j}\cdot(2^{m-g}-1)\\ &= \sum_{j=1}^{g}\binom{g}{j} - g + \sum_{j=1}^{g}\binom{g}{j}\cdot 2^{m-g} - \sum_{j=1}^{g}\binom{g}{j}\\ &= \sum_{j=1}^{g}\binom{g}{j}\cdot 2^{m-g} - g.\end{aligned}$$

Hence, the bounds of Theorems 2, 3, and 5, change according to this value.

g-resilient t-conference schemes Suppose that it is known an upper bound t on the maximum size of the conferences of $\mathcal{U} = \{U_1, \ldots, U_m\}$ and an upper bound g on the maximum size of the coalitions of adversaries. Assuming that $\mathcal{C}$ contains all the possible conferences of $\mathcal{U}$ of size at most t, then $|\mathcal{C}| = \sum_{j=2}^{t}\binom{m}{j}$ and

$$\begin{aligned}\ell &= \sum_{j=2}^{t}\binom{g}{j} + \sum_{s=2}^{t}\sum_{j=1}^{s-1}\binom{g}{j}\cdot\binom{m-g}{s-j}\\ &= \sum_{s=2}^{t}\sum_{j=1}^{s}\binom{g}{j}\cdot\binom{m-g}{s-j}.\end{aligned}$$

It is easy to see that the bounds of Theorems 2, 3, and 5 are determined by the above value for ℓ.

6 An Optimal Protocol

A construction of a $(k, n, \mathcal{C})$-DKDS, based on a family of ℓ-wise independent functions[3], has been proposed in [5].

The scheme enables ℓ conferences in $\mathcal{C}$, *not known* a priori, to securely compute a conference key. The family of ℓ-wise independent functions chosen in [5] to construct the $(k, n, \mathcal{C})$-DKDS is the family of all bivariate polynomials $P(x, y)$ over a given finite field Z_q, in which the degree of x is $k-1$ and the degree of y is $\ell - 1$. The protocol can be described as follows

[3] A function is ℓ-wise independent if the knowledge of the value of the function in $\ell - 1$ different points of the domain does not convey any information on the value of the function in another point.

Initialization Phase

- Let $\ell = \max_{G \subseteq \mathcal{U}} \ell_G$ be the maximum number of conference keys that a coalition G of users can compute.
- Each of the servers $S_1,\ldots,S_k$, performing the initialization phase, constructs a random bivariate polynomial $P^i(x,y)$ of degree $k-1$ in x, and $\ell-1$ in y by choosing $k \cdot \ell$ random elements in Z_q.
- Then, for $i=1,\ldots,k$, server S_i evaluates the polynomial $P^i(x,y)$ in the identity j of S_j, and sends $Q_j^i(y) = P^i(j,y)$ to the server S_j, for $j = 1,\ldots,n$.
- For $j=1,\ldots,n$, each server S_j computes his private information a_j, adding the k polynomials of degree $\ell-1$, obtained from the k servers performing the initialization phase. More precisely,

$$a_j = Q_j(y) = \sum_{i=1}^{k} Q_j^i(y).$$

A user who needs a conference key, sends a key request to the servers as follows

Key Request Phase

- A user in conference C_h, who wants to compute the conference key, sends to at least k servers, say $S_{i_1},\ldots,S_{i_k}$, a request for the conference key
- Each server S_{i_j}, invoked by the user, checks that the user belongs to C_h, and sends to the user the value $Q_{i_j}(h)$, i.e., the value of the polynomial $Q_{i_j}(y)$ evaluated in $y=h$.

Finally, using the k values received from the servers $S_{i_1},\ldots,S_{i_k}$, and applying the Lagrange formula for polynomial interpolation, each user in C_h recovers the secret key $P(0,h) = \sum_{i=1}^{k} P^i(0,h)$. More precisely,

Key Computation Phase

- Each user computes, for $j=1,\ldots,k$, the coefficients

$$b_j = \prod_{1 \le s \le k, s \ne j} \frac{i_s}{i_s - i_j}.$$

Then, he recovers $P(0,h)$ computing the $\sum_{j=1}^{k} b_j y_{i_j}$ where, for $j=1,\ldots,k$, $y_{i_j} = Q_{i_j}(h)$, the value received from the server S_{i_j}.

The protocol satisfies Definition 1 and meets the bounds established by Theorems 1, 2, 3, and 5.

7 Open Problems

Some questions arise from this analysis. We have assumed that each user has private connections with *all* the n servers of the network. It can be interesting to study the same problem assuming that, for each user, there are d possible different secure connections with the servers, where $k \leq d \leq n$. A real world motivation for this setting could be that each user has only connections with geographically close servers.

Another interesting research line is in studying *computationally secure* distributed key distribution schemes. In [5] some constructions have been proposed. With this approach some resource requirements can be reduced. It is interesting to see if more efficient and secure schemes of those suggested in [5] can be designed.

References

1. C. Blundo, and P. D'Arco, An Information Theoretic Model for Distributed Key Distribution, *Proceedings of the 2000 IEEE International Symposium on Information Theory*, pp. 270, 2000. 365
2. C. Blundo, A. De Santis, and U. Vaccaro, Randomness in Distribution Protocols, *Information and Computation*, vol. 131, no. 2, pp. 111–139, 1996. 364
3. Cover T. M. and Thomas J. A., Elements of Information Theory, *John Wiley & Sons*, 1991. 368
4. Knuth D. E. and Yao A. C., The Complexity of Nonuniform Random Number Generation, *Algorithms and Complexity*, Academic Press, pp. 357–428, 1976. 364
5. M. Naor, B. Pinkas, and O. Reingold, Distributed Pseudo-random Functions and KDCs, *Advances in Cryptology - Eurocrypt 99, Lecture notes in Computer Science*, vol. 1592, pp. 327–346, 1999. 357, 358, 359, 366, 368
6. R. M. Needham and M. D. Schroeder, Using Encryption for Authentication in Large Networks of Computers, *Communications of ACM*, vol. 21, pp. 993–999, 1978. 358
7. B. C. Neuman and T. Tso, Kerberos: An Authentication Service for Computer Networks, *IEEE Transaction on Communications*, vol. 32, pp. 33–38, 1994. 358
8. A. Shamir, How to Share a Secret, *Communications of ACM*, vol. 22, n. 11, pp. 612–613, 1979.
9. D. R. Stinson, On Some Methods for Unconditional Secure Key Distribution and Broadcast Encryption, *Design, Codes and Cryptography*, vol. 12, pp. 215-243, 1997. 358

A Information Theory Elements

This appendix briefly recalls some elements of information theory (see [3] for details).

Let $\mathbf{X}$ be a random variable taking values on a set X according to a probability distribution $\{P_{\mathbf{X}}(x)\}_{x\in X}$. The *entropy* of $\mathbf{X}$, denoted by $H(\mathbf{X})$, is defined as

$$H(\mathbf{X}) = -\sum_{x \epsilon X} P_{\mathbf{X}}(x) \log P_{\mathbf{X}}(x),$$

where the logarithm is to the base 2. The entropy satisfies $0 \leq H(\mathbf{X}) \leq \log |X|$, where $H(\mathbf{X}) = 0$ if and only if there exists $x_0 \in X$ such that $Pr(\mathbf{X} = x_0) = 1$; whereas, $H(\mathbf{X}) = \log |X|$ if and only if $Pr(\mathbf{X} = x) = 1/|X|$, for all $x \in X$.

Given two random variables $\mathbf{X}$ and $\mathbf{Y}$ taking values on sets X and Y, respectively, according to the probability distribution $\{P_{\mathbf{XY}}(x,y)\}_{x\in X, y\in Y}$ on their cartesian product, the *conditional entropy* $H(\mathbf{X}|\mathbf{Y})$ is defined as

$$H(\mathbf{X}|\mathbf{Y}) = -\sum_{y\in Y}\sum_{x\in X} P_{\mathbf{Y}}(y) P_{\mathbf{X}|\mathbf{Y}}(x|y) \log P_{\mathbf{X}|\mathbf{Y}}(x|y).$$

It is easy to see that

$$H(\mathbf{X}|\mathbf{Y}) \geq 0. \tag{1}$$

with equality if and only if X is a function of Y.

Given $n+1$ random variables, $\mathbf{X}_1 \ldots \mathbf{X}_n \mathbf{Y}$, the entropy of $\mathbf{X}_1 \ldots \mathbf{X}_n$ given $\mathbf{Y}$ can be written as

$$H(\mathbf{X}_1 \ldots \mathbf{X}_n|\mathbf{Y}) = H(\mathbf{X}_1|\mathbf{Y}) + H(\mathbf{X}_2|\mathbf{X}_1\mathbf{Y}) + \cdots + H(\mathbf{X}_n|\mathbf{X}_1 \ldots \mathbf{X}_{n-1}\mathbf{Y}). \tag{2}$$

The *mutual information* between $\mathbf{X}$ and $\mathbf{Y}$ is given by

$$I(\mathbf{X};\mathbf{Y}) = H(\mathbf{X}) - H(\mathbf{X}|\mathbf{Y}).$$

Since, $I(\mathbf{X};\mathbf{Y}) = I(\mathbf{Y};\mathbf{X})$ and $I(\mathbf{X};\mathbf{Y}) \geq 0$, it is easy to see that

$$H(\mathbf{X}) \geq H(\mathbf{X}|\mathbf{Y}), \tag{3}$$

with equality if and only if $\mathbf{X}$ and $\mathbf{Y}$ are independent. Therefore, given n random variables, $\mathbf{X}_1 \ldots \mathbf{X}_n$, it holds that

$$H(\mathbf{X}_1 \ldots \mathbf{X}_n) = \sum_{i=1}^{n} H(\mathbf{X}_i|\mathbf{X}_1 \ldots \mathbf{X}_{i-1}) \leq \sum_{i=1}^{n} H(\mathbf{X}_i). \tag{4}$$

Moreover, the above relation implies that, for each $k \leq n$,

$$H(\mathbf{X}_1 \ldots \mathbf{X}_n) \geq H(\mathbf{X}_1 \ldots \mathbf{X}_k). \tag{5}$$

Given three random variables, $\mathbf{X}$, $\mathbf{Y}$, and $\mathbf{Z}$, the *conditional mutual information* between $\mathbf{X}$ and $\mathbf{Y}$ given $\mathbf{Z}$ can be written as

$$I(\mathbf{X};\mathbf{Y}|\mathbf{Z}) = H(\mathbf{X}|\mathbf{Z}) - H(\mathbf{X}|\mathbf{Z}\ \mathbf{Y}) = H(\mathbf{Y}|\mathbf{Z}) - H(\mathbf{Y}|\mathbf{Z}\ \mathbf{X}) = I(\mathbf{Y};\mathbf{X}|\mathbf{Z}). \tag{6}$$

Since the conditional mutual information $I(\mathbf{X};\mathbf{Y}|\mathbf{Z})$ is always non-negative we get

$$H(\mathbf{X}|\mathbf{Z}) \geq H(\mathbf{X}|\mathbf{Z}\ \mathbf{Y}). \tag{7}$$

Online Advertising: Secure E-coupons

Stelvio Cimato and Annalisa De Bonis

Dipartimento di Informatica ed Applicazioni, Università di Salerno
84081 Baronissi (SA), Italy
{cimato,debonis}@dia.unisa.it

Abstract. A form of advertisement which is becoming popular on the web is based on digital coupon (*e-coupon*) distribution. E-coupons are the digital analogue of paper coupons which are used to provide customers with incentives to buy some merchandise. Nowadays, the potential of digital coupons has not been fully exploited on the web. This is mostly due to the lack of an "efficient" protocol for e-coupon distribution which ensures that e-coupons are released by advertisers according to the merchant's specification and that customers use them properly. In this paper we propose a protocol for e-coupons which satisfies these security requirements. Our protocol is lightweight and preserves users privacy since it does not require users' registration.

1 Introduction

New forms of business have been created by the spreading diffusion of Internet and the WWW. Actually, the most diffused approaches for making money on the Internet are the exchange of physical goods, services and commercial information through advertising. The annual volume of money involved in online advertising is on the order of billions of dollars and is still growing. According to several forecasts, an important share of the whole advertising market will be conquered by online advertising.

What is happening is that traditional mechanisms of communication between customers and advertisers are being entirely transposed to digital communication, exploiting Internet as an alternative channel to the traditional ones, such as TV and newspapers. In this perspective, merchants buy portions of a web page in the same way they choose space on a newspapers, selecting the most visited sites to get in touch with the largest number of people. In the context of online advertising and e-commerce, however, the electronic nature of the medium offers both new advantages and new risks which have to be understood and faced to better exploit all the technological opportunities available.

Both advertising and e-commerce applications can be improved by developing new techniques to measure the impact of an advertisement campaign and to construct new means to attract potential customers. Traditional means to rate the success on an advertisement campaign, usually based on statistics, does not apply well to online advertisements where the number of users and the differences among them are very high. *Metering schemes* [4] have been introduced

A. Restivo, S. Ronchi Della Rocca, L. Roversi (Eds.): ICTCS 2001, LNCS 2202, pp. 370–383, 2001.

as alternative systems to measure the exposure of online advertisements. A metering scheme is a protocol which allows to count the number of client visits received by web sites. In [10,3] metering schemes have been provided, which are cryptographically secure against attempts of web sites to inflate the count of client visits they have received.

Coupons are another traditional form of advertising. Coupons are normally found in newspapers, mail, or handbills, providing customers with an incentive to buy certain products by offering discounts or gifts on the purchase of the merchandise. They can be issued by both manufacturers and merchants with the purpose of promoting new brands or the switching between two brands or of increasing the sales of an existing product [2]. Since customers can have an immediate saving of money, coupons represent a very efficient tool to promote a particular product. Recently, the Internet has become a distribution medium for traditional coupons, where customers have to print the coupon contained in a Web page and redeem it like any other coupon [12,11,5].

In [7] it has been introduced the concept of *e-coupon* as a new mechanism of advertising on the web. E-coupons are the electronic version of the real-world coupons which can be redeemed at online stores during e-commerce transactions. The advantages of e-coupons over their paper counterpart are in terms of facilities for the targeting and ease of use in the redemption phase. Furthermore, e-coupons constitute a more powerful informative tool since they make it possible to collect information on several characteristics of the ad access, such as the time or the context in which it was made. However, the utilization of e-coupons for advertising poses some problems, derived from their e-nature, such as duplication, double-spending and forging.

At the moment, there is no common accepted model describing electronic coupons, and only recently frameworks for e-coupon distribution are being developed [1,8]. In this paper we present a suitable model for e-coupons comprising a number of characteristics which make them efficiently usable in e-commerce applications. Furthermore we present two protocols for their generation and distribution, such that a number of security requirements are respected. E-coupons are indeed conceived to be used in commercial transactions, and all the participants (users, advertisers and sellers) should be ensured on the mechanisms of generation and redemption. Our solution is deliberately lightweight with respect to other proposal which rely on digital signature schemes and certification authorities [7].

In Section 2 we introduce our model for e-coupon, distinguishing between *static* and *dynamic* coupons. In Section 3 we introduce the protocols for the distribution of both kinds of coupons, discussing their security requirements. Finally in Section 4 we discuss the design of a viable implementation of the e-coupon generation and distribution model presented.

2 The Model

Our scenario comprises *merchants* who are willing to advertise their products, *advertisers* who display ads on their web sites, and *customers* who browse the web. Merchants stipulate a contract with all the advisers whose web sites she would like to display her ads on. This contract settles the terms of the advertisement campaign, including the specification of the e-coupon promotion. Indeed, the merchant should instruct the advertiser on the product she would like to promote, the amount of discount to be offered, the eventual conditions of the e-coupon's redemption, and other particulars of the promotion.

The advertisers authorized by a merchant to release e-coupons on her behalf have to generate e-coupons according to the merchant's specification.

Users who connect to the web site of an advertiser can download the e-coupons hosted by that web site. A user who wants to benefit of the e-coupon's offer will present the e-coupon for redemption to the merchant. Merchants can obtain information on the success of their advertisement campaigns from the number of e-coupons redeemed by customers. In this way she can check whether the amount of money paid to each advertiser to host her ads is worthwhile.

We will assume that each participant aims to maximize her benefit/cost ratio. This assumption implies that:

- Merchants want to maximize the revenue coming from her advertisement investment.
- Users download the coupon providing the most attractive offer.

We will consider two different kinds of e-coupons which we will refer to as *static* e-coupons and *dynamic* e-coupons, respectively. Static e-coupons are very much like coupons which are distributed by magazines, whereas dynamic e-coupons introduce new features which are particularly useful in the context of electronic commerce.
Static e-coupons have the following characteristics:

- They are static in the sense that their value does not depend on when they have been downloaded by the user.
- Eventually they may indicate a validity period and it might be even the case that their value would change, possibly decrease, during this period. However as for their paper counterpart, there is no mean to determine the exact time when the user has obtained them.

Dynamic e-coupon have the following characteristics:

- They contain information on the time when they have been downloaded by the user.
- They push the users to purchase the advertised merchandise as soon as possible by offering a discount whose value starts decreasing from the moment they are downloaded. In this way users are discouraged from shopping around in order to find a better priced product, and at the same time, merchants have an immediate feedback of the goodness of their ad campaign.

Dynamic e-coupons are particularly useful for advertising products and services which are directly sold on the web. Indeed, persons who buy goods on the web use to visit a consistent number of e-commerce sites in order to purchase the merchandise at the most convenient price.

2.1 Security Requirements

We assume that each participant aims to obtain the maximum possible benefit from the system. We also contemplate the eventuality that participants would cheat in order to achieve their objective. Users might alter the e-coupon data in order to benefit of a more advantageous offer. For example, users may modify the e-coupon value or its validity period. In the dynamic setting users may also attempt to date e-coupons after the real emission time. A postdated e-coupon will allow the dishonest user to delay the e-coupon redemption. In this way the dishonest user will have more time to browse the web and to compare the e-coupon's offer with other offers available on the web. Instead of altering the data of a legally released e-coupon, a dishonest user might try to generate a fake e-coupon which would resemble an authentic one.

Notice that e-coupons, as any other digital object, can be easily duplicated. For that reason a user who has downloaded an e-coupon may decide to make several copies of it and distribute these copies to her friends or use those duplicates herself. Obviously it is impossible for the merchant to distinguish a duplicate from the original which has been actually downloaded from the web site of the advertiser. Depending on the market strategy adopted by the merchant, this circumstance may be more or less acceptable. There are situations in which duplicate redemptions is highly undesirable for the merchant. As an example, consider the case of a merchant who offers discounts on a new product in order to push users to try it for the first time. The merchant's policy consists of offering discounts only on a very restricted number of items. In this case if the number of redeemed e-coupons is far larger than the expected one, then the advertisement campaign will turn into an economic loss for the merchant. Duplicates redemption will also have another undesired effect for the merchant. Indeed, the merchant could obtain information on the exposure of her advertisements on a given web site from the number of redeemed e-coupons which have been downloaded from that web site. Obviously, if the number of redeemed e-coupons is far larger than the number of e-coupons which have been really downloaded from the web site, then it represents a completely meaningless information for the merchant. Advertisers have interest in avoiding duplicate redemption, too. Indeed, users who obtain duplicated e-coupons will probably miss of visiting the advertisers' web sites. Consequently these users will not view the other ads displayed by the advertisers. Clearly, this is a drawback for the advertisers whose revenues depend on the exposure of all displayed ads.

Consequently, the system should allow merchants to detect users' attempts of redeeming the same e-coupon more than once. In this way a merchant may decide to accept a given e-coupon only from the user which provides it as first. For that reason, it would be impossible for a user to use the same e-coupon more than once. Furthermore, it would be not convenient for the user who actually downloads the e-coupon to give out a duplicate to another person who might use it as first. This security problem is commonly referred to as the *double-spending* problem.

Our system should also contemplate the possibility that the advertisers collude with users. An advertiser might for example release a postdated e-coupon or alter the e-coupon value or validity period in order to advantage the user.

A system for e-coupon generation should protect the merchant against the above described misconduct of advertisers and users. Hence, a system for distributing e-coupons should verify the following security requirements.

- **Advertisers' Loyalty:** The merchant should be able to determine whether the e-coupon has been validly released by an advertiser on her behalf. In other words, the merchant should be able to verify whether the e-coupon has been released by the merchant according to the merchant's specifications.
- **Users' Integrity:** The merchant should be able to detect whether the e-coupon has been released by an authorized advertiser on her behalf and/or whether the e-coupon data have not been manipulated by the user.
- **Double Spending Freeness:** The merchant should avoid e-coupons' double spending.

In the dynamic setting a system for e-coupon generation should also verify the following security requirement.

- **E-coupon Postdating Freeness:** The merchant should detect whether an e-coupon carries an illegal emission date.

3 A Protocol for E-coupon

Our scenario contemplates ℓ merchants, say $M_1, \ldots, M_\ell$, m advertisers, say $A_1, \ldots, A_m$, and n users, say $U_1, \ldots, U_n$. For $i = 1, \ldots, \ell$ and $j = 1, \ldots, m$ merchant M_i stipulates a contract with advertiser A_j which authorizes A_j to release e-coupons on M_i's behalf. Users visiting the advertisers' sites can download the e-coupons to purchase goods at M_i's web site.

E-coupon data authentication is provided by *message authentication codes (MACs)*, whose definition is given below.

Definition 1. *A message authentication code (MAC) algorithm is a family of functions h_k parameterized by a secret key k, verifying the following properties:*

1. ease of computation*: for a known function h_k, given a value k and an input x, $h_k(x)$ is easy to compute.*
2. compression*: h_k maps an input x of arbitrary finite bitlength to an output $h_k(x)$ of finite bitlength n.*
   Furthermore, given a description of the function family h, for every fixed allowable value of k (unknown to an adversary), the following property holds:

3. computational resistance*: given zero or more text-MAC pairs* $(x_i, h_k(x_i))$*, it is computationally infeasible to compute any text-MAC pair* $(x, h_k(x))$ *for any new input* $x \neq x_i$ *(including possibly for* $h_k(x) = h_k(x_i)$ *for some* i*).*

As an example, our protocol could be implemented by using MD5-MAC or the MAC algorithm based on DES block cipher. We refer the reader to [9] for an extended treatise on MACs.

We assume that for $i = 1, \ldots, \ell$, merchant M_i has been assigned a secret key k_{M_i} which is known only to her, and that for $i = 1, \ldots, \ell$ and $j = 1, \ldots, m$, there is a key k_{M_i,A_j} which is shared by M_i and A_j and is known only to them. For $i = 1, \ldots, \ell$ and $j = 1, \ldots, m$, let $h_{k_{M_i}}$ $h_{k_{M_i,A_j}}$ be two MAC algorithm functions parameterized by k_{M_i} and k_{M_i,A_j}, respectively.

Initialization Phase Merchant M_i provides the advertisers with a "framework" for the e-coupons to be released for a given period of time.

Let M_i be a merchant who is willing to advertise her products on A_j's web site. Merchant M_i releases to A_j an e-coupon framework which will be used by the advertiser to generate the e-coupons. This framework will be the same for both the static and the dynamic models. The framework released by the merchant carries the following information.

a) *m_data*: e-coupon specification data, such as,
   - *M_name*: the identity of the merchant;
   - *P_name*: the name of the promoted good;
   - *O_name*: type, value, and period of validity of the offer (discount on purchased items, three items for the price of two, gifts, samples, etc.);
   - *A_name*: the name of the advertiser.

b) $h_{k_{M_i}}(m\_data)$.

In addition to the above data, the e-coupon released by the advertiser A_j contains further information whose nature depends on the type of e-coupon we are considering.

First we will describe the information added by the advertiser to static e-coupons and then we will show how to extend the protocol for the static model in order to make it work under the dynamic model.

3.1 Static E-coupons

Static E-coupon Generation In the static setting the released e-coupon will carry the following data in addition to those introduced by the merchant.

c) the serial number SN of the coupon;

d) $h_{k_{M_i,A_j}}(m\_data||SN)$.

The serial number increases every time a new user downloads an e-coupon from the site. The serial number SN along with the *m_data* specified by the merchant constitute the e-coupon's significant data, whereas the values $h_{k_{M_i}}(m\_data)$ and $h_{k_{M_i,A_j}}(m\_data||SN)$ are used for security reasons.

Static E-coupon Verification When merchant M_i is presented an e-coupon, then she computes the values $h_{k_{M_i}}(m\_data)$ and $h_{k_{M_i,A_j}}(m\_data||SN)$ and accepts the e-coupon if and only if these values coincide with those stored in the e-coupon.

3.2 Dynamic E-coupons

In the dynamic setting the e-coupon contains information which would allow the merchant to verify its release time.

In addition to $h_{k_{M_i}}$ and $h_{k_{M_i,A_j}}$, the protocol for dynamic e-coupons uses another publicly known function q which is assumed to be a *collision resistant* hash function according to the following definition.

Definition 2. *A hash function $q : D \rightarrow C$ is collision resistant if and only if it is computationally infeasible to find a pair of distinct elements x and y of D such that $q(x) = q(y)$.*

Two examples of popular hash functions used in many practical applications are SHA-1 and MD5 (see [9]).

Dynamic E-coupons Generation In the following we will suppose, for the sake of simplicity, that the e-coupons' serial numbers are consecutive integers. In the dynamic setting, advertiser A_j will introduce the following information into the e-coupon.

c) SN: the serial number of the e-coupon;
d) *time*: the date and time at which the coupon has been downloaded;
e) *u_data*: a piece of information released by the user (e.g., the user's IP address number);
f) $q_{SN} = q(u\_data||q_{SN-1})$;
g) $h_{k_{M_i,A_j}}(m\_data||SN||time||u\_data||q_{SN})$.

Dynamic E-coupon Verification Our protocol assumes that a merchant can ask each advertiser for the *u_data*'s of the e-coupons released in a given time frame. For example, we may assume that every day a merchant obtain from each advertiser the list of the e-coupons released on the previous day. We assume that the *u_data*'s appear in the list in the same order the corresponding e-coupons have been downloaded.

Let $u\_data_1, u\_data_2, \ldots, u\_data_n$ be the list of the *u_data*'s for the e-coupons released in a given time frame by advertiser A_j on M_i's behalf, and let us assume, for the sake of simplicity, that the corresponding serial number be the integers $1, 2, \ldots, n$. Let q_0 denote the value of q computed for the last e-coupon released in the previous time frame. We say that the values $q_0, q_1, \ldots, q_n$ form a *dependence chain*. A merchant verifies the consistency of the sequence $q_0, q_1, \ldots, q_n$ by computing the values $q(u\_data_1, q_0), q(u\_data_2, q_1), \ldots, q(u\_data_n, q_{n-1})$, and by checking, for any $i = 1, \ldots, n$, if $q(u\_data_i, q_{i-1}) = q_i$. If for some i it results $q(u\_data_i, q_{i-1}) \neq q_i$ then we say that q_i violates the dependence chain.

We assume that in every time frame the merchant verifies the consistency of the dependence chain for the e-coupons released in the previous time frame. If the merchants finds a value q_i which violates the dependence chain then she realizes that A_j is dishonest and takes the appropriate countermeasures.

When merchant M_i is presented an e-coupon, she computes the value $h_{k_{M_i}}(m\_data)$ and $h_{k_{M_i,A_j}}(m\_data||SN||time||u\_data||q_{SN})$ and checks whether these values coincide with those stored in the e-coupon. If the given e-coupon has been downloaded in some previous time frame, then the merchant disposes of the dependence chain for the e-coupons downloaded in that time frame, and consequently she can check whether the value q_{SN} in the e-coupon is equal to the corresponding element in that dependence chain. If the performed tests give positive result then the merchant accepts the e-coupon. If the given e-coupon has been downloaded in the same time frame it has been presented for redemption, then M_i performs only the first two tests and accepts the e-coupon if it passes these two tests.

We assume that the merchant be mainly interested in protecting herself against dishonest advertisers rather than in avoiding redemption of a few post-dated e-coupons. Indeed, it is very important for her to detect an advertiser's misconduct which in the long run could seriously damage her advertisement campaign.

Protocol Security In the following we will prove that the above protocol verifies the security requirements settled in Section 2. First we will prove that the protocol satisfies the first three security requirements. These three requirements have to be verified by both the protocol for static e-coupons and that for dynamic e-coupons. Then, we will prove that the protocol for dynamic e-coupons verifies the fourth security requirement, too.

It is easy to see that if the advertisers behave correctly in the sense that they release e-coupons according to the merchant specification, and if the users do not manipulate, duplicate or falsify e-coupons, then the protocol works pro-perly. Indeed, our protocol is such that a merchant accepts e-coupons which have been generated by authorized advertisers according to her specifications and whose significant information has not been altered.

We will show that our protocol satisfies the security requirements if we admit the possibility that advertisers and users cheat.

Advertiser's Loyalty Our protocol allows to detect any attempt by advertisers to release an e-coupon which does not correspond to the merchant specifications. The crucial observation is that the dishonest advertisers need to modify the information inserted by the merchant into the e-coupon framework. Let m_data denote the information inserted by merchant M_i into the e-coupon and suppose that a dishonest advertiser has replaced m_data with a different value m_data^*. Since the advertiser does not know k_{M_i}, then she cannot directly compute $h_{k_{M_i}}(m\_data^*)$. Moreover, since the function $h_{k_{M_i}}$ is computation resistant, then it is computationally infeasible for her to compute $h_{k_{M_i}}(m\_data^*)$

for any value m_data^* different from the original value m_data. Consequently, the dishonest advertiser will introduce in field b) of the e-coupon a value different from $h_{k_{M_i}}(m\_data^*)$. On e-coupon verification, the merchant will compute $h_{k_{M_i}}(m\_data^*)$ and find out that this value is different from the one stored in the e-coupon. Consequently, she will reject the e-coupon.

User's Integrity Our protocol allows the merchant to detect whether one of the two following cases has occurred.

case 1. The e-coupon has not been generated by an authorized advertiser.
case 2. The e-coupon data have been manipulated by the user.

case 1: If the fake e-coupon is identical to a legally released one, in the sense that the values of all significant data in the e-coupon are the same as those of a legally released e-coupon, then it is actually a duplicate of the legal e-coupon. We will discuss later the security of our protocol with respect to duplicates.

Let us consider the case when an unauthorized person generates a fake e-coupon which resembles a legally released e-coupon but which does not contain the same significant data of a legally released e-coupon. The security problem represented by such an illegal e-coupon is the same one represented by a legally released e-coupon which has been manipulated by a dishonest user. For that reason this case can be assimilated to **case 2** and will be treated below.
case 2: Suppose a dishonest user wants to manipulate the information in an e-coupon legally released by advertiser A_j on M_i's behalf. As already observed in the previous paragraph, it is computationally infeasible for a person who does not know k_{M_i} to compute the correct value of $h_{k_{M_i}}$ for the manipulated data. By a similar argument, it is possible to show that it is computationally infeasible for a person who does not know k_{M_i,A_j}, the value of $h_{k_{M_i,A_j}}$ corresponding to the manipulated data. Consequently, our protocol allows to detect any attempt by the user to manipulate either the data specified by M_i or those specified by A_j.

Double Spending Freeness The system protects the merchant against users' double spending attempts by having the advertiser release e-coupons which are identified by serial numbers. The merchant will accept a given e-coupon only from the user which provides it for the first time. In this way a dishonest user cannot benefit more than once of the same e-coupon. Further, she is discouraged from giving out a duplicate of her e-coupon to another user who might use it as first.

The merchant can avoid redeeming the same e-coupon more than once by simply maintaining a record of the e-coupons which have been already redeemed. Every time an e-coupon is presented to the merchant, its serial number is compared with those of the already redeemed e-coupons.

E-coupon Postdating Freeness In the dynamic setting the emission date might be altered by the advertiser to advantage the user who receives the e-coupon (they cooperate to fool the merchant). Indeed, a user who receives a postdated e-coupon can delay the purchase of the promoted good and look for a better priced offer. The serial number can be of help in preventing such a circumstance. Indeed, the serial number of an e-coupon carrying a certain emission date should be smaller than one carrying a later emission date. The advertiser does not know in advance how many users will visit the site at the time she generates the postdated coupon, and, for that reason, she should generate an illegal coupon with a very faraway date to be sure she has enough serial numbers for the users who will visit the site. However, the serial number does not guarantee that a postdated e-coupon will be detected. The only way to prevent the advertiser to postdate a given e-coupon is to introduce in each e-coupon pieces of information depending on previously released e-coupons. This information should be generable only with the cooperation of the users who download the e-coupons. Indeed, suppose that U_c connects to the advertiser's web site before U_d and that the advertiser wants the e-coupon downloaded by user U_c appear as to be downloaded after that of user U_d. In this case the e-coupon of U_c should contain a piece of information depending also on the information released later by U_d. Consequently, the advertiser in order to postdate the e-coupon of U_c must introduce a piece of information which violates the dependence relation from previously downloaded e-coupons. We have denoted the information released by the user with the *u_data*. An information which could be used as *u_data* is the IP address number of the user which downloads the e-coupon from the advertiser's site.

Suppose that an advertiser and a user have colluded in order to postdate an e-coupon downloaded in a given time frame. Assume that the postdated e-coupon has been downloaded soon after the one carrying serial number r at time t_1. Suppose that instead of carrying the serial number $r+1$ and $time = t_1$, the postdated e-coupon carries serial number $SN = s$, with $s > r+1$, and $time = t_2$, with $t_2 > t_1$. If a honest user visits the site at time t with $t_1 < t < t_2$, then the advertiser must give to this user an e-coupon with $time = t$ and a serial number v comprised between r and s. Obviously the value q_s in the illegal e-coupon should depend on the *u_data* value of the e-coupon downloaded at time t. Since these data are not available at time t_1, then the value q_s computed by the advertiser violates the dependence chain. The merchant detects this anomaly when it checks the consistency of the dependence chain. The only possibility which the advertiser has of generating a value q_s which does not violate the dependence chain is to postdate all the e-coupons downloaded between time t_1 and time t_2. In other words the advertiser should convince the merchant that these e-coupons have been downloaded after time t_1. Since the user who downloads the e-coupon at time t is honest, then she should not realize that her e-coupon has been postdated. Consequently, the *time* value of her e-coupon should be equal to t. Suppose that both the dishonest user who has downloaded the postdated e-coupon and the honest user who has downloaded the e-coupon at time t decide to redeem their e-coupons. Then, the merchant will find out that the *time* value of the e-coupon presented by the honest user is anterior to that of the postdated e-coupon. Since this order does not correspond to the order of the *u_data*'s of the two e-coupons in the merchant's *u_data* list, then the merchant will detect the illegal misconduct of the advertiser who released those e-coupons.

4 Implementation

In this Section we will discuss some aspects related to the implementation of our proposed protocol. We are developing a prototype which is based on CGI scripting to perform server side computation and a plugin to perform client side computation. Our proposal has the advantage of reducing as far as possible the changes to the usual client-server communication pattern over Internet.

4.1 Size of an E-coupon

Since the e-coupon is used during the interactions between the customer and the merchant, and the customer and the advertiser, it is important to limit the size of the e-coupon, to avoid a communication overhead for the user. In the following we analyze the size of an e-coupon constructed according to the requirements described in Section 2.

We will use a MAC algorithm based on a DES block cipher to implement the functions $h_{k_{M_i}}$ and $h_{k_{M_i,A_j}}$. Both keys k_{M_i} and k_{M_i,A_j} are 56-bit DES key. In the static case, each e-coupon will be composed of the *m_data* which can be of arbitrary length, and of a sequence of 160 bit containing: the results of the computation of the two hash functions $h_{k_{M_i}}$ and $h_{k_{M_i,A_j}}$, each one 64 bit long and the serial number SN (32 bit long). Assuming 100 bytes for each field constituting the *m_data* (i.e., *M_name*, *A_name*, *P_name*, *O_name*), the total length of a static e-coupon is approximately of 420 bytes.

A dynamic e-coupon will also have 6 bytes for the generation time and 4 bytes containing the user data (the user's IP address). A dynamic e-coupon will have then 30 bytes in addition to those of a static one, for a total length of approximately 450 bytes.

4.2 Advertiser-Merchant Interaction

In our model the interaction between the advertiser and the merchant occurs at the beginning of an advertisement campaign during the inizialitazion phase of the protocol. During this phase, the merchant has to communicate to the advertiser the e-coupon specification as described in Section 3. Such communication can be done off-line or using some secured channel. After this communication, the advertiser is able to set all the parameters for the e-coupon generation.

In the dynamic setting, eventually the advertiser and the merchant interact also during the advertisement campaign. Indeed, in order to increase the control on the advertiser's activity, the merchant can split the duration of the advertisement campaign in several time frames (e.g. one day), at the end of which the advertiser is obliged to provide the log file relative to the generated e-coupons. The merchant can use that information to reconstruct the generation process and to recognize any possible postdating attempt by the advertiser. The verification is performed off-line, by examining the log file relative to the elapsed time frame, and following the e-coupon dependence chain to control the correctness of its generation with respect to the received e-coupons.

4.3 Customer-Advertiser Interaction

The interaction between the customer and the advertiser is the crucial point when the exchange of the e-coupon occurs. In our model, we assume that the interaction between the client and the advertiser is started when the customer chooses to accept an attractive offer hosted in the advertiser's site. In this case the e-coupon is presented as a link (or a banner) which contains a particular mime-type which activates the client plugin. The plugin should be freely distributed and downloadable avoiding any registration phase; indeed customers could be discouraged to install a plugin if personal data should be given to have access to the plugin code. After clicking on an e-coupon link the e-coupon plugin is executed with the task to store the e-coupon data in a file which is hosted on the client's hard disk, constituting the customer's wallet.

On the other side, the advertiser's CGI script generates the e-coupon by collecting all the data relative to the e-coupon (the updated serial number, the m_data and the IP address contained in the client http request), calculates the hash values of both the data and the dependence chain, and stores the user data in a file which successively would be communicated to the merchant for the verification.

4.4 Customer-Merchant Interaction

The customer interacts with the merchant to spend an e-coupon contained in its own wallet. When the customer decides to use one of the e-coupons contained in his wallet to purchase some merchandise, the redemption phase is started. The e-coupon has the address of the web site where the purchase can be made, and extra data. This phase involves the sending of data from the customer to the merchant, who must perform some extra computation to verify that the e-coupon is valid and that the purchase can be finalized.

On the client side, e-coupons are stored in a local file which is used by the browser to visualize all the e-coupons contained in the wallet. The file which is generated by the plugin contains html code for each e-coupon, which is then represented by the URL of the merchant plus extra data constituting the body of the e-coupon. Since no further computation is requested on the client side, it is not necessary to activate the plugin, but the data which are contained in the e-coupon are sent to the merchant through an HTTP GET request. Whenever the customer wants to redeem an e-coupon he clicks on the e-coupon establishing a connection between the browser and the merchant site. The CGI script running on the merchant site at the URL contained in the e-coupon has the task to accept the e-coupon, to verify its validity and eventually finalize the purchase applying the conditions of the offer. The verification in this phase aims to control the well-formedness of the e-coupon sent by the customer. An additional verification phase is conceived to control the advertiser's activity in the process of the e-coupon generation. This kind of verification can be performed off-line, and the validity of a received e-coupon can be controlled by simulating the generation of e-coupons, following the dependence chain provided by the merchant and controlling that all the received e-coupons have been correctly constructed.

5 Some Conclusions

In this paper, we provided a suitable model for e-coupons such that a number of security assumptions can be done on their generation and distribution. We presented also two protocols which respect the security requirements and a viable implementation. Our model is deliberately lightweight with respect to other proposal [7] which make use of digital signature. The mechanisms ruling the coupon generation and redemption cannot represent a substantial overhead which could prevent the interested users to subscribe a commercial offer. On the other hand, our model shows the same security features which guarantees each other against malicious behavior of each participant to the protocol. Furthermore, no registration phase is requested, such that customers have not to disclose their personal data.

We are considering some improvements to our protocol, in order to make it work in a more general context. In particular we are concerned with the problem of verifying the e-coupon generation time. Indeed, in the dynamic setting, our protocol relies on the fact that at least one honest user redeems a legally released e-coupon. However, we may assume that a merchant might play the role of a honest user who downloads e-coupons from the advertisers' sites at random times. In order to relax such assumption, we are considering the application of timestamping techniques [6] to improve the mechanism of e-coupon generation, so that both customers and merchants increase their confidence in the advertisers' activity.

References

1. R. Anand, M. Kumar, and A. Jhingran. Distributing e-coupons on the internet. In *9th Conference on Internet Society (INET '99)*, San Jose, 1999. 371
2. R. Anand, M. Kumar, A. Jhingran, and R. Mohan. Sales promotions on the internet. In *Third Usenix Workshop on E-Commerce*, Boston, 1998. 371
3. C. Blundo, A. De Bonis, and B. Masucci. Metering schemes with pricing. In M. Herlihy, editor, *14th International Conference on Distributed Computing (DISC 2000)*, volume 1914 of *Lecture Notes in Computer Science*, Toledo, 2000. Springer-Verlag, Berlin. 371
4. M. Franklin and D. Malkhi. Auditable metering with ligthweigth security. In R. Hirschfeld, editor, *Financial Criptography (FC '97)*, volume 1318 of *Lecture Notes in Computer Science*, pages 151–160. Springer-Verlag, Berlin, 1997. 370

5. New York Times (C. Greenman). The trouble with rebates. `http://www.nytimes.com/library/tech/98/09/cyber/articles/13advertising.html`, September 16, 1999. 371
6. S. Haber and W. S. Stornetta. How to time-stamp a digital document. *Journal of Cryptology*, 3(2):99–111, 1991. 382
7. M. Jakobsson, P. D. MacKenzie, and J. P. Stern. Secure and ligthweight advertising on the web. In *9th World Wide Web Conference (WWW9)*, 1999. 371, 382
8. IBM India Research Lab. E-coupons. `http://www.research.ibm.com/irl/projects/ecoupons/`. 371
9. A. Menezes, P. van Oorschot, and S. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996. 375, 376
10. M. Naor and B. Pinkas. Secure and efficient metering. In K. Nyberg, editor, *International Conference on the Theory and Application of Cryptographic Techniques (Eurocrypt '98)*, volume 1403 of *Lecture Notes in Computer Science*, pages 576–590, Espoo, Finland, 1998. Springer-Verlag, Berlin. 371
11. New York Times (M. Slatalla). Turning coupon users from clippers into clickers, April 1, 1999. 371
12. New York Times (B. Tedeschi). Is coupon clicking the next advertising trend?, May 12, 1998. 371

A Calculus and Complexity Bound for Minimal Conditional Logic

Nicola Olivetti[1] and Camilla B. Schwind[2]

[1] Dipartimento di Informatica - Università di Torino
Italy
olivetti@di.unito.it
[2] LIM, Faculté des Sciences de Luminy
Marseille, France
schwind@lim.univ-mrs.fr

Abstract. In this paper, we introduce a cut-free sequent calculus for minimal conditional logic **CK** and three extensions of it: namely, with ID, MP and both of them. The calculus uses labels and transition formulas and can be used to prove decidability and space complexity bounds for the respective logics. As a first result, we show that **CK** can be decided in $O(n \log n)$ space.

1 Introduction

Conditional logics have a long history. They have been studied first by Lewis [23,24,2,27] to formalize a kind of hypothetical reasoning (if A were the case then B) that cannot be captured by classical logic with its material implication. More recently, they have been rediscovered in computer science and artificial intelligence for their potential application in a number of areas (see [4]), such as knowledge representation, non-monotonic reasoning, deductive databases, and natural language semantics. For instance, in knowledge representation, conditional logics have been used to represent and reason about prototypical properties [14], the have been used to model knowledge and database update [18], belief revision [1,15,16]. They can provide an axiomatic foundation of non-monotonic reasoning [21], as it turns out that all forms of inferences studied in the framework of non-monotonic logics are particular cases of conditional axioms [5]. Causal inference, which is very important for applications in action planning [26], has been modelled by conditional logics. They have been used to model hypothetical queries in deductive databases and logic programming; for instance conditional logic **CK**+ID is the base of the logic programming language defined in [12]. In system diagnosis, conditional logics can been used to reason hypothetically about the expected functioning of system components with respect to the observed faults [14]. Finally, they find obvious applications to the semantics of natural language, in particular to give a formal treatment of hypothetical and counterfactual sentences (those conditionals whose antecedent is known to be false) [24].

A. Restivo, S. Ronchi Della Rocca, L. Roversi (Eds.): ICTCS 2001, LNCS 2202, pp. 384–404, 2001.

In spite of their significance, very few proof systems have been proposed for conditional logics: we just mention [22,19,3,17,13,7]. One possible reason of the underdevelopment of proof-methods for conditional logics is the lack of a universally accepted semantics for them. This is in sharp contrast, for instance, with modal or temporal logics logics which have a consolidated semantics based on a standard kind of Kripke structures.

Similarly to modal logics, the semantics of conditional logics can be defined in terms of possible world structures. In this respect, conditional logics can be seen as a generalization of modal logics (or a type of multi-modal logic) where the conditional operator is a sort of modality indexed by a formula of the same language. The two most popular semantics for conditional logics are the so-called *sphere semantics* [23] and *selection function semantics* [24]. Both are possible-world semantics, but are based on different (though related) algebraic notions.

For our purposes, selection function semantics appear to be simpler and it is also more general. In this approach, truth values are assigned to formulas depending on a world; intuitively, the selection function f selects, for a world w and a formula A, the set of worlds $f(w, A)$ which are "most-similar to w" or "closer to w" given the information A. In *normal* conditional logics, the function f depends on the set of worlds satisfying A rather than on A itself, so that $f(w, A) = f(w, A')$ whenever A and A' are true in the same worlds. A conditional sentence $A \Rightarrow B$ is true in w whenever B is true in every world selected by f for A and w.

In this paper we propose a sequent calculus for the minimal normal conditional logic **CK** and its extensions with the axioms/semantic conditions ID and MP.

In our sequent calculi we use labels to represent worlds within the selection function semantics. Their completeness is an immediate property of cut-adimissibility property. *Explicit* or labelled proof systems have been provided for a wide range of modal and substructural logics and go back at least to Fitting's tableaux for modal logics [8]. A systematic development of labelled proof systems have been proposed in [28] and [11]. However the development of this kind of proof systems for conditional logics, with the exception of [17], is still unexplored.

To the best of our knowledge, this is the first analytic proof method for **CK** and the mentioned extensions. We begin a complexity analysis of these logics, by showing that one can get decidability and complexity bounds for **CK** by an entirely proof-theoretical argument, based on contraction elimination. In particular we show that **CK** is decidable in polynomial space, namely in $O(n \log n)$ space.

2 Systems of Propositional Conditional Logic

Conditional logic is an extension of propositional logic by the conditional operator $\Rightarrow$. We consider a propositional language $\mathcal{L}$ over a set of propositional variables ATM. Formulas of $\mathcal{L}$ are built from propositional variables by means of the boolean operators $\rightarrow, \bot$ and the conditional operator $\Rightarrow$. The other boolean operators are defined by the usual equivalences.

As explained above, we adopt here the so-called propositional selection function semantics [24].

Definition 1 (CK semantics). *A selection function model for* $\mathcal{L}$ *is a triple* $\mathcal{M} =< \mathcal{W}, f, [\,] >$, *where*

- $\mathcal{W}$*is a non-empty set of items called* worlds*,*
- f *is a function of type* $f : \mathcal{W} \times 2^{\mathcal{W}} \longrightarrow 2^{\mathcal{W}}$*;* f *is called the* selection *function,*
- $[\,]$ *is an evaluation function of type* $ATM \longrightarrow 2^{\mathcal{W}}$*.*

$[\,]$ *assigns to an atom* p *the set of world where* p *is true. Hence,* $w \in [p]$ *means that atom* p *is true in the world* w*. The evaluation function* $[\,]$ *can be extended to every formula by means of the following inductive clauses.*

$$[\bot] = \emptyset,$$
$$[A \rightarrow B] = (\mathcal{W} - [A]) \cup [B]^{1},$$
$$[A \Rightarrow B] = \{w \in \mathcal{W} \mid f(w, [A]) \subseteq [B]\}.$$

We say that a formula A *is* valid *in a model* $\mathcal{M}$ *as above if* $[A] = \mathcal{W}$*. A formula* A *is* valid *(denoted by* $\models_{CK} A$*) if it is valid in every model* $\mathcal{M}$*.*

The above is the semantics of the basic conditional logic **CK**, where no specific properties are assumed on the selection function f. Notice that the value of f depends on the set of worlds satisfying a formula, rather than on the formula itself, i.e. f is defined for w and $[A]$ rather than w and A. Consequently, we have $f(w, [A]) = f(w, [A'])$, whenever $A \leftrightarrow A'$ is valid in $\mathcal{M}$. This requisite is called normality. Logic **CK** is the minimal normal conditional logic and has the same role as modal logic K in the family of normal modal logics. **CK** is axiomatized by considering the following axioms and rules :

- all tautologies of classical propositional logic.
- (Modus Ponens) $\dfrac{A \quad A \rightarrow B}{B}$
- (RCEA) $\dfrac{A \leftrightarrow B}{(A \Rightarrow C) \leftrightarrow (B \Rightarrow C)}$
- (RCK) $\dfrac{(A_1 \wedge \ldots \wedge A_n) \rightarrow B}{(C \Rightarrow A_1 \wedge \ldots \wedge C \Rightarrow A_n) \rightarrow (C \Rightarrow B)}$

Logical inference is defined as usual and is denoted by $\vdash_{CK}$. **CK** is sound and complete with respect to the selection function semantics [24].

Theorem 1 (Nute [24]). $\vdash_{CK} A$ *if and only if* $\models_{CK} A$*.*

[1] By the standard equivalence we get $[\neg A] = \mathcal{W} - [A]$, $[A \wedge B] = [A] \cap [B]$, $[A \vee B] = [A] \cup [B]$.

Moreover, we consider also the following extensions of **CK**, namely **CK**+ID, **CK**+MP, and **CK**+ID+MP. The semantic conditions (MP) and (ID) corresponds to well-known axioms shown in the following table.

System	Axiom	Model condition
(**ID**)	$A \Rightarrow A$	$f(x, [A]) \subseteq [A]$
(**MP**)	$(A \Rightarrow B) \rightarrow (A \rightarrow B)$	$w \in [A] \rightarrow w \in f(w, [A])$

ID represents a usually expected property of the conditional operator[2]: if we make the hypothesis A, then A follows. The corresponding semantic property says that the worlds selected by f according to a formula A actually satisfy A. Following the standard terminology, let us call A-worlds the elements of $[A]$. By ID we can say that $A \Rightarrow B$ is true at w if B holds in the subset of A-worlds that are "closest" to w. Property MP, called the conditional modus ponens, means that if the w satisfies the hypothesis A it will be among the closest worlds to itself according to A. Logic **CK**+MP is called *weakly material* by Nute [24].

Theorem 1 extends to **CK**+ S, where $S \subseteq \{(ID), (MP)\}$.

3 Sequent Calculus

The sequent calculus that we present in this section makes use of labels to represent possible worlds. In order to express that w' belongs to the set of worlds $f(w, [A])$, we use the expression $w \xrightarrow{A} w'$, which can also be understood as specifying a transition from w to w' labelled by formula A[3].

Let us fix a language $\mathcal{L}$ and a denumerable alphabet of labels $\mathcal{A}$ whose elements are denoted by $x, y, z \ldots$. The constituents of sequents are expressions of the form $x : A$, called *labelled sentences*, whose meaning is that A holds in a world x, and $x \xrightarrow{B} y$, called *transition formulas* (or just transitions) whose meaning is that $y \in f(x, [B])$, where $x, y \in \mathcal{A}, A, B \in \mathcal{L}$. We call *labelled formulas* both kind of expressions and we use metavariable F, G with possible subscripts, to denote them.

The intuitive meaning of $\Gamma \vdash \Delta$ is: every model that satisfies all labelled formulas of Γ in the respective worlds (specified by the labels) satisfies at least one of the labelled formulas of Δ (in those worlds). This is made precise by the notion of *validity* of a sequent as given in the next definition. We use the notation **CK**+ S, where $S \subseteq \{ID, MP\}$, to refer to **CK** and its extensions.

[2] However, this property is not assumed for instance in the causal interpretation of $A \Rightarrow B$ (roughly speaking A causes B) [26]. It can also be questioned in the *belief revision* interpretation of the conditional: $A \Rightarrow B$ means B follows from the knowledge of the agent revised by A [10].

[3] The use of transition formulas makes apparent some connection of conditional logic with dynamic logics of programs: both are extensions of modal logic which can model state (or world) transitions, but whereas in the latter the transitions are associated to *actions* or program statements, in the former the transitions are associated to *propositions*.

Definition 2 (CK+S-validity). *Given a* **CK**+S *model* $\mathcal{M} = \langle \mathcal{W}, f, [\,] \rangle$ *for* $\mathcal{L}$, *and a label alphabet* $\mathcal{A}$, *we consider any* mapping $I : \mathcal{A} \rightarrow \mathcal{W}$. *Let* F *be a labelled formula, we define* $\mathcal{M} \models_I F$ *as follows*

$$\mathcal{M} \models_I x : A \textit{ iff } I(x) \in [A] \quad \textit{and} \quad \mathcal{M} \models_I x \xrightarrow{A} y \textit{ iff } I(y) \in f(I(x), [A])$$

We say that $\Gamma \vdash \Delta$ *is* valid *in* $\mathcal{M}$ *if for every mapping* $I : \mathcal{A} \rightarrow \mathcal{W}$, *if* $\mathcal{M} \models_I F$ *for every* $F \in \Gamma$, *then* $\mathcal{M} \models_I G$ *for some* $G \in \Delta$. *We say that* $\Gamma \vdash \Delta$ *is* **CK**-valid *if it is valid in every* $\mathcal{M}$.

We give a sequent calculus **Seq**S, where $S \subseteq \{ID, MP\}$, for **CK** and its extensions with ID and MP. A *sequent* is a pair of multi-sets of labelled formulas $\langle \Gamma, \Delta \rangle$, denoted by $\Gamma \vdash \Delta$. As usual, Γ, A is an abbreviation for $\Gamma \cup \{ A \}$ and Γ, Γ' for $\Gamma \cup \Gamma'$, where $\cup$ is multiset union. The calculi **Seq**S comprise the axioms and the rules of figure 1. More precisely **SeqCK** contains the axioms **(AX)**,**(A$\bot$)**, the structural rules **(WeakL)**, **(WeakR)**, **(ContrL)**,**(ContrR)**, the logical rules **($\rightarrow$ L)**,**($\rightarrow$ R)**, **($\Rightarrow$ L)**, **($\Rightarrow$ R)**, and the transition rule **(EQ)** and

- **Seq**ID contains in addition the transition rule **(ID)**
- **Seq**MP contains in addition the transition rule **(MP)**
- **Seq**ID+MP contains both **(ID),(MP)**.

The rules for the other boolean operators can be derived from the rules/axioms for $\rightarrow$ and $\bot$ as usual.

Example 1. We show a derivation of the (ID) axiom.

$$\cfrac{\cfrac{y : A \vdash y : A}{x \xrightarrow{A} y \vdash y : A}\,(ID)}{\vdash x : A \Rightarrow A}\,(\Rightarrow L)$$

Example 2. We show a derivation of the (MP) axiom.

$$\cfrac{\cfrac{\cfrac{\cfrac{x : A \vdash x : A}{x : A \vdash x \xrightarrow{A} x}\,(MP) \qquad x : B \vdash x : B}{x : A \Rightarrow B, x : A \vdash x : B}\,(\Rightarrow L)}{x : A \Rightarrow B \vdash x : A \rightarrow B}}{\vdash x : (A \Rightarrow B) \rightarrow (A \rightarrow B)}$$

The calculi **Seq**S are sound and complete with respect to the semantics.

Theorem 2 ((Soundness)). *If* $\Gamma \vdash \Delta$ *is derivable in* **Seq**S, *where* $S \subseteq \{(ID), (MP)\}$ *then it is valid in the corresponding system.*

Proof. By induction on the height of a derivation of $\Gamma \vdash \Delta$. As an example, we examine the cases of ($\Rightarrow R$) and (MP). The other cases are left to the reader.

$$(\mathbf{AX})\ \Gamma, F \vdash \Delta, F \qquad\qquad (\mathbf{A\bot})\ \Gamma, x:\bot \vdash \Delta$$

$$(\mathbf{ContL})\ \frac{\Gamma, F, F \vdash \Delta}{\Gamma, F \vdash \Delta} \qquad\qquad (\mathbf{ContR})\ \frac{\Gamma \vdash \Delta, F, F}{\Gamma \vdash \Delta, F}$$

$$(\mathbf{WeakL})\ \frac{\Gamma \vdash \Delta}{F, \Gamma \vdash \Delta} \qquad\qquad (\mathbf{WeakR})\ \frac{\Gamma \vdash \Delta}{\Gamma \vdash \Delta, F}$$

$$(\rightarrow \mathbf{L})\ \frac{\Gamma, x:A \vdash x:B, \Delta}{\Gamma \vdash x:A \rightarrow B, \Delta} \qquad\qquad (\rightarrow \mathbf{R})\ \frac{\Gamma \vdash x:A, \Delta \quad \Gamma, x:B \vdash \Delta}{\Gamma, x:A \rightarrow B \vdash \Delta}$$

$$(\mathbf{EQ})\ \frac{u:A \vdash u:B \qquad u:B \vdash u:A}{\Gamma, x \xrightarrow{A} y \vdash x \xrightarrow{B} y, \Delta}$$

$$(\Rightarrow \mathbf{L})\ \frac{\Gamma \vdash x \xrightarrow{A} y, \Delta \qquad \Gamma, y:B \vdash \Delta}{\Gamma, x:A \Rightarrow B \vdash \Delta} \qquad\qquad (\Rightarrow \mathbf{R})\ \frac{\Gamma, x \xrightarrow{A} y \vdash y:B, \Delta}{\Gamma \vdash x:A \Rightarrow B, \Delta}\ (y \notin \Gamma, \Delta)$$

$$(\mathbf{ID})\ \frac{\Gamma, y:A \vdash \Delta}{\Gamma, x \xrightarrow{A} y, \vdash \Delta} \qquad\qquad (\mathbf{MP})\ \frac{\Gamma \vdash x:A, \Delta}{\Gamma \vdash x \xrightarrow{A} x, \Delta}$$

Fig. 1. Sequent rules for **CK**(+ID+MP)

($\Rightarrow$ R) Let $\Gamma \vdash \Delta, x : A \Rightarrow B$ be derived from (1) $\Gamma, x \xrightarrow{A} y \vdash \Delta, y : B$, where y does not occur in Γ, Δ and it is different from x. By induction hypothesis we know that the latter sequent is valid. Suppose the former is not, and that it is not valid in a model $\mathcal{M} = \langle \mathcal{W}, f, [\ ] \rangle$, via a mapping I, so that we have:

$$\mathcal{M} \models_I F \text{ for every } F \in \Gamma,\ \mathcal{M} \not\models_I F \text{ for any } F \in \Delta \text{ and } M \not\models_I x : A \Rightarrow B.$$

As $M \not\models_I x : A \Rightarrow B$ there exists $w \in f(I(x), [A]) - [B]$. We can define an interpretation $I'(z) = I(z)$ for $z \neq x$ and $I'(z) = w$. Since y does not occur in Γ, Δ and is different from x, we have that $\mathcal{M} \models_{I'} F$ for every $F \in \Gamma$, $\mathcal{M} \not\models_{I'} F$ for any $F \in \Delta$, $\mathcal{M} \not\models y : B$ and $\mathcal{M} \models_{I'} x \xrightarrow{A} y$, against the validity of (1).
(MP) Let $\Gamma \vdash \Delta, x \xrightarrow{A} x$ be derived from (2) $\Gamma \vdash \Delta, x : A$. Let (2) be valid and let $\mathcal{M} = \langle \mathcal{W}, f, [\ ] \rangle$ be a model satisfying the MP condition. Suppose that for one mapping I, $\mathcal{M} \models_I F$ for every $F \in \Gamma$, then by the validity of (2) either $\mathcal{M} \models_I G$ for some $G \in \Delta$, or $\mathcal{M} \models_I x : A$. In the latter case, we have $I(x) \in [A]$, thus $I(x) \in f(I(x), [A])$, by MP, this means that $\mathcal{M} \models_I x \xrightarrow{A} x$.

Completeness is an easy consequence of the admissibility of cut. By cut we mean the following rule:

$$\frac{\Gamma \vdash \Delta, F \quad F, \Gamma \vdash \Delta}{\Gamma \vdash \Delta}\ cut$$

where F is any labelled formula. To prove cut adimissibility, we need the following lemma about label substitution.

Lemma 1. *If a sequent $\Gamma \vdash \Delta$ has a derivation of height h, then $\Gamma[x/y] \vdash \Delta[x/y]$ has a derivation of height h, where $\Gamma[x/y] \vdash \Delta[x/y]$ is the sequent obtained from $\Gamma \vdash \Delta$ by replacing a label x by a label y wherever it occurs.*

Proof. By a straightforward induction on the height of a derivation.

Theorem 3. *If $\Gamma \vdash \Delta, F$ and $F, \Gamma \vdash \Delta$ are derivable, so is $\Gamma \vdash \Delta$.*

Proof. As usual, the proof proceeds by a double induction over the complexity of the cut formula and the sum of the heights of the derivations of the two premises of the cut inference, in the sense that we replace one cut by one or several cuts on formulas of smaller complexity, or on sequents derived by shorter derivations. We have several cases: (i) one of the two premises is an axiom, (ii) the last step of *one* of the two premises is obtained by a rule in which F is *not* the principal formula[4], (iii) F is the principal formula in the last step of *both* derivations.

(i) If one of the two premises is an axiom then either $\Gamma \vdash \Delta$ is an axiom, or the premise which is not an axiom contains two copies of F and $\Gamma \vdash \Delta$ can be obtained by contraction.

(ii) We distinguish two cases: the sequent where F is not principal is derived by any rule (R), except the (EQ) rule. This case is standard, we can permute (R) over the cut: i.e. we cut the premise(s) of (R) and then we apply (R) to the result of cut. If one of the sequents, say $\Gamma \vdash \Delta, F$ is obtained by the (EQ) rule, where F is not principal, then also $\Gamma \vdash \Delta$ is derivable by the (EQ) rule and we are done.

(iii) F is the principal formula in both the inferences steps leading to the two cut premises. There are six subcases: F is introduced by (a) a classical rule, (b) by $(\Rightarrow L), (\Rightarrow R)$, (c) by (EQ), (d) F by (EQ) on the left and by (ID) on the right, (e) by (MP) on the left and by (EQ) on the right, (f) by (ID) on the left and by (MP) on the right. The list is exhaustive.

(a) This case is standard and left to the reader.

(b) $F = x : A \Rightarrow B$ is introduced by $(\Rightarrow R)$ and $(\Rightarrow L)$. Then we have

$$\dfrac{\dfrac{(*)\, \Gamma, x \xrightarrow{A} z \vdash z : B, \Delta}{\Gamma \vdash x : A \Rightarrow B, \Delta}\,(\Rightarrow R) \qquad \dfrac{\Gamma \vdash x \xrightarrow{A} y, \Delta \quad \Gamma, y : B \vdash \Delta}{\Gamma, x : A \Rightarrow B \vdash \Delta}\,(\Rightarrow L)}{\Gamma \vdash \Delta}\,(cut)$$

where z does not occur in Γ, Δ and $z \neq x$; By lemma 1, we obtain that

$$\Gamma, x \xrightarrow{A} y \vdash y : B, \Delta$$

is derivable by a derivation of no greater height than (*); thus we can

[4] The principal formula of an inference step is the formula introduced by the rule applied in that step.

replace the cut as follows

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash x \xrightarrow{A} y, \Delta}{\Gamma \vdash x \xrightarrow{A} y, \Delta, y : B}\,(WeakR) \qquad \Gamma, x \xrightarrow{A} y \vdash y : B, \Delta}{\Gamma \vdash \Delta, y : B}\,(cut) \qquad \Gamma, y : B \vdash \Delta}{\Gamma \vdash \Delta}\,(cut)$$

The upper cut uses the induction hypothesis on the height, the lower the induction hypothesis on the complexity of the formula.

(c) $F = x \xrightarrow{B} y$ is introduced by (EQ) in both premises, we have where

$$\dfrac{\dfrac{(5)\, u : A \vdash u : B \quad (6)\, u : B \vdash u : A}{\Gamma', x \xrightarrow{A} y \vdash x \xrightarrow{B} y, \Delta'}\,(EQ) \qquad \dfrac{(7)\, u : B \vdash u : C \quad (8)\, u : C \vdash u : B}{\Gamma', x \xrightarrow{B} y \vdash x \xrightarrow{C} y, \Delta'}\,(EQ)}{\Gamma', x \xrightarrow{A} y \vdash x \xrightarrow{C} y, \Delta'}\,(cut)$$

$\Gamma = \Gamma', x \xrightarrow{A} y$, $\Delta = x \xrightarrow{C} y, \Delta'$. (5)-(8) have been derived by a shorter derivation; thus we can replace the cut by cutting (5) and (7) on the one hand, and (8) and (6) on the other, which give respectively

(9) $u : A \vdash u : C$ and (10) $u : C \vdash u : A$.

Using (EQ) we obtain $\Gamma', x \xrightarrow{A} y \vdash x \xrightarrow{C} y, \Delta'$

(d) $F = x \xrightarrow{B} y$ is introduced on the left by (EQ) rule and it is introduced on the right by (ID). Thus we have

$$\dfrac{\dfrac{u : A \vdash u : B \quad u : B \vdash u : A}{\Gamma', x \xrightarrow{A} y \vdash \Delta, x \xrightarrow{B} y}\,(EQ) \qquad \dfrac{\Gamma', x \xrightarrow{A} y, y : B \vdash \Delta}{x \xrightarrow{B} y, \Gamma', x \xrightarrow{A} y \vdash \Delta}\,\text{(ID)}}{\Gamma', x \xrightarrow{A} y \vdash \Delta}\,\text{(cut)}$$

where $\Gamma = \Gamma', x \xrightarrow{A} y$. By lemma 1 and weakening, the sequent $\Gamma', x \xrightarrow{A} y, y : A \vdash y : B, \Delta$ can be derived by a derivation of the same height as $u : A \vdash u : B$. Thus, the cut is replaced as follows

$$\dfrac{\dfrac{\dfrac{\Gamma', x \xrightarrow{A} y, y : A \vdash y : B, \Delta \quad \Gamma', x \xrightarrow{A} y, y : B \vdash \Delta}{\Gamma', x \xrightarrow{A} y, y : A \vdash \Delta}\,(cut)}{\Gamma', x \xrightarrow{A} y, x \xrightarrow{A} y \vdash \Delta}\,\text{(ID)}}{\Gamma', x \xrightarrow{A} y, \vdash \Delta}\,\text{(ContrL)}$$

(e) $F = x \xrightarrow{A} y$ is introduced on the left by (MP) rule and it is introduced on the right by (EQ). Thus we have

$$\dfrac{\dfrac{\Gamma \vdash x : A, \Delta'}{\Gamma \vdash \Delta', x \xrightarrow{A} x}\,(\mathrm{MP}) \qquad \dfrac{u : A \vdash u : B \quad u : B \vdash u : A}{\Gamma, x \xrightarrow{A} x \vdash \Delta', x \xrightarrow{B} x}\,(EQ)}{\Gamma \vdash \Delta', x \xrightarrow{B} x}\,(\mathrm{cut})$$

where $\Delta = \Delta', x \xrightarrow{B} x$. By lemma 1 and weakening, the sequent $\Gamma, x : A \vdash x : B, \Delta$ can be derived by a derivation of the same height as $u : A \vdash u : B$. Thus the cut is replaced as follows:

$$\dfrac{\dfrac{\Gamma \vdash x : A, \Delta' \quad \Gamma, x : A \vdash \Delta', x : B}{\Gamma \vdash \Delta', x : B}\,(cut)}{\Gamma \vdash \Delta', x \xrightarrow{B} x}\,(\mathrm{MP})$$

(f) $F = x \xrightarrow{A} y$ is introduced on the right by (MP) rule and on the left by (ID). Thus we have

$$\dfrac{\dfrac{\Gamma \vdash x : A, \Delta}{\Gamma \vdash \Delta, x \xrightarrow{A} x}\,(MP) \qquad \dfrac{\Gamma, x : A \vdash \Delta}{\Gamma, x \xrightarrow{A} x \vdash \Delta}\,(ID)}{\Gamma \vdash \Delta}\,(\mathrm{cut})$$

We replace this cut by the following:

$$\dfrac{\Gamma \vdash x : A, \Delta \quad \Gamma, x : A \vdash \Delta}{\Gamma \vdash \Delta}\,(cut)$$

Theorem 4 ((Completeness)). *If A is valid in* **CK**+*S, where $S \subseteq \{(ID), (MP)\}$ then $\vdash x : A$ is derivable in* **Seq**S.

Proof. We must show that the axioms are derivable and that the set of derivable formulas is closed under (Modus Ponens), (RCEA), and (RCK). A derivation of axioms (ID) and (MP) is shown in examples 1 and 2 respectively.
(Modus Ponens) suppose that $\vdash x : A \rightarrow B$ and $\vdash x : A$ are derivable. We easily have that $x : A \rightarrow B, x : A \vdash x : B$ is derivable too. Since cut is admissible, by two cuts we obtain $\vdash x : B$.
(RCEA), we have to show that if $A \leftrightarrow B$ is derivable, then also $(A \Rightarrow C) \leftrightarrow (B \Rightarrow C)$ is so. The formula $A \leftrightarrow B$ is an abbreviation for $(A \rightarrow B) \wedge (B \rightarrow A)$. Suppose that $\vdash x : A \rightarrow B$ and $\vdash x : B \rightarrow A$ are derivable, we can derive $x : B \Rightarrow C \vdash x : A \Rightarrow C$ as follows: (the other half is symmetrical).

$$\dfrac{\dfrac{\dfrac{x : A \vdash x : B \quad x : B \vdash x : A}{x \xrightarrow{B} y \vdash x \xrightarrow{A} y, y : C}\,(EQ) \qquad x \xrightarrow{B} y, y : C \vdash y : C}{x \xrightarrow{B} y, x : A \Rightarrow C \vdash y : C}\,(\Rightarrow L)}{x : A \Rightarrow C \vdash x : B \Rightarrow C}\,(\Rightarrow R)$$

(RCK), Suppose that (1) $\vdash x : B_1 \wedge B_2 \ldots \wedge B_n \rightarrow C$ is derivable, $x : B_1, \ldots, x : B_n \vdash x : C$ must be derivable too. We set $\Gamma_i = x : A \Rightarrow B_i, x : A \Rightarrow B_{i+1}, \ldots x : A \Rightarrow B_n$, for $1 \leq i \leq n$. Then we have (we omit side formulas in the sequent $x \xrightarrow{A} y \vdash x \xrightarrow{A} y$)

$$\dfrac{x \xrightarrow{A} y \vdash x \xrightarrow{A} y \quad x : B_1, \ldots, x : B_n \vdash x : C}{x \xrightarrow{A} y, x : A \Rightarrow B_n, x : B_1, \ldots, x : B_{n-1} \vdash y : C}\ (\Rightarrow L)$$

$$\vdots$$

$$\dfrac{\dfrac{x \xrightarrow{A} y \vdash x \xrightarrow{A} y \qquad\qquad x \xrightarrow{A} y, \Gamma_2, y : B_1 \vdash y : C}{x \xrightarrow{A} y, x : A \Rightarrow B_1, x : A \Rightarrow B_2, \ldots x : A \Rightarrow B_n \vdash y : C}\ (\Rightarrow L)}{x : A \Rightarrow B_1, x : A \Rightarrow B_2, \ldots x : A \Rightarrow B_n \vdash x : A \Rightarrow C}\ (\Rightarrow R)$$

4 Complexity of CK

The sequent calculus we have presented in the previous section can be used to prove the decidability of the respective conditional logics and possibly to obtain complexity bounds for deduction in these systems. We analyse the case of **CK**.

These logics are known to be decidable, since they have the finite model property as shown by Nute [24]. However, the Nute's upper bound is hyper-exponential. A complexity analysis has not been carried out yet for most of the systems. In this section we show that **CK** belongs to PSPACE and it can be decided in $O(n \log n)$ space, where n is the length of the formula.

It is easy to show that **CK** is PSPACE-*hard* using a standard translation of modal logic K to **CK**; the former is known to be PSPACE-complete. To prove that **CK** is PSPACE, we show that contraction is eliminable in **SeqCK**. We adapt the technique used in [28]. This will give a linear (whence polynomial) bound on derivation length. From this fact the result will follow rather easily.

We begin by some simple facts. First observe that for **SeqCK** the ($\Rightarrow$ L)-rule can be restricted as follows:

$$\dfrac{\Gamma \vdash x \xrightarrow{A} y \quad \Gamma, y : B \vdash \Delta}{\Gamma, x : A \Rightarrow B \vdash \Delta}$$

The reason is that:

Fact 1

if $\Gamma \vdash \Delta, x \xrightarrow{A} y$ is derivable then either $\Gamma \vdash \Delta$ is derivable or $\Gamma = \Gamma', x \xrightarrow{A'} y$ and $x \xrightarrow{A'} y \vdash x \xrightarrow{A} y$ is derivable.

In other words a transition formula can only be proved by means of (EQ) rule, otherwise it must be introduced by weakening.

From now on we consider the proof system with the above rule in place of the old ($\Rightarrow$ L).

Another fact that can easily be checked is that all rules, except ($\Rightarrow$ L), permutes over any other rule[5]. Moreover, the rule ($\Rightarrow$ L) permutes over any other rule except ($\Rightarrow$ R). ($\Rightarrow$ L) does not permute over ($\Rightarrow$ R) exactly when the transition formula on the left premise of ($\Rightarrow$ L) is provable by a transition formula introduced (looking backwards) by the ($\Rightarrow$ R) rule.

The first significant fact about **CK** is that it satisfies the disjunction property for conditional formulas, namely

$$\vdash (A \Rightarrow B) \vee (C \Rightarrow D) \text{ implies } \vdash A \Rightarrow B \text{ or } \vdash C \Rightarrow D$$

as we show below. This property can be generalized to a certain extent by considering sequents -with non-empty antecedent and consequent:

$$\Gamma \vdash \Delta, x : A \Rightarrow B, x : C \Rightarrow D \text{ implies } \Gamma \vdash \Delta, x : A \Rightarrow B \text{ or } \Gamma \vdash \Delta, x : C \Rightarrow D.$$

Of course, it cannot hold for any Γ and Δ : consider trivially $x : (F_1 \rightarrow \bot) \rightarrow F_2 \vdash x : F_1, x : F_2$, where F_1 and F_2 are two conditional formulas, we do not necessarily have $x : (F_1 \rightarrow \bot) \rightarrow F_2 \vdash x : F_1$ or $x : (F_1 \rightarrow \bot) \rightarrow F_2 \vdash x : F_2$.

Intuitively, the property holds for Γ and Δ that do not cause any branching by formulas with label x or its predecessors (if any) according to the transition formulas. To this purpose we define:

Definition 3. *Let T be a set of transition formulas, let the set of x-branching formulas with respect to T, denoted by $\mathcal{B}(x, T)$ be defined as follows:*

$x : A \rightarrow B \in \mathcal{B}(x, T)$
$u : A \rightarrow B \in \mathcal{B}(x, T)$ *if* $T \vdash u \xrightarrow{D} x$ *for some formula D.*
$u : A \Rightarrow B \in \mathcal{B}(x, T)$ *if* $T \vdash u \xrightarrow{A} v$ *and* $v : B \in \mathcal{B}(x, T)$

Given a sequent $\Gamma \vdash \Delta$, where $\Gamma = \Gamma', T$, T is a set of transition formulas and Γ' does not contain transition formulas, we say that Γ (Δ) is x-branching if there is a formula $u : A \in \mathcal{B}(x, T)$ which occurs positively in Γ [6] *(negatively in Δ).*

[5] The previous considerations also entail that for **SeqCK** the (EQ) rule can be plugged into ($\Rightarrow$ L)-rule and removed as an independent rule. The new ($\Rightarrow$ L)-rule would be:

$$\frac{u : A \vdash u : A' \quad u : A' \vdash u : A \quad \Gamma, x \xrightarrow{A'} y, y : B \vdash \Delta}{\Gamma, x \xrightarrow{A'} y, x : A \Rightarrow B \vdash \Delta}$$

This reformulation may be useful for proof search; a similar rule is part of the tableau system presented in [25].

[6] Positive and negative occurrences of a formula are defined in the standard way: A occurs positively in A, if $B \rightarrow C$ occurs positively (negatively) in A, then C occurs positively (negatively) in A and B occurs negatively (positively) in A, if $B \Rightarrow C$ occurs positively (negatively) in A, then B occurs positively (negatively) in A. By extension, we say that a formula C occurs positively (negatively) in a multiset Γ if C occurs positively (negatively) in some formula of Γ.

The disjunction property holds for Γ and Δ that are not x-branching. The reason is twofold: on the one hand, for such Γ and Δ only the formulas on the path from x going backwards through the transition formulas (i.e. on the worlds $u_1 \xrightarrow{A_1} u_2 \xrightarrow{A_2} \ldots \xrightarrow{A_n} x$) can contribute to a proof of a formula with label x. This is proved by lemma 2 below. On the other hand, no formula on that path can create a branching. Given a multiset of transition formulas T and a multiset of labelled formulas Σ, we define

$$\begin{aligned} T_x = \{u \xrightarrow{A} v \in T \mid \exists u_0, \ldots u_m, A_1, \ldots A_m (m > 0),\\ u_0 = u, u_1 = v, u_m = x, A = A_1, u_{i-1} \xrightarrow{A_i}, u_i \in T\} \\ \Sigma^T_x = \{u : C \in \Sigma \mid u \xrightarrow{D} v \in T_x\} \cup \{x : C \mid x : C \in \Sigma\}. \end{aligned}$$

Given the sequent $\Gamma \vdash \Delta$, let us write $\Gamma = \Gamma', T$, where T is a set of transition formulas and Γ' does not contain transition formulas; let $\Gamma^T_x = \Gamma'^T_x, T_x$.

Lemma 2. *if Γ, Δ are not x-branching, then $\Gamma \vdash \Delta, x : A$ is derivable implies either $\Gamma^T_x \vdash \Delta^T_x, x : A$ is derivable or $\Gamma \vdash \Delta$ is derivable.*

Proof. By induction on the derivation height.

Proposition 1. *Let Γ and Δ be not x-branching, then we have*

$$\Gamma \vdash \Delta, x : A \Rightarrow B, x : C \Rightarrow D \textit{ implies } \Gamma \vdash \Delta, x : A \Rightarrow B \textit{ or } \Gamma \vdash \Delta, x : C \Rightarrow D.$$

Proof. Suppose that $\Gamma \vdash \Delta, x : A \Rightarrow B, x : C \Rightarrow D$ is derivable. By the permutation properties we can assume that there is a proof which terminates with the introduction of the two conditionals by ($\Rightarrow$ R) rule. Thus

$$\Gamma, x \xrightarrow{A} y, x \xrightarrow{C} z \vdash \Delta, y : B, z : D, \Delta \text{ is derivable.}$$

Let T be the set of transition formulas in Γ. Notice that since y, z do not occur in Γ, Δ and x, y, z are all distinct, we have that Γ and Δ are not y-branching (nor z-branching). Moreover observe that, letting $T' = T, x \xrightarrow{A} y$, we have $(\Gamma, x \xrightarrow{A} y)^{T'}_y = \Gamma^T_x x, x \xrightarrow{A} y$. Thus from lemma 2, we obtain either $\Gamma^T_x, x \xrightarrow{A} y \vdash \Delta^T_y, y : B$ is derivable or $\Gamma, x \xrightarrow{A} y, x \xrightarrow{C} z \vdash \Delta, z : D, \Delta$ is derivable. In the first case, by weakening we obtain $\Gamma, x \xrightarrow{A} y \vdash \Delta, y : B, \Delta$, from which we derive $\Gamma \vdash \Delta, x : A \Rightarrow B$ by the ($\Rightarrow$ R) rule. In the latter case, we observe first that $x \xrightarrow{A} y$ can be deleted as it cannot contribute to the proof, since y does not occur anywhere. Thus we have that $\Gamma, x \xrightarrow{C} z \vdash \Delta, z : D, \Delta$ is derivable and we conclude by the ($\Rightarrow$ R) rule again.

Using the previous fact we can easily show that we can eliminate right contractions on conditional formulas, namely

Proposition 2. *If $\vdash x : D$ is derivable, then it has a proof where there are no right contractions on conditional formulas.*

Proof. By permutation properties, a proof Π ending with

$$\Gamma \vdash \Delta, x : A \Rightarrow B, x : A \Rightarrow B$$

can be transformed into a proof Π', where all rules introducing x-branching formulas are permuted over the other rules (i.e. they are applied at the bottom of the tree). As an example, let the end sequent of Π have the form
$\Gamma, x : C \rightarrow D \vdash \Delta, x : A \Rightarrow B, x : A \Rightarrow B$,
we have that the lower sequent is x-branching, (at least) because of $x : C \rightarrow D$. We can permute Π so that the last step is the introduction of the x-branching formula $x : C \rightarrow D$ from the two sequents:

$$\Gamma \vdash \Delta, x : C, x : A \Rightarrow B, x : A \Rightarrow B \text{ and } \Gamma, x : D \vdash \Delta, x : A \Rightarrow B, x : A \Rightarrow B.$$

We have decomposed the x-branching formula, if the two sequents are still x-branching we perform a similar permutation upwards, so that at the end every branch of Π' will contain a sequent $\Gamma_i \vdash \Delta_i, x : A \Rightarrow B, x : A \Rightarrow B$, such that Γ_i, Δ_i are no longer x-branching. Then we can apply the previous proposition and obtain that for each i,

$\Gamma_i \vdash \Delta_i, x : A \Rightarrow B$ is derivable.

Thus, deleting one occurrence of $x : A \Rightarrow B$ in the consequent of any sequent in Π' below $\Gamma_i \vdash \Delta_i, x : A \Rightarrow B, x : A \Rightarrow B$ we get a derivation of $\Gamma \vdash \Delta, x : A \Rightarrow B$.

The previous propositions can be used to prove that contraction can be completely eliminated from **SeqCK** derivations. The relevant case is again the one of contraction on conditional formulas. Intuitively, the reason why we can eliminate contractions on conditional formulas is that if two transitions $x \xrightarrow{A} y$ and $x \xrightarrow{A} z$ are introduced by ($\Rightarrow$ R) in a proof of an end sequent $\vdash x : D$ (looking backwards), then x, y, z are all distinct. This matters for both right and left contraction. Regarding to right contraction, as we have argued above, we have that formulas on possible worlds y and z cannot interact; thus only one of the two y or z will be needed to carry on the derivation, and we can delete the other and the relative transition. This is the reason behind the disjunctive property itself and has a semantic counterpart: the models of **CK** can be assumed to have a tree-like structure where the links are the given by transition formulas.

The reason why we can eliminate a left contraction on conditional formulas is related, but different. Since given $x \xrightarrow{A} y$ and $x \xrightarrow{A} z$ as above, x, y, z are all distinct a conditional $x : A \Rightarrow B$ cannot be used to propagate B neither in x, nor in z. Thus $x : A \Rightarrow B$ does not need to be "used" (e.g. be the principal formula of ($\Rightarrow$ L)-rule) more than *once* along each branch of the proof, although $x : A \Rightarrow B$ might be used in several branches. But, the multiple use on different branches is accomplished by the branching rules and does not require an explicit use of contraction.

Theorem 5. *If* $\vdash x : A$ *is derivable in* **SeqCK***, then it has a derivation where there is no application of contraction.*

Proof. (Sketch) First observe that we do not have to worry about contractions on transition formulas. They can always be eliminated by **Fact 1** (at the beginning of section 4). Then the proof proceeds similarly to the one of theorem 9.1.1 in [28] by triple induction respectively: (i) on the number of contractions in a proof of the sequent, (ii) on the complexity of the formula involved in a contraction step, and (iii) the rank of the contraction, that is defined as the largest number of steps between the conclusions of a contraction and an upward sequent containing at least one of the two copies of the formula that is contracted. Since we can assume (by permutation properties) that a contraction follows immediately the introduction of one of the two copies of the formula (say the leftmost) the rank measures the *maximal number of steps* between the introduction of the first and second copy of the formula.
The minimum rank is hence 2, when the last two steps (before the contraction step) introduce the two copies one after the other. The third induction, on the rank, is needed because of the rule ($\Rightarrow$ L) which does not permute over the ($\Rightarrow$ R) rule: we cannot assume that there is a proof which introduces the two copies of the conditional formulas by ($\Rightarrow$ L) rule one after the other, (this happens when the introduction of the first copy is separated by the introduction of the second copy by ($\Rightarrow$ R) rule occurring in the middle) and we need to consider separately the two cases.

To carry on the proof, suppose that a derivation Π of $\vdash x_1 : A$ contains $i + 1$ contractions. Concentrate on an uppermost instance of contraction, say on a formula $x : D$ (so that the portion of the derivation above this step is contraction-free). In order to eliminate this contraction step, thereby obtaining a proof Π' that contain i contractions, we proceed by induction on the complexity of D and then by induction on the of the contraction step. Due to space limitations, we only sketch the proof of the most difficult case (see below). We use proposition 2 for eliminating a right contraction on a conditional formula, whereas a left contraction of rank = 2 can be eliminated by an argument based on the above informal considerations.

We analyze in details the most difficult case, the one of a left contraction on conditional formulas whose rank is > 2.

Suppose a proof Π of $\vdash x : A$ contains a contraction on the sequent $\Gamma, x : C \Rightarrow D, x : C \Rightarrow D \vdash \Delta$ on the formula $C \Rightarrow D$. To make the proof non-trivial, we assume that the first copy of $C \Rightarrow D$ is introduced by ($\Rightarrow$ L) immediately above the contraction step, whereas the second copy is introduced in a step further above by ($\Rightarrow$ L). Moreover, to introduce the first copy it is used a transition formula by the (EQ) rule which is present in the conclusion of the contraction step. We have two subcases: (a) to introduce the second copy it is used a transition formula which is present (or provable by the (EQ) rule from a transition formula) in the conclusion of the contraction step, or (b) it is used a transition formula which is not provable from a transition formula in the conclusion of the contraction step. In case (a), we can permute the ($\Rightarrow$ L), which introduces the second copy, over the following rule, so that we decrease the rank and we can apply the induction hypothesis. In case (b), the situation is more complex: the transition formula used to introduce the second copy of $C \Rightarrow D$ comes from an application of ($\Rightarrow$ R) *above* the contraction step. The situation is then as follows:

$$
\begin{array}{c}
\dfrac{\Gamma_2, x \xrightarrow{C''} z \vdash x \xrightarrow{C} z \qquad \begin{array}{c}\Pi_3\\ \Gamma_2, z : D \vdash \Delta_2\end{array}}{x : C \Rightarrow D, \Gamma_2, x \xrightarrow{C''} z \vdash \Delta_2}\,(\Rightarrow L)\\
\Pi_2\\
\dfrac{x : C \Rightarrow D, \Gamma_1, x \xrightarrow{C''} z \vdash z : E, \Delta_1}{x : C \Rightarrow D, \Gamma_1 \vdash x : C'' \Rightarrow E, \Delta_1}\,(\Rightarrow R)\\
\Pi_1\\
\dfrac{\dfrac{\Gamma, x \xrightarrow{C'} y \vdash x \xrightarrow{C} y \qquad y : D, x : C \Rightarrow D, \Gamma, x \xrightarrow{C'} y \vdash \Delta}{\Gamma, x : C \Rightarrow D, x : C \Rightarrow D, x \xrightarrow{C'} y \vdash \Delta}\,(\Rightarrow L)}{\Gamma, x : C \Rightarrow D \vdash \Delta}\,(ContrL)\\
\Pi_0\\
\vdash x_1 : A
\end{array}
$$

We abbreviate subderivations by Π_i. Observe that if Π_2 is empty we cannot permute the upper ($\Rightarrow$ L) over the lower ($\Rightarrow$ R), in order to diminish the rank. However we can observe the following: (i) x, y, z are all distinct, (ii) $x : C'' \Rightarrow E$ cannot come from (be a subformula of) $y : D$, it must already be a subformula of a formula in Γ or Δ. Thus we can divide the subproof Π_1 in two parts Π_1' and Π_1'', such that Π_1'' introduces $y : D$ and $x \xrightarrow{C'} y$ (possibly empty if $y : D$ and $x \xrightarrow{C'} y$ are already in Γ_1), and $x : C'' \Rightarrow E$ is used in a rule in Π_1'. Given this separation, we can permute the ($\Rightarrow$ R) over the lower ($\Rightarrow$ L) so that we obtain:

$$
\begin{array}{c}
\dfrac{\Gamma_2, x \xrightarrow{C''} z \vdash x \xrightarrow{C} z \qquad \begin{array}{c}\Pi_3\\ \Gamma_2, z : D \vdash \Delta_2\end{array}}{x : C \Rightarrow D, \Gamma_2, x \xrightarrow{C''} z \vdash \Delta_2}\,(\Rightarrow L)\\
\Pi_2\\
x : C \Rightarrow D, \Gamma_1, x \xrightarrow{C''} z \vdash z : E, \Delta_1\\
\Pi_1''\\
\dfrac{\dfrac{\Gamma^*, x \xrightarrow{C'} y \vdash x \xrightarrow{C} y \qquad y : D, x : C \Rightarrow D, \Gamma^*, x \xrightarrow{C'} y, x \xrightarrow{C''} z \vdash z : E, \Delta^*}{x : C \Rightarrow D, x : C \Rightarrow D, \Gamma^*, x \xrightarrow{C'} y, x \xrightarrow{C''} z \vdash z : E, \Delta^*}\,(\Rightarrow L)}{x : C \Rightarrow D, x : C \Rightarrow D, \Gamma^*, x \xrightarrow{C'} y \vdash x : C'' \Rightarrow E, \Delta^*}\,(\Rightarrow R)\\
\Pi_1'\\
\dfrac{\Gamma, x : C \Rightarrow D, x : C \Rightarrow D, x \xrightarrow{C'} y \vdash \Delta}{\Gamma, x : C \Rightarrow D, x \xrightarrow{C'} y \vdash \Delta}\,(ContrL)\\
\Pi_0\\
\vdash x_1 : A
\end{array}
$$

Then we can permute the (⇒ R) over the contraction step, so that we obtain

$$
\begin{array}{c}
\cfrac{\cfrac{\cfrac{\Gamma^*, x \xrightarrow{C'} y, x \xrightarrow{C''} z \vdash x \xrightarrow{C} y \qquad \begin{array}{c} \cfrac{\Gamma_2, x \xrightarrow{C''} z \vdash x \xrightarrow{C} z \quad \begin{array}{c}\Pi_3\\ \Gamma_2, z : D \vdash \Delta_2\end{array}}{x : C \Rightarrow D, \Gamma_2, x \xrightarrow{C''} z \vdash \Delta_2}\,(\Rightarrow L)\\ \Pi_2\\ x : C \Rightarrow D, \Gamma_1, x \xrightarrow{C''} z \vdash z : E, \Delta_1\\ \Pi_1''\\ y : D, x : C \Rightarrow D, \Gamma^*, x \xrightarrow{C'} y, x \xrightarrow{C''} z \vdash z : E, \Delta^*\end{array}}{x : C \Rightarrow D, x : C \Rightarrow D, \Gamma^*, x \xrightarrow{C'} y, x \xrightarrow{C''} z \vdash z : E, \Delta^*}\,(\Rightarrow L)}{x : C \Rightarrow D, \Gamma^*, x \xrightarrow{C'} y, x \xrightarrow{C''} z \vdash z : E, \Delta^*}\,(ContrL)}{x : C \Rightarrow D, \Gamma^*, x \xrightarrow{C'} y \vdash x : C'' \Rightarrow E, \Delta^*}\,(\Rightarrow R)\\
\Pi_1'^*\\
\Gamma, x : C \Rightarrow D, x \xrightarrow{C'} y \vdash \Delta\\
\Pi_0\\
\vdash x_1 : A
\end{array}
$$

The subderivation $\Pi_1'^*$ is obtained from Π_1' by deleting one copy of of $x : C \Rightarrow D$ from every descendant of $x : C \Rightarrow D, \Gamma^*, x \xrightarrow{C'} y \vdash x : C'' \Rightarrow E, \Delta^*$. Now the rank of the contraction step is decreased (the instance of the (⇒ L) rule which introduce the two copies are closer to each other), and we can apply the induction hypothesis.

Since we can eliminate contraction, it is relatively easy to prove both decidability and a space complexity bound.

First we observe that in all rules (except contraction obviously) the premises have a smaller complexity than the conclusion. To this regard, let $\mid A \mid$ be the number of symbols in the string representation of A, then define

$$cp(x : A) = 2* \mid A \mid \text{ and } cp(x \xrightarrow{A} y) = 2* \mid A \mid +1.$$

(so that $cp(x : A) < cp(x \xrightarrow{A} y) < cp(x : A \Rightarrow B)$). Let the complexity of a sequent $\Gamma \vdash \Delta$ be be the sum of the complexity $cp(F)$ of every labelled formula F occurring in the sequent. By inspection of the rules, it is easy to see that each premise of every rule has a smaller complexity than its conclusion.

By this fact we get that the length of each branch in a proof of a sequent $\vdash x : A$ is bounded by $O(\mid A \mid)$.

Secondly, observe that the rules are analytic, so that the premises contains only (labelled) subformulas of the formulas in the conclusion. Moreover, in the search of a proof of $\vdash x : A$, with $\mid A \mid= n$, new labels are introduced only by (positive) conditional subformulas of A. Thus, the number of different labels occurring in a proof is $O(n)$; it follows that the total number of distinct labelled formulas is $O(n^2)$, and only $O(n)$ of them can actually occur in each sequent.

This itself gives decidability:

Theorem 6. *Logic* **CK** *is decidable.*

Proof. We just observe that there is only a finite number of derivations to check of a given sequent $\vdash x : A$, as both the length of a proof and the number of labelled formulas which may occur in it is finite.

Notice that, as usual, a proof may have an exponential size because of the branching introduced by the rules. However we can obtain a much sharper space complexity bound using a standard technique [20,28], namely we do not need to store the whole proof, but only a sequent at a time plus additional information to carry on the proof search. In searching a proof there are two kinds of branching to consider: AND-branching caused by the rules with multiple premises and OR-branching (backtracking points in a depth first search) caused by the choice of the rule to apply, and how to apply it in the case of ($\Rightarrow$ L).

We store only one sequent at a time and maintain a stack containing information sufficient to reconstruct the branching points of both types. Each stack entry contains the principal formula (either a labelled sentence $x; B$, or a transition formula $x \xrightarrow{B} y$), the name of the rule applied and an index which allows to reconstruct the other branches on return to the branching points. The stack entries represent thus backtracking points and the index within the entry allows one to reconstruct both the AND branching and to check whether there are alternatives to explore (OR branching). The working sequent on a return point is recreated by replaying the stack entries from the bottom of the stack using the information in the index (for instance in the case of ($\Rightarrow$ L) applied to the principal formula $x : A \Rightarrow B$, the index will indicate which premise-first or second-we have to expand and the label y involved in the transition formula $x \xrightarrow{A} y$).

A proof begins with the end sequent $\vdash x : A$ and the empty stack. Each rule application generates a new sequent and extends the stack. If the current sequent is an axiom we pop the stack until we find an AND branching point to be expanded. If there are not, the end sequent $\vdash x : A$ is provable and we have finished. If the current sequent is not an axiom and no rule can be applied to it, we pop the stack entries and we continue at the first available entry with some alternative left (a backtracking point). If there are no such entries, the end sequent is not provable.

The entire process must terminate since: (i) the depth of the stack is bounded by the length of a branch proof, thus it is $O(n)$, where $\mid A \mid= n$, (ii) the branching is bounded by the number of rules, the number of premises of any rule and the number of formulas occurring in one sequent, the last being $O(n)$.

To evaluate the space requirement, we have that each subformula of the initial formula can be represented by a positional index into the initial formula, which requires $O(\log n)$ bits. Moreover, also each label can be represented by $O(\log n)$ bits. Thus, to store the working sequent we need $O(n \ \log n)$ space, since there may occur $O(n)$ labelled subformulas. Similarly, each stack entry requires $O(\log n)$ bits, as the name of the rule requires constant space and the index $O(\log n)$ bits. Having depth $O(n)$, to store the whole stack requires $O(n \ \log n)$ space. Thus we obtain:

Theorem 7. *Provability in* **CK** *is decidable in* $O(n \ \log n)$ *space.*

We strongly conjecture that the same result can be obtained for **CK**+ID. For **CK**+MP and **CK**+MP+ID, we conjecture that contraction cannot be eliminated, but it can be polynomially bounded. If the latter conjecture is right, it would give a higher, but still polynomial, space complexity bound. We will carry on a wider investigation in future research.

5 Extension to other Systems

There are a number of extensions of **CK**(+ID+MP) which have been considered in the literature. Each one of them is characterized by a set of axioms/semantic conditions. We list a few of them, without being exhaustive:

$$\begin{array}{rl}
\text{(AC)} & (A \Rightarrow B) \wedge (A \Rightarrow C) \rightarrow (A \wedge C \rightarrow B) \\
 & \text{If } f(w,[A]) \subseteq [B] \text{ then } f(w,[A \wedge B]) \subseteq f(w,[A]) \\
\text{(RT)} & (A \wedge B \Rightarrow C) \wedge (A \Rightarrow B) \rightarrow (A \Rightarrow C) \\
 & \text{If } f(w,[A]) \subseteq [B] \text{ then } f(w,[A]) \subseteq f(w,[A \wedge B] \\
\text{(CV)} & (A \Rightarrow B) \wedge \neg(A \Rightarrow \neg C) \rightarrow (A \wedge C \Rightarrow B) \\
 & \text{If } f(w,[A]) \subseteq [B] \text{ and } f(w,[A]) \cap [C] \neq \emptyset \text{ then } f(w,[A \wedge C]) \subseteq [B] \\
\text{(CA)} & (A \Rightarrow C) \wedge (B \Rightarrow C) \rightarrow (A \vee B \Rightarrow C) \\
 & f(w,[A \vee B]) \subseteq f(w,[A]) \cup f(w,[B]) \\
\text{(CEM)} & (A \Rightarrow B) \vee (A \Rightarrow \neg B) \\
 & \mid f(w,[A]) \mid \leq 1
\end{array}$$

These axioms/conditions are part of well-known conditional logics [24]. Some of these conditions can be used to formalize non-monotonic inferences. For instance AC corresponds to the property of *cumulativity* and CV to the property of *rational monotony*. Preferential entailment corresponds to the first degree fragment (i.e. without nested conditionals) of **CK**+ID+AC+CUT+CA[21].

We can think of extending our sequent calculi to these logics. One would like to have a modular proof system in the form of a sequent calculus where each semantic condition/axiom corresponds to a well-defined group of rules. To this regard, it is not difficult to devise rules capturing these semantic conditions. For instance, a possible rule for AC is the following:

$$(\mathbf{AC})\ \frac{\Gamma, x \xrightarrow{A} y \vdash \Delta \qquad \Gamma, x \xrightarrow{A} z \vdash z : B, \Delta}{\Gamma, x \xrightarrow{A \wedge B} y \vdash \Delta}\ (z \notin \Gamma, \Delta)$$

However, the addition of **(AC)** rule to **SeqCK** does not give a complete system unless *we allow cut*, or in other words, in the system **SeqCK**+AC cut is not eliminable. To see this consider the formula

$$(A \Rightarrow (B \wedge C)) \rightarrow ((A \wedge B) \Rightarrow C).$$

This formula is provable by **SeqCK**+AC without cut. But the equivalent formula

$$(A \Rightarrow (B \wedge C)) \rightarrow (\neg(\neg A \vee \neg B)) \Rightarrow C).$$

is not. In order to apply the rule for AC, we must be able to prove the equivalence of $\neg(\neg A \vee \neg B)$ and $A \wedge B$. We cannot do this unless we allow cut on transition formulas, or we incorporate the (EQ)-test within the AC rule itself.

The same problem seems to arise with the other axiom/semantic conditions where we need to identify a subformula which is characterized by its syntactic structure (such as $A \wedge B$) above. We can conclude that a straightforward encoding of the semantic conditions we have exemplified results in a non-analytic calculus where cut cannot be eliminated. This is somewhat expected as the selection function cannot be assumed to satisfy any *compositionality* principle: i.e. the value of $f(w, [A\#B])$ for any connective $\#$ is not a function of $f(w, [A])$ and $f(w, [B])$, at most f satisfies some constraints as the above ones. However, further research is needed to see how and whether we can capture the above semantic conditions and alike within the labelled calculus by analytic rules. It might be that some *combination* of semantic conditions can be translated into a set of analytic rules, although each single condition cannot. It might also be that, in order to represent these semantic conditions, we need sequents with a more complex structure, than transition formulas and labelled sentences. These are hypotheses for future research.

6 Conclusions

In this work we have provided a labelled calculus for minimal conditional logic **CK** and its extensions with conditions ID and MP. The calculus is cut-free and analytic. By a proof-theoretical analysis of **CK** we have shown that **CK** is decidable in $O(n \log n)$ space. To the best of our knowledge, sequent calculi for these logics have not been previously studied and the complexity bound for **CK** is new. In future research we intend to expand our work in two directions: extend the complexity analysis to other systems and find labelled analytic calculi for other conditional logics. We briefly remark on some related work.

Some complexity results for conditional logic have been provided in [9], however the case of minimal logic **CK** is not considered by the authors; moreover, the results are obtained semantically, by arguing about the size of possible countermodels, and they only apply to logics stronger than **CK**+ID+MP for which there is a sphere-model semantics.

In [25], it is presented a labelled tableaux system for various systems of conditional logics along the same lines. The tableaux calculus is defined inductively on the levels of nesting of the conditional operator: the equivalence of nested conditional subformulas is checked by a recursive "call' of the tableaux prover. The sequent calculus we present in this present paper does not need the inductive construction of the tableau system. The equivalence test of subformulas is fully integrated into the sequent calculus.

An approach similar to ours has been followed in [17]. The authors develop labelled tableau for the *first-degree* fragment (i.e. without nested conditionals) of conditional logic **CK**+ID+AC+RT+CA. Formulas are labelled by path of worlds (containing also variable worlds) which seem to correspond to (sets of) chains of transition formulas in our setting. They use an unification procedure to propagate positive conditionals; moreover the unification procedure itself takes care of checking the equivalence of antecedents. Their tableau system contains a cut-rule, called PB, which *cannot* be eliminated.

Acknowledgements We are grateful to the referees for their careful reading, constructive criticisms, and interesting remarks.

References

1. C. Boutilier, Conditional logics of normality: a modal approach. *Artificial Intelligence*, 68:87–154, 1994. 384
2. B. F. Chellas, Basic Conditional logics, *J. of Philosophical Logic*, 4:133-153,1975. 384
3. G. Crocco and L. Fariñas del Cerro, *Structure, Consequence relation and Logic*, volume 4, pages 239–259. Oxford University Press, 1992. 385
4. G. Crocco, L. Fariñas del Cerro, and A. Herzig, *Conditionals: From philosophy to computer science*, Oxford University Press, Studies in Logic and Computation, 1995. 384
5. G. Crocco and P. Lamarre, *On the Connection between Non-Monotonic Inference Systems and Conditional Logics*, In B. Nebel and E. Sandewall, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the 3rd International Conference*, pages 565-571, 1992. 384
6. J. P. Delgrande, A first-order conditional logic for prototypical properties. *Artificial Intelligence*, (33):105–130, 1987.
7. H. C. M. de Swart, A Gentzen-or Beth-type system, a practical decision procedure and a constructive completeness proof for the counterfactual logics VC and VCS, *Journal of Symbolic Logic*, 48:1-20, 1983. 385
8. M. Fitting, *Proof methods for Modal and Intuitionistic Logic*, vol 169 of Synthese library, D. Reidel, Dordrecht, 1983. 385
9. N. Friedman and J. Halpern, On the complexity of conditional logics. In *Principles of Knowledge Representation and Reasoning: Proceedings of the 4th International Conference, KR'94*, pages 202–213. 402
10. N. Friedman and J. Halpern. Belief Revision: A critique, J. of Logic, Language and Information, 8(4): 401–420, 1999. 387
11. D. M. Gabbay, *Labelled Deductive Systems* (vol I), Oxford Logic Guides, Oxford University Press, 1996. 385
12. D. M. Gabbay L.Giordano, A.Martelli, N.Olivetti and M. L. Sapino, Conditional Reasoning in Logic Programming. *J. of Logic Programming* 44(1-3):37–74, 2000. 384

13. I. P. Gent, A sequent or tableaux-style system for Lewis's counterfactual logic VC. *Notre Dame j. of Formal Logic*, 33(3): 369-382, 1992. 385
14. M. L. Ginsberg. Counterfactuals. *Artificial Intelligence*, 30(2):35–79, 1986. 384
15. L. Giordano, V. Gliozzi and N. Olivetti, A conditional logic for belief revision. In *Proc. European Workshop on Logics in Artificial Intelligence JELIA 98*, Springer LNAI 1489, pp. 294-308, 1998. 384
16. L. Giordano, V. Gliozzi, and N. Olivetti. Iterated Belief Revision and Conditional Logic. *Studia Logica*, to appear, 2001. 384
17. A. Artosi, G. Governatori, and A. Rotolo: A Labelled Tableau Calculus for Nonmonotonic (Cumulative) Consequence Relations. In *Proc.of TABLEAUX 2000*, LNCS 1847, pp. 82-97, 2000, 385, 403
18. G. Grahne, Updates and Counterfactuals, *in* Journal of Logic and Computation, Vol 8 No.1:87-117, 1998. 384
19. C. Groeneboer and James Delgrande, A general approach for determining the validity of commonsense assertions using conditional logics. *International Journal of Intelligent Systems*, (5):505–520, 1990. 385
20. J. Hudelmaier, An $O(n\ log\ n)$-space decision procedure for intuitionistic propositional logic. *Journal of Logic and Computation*, **3**, 63–75, 1993. 400
21. Sarit Kraus, Daniel Lehmann, and Menachem Magidor. Nonmonotonic reasoning, preferential models and cumulative logics. *Artificial Intelligence*, 44:167–202, 1990. 384, 401
22. P. Lamarre. A tableaux prover for conditional logics. In *Principles of Knowledge Representation and Reasoning: Proceedings of the 4th International Conference, KR'94*, pages 572–580. 385
23. D. Lewis, *Counterfactuals.* Basil Blackwell Ltd, 1973. 384, 385
24. D. Nute, *Topics in Conditional Logic*, Reidel, Dordrecht, 1980. 384, 385, 386, 387, 393, 401
25. N. Olivetti and C. Schwind, *Analytic Tableaux for Conditional Logics*, Technical Report, University of Torino, 2000. 394, 402
26. C. B. Schwind, Causality in Action Theories. Electronic Articles in Computer and Information Science, Vol. 3 (1999), section A, pp. 27-50. 384, 387
27. R. Stalnaker, A Theory of Conditionals, *in* N. Rescher (ed.), Studies in Logical Theory, American Philosophical Quarterly, Monograph Series no.2, Blackwell, Oxford: 98-112, 1968. 384
28. L. Viganò, *Labelled Non-classical Logics.* Kluwer Academic Publishers, Dordrecht, 2000. 385, 393, 397, 400

Reasoning about Complex Actions with Incomplete Knowledge: A Modal Approach

Matteo Baldoni[1], Laura Giordano[2], Alberto Martelli[1], and Viviana Patti[1]

[1] Dipartimento di Informatica, Università degli Studi di Torino
C.so Svizzera 185, I-10149 Torino (Italy)
{baldoni,mrt,patti}@di.unito.it

[2] Dipartimento di Scienze e Tecnologie Avanzate, Università degli Studi del Piemonte Orientale
C.so Borsalino 54, I-15100 Alessandria (Italy)
laura@di.unito.it

Abstract. In this paper we propose a modal approach for reasoning about dynamic domains in a logic programming setting. We present a logical framework for reasoning about actions in which *modal inclusion axioms* of the form $\langle p_0 \rangle \varphi \subset \langle p_1 \rangle \langle p_2 \rangle \ldots \langle p_n \rangle \varphi$ allow *procedures* to be defined for building *complex actions* from elementary actions. The language is able to handle knowledge producing actions as well as actions which remove information. Incomplete states are represented by means of epistemic operators and test actions can be used to check whether a fluent is true, false or undefined in a state. We give a *non-monotonic* solution for the frame problem by making use of persistency assumptions in the context of an abductive characterization. A *goal directed proof procedure* is defined, which allows reasoning about complex actions and generating conditional plans.

1 Introduction

Reasoning about the effects of actions in a dynamically changing world is one of the problems an intelligent agent has to face. Often an agent has incomplete knowledge about the state of the world and it needs to perform sensing actions [25] to acquire new information for determining how to act.

There are several proposals in the literature for reasoning about actions in the presence of sensing which have been developed along the line of Scherl and Levesque paper [25]. Let us mention the work on the high level robot programming language GOLOG [11,12], which is based on a theory of actions in the situation calculus. Other proposals have been developed by extending the action description language $\mathcal{A}$ [17], as in [23,8], while in [27] a formal account of a robot's knowledge about the state of its environment has been developed in the context of the fluent calculus.

In this paper, we tackle the problem of reasoning about complex actions with incomplete knowledge in a modal action logic. The adoption of dynamic logic or a modal logic to formalize reasoning about actions and change is common to many

A. Restivo, S. Ronchi Della Rocca, L. Roversi (Eds.): ICTCS 2001, LNCS 2202, pp. 405–426, 2001.

proposals [10,24,9,26,18] and it allows very natural representation of actions as state transitions, through the accessibility relation of Kripke structures.

We introduce an action theory on the line of [6,18,19], in which actions are represented by modalities, and we extend it by allowing sensing actions as well as complex actions definitions. Our starting point is the modal logic programming language for reasoning about actions presented in [6]. Such language mainly focuses on *ramification problem* but does not provide a formalization of incomplete initial states with an explicit representation of *undefined* fluents. Such an explicit representation is needed if we want to model an agent which is capable of reasoning and acting on the basis of its (dis)beliefs. In particular, an agent might want to take actions to acquire new knowledge on the world, if its knowledge is incomplete. These knowledge producing actions are usually called *sensing* actions.

In this paper, we aim at extending the action language presented in [6] to represent *incomplete states* and to deal with *sensing actions*. Note that, based on the logical framework, we aim at defining an agent programming language. In this context, we need to describe the behavior of an intelligent agent that chooses a course of actions conditioned on its beliefs on the environment and uses sensors for acquiring or updating its knowledge about the real world. Keeping the point of view of the agent, as we do, the only relevant characterization concerns the internal dynamics of the agent, which can be regarded as a result of executing actions on the mental state. As a consequence, we only keep the agent's representation of the world, while in other formalizations of sensing actions [25,8], where the focus is on developing a theory of actions and knowledge rather than on modeling agent behaviors, both the mental state of the agent and the real state of the world are represented. In order to represent the mental state of an agent, we introduce an epistemic level in our logical framework.In particular, by using modalities, we represent the mental state of an agent as a set of epistemic fluents. Then, concerning world actions, i.e. actions affecting the real world, we only model what the agent knows about action's effects based on knowledge preconditions and we consider sensing actions as input actions which produce fresh knowledge on the value of some fluents in the real world. As a consequence, we simply model sensing actions as non-deterministic actions, whose outcome can not be predicted by the agent.

Another aim of the paper is to extend the action language to deal with *complex actions*. The definition of complex actions we introduce draws from dynamic logic [20] for the definition of action operators like sequence, test and non-deterministic choice. However, rather than referring to an Algol-like paradigm for describing complex actions, as in [22], we refer to a Prolog-like paradigm: complex actions are defined through (possibly recursive) definitions, given by means of Prolog-like clauses.

In particular, we show that in modal logics, we can express complex actions' definitions by means of a suitable set of axioms of the form

$$\langle p_0 \rangle \varphi \subset \langle p_1 \rangle \langle p_2 \rangle \ldots \langle p_n \rangle \varphi .$$

If p_0 is a procedure name, and the $p_i (i = 1, \ldots, n)$ are either procedure names, or atomic or test actions, the above axiom can be interpreted as a procedure definition, which can then be executed in a goal directed way, similarly to standard logic programs. These axioms have the form of inclusion axioms, which were the subject of a previous work [5,2], in which we have analyzed the class of multimodal logics characterized by axioms of the form $[s_1] \ldots [s_m]\varphi \subset [p_1] \ldots [p_n]\varphi$ where $[s_i]$ and $[p_i]$ are modal operators. These axioms have interesting computational properties because they can be considered as rewriting rules.

We show that the temporal projection problem and the planning problem can be formalized in our language. Furthermore we develop proof procedures for reasoning about complex actions (including sensing actions) and for constructing *conditional plans* to achieve a given goal from an incompletely specified initial state. We can prove in the language that such generated plans are *correct*, i.e. achieve the desired goal for a given initial state.

2 The Modal Action Logic

In our action logic each atomic action is represented by a modality. We distinguish between two kinds of atomic actions: *sensing actions*, which affect the internal state of the agent by enhancing its knowledge on the environment and *non-sensing actions* (or *world actions*), that is actions which have actual effects on the external world. We denote by $\mathcal{S}$ the set of sensing actions and by $\mathcal{A}$ the set of world actions. For each action $a \in \mathcal{A}$ ($s \in \mathcal{S}$) we introduce a modality $[a]$ ($[s]$). A formula $[a]\alpha$ means that α holds after any execution of action a, while $\langle a \rangle\alpha$ means that there is a possible execution of action a after which α holds (similarly for the modalities for sensing actions). We also make use of the modality $\Box$, in order to denote those formulas that hold in all states. The intended meaning of a formula $\Box\alpha$ is that α holds after any sequence of actions. In order to represent complex actions, the language contains also a finite number of modalities $[p_i]$ and $\langle p_i \rangle$ (universal and existential modalities respectively), where p_i is a constant denoting a procedure name. Let us denote by $\mathcal{P}$ the set of such procedure names. The modal operator $\mathcal{B}$ is used to model agent's beliefs. Moreover, we use the modality $\mathcal{M}$, which is defined as the dual of $\mathcal{B}$, i.e. $\mathcal{M}\alpha \equiv \neg\mathcal{B}\neg\alpha$. Intuitively, $\mathcal{B}\alpha$ means that α is believed to be the case, while $\mathcal{M}\alpha$ means that α is considered to be possible.

A *fluent literal* l is defined to be f or $\neg f$, where f is an atomic proposition (*fluent name*). Since we want to reason about the effects of actions on the internal state of an agent, we define a state as a set of *epistemic fluent literals*. An epistemic fluent literal F is a modal atom $\mathcal{B}l$ or its negation $\neg\mathcal{B}l$, where l is a fluent literal. An *epistemic state* S is a set of epistemic literals satisfying the requirement that for each fluent literal l, either $\mathcal{B}l \in S$ or $\neg\mathcal{B}l \in S$. In essence a state is a complete and consistent set of epistemic literals, and it provides a three-valued interpretation in which each literal l is *true* when $\mathcal{B}l$ holds, *false* when $\mathcal{B}\neg l$ holds, and *undefined* when both $\neg\mathcal{B}l$ and $\neg\mathcal{B}\neg l$ hold (denoted by $\mathcal{U}l$).

All the modalities of the language are normal, that is, they are ruled at least by axiom K. In particular, the modality $\Box$, is ruled by the axioms of logic $S4$. Since it is used to denote information which holds in any state, after any sequence of primitive actions, the $\Box$ modality interacts with the atomic actions modalities through the interaction axiom schemas $\Box\varphi \supset [a]\varphi$ and $\Box\varphi \supset [s]\varphi$, for all $a \in \mathcal{A}$ and $s \in \mathcal{S}$. The epistemic modality $\mathcal{B}$ is serial, that is, in addition to axiom schema K we have the axiom schema $\mathcal{B}\varphi \supset \neg\mathcal{B}\neg\varphi$. Seriality is needed to guarantee the consistency of states: it is not acceptable a state in which, for some literal l, both $\mathcal{B}l$ holds and $\mathcal{B}\neg l$ holds.

2.1 World Actions

World actions allow the agent to affect the environment. In our formalization we only model the epistemic state of the agent while we do not model the real world. This is the reason we will not represent the actual effects of world actions, formalizing only what the agent knows about these effects based on knowledge preconditions. For each world action, the domain description contains a set of *simple action clauses*, that allow one to describe direct effects and preconditions of primitive actions on the epistemic state. Basically, simple action clauses consist of *action laws* and *precondition laws.* [1]

Action laws define direct effects of actions in $\mathcal{A}$ on an epistemic fluent and allow actions with conditional effects to be represented. They have form:

$$\Box(\mathcal{B}l_1 \wedge \ldots \wedge \mathcal{B}l_n \supset [a]\mathcal{B}l_0) \tag{1}$$

$$\Box(\mathcal{M}l_1 \wedge \ldots \wedge \mathcal{M}l_n \supset [a]\mathcal{M}l_0) \tag{2}$$

(1) means that in any state ($\Box$), if the set of literals $l_1, \ldots, l_n$ (representing the preconditions of the action a) is believed then, after the execution of a, l_0 (the effect of a) is also believed. (2) is necessary in order to deal with *ignorance* about preconditions of the action a. It means that the execution of a may affect the beliefs about l_0, when executed in a state in which the preconditions are considered to be possible. When the preconditions of a are unknown, this law allows to conclude that the effects of a are unknown as well.

Example 1. Let us consider the example of a robot which is inside a room (see Fig. 1). Two sliding doors, 1 and 2, connect the room to the outside and $toggle\_switch(I)$ denote the action of toggling the switch next to door I, by which door opens if it is closed and closes if it is open. This is a suitable set of action laws for this action:

(a) $\Box(\mathcal{B}\neg open(I) \supset [toggle\_switch(I)]\mathcal{B}open(I))$
(b) $\Box(\mathcal{M}\neg open(I) \supset [toggle\_switch(I)]\mathcal{M}open(I))$
(c) $\Box(\mathcal{B}open(I) \supset [toggle\_switch(I)]\mathcal{B}\neg open(I))$
(d) $\Box(\mathcal{M}open(I) \supset [toggle\_switch(I)]\mathcal{M}\neg open(I))$

[1] In this paper we do not introduce constraints or causal rules among fluents. However, causal rules could be easily introduced by allowing a causality operator, as in [18,19] to which we refer for a treatment of ramification in a modal setting.

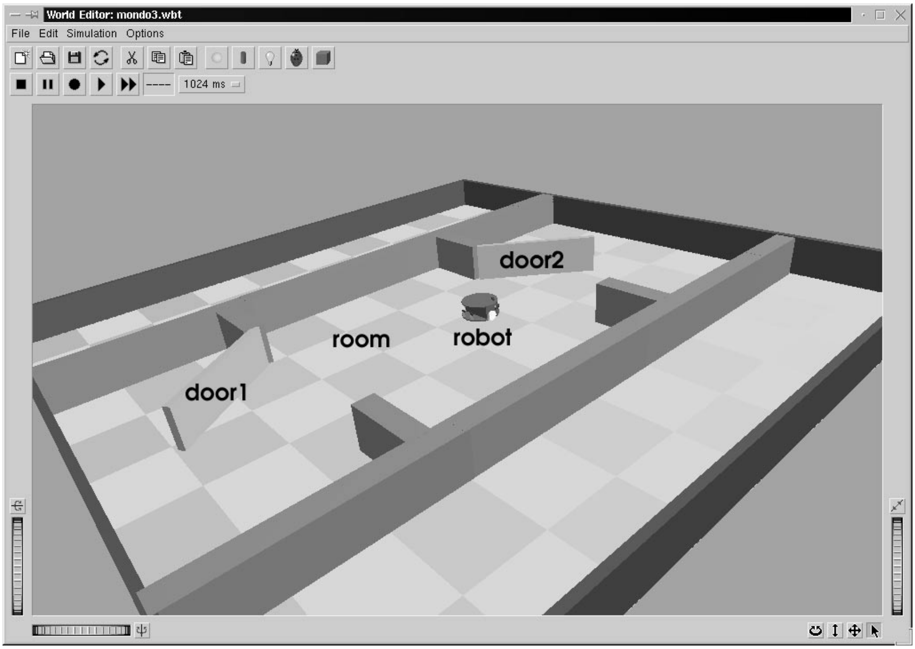


Fig. 1. A snapshot of our robot. Initially it is inside the room, in front of door number 2

Note that, in order to avoid introducing many variant of the same clauses, as a shorthand, we use the metavariables I, J, where $I, J \in \{door1, door2\}$ and $I \neq J$.

Precondition laws allow to specify knowledge preconditions for actions, i.e. those epistemic conditions which make an action executable in a state. They have form:

$$\Box(\mathcal{B}l_1 \wedge \ldots \wedge \mathcal{B}l_n \supset \langle a \rangle true) \tag{3}$$

meaning that in any state, if the conjunction of epistemic literals $\mathcal{B}l_1, \ldots, \mathcal{B}l_n$ holds, then a can be executed. For instance, according to the following clause, the robot must know to be in front of a door I if it wants to open (or close) it by executing $toggle\_switch(I)$:

(e) $\Box(\mathcal{B}in\_front\_of(I) \supset \langle toggle\_switch(I) \rangle true)$

Knowledge Removing Actions Up to now, we considered actions with deterministic effects on the world, i.e. actions in which the outcome can be predicted. The execution of such actions causes the agent to have knowledge about their effects, because the action is said to *deterministically* cause the change of a given set of fluents. However effects of actions can be non-deterministic and, then, unpredictable. In such a case, the execution of the action causes the agent to *lose knowledge* about its possible effects, because the action could unpredictably cause the change of some fluent. In our framework, we can model actions with non-deterministic effects as actions which *may* affect the knowledge about the value of a fluent, by simply using action laws of form (2) but without adding the corresponding law of the form (1).

Example 2. Let us consider an action $drop(I)$ of dropping a glass I from a table. We want to model the fact that dropping a *fragile* glass may possibly make the glass broken. It can be expressed by using a suitable action law of the form (2):

$$\Box(\mathcal{M}fragile(I) \supset [drop(I)]\mathcal{M}broken(I)).$$

It means that, in the case the agent considers possible that the glass is fragile, then, after dropping it, it considers possible that it has become broken. Note that, since $\mathcal{B}\alpha$ entails $\mathcal{M}\alpha$, the action law above can also be applied in the case the agent believes that the glass is fragile, to conclude that it is possibly broken. If action *drop* is executed in a state in which $\mathcal{B}fragile$ and $\mathcal{B}\neg broken$ hold, in the resulting state $\mathcal{M}broken$ (i.e. $\neg\mathcal{B}\neg broken$) will hold: the agent does not know anymore if the glass is broken or not.

2.2 Sensing Actions: Gathering Information from the World

Let us now consider sensing actions, which allow an agent to *gather information from the environment*, enhancing its knowledge about the value of a fluent. In our representation sensing actions are defined by modal inclusion axioms [2], in terms of *ad hoc* primitive actions. We represent a *binary* sensing action $s \in \mathcal{S}$, for knowing whether the fluent l or its complement $\neg l$ is true, by means of axioms of our logic that specify the effects of s on agent knowledge as the non-deterministic choice between two primitive actions, the one causing the belief $\mathcal{B}l$, and the other one causing the belief $\mathcal{B}\neg l$. For each binary sensing action $s \in \mathcal{S}$ we have an axiom of form: $[s]\varphi \equiv [s^{\mathcal{B}l} \cup s^{\mathcal{B}\neg l}]\varphi$. The operator $\cup$ is the *choice operator* of dynamic logic, which expresses the non-deterministic choice among two actions: executing the choice $a \cup b$ means to execute non-deterministically either a or b. This is ruled by the axiom schema $\langle a \cup b\rangle\varphi \equiv \langle a\rangle\varphi \vee \langle b\rangle\varphi$ [20]. The actions $s^{\mathcal{B}l}$ and $s^{\mathcal{B}\neg l}$ are primitive actions in $\mathcal{A}$ and they can be regarded as being predefined actions, ruled by the simple action clauses:

$$\begin{array}{ll} \Box(\mathcal{B}l_1 \wedge \ldots \wedge \mathcal{B}l_n \supset \langle s^{\mathcal{B}l}\rangle true) & \Box(\mathcal{B}l_1 \wedge \ldots \wedge \mathcal{B}l_n \supset \langle s^{\mathcal{B}\neg l}\rangle true) \\ \Box(true \supset [s^{\mathcal{B}l}]\mathcal{B}l) & \Box(true \supset [s^{\mathcal{B}\neg l}]\mathcal{B}\neg l) \end{array}$$

Note that, executability preconditions of sensing action s, are represented by executability preconditions $\mathcal{B}l_1, \ldots, \mathcal{B}l_n$ of the *ad hoc* defining action $s^{\mathcal{B}l}$ and $s^{\mathcal{B}\neg l}$. This is the reason they have to be the same.

Summarizing, the formulation above expresses the fact that s can be executed in a state where preconditions hold, leading to a new state where the agent has a belief about l: he may either believe that l or that $\neg l$.

Example 3. Let $sense\_door(I) \in \mathcal{S}$ denote the action of sensing whether a door I is open, which is executable if the robot knows to be in front of I. This is the suitable axiom representing knowledge precondition and effects:

(f) $[sense\_door(I)]\varphi \equiv [sense\_door(I)^{\mathcal{B}open(I)} \cup sense\_door(I)^{\mathcal{B}\neg open(I)}]\varphi$

where the primitive actions $sense\_door(I)^{\mathcal{B}open(I)}$ and $sense\_door(I)^{\mathcal{B}\neg open(I)}$ are ruled by the set of laws:

(g) $\Box(\mathcal{B}in\_front\_of(I) \supset \langle sense\_door(I)^{\mathcal{B}open(I)}\rangle true)$
(h) $\Box(true \supset [sense\_door(I)^{\mathcal{B}open(I)}]\mathcal{B}open(I))$
(i) $\Box(\mathcal{B}in\_front\_of(I) \supset \langle sense\_door(I)^{\mathcal{B}\neg open(I)}\rangle true)$
(j) $\Box(true \supset [sense\_door(I)^{\mathcal{B}\neg open(I)}]\mathcal{B}\neg open(I))$

More in general, we can deal with sensing on a *finite set of literals*, where executing a sensing action leads to a new state where the agent knows which literal is true among an associated set of literals. More formally, we associate to each sensing action $s \in \mathcal{S}$ a set $dom(s)$ of literals. The effect of s will be to know which literal in $dom(s)$ is true. This is modeled by introducing an axiom of the form:

$$[s]\varphi \equiv [\bigcup_{l \in dom(s)} s^{\mathcal{B}l}]\varphi \tag{4}$$

where the primitive action $s^{\mathcal{B}l}$ ($\in \mathcal{A}$), for each $l, l' \in dom(s)$, $l \neq l'$, is ruled by the following simple action clauses:

$$\Box(\mathcal{B}l_1 \wedge \ldots \wedge \mathcal{B}l_n \supset \langle s^{\mathcal{B}l}\rangle true) \tag{5}$$

$$\Box(true \supset [s^{\mathcal{B}l}]\mathcal{B}l) \tag{6}$$

$$\Box(true \supset [s^{\mathcal{B}l}]\mathcal{B}\neg l') \tag{7}$$

Clause (5) means that in any state, if the set of literal $\mathcal{B}l_1 \wedge \ldots \wedge \mathcal{B}l_n$ holds, then the action $s^{\mathcal{B}l}$ can be executed. The other ones describe the effects of $s^{\mathcal{B}l}$: in any state, after the execution of $s^{\mathcal{B}l}$ l is believed (6), while all the other fluents belonging to $dom(s)$ are believed to be false (7). Note that the binary sensing action on a fluent l, is a special case of sensing where the associated finite set is $\{l, \neg l\}$.

2.3 Complex Actions

In our modal action theory, complex actions are defined on the basis of other complex actions, atomic actions and test actions. *Test actions* are needed for testing if some fluent holds in the current state and for expressing conditional complex actions. Like in dynamic logic [20], if ψ is a proposition then $\psi?$ can be used as a label for a modal operator, such as $\langle\psi?\rangle$. Test modalities are characterized by the axiom schema $\langle\psi?\rangle\varphi \equiv \psi \wedge \varphi$.

A *complex action* is defined by means of a suitable set of inclusion axiom schemas of our modal logic, having the form [2]:

$$\langle p_0 \rangle \varphi \subset \langle p_1 \rangle \langle p_2 \rangle \dots \langle p_n \rangle \varphi. \tag{8}$$

If p_0 is a procedure name in $\mathcal{P}$, and p_i $(i = 1, \dots, n)$ are either procedure names, or atomic actions or test actions, axiom (8) can be interpreted as a procedure definition. Procedures definition can be *recursive* and they can also be *non-deterministic*, when they are defined by a collection of axioms of the form specified above. Intuitively, they can be executed in a goal directed way, similarly to standard logic programs. Indeed the meaning of (8) is that if in a state there is a possible execution of p_1, followed by an execution of p_2, and so on up to p_n, then in that state there is a possible execution of p_0.

Remark 1. Complex actions' definitions are inclusion axioms. [2,5] presents a tableaux calculus and some decidability results for logics characterized by this kind of axioms, where inclusion axioms are interpreted as rewriting rules. In particular in [5,2] it is shown that the general satisfiability problem is decidable for right regular grammar logics, but it is undecidable for the class of context-free grammar logics. Moreover, in [2] a tableaux-based proof procedure is presented for a broader class of logics, called incestual modal logics, in which the operators of union and composition are used for building new labels for modal operators. Such class includes grammar logic. These results were recently extended and generalized by Demri [14]. In particular in [14] it is shown that *every* regular grammar logics is decidable, where also more expressive logics including structured modalities of the form $a; b$, $a \cup b$ and $a?$ are considered. Grammar logics with definitions of complex actions as inclusion axioms could fall in the class of context-free grammar logics or in the decidable class of regular grammar logics, depending on the form of the axioms. As concern the complexity problem, we refer also to [14] where some complexity results for grammar logics are presented.

Procedures can be used to describe the complex behavior of an agent, as shown in the following example.

Example 4. Let us suppose that our robot has to achieve the goal of closing a door I of the room (see Fig. 1). By the following axioms we can define $close\_door(I)$, i.e. the procedure specifying the action plans the robot may execute for achieving the goal of closing the door I.

(k) $\langle close\_door(I) \rangle \varphi \subset \langle \mathcal{B} \neg open(I)? \rangle \varphi$

(l) $\langle close\_door(I) \rangle \varphi \subset \langle (\mathcal{B} open(I) \wedge \mathcal{B} in\_front\_of(I))? \rangle \langle toggle\_switch \rangle \varphi$

(m) $\langle close\_door(I) \rangle \varphi \subset \langle (\mathcal{U} open(I) \wedge \mathcal{B} in\_front\_of(I))? \rangle$
$\langle sense\_door(I) \rangle \langle close\_door(I) \rangle \varphi$

(n) $\langle close\_door(I) \rangle \varphi \subset \langle (\mathcal{M} open(I) \wedge \mathcal{B} \neg in\_front\_of(I))? \rangle$
$\langle go\_to\_door(I) \rangle \langle close\_door(I) \rangle \varphi$

[2] For sake of brevity, sometimes we will write these axioms as $\langle p_0 \rangle \varphi \subset \langle p_1; p_2; \dots; p_n \rangle \varphi$, where the operator ";" is the sequencing operator of dynamic logic: $\langle a; b \rangle \varphi \equiv \langle a \rangle \langle b \rangle$.

The definition of $close\_door$ is recursive. The complex behavior defined is based on the primitive actions $toggle\_switch(I)$, $go\_to\_door(I)$ and on the sensing action $sense\_door(I)$. $toggle\_switch(I)$ is ruled by the action laws (a-d) in Example 1, and by precondition law (e) above. $sense\_door(I)$ is ruled by axiom (f) and by laws (g-j) in Example 3, while the simple action clauses for $go\_to\_door(I)$ are given in the following:

(o) $\Box(\mathcal{B}\neg in\_front\_of(I) \wedge \mathcal{B}\neg out\_room \supset \langle go\_to\_door(I)\rangle true)$
(p) $\Box(true \supset [go\_to\_door(I)]\mathcal{B}in\_front\_of(I))$
(q) $\Box(true \supset [go\_to\_door(I)]\mathcal{M}in\_front\_of(I))$
(r) $\Box(\mathcal{B}in\_front\_of(J) \supset [go\_to\_door(I)]\mathcal{B}\neg in\_front\_of(J))$
(s) $\Box(\mathcal{M}in\_front\_of(J) \supset [go\_to\_door(I)]\mathcal{M}\neg in\_front\_of(J)))$

Now we can define all_door_closed, which builds upon $close\_door(I)$ and specifies how to achieve the goal of closing all doors, assuming the robot to be initially inside the room.

(t) $\langle all\_doors\_closed\rangle\varphi \subset \langle close\_door(door1)\rangle\langle close\_door(door2)\rangle\varphi.$
(u) $\langle all\_doors\_closed\rangle\varphi \subset \langle close\_door(door2)\rangle\langle close\_door(door1)\rangle\varphi.$

Notice that the two clauses defining the procedure all_doors_closed are not mutually exclusive: the doors can be closed in any order. The clauses specify *alternative* recipes that the robot can follow to close all the doors, each of them leading the robot to reach a different position at the end of the task.

2.4 Reasoning on Dynamic Domain Descriptions

In general, a particular dynamic domain will be described in terms of suitable laws and axioms describing precondition and effects of atomic actions, axioms describing the behavior of complex actions and a set of epistemic fluents describing the initial epistemic state

Definition 1 (Dynamic Domain Description). *Given a set $\mathcal{A}$ of atomic world actions, a set $\mathcal{S}$ of sensing actions, and a set $\mathcal{P}$ of procedure names, let $\Pi_{\mathcal{A}}$ be a set of simple action clauses for world actions, $\Pi_{\mathcal{S}}$ a set of axioms of form (4) for sensing actions, $\Pi_{\mathcal{P}}$ a set of axioms of form (8). A* dynamic domain description *is a pair (Π, S_0), where Π is the tuple $(\Pi_{\mathcal{A}}, \Pi_{\mathcal{S}}, \Pi_{\mathcal{P}})$ and S_0 is a consistent and complete set of epistemic fluent literals representing the beliefs of the agent in the initial state.*

Note that $\Pi_{\mathcal{A}}$ contains also the simple actions clauses for the primitive actions that are elements of the non deterministic choice in axioms for sensing actions.

Example 5. An example of domain description is obtained by taking as $\Pi_{\mathcal{A}}$ the set of simple action clauses in Examples 1, 3 and 4 plus the formula (e), as $\Pi_{\mathcal{S}}$ the axiom (f) in Example 3 and as $\Pi_{\mathcal{P}}$ the set of procedure axioms (k-m, t-u) in Example 4. One possible initial set of beliefs is given by state $s = \{\mathcal{B}in\_front\_of(door2), \mathcal{B}\neg in\_front\_of(door1), \mathcal{B}\neg out\_room, \mathcal{U}open(door1), \mathcal{B}open(door2)\}$.

Given a domain description, we can formalize a well known form of reasoning about actions, called *temporal projection*, where the reasoning task is to predict the future effects of actions on the basis of (possibly incomplete) information on preceding states. In particular, we formalize the *temporal projection problem* "given an action sequence $a_1, \dots, a_n$, does the condition Fs hold after the execution of the actions sequence starting from the initial state?" by the query $\langle a_1 \rangle \dots \langle a_n \rangle Fs$ $(n \geq 0)$, where Fs is a conjunction of epistemic literals. [3] We can generalize this query to complex actions $p_1, p_2, \dots, p_n$ by:

$$\langle p_1 \rangle \langle p_2 \rangle \dots \langle p_n \rangle Fs \quad (n \geq 0) \tag{9}$$

where p_i, $i = 1, \dots, n$, is either an atomic action (including sensing actions), or a procedure name, or a test. If $n = 0$ we simply write the above goal as Fs. Query (9) succeeds if it is possible to find a (terminating) execution of $p_1; p_2; \dots; p_n$ leading to a state where Fs holds. Intuitively, when we are faced with a query $\langle p \rangle Fs$ we look for those *terminating execution sequences* which are plans to bring about Fs. In this way we can formalize the *planning problem*: "given an initial state and a condition Fs, is there a sequence of actions that (when executed from the initial state) leads to a state in which Fs holds?". The procedure definitions constrain the search space of reachable states in which to search for the wanted sequence[4].

Example 6. Consider the domain description in Example 5, with the difference that the robot knows that also *door*1 is open. The query

$$\langle all\_door\_closed \rangle (\mathcal{B}\neg open(door1) \wedge \mathcal{B}\neg open(door2))$$

amounts to ask whether it is possible to find a terminating execution of the procedure *all_door_closed* (a plan) which leads to a state where both doors are closed. One terminating execution sequence is the following:

$$toggle\_switch(door2); go\_to\_door(door1); toggle\_switch(door1)$$

3 The Frame Problem

The frame problem is known in the literature on formalization of dynamic domains as the problem of specifying those fluents which remain unaffected by the

[3] Notice that, since primitive actions $a \in \mathcal{A}$ defined in our domain descriptions are *deterministic* w.r.t the epistemic state (see semantic property 2(c) in section 4.1), the equivalence $\langle a \rangle Fs \equiv [a]Fs \wedge \langle a \rangle true$ holds for actions a defined in the domain description, and then, the success of the existential query $\langle a_1 \rangle \dots \langle a_n \rangle Fs$ entails the success of the universal query $[a_1] \dots [a_n]Fs$.

[4] Note that, as a special case, we can define a procedure p which repeatedly selects any atomic action, so that all the atomic action sequences can be taken into account.

execution of a given action. In our formalization, we provide a non-monotonic solution to the *frame problem*. Intuitively, the problem is faced by using persistency assumptions: when an action is performed, any epistemic fluent F which holds in the state before executing the action is assumed to hold in the resulting state unless the action makes it false. As in [6], we model persistency assumptions by abductive assumptions: building upon the monotonic interpretation of a dynamic domain description we provide an abductive semantics to account for this non-monotonic behavior of the language.

First of all, let us introduce some definitions. Given a dynamic domain description (Π, S_0), let us call $\mathcal{L}_{(\Pi,S_0)}$ the propositional modal logic on which (Π, S_0) is based. The axiomatization of $\mathcal{L}_{(\Pi,S_0)}$ contains all the axioms defined at the beginning of section 2 and the axioms $\Pi_{\mathcal{P}}$ and in $\Pi_{\mathcal{S}}$, characterizing complex actions and sensing actions, respectively. The action laws for primitive actions in $\Pi_{\mathcal{A}}$ and the initial beliefs in S_0 define a *theory* fragment $\Sigma_{(\Pi,S_0)}$ in $\mathcal{L}_{(\Pi,S_0)}$. The model theoretic semantics of the logic $\mathcal{L}_{(\Pi,S_0)}$ is given through a standard Kripke semantics with inclusion properties among the accessibility relations [1]. The abductive semantics builds on monotonic logic $\mathcal{L}_{(\Pi,S_0)}$ and is defined in the style of Eshghi and Kowalski's abductive semantics for negation as failure [16]. We define a new set of atomic propositions of the form $\mathbf{M}[a_1][a_2]\dots[a_m]F$ and we take them as being *abducibles*.[5] Their meaning is that the epistemic fluent F can be assumed to hold in the state obtained by executing primitive actions $a_1, a_2, \dots, a_m$. Each abducible can be assumed to hold, provided it is consistent with the domain description (Π, S_0) and with other assumed abducibles. More precisely, we add to the axiom system of $\mathcal{L}_{(\Pi,S_0)}$ the *persistency axiom schema*:

$$[a_1][a_2]\dots[a_{m-1}]F \wedge \mathbf{M}[a_1][a_2]\dots[a_{m-1}][a_m]F \supset [a_1][a_2]\dots[a_{m-1}][a_m]F \quad (10)$$

where $a_1, a_2, \dots, a_m$ $(m > 0)$ are primitive actions, and F is an epistemic fluent (either $\mathcal{B}l$ or $\mathcal{M}l$). Its meaning is that, if F holds after the action sequence $a_1, a_2, \dots, a_{m-1}$, and F can be assumed to persist after action a_m (i.e., it is consistent to assume $\mathbf{M}[a_1][a_2]\dots[a_m]F$), then we can conclude that F holds after performing the sequence of actions $a_1, a_2, \dots, a_m$.

Given a domain description (Π, S_0), let $\models$ be the satisfiability relation in the monotonic modal logic $\mathcal{L}_{(\Pi,S_0)}$ defined above.

Definition 2 (Abductive solution for a dynamic domain description). *A set of* abducibles Δ *is an* abductive solution *for* (Π, S_0) *if, for every epistemic fluent* F*:*

a) $\forall \mathbf{M}[a_1][a_2]\dots[a_m]F \in \Delta$, $\Sigma_{(\Pi,S_0)} \cup \Delta \not\models [a_1][a_2]\dots[a_m]\neg F$
b) $\forall \mathbf{M}[a_1][a_2]\dots[a_m]F \notin \Delta$, $\Sigma_{(\Pi,S_0)} \cup \Delta \models [a_1][a_2]\dots[a_m]\neg F$.

[5] Notice that $\mathbf{M}$ is not a modality. Rather, $\mathbf{M}\alpha$ is the notation used to denote a new atomic proposition associated with α. This notation has been adopted in analogy to default logic, where a justification $\mathbf{M}\alpha$ intuitively means "α is consistent".

Condition a) is a *consistency* condition, which guarantees that each assumption cannot be assumed if its "complementary" formula holds. Condition b) is a *maximality* condition which forces an abducible to be assumed, unless its "complement" is proved. When an action is applied in a certain state, persistency of those fluents which are not modified by the direct effects of the action, is obtained by maximizing persistency assumptions.

Let us now define the notion of abductive solution for a query in a domain description.

Definition 3 (Abductive solution for a query). *Given a domain description* (Π, S_0) *and a query* $\langle p_1; p_2; \ldots; p_n \rangle Fs$, *an* abductive solution for the query in (Π, S_0) *is defined to be an abductive solution* Δ *for* (Π, S_0) *such that* $\Sigma_{(\Pi,S_0)} \cup \Delta \models \langle p_1; p_2; \ldots; p_n \rangle Fs$.

The consistency of an abductive solution, according to Definition 2, is guaranteed by the seriality of $\mathcal{B}$ (from which $\neg(\mathcal{B}l \wedge \mathcal{B}\neg l)$ holds for any literal l). However the presence of action laws with contradictory effects for a given action may cause unintended solutions which are obtained by the contraposition of action laws. Such unintended solutions can be avoided by introducing an *e-consistency* requirement on domain descriptions, as for the language $\mathcal{A}$ in [15]. Essentially we require that, for any set of action laws (for a given action) which may be applicable in the same state, the set of their effects is consistent. Assuming that the domain description is *e-consistent*, the following property holds for abductive solutions.

Proposition 1. *Given an e-consistent dynamic domain description* (Π, S_0)*, there is a unique abductive solution for* (Π, S_0).

4 Proof Procedure: Finding Correct Plans

In section 4.1 we present a proof procedure which constructs a linear plan, by making assumptions on the possible result of sensing actions which are needed for the plan to reach the wanted goal. In section 4.2 we introduce a proof procedure that constructs a conditional plan which achieves the goal for all the possible outcomes of the sensing actions.

4.1 Linear Plan Generation

In this section we introduce a *goal directed proof procedure* based on *negation as failure* (NAF) which allows a query to be proved from a given dynamic domain description. From a procedural point of view our non-monotonic way of dealing with the frame problem consists in using negation as failure, in order to verify that the *complement* of the epistemic fluent F is not made true in the state resulting from an action execution, while in the modal theory we adopted an abductive characterization to deal with persistency. However, it is well studied how to give an abductive semantics for NAF [16].

The first part of the proof procedure, denoted by "$\vdash_{ps}$" and presented in Fig. 2, deals with the execution of complex actions, sensing actions, primitive actions and test actions. The proof procedure reduces the complex actions in the query to a sequence of primitive actions and test actions, and verifies if execution of the primitive actions is possible and if the test actions are successful. To do this, it reasons about the execution of a sequence of primitive actions from the initial state and computes the values of fluents at different states. During a computation, a *state* is represented by a sequence of primitive actions $a_1, \ldots, a_m$. The value of fluents at a state is not explicitly recorded but it is computed when needed in the computation. The second part of the procedure, denoted by "$\vdash_{fs}$" and presented in Fig. 3, allows the values of fluents in a state to be determined.

A query of the form $\langle p_1; p_2; \ldots ; p_n\rangle Fs$, where p_i, $1 \leq i \leq n$ $(n \geq 0)$, is either a primitive action, or a sensing action, or a procedure name, or a test, succeeds if it is possible to execute $p_1, p_2, \ldots, p_n$ (in the order) starting from the current state, in such a way that Fs holds at the resulting state. In general, we will need to establish if a goal holds at a given state. Hence, we will write:

$$a_1, \ldots, a_m \vdash_{ps} \langle p_1; p_2; \ldots ; p_n\rangle Fs \text{ with answer (w.a.) } \sigma$$

to mean that the query $\langle p_1; p_2; \ldots ; p_n\rangle Fs$ can be proved from the domain description (Π, S_0) at the state $a_1, \ldots, a_m$ with answer σ, where σ is an action sequence $a_1, \ldots, a_m, \ldots a_{m+k}$ which represents the state resulting by executing $p_1, \ldots, p_n$ in the current state $a_1, \ldots, a_m$. We denote by ε the initial state.

The five rules of the derivation relation $\vdash_{ps}$ in Fig. 2 define, respectively, *how to execute* procedure calls, test actions, sensing actions and primitive actions:[6]

To execute a complex action p we non-deterministically replace the modality $\langle p\rangle$ with the modality in the antecedent of a suitable axiom for it (rule 1). To execute a test action $(Fs)?$, the value of Fs is checked in the current state. If Fs holds in the current state, the state action is simply eliminated, otherwise the computation fails (rule 2). To execute a primitive action a, first we need to verify if that action is possible by using the precondition laws. If these conditions hold we can move to a new state in which the action has been performed (rule 3). To execute a sensing action s (rule 4) we non-deterministically replace it with one of the primitive actions which define it (see Section 2.2), that, when it is executable, will cause $\mathcal{B}l$ and $\mathcal{B}\neg l'$, for each $l' \in dom(s)$, with $l \neq l'$. Rule 5) deals with the case when there are no more actions to be executed. The sequence of primitive actions to be executed $a_1, \ldots, a_m$ has been already determined and, to check if Fs is true after $a_1, \ldots, a_m$, proof rules 6)-10) below are used.

The second part of the procedure (see Fig. 3) determines the derivability of an epistemic fluent conjunction Fs at a state $a_1, \ldots, a_m$, denoted by $a_1, \ldots, a_m \vdash_{fs} Fs$, and it is defined inductively on the structure of Fs.

[6] Note that it can deal with a more general form of action laws and precondition laws than the ones presented in Section 2. In particular, it deals with action law of the form $\Box(Fs \supset [a]F)$ and precondition law of the form $\Box(Fs \supset \langle a\rangle true)$, where Fs is an arbitrary conjunction of epistemic fluents and F is an epistemic fluent, respectively.

$$1)\quad \frac{a_1,\dots,a_m \vdash_{ps} \langle p'_1;\dots;p'_{n'};p_2;\dots;p_n\rangle Fs \text{ w. a. } \sigma}{a_1,\dots,a_m \vdash_{ps} \langle p;p_2;\dots;p_n\rangle Fs \text{ w. a. } \sigma} \qquad \begin{array}{c}\text{where } p \in \mathcal{P} \text{ and}\\ \langle p\rangle\varphi \subset \langle p'_1;\dots;p'_{n'}\rangle\varphi\\ \in \Pi_{\mathcal{P}}\end{array}$$

$$2)\quad \frac{a_1,\dots,a_m \vdash_{fs} Fs' \quad a_1,\dots,a_m \vdash_{ps} \langle p_2;\dots;p_n\rangle Fs \text{ w. a. } \sigma}{a_1,\dots,a_m \vdash_{ps} \langle (Fs')?;p_2;\dots;p_n\rangle Fs \text{ w. a. } \sigma}$$

$$3)\quad \frac{a_1,\dots,a_m \vdash_{fs} Fs' \quad a_1,\dots,a_m,a \vdash_{ps} \langle p_2;\dots;p_n\rangle Fs \text{ w. a. } \sigma}{a_1,\dots,a_m \vdash_{ps} \langle a;p_2;\dots;p_n\rangle Fs \text{ w. a. } \sigma} \qquad \begin{array}{c}\text{where } a \in \mathcal{A} \text{ and}\\ \Box(Fs' \supset \langle a\rangle true)\\ \in \Pi_{\mathcal{A}}\end{array}$$

$$4)\quad \frac{a_1,\dots,a_m \vdash_{ps} \langle s^{\mathcal{B}l};p_2;\dots;p_n\rangle Fs \text{ w. a. } \sigma}{a_1,\dots,a_m \vdash_{ps} \langle s;p_2;\dots;p_n\rangle Fs \text{ w. a. } \sigma} \qquad \begin{array}{c}\text{where } s \in \mathcal{S} \text{ and}\\ l \in dom(s)\end{array}$$

$$5)\quad \frac{a_1,\dots,a_m \vdash_{fs} Fs}{a_1,\dots,a_m \vdash_{ps} \langle \varepsilon\rangle Fs \text{ w. a. } \sigma} \qquad \begin{array}{c}\text{where}\\ \sigma = a_1;\dots;a_m\end{array}$$

Fig. 2. The derivation relation $\vdash_{ps}$

An epistemic fluent F holds at state $a_1, a_2, \dots, a_m$ if: either F is an immediate effect of action a_m, whose preconditions hold in the previous state (rule 7a); or the last action, a_m, is an *ad hoc* primitive action s^F (introduced to model the sensing action s), whose effect is that of adding F to the state(rule 7b); or F holds in the previous state $a_1, a_2, \dots, a_{m-1}$ and it persists after executing a_m (rule 7c); or $a_1, a_2, \dots, a_m$ is the initial state and F is in it. Notice that rule 7(c) allows to deal with the *frame problem*: F persists from a state $a_1, a_2, \dots, a_{m-1}$ to the next state $a_1, a_2, \dots, a_m$ unless a_m makes $\neg F$ true, i.e. it persists if $\neg F$ fails from $a_1, a_2, \dots, a_m$. In rule 7c **not** represents *negation as failure.*

We say that a query $\langle p_1;p_2;\dots;p_n\rangle Fs$ *succeeds* from a dynamic domain description (Π, S_0) if it is operationally derivable from (Π, S_0) in the initial state ε by making use of the above proof rules with the *execution trace* σ as answer (i.e. $\varepsilon \vdash_{ps} \langle p_1;p_2;\dots;p_n\rangle Fs$ with answer σ). Notice that the proof procedure does not perform any consistency check on the computed abductive solution. However, under the assumption that the domain description is e-consistent and that the beliefs on the initial state S_0 are consistent, soundness of the proof procedure above can be proved w.r.t. the unique acceptable solution.

Theorem 1. *Let (Π, S_0) be an e-consistent dynamic domain description and let $\langle p_1;p_2;\dots;p_n\rangle Fs$ be a query. Let Δ be the unique abductive solution for (Π, S_0). If $\langle p_1;p_2;\dots;p_n\rangle Fs$ succeeds from (Π, S_0) with answer σ, then $\Sigma_{(\Pi,S_0)} \cup \Delta \models \langle p_1;p_2;\dots;p_n\rangle Fs$.*

The proof is omitted for sake of brevity. It is by induction on the rank of the derivation of the query, and it makes use of a soundness and completeness result for the monotonic part of the proof procedure presented in this section w.r.t. the monotonic part of the semantics. Indeed, if the assumptions $\mathbf{M}[a_1][a_2]\dots[a_m]F$ are regarded as facts rather then abducibles and they are added to the program,

6) $$\frac{}{a_1, \ldots, a_m \vdash_{fs} \mathit{true}}$$

7a) $$\frac{a_1, \ldots, a_{m-1} \vdash_{fs} Fs'}{a_1, \ldots, a_m \vdash_{fs} F} \qquad \text{where } m > 0 \text{ and } \Box(Fs' \supset [a_m]F) \in \mathit{\Pi}_{\mathcal{A}}$$

7b) $$\frac{}{a_1, \ldots, a_m \vdash_{fs} F} \qquad \text{where } a_m = s^F$$

7c) $$\frac{\textbf{not } a_1, \ldots, a_m \vdash_{fs} \neg F \quad a_1, \ldots, a_{m-1} \vdash_{fs} F}{a_1, \ldots, a_m \vdash_{fs} F} \qquad \text{where } m > 0$$

7d) $$\frac{}{\varepsilon \vdash_{fs} F} \qquad \text{where } F \in S_0$$

8) $$\frac{a_1, \ldots, a_m \vdash_{fs} Fs_1 \quad a_1, \ldots, a_m \vdash_{fs} Fs_2}{a_1, \ldots, a_m \vdash_{fs} Fs_1 \wedge Fs_2}$$

9) $$\frac{a_1, \ldots, a_m \vdash_{fs} \mathcal{B}l}{a_1, \ldots, a_m \vdash_{fs} \mathcal{M}l}$$

Fig. 3. The derivation relation $\vdash_{fs}$

the non-monotonic step 7c) in the proof procedure can be replaced by a monotonic one. The resulting monotonic proof procedure can be shown to be sound and complete with respect to the Kripke semantics of the modal logic $\mathcal{L}_{(\mathit{\Pi},S_0)}$.

Our proof procedure computes just one solution, while abductive semantics may give multiple solutions for a domain description. As stated in proposition 1, the requirement of e-consistency ensures that a domain description has a unique abductive solution. Under these condition we argue that completeness of the proof procedure can be proved.

Since a query $\langle p_1; \ldots; p_n\rangle Fs$ is an existential formula, a successful answer σ represents a possible execution of the sequence $p_1, \ldots, p_n$. Indeed, for the answer σ we can prove the Proposition 2. Property (a) says that σ is a possible execution of $p_1, \ldots, p_n$ while (b) says that the plan σ is *correct* w.r.t. Fs. Notice that, since σ is a sequence of primitive actions $a \in \mathcal{A}$, property (b) is a consequence of the fact that there is only one epistemic state reachable executing an action a, i.e. primitive actions are *deterministic*, as stated by property (c).

Proposition 2. *Let $(\mathit{\Pi}, S_0)$ be an e-consistent dynamic domain description and let $\langle p_1; p_2; \ldots; p_n\rangle Fs$ be a query. Let $\mathit{\Delta}$ be the unique abductive solution for $(\mathit{\Pi}, S_0)$. If $\varepsilon \vdash_{ps} \langle p_1; p_2; \ldots; p_n\rangle Fs$ with answer σ then:*

(a) $\mathit{\Sigma}_{(\mathit{\Pi},S_0)} \cup \mathit{\Delta} \models \langle\sigma\rangle Fs \supset \langle p_1; p_2; \ldots; p_n\rangle Fs$;
(b) $\mathit{\Sigma}_{(\mathit{\Pi},S_0)} \cup \mathit{\Delta} \models [\sigma]Fs$;
(c) $\mathit{\Sigma}_{(\mathit{\Pi},S_0)} \cup \mathit{\Delta} \models \langle a\rangle Fs \supset [a]Fs$

4.2 Conditional Plan Generation

In this section we introduce a proof procedure that constructs a conditional plan which achieves the goal for all the possible outcomes of the sensing actions. Let us start with an example.

Example 7. Consider the Example 5 and the query

$$\langle all\_door\_closed\rangle(\mathcal{B}\neg open(door1) \wedge \mathcal{B}\neg open(door2))$$

We want to find an execution of *all_door_closed* reaching a state where all the doors of the room are closed. When it is unknown in the initial state if $door1$ is open, the action sequence the agent has to perform to achieve the goal depends on the outcome of the sensing action $sense\_door(door1)$. Indeed, after performing the action sequence $toggle\_switch(door2); go\_to\_door(door1)$ the robot has to execute the sensing on $door1$ in order to know if it is open or not. The result of sensing conditions the robot's future course of actions: if it comes to know that the door it is closed, it will execute the action $toggle\_switch(door1)$, otherwise it will not do anything. Given the query above, the proof procedure described in the previous section extracts the following primitive action sequences, making assumptions on the possible results of $sense\_door(door1)$:

- $toggle\_switch(door2); go\_to\_door(door1);$
  $sense\_door(door1)^{\mathcal{B}open(door1)}; toggle\_switch(door1)$ and
- $toggle\_switch(door2); go\_to\_door(door1); sense\_door(door1)^{\mathcal{B}\neg open(door1)}$.

Instead the proof procedure we are going to present, given the same query, will look for a conditional plan that achieves the goal $\mathcal{B}\neg open(door1) \wedge \mathcal{B}\neg open(door2)$ for *any outcome* of the sensing, as the following:

$toggle\_switch(door2);$
$go\_to\_door(door1);$
$sense\_door(door1);$
$\quad((\mathcal{B}open(door1)?);$
$\quad\quad toggle\_switch(door1))\cup$
$\quad(\mathcal{B}\neg open(door(1)?)))$

Intuitively, given a query $\langle p\rangle Fs$, the proof procedure we are going to define computes a conditional plan σ (if there is one), which determines the actions to be executed for all possible results of the sensing actions. All the executions of the conditional plan σ are possible behaviours of the procedure p. Let us define inductively the structure of such conditional plans.

Definition 4. *Conditional plan*

1. *a (possibly empty) action sequence* $a_1; a_2; \ldots; a_n$ *is a conditional plan;*

2. if $a_1; a_2; \ldots a_n$ is an action sequence, $s \in \mathcal{S}$ is a sensing action, and $\sigma_1, \ldots, \sigma_t$ are conditional plans then $a_1; a_2; \ldots; a_n; s; ((\mathcal{B}l_1?); \sigma_1 \cup \ldots \cup (\mathcal{B}\neg l_t?); \sigma_t)$ is a conditional plan, where $l_1, \ldots, l_t \in dom(s)$.

Given a query $\langle p_1; p_2; \ldots; p_n \rangle Fs$ the proof procedure constructs, as answer, a conditional plan σ such that: 1) all the executions of σ are possible executions of $p_1; p_2; \ldots; p_n$ and 2) all the executions of σ lead to a state in which Fs holds. The proof procedure is defined on the bases of the previous one. We simply need to replace step 4) above (dealing with the execution of sensing actions) with the following step:

$$\text{4-bis)}\ \frac{\forall l_i \in \mathcal{F},\ a_1, \ldots, a_m \vdash_{ps} \langle s^{\mathcal{B}l_i}; p_2; \ldots; p_n \rangle Fs \text{ w. a. } a_1; \ldots; a_m; s^{\mathcal{B}l_i}; \sigma'_i}{a_1, \ldots, a_m \vdash_{ps} \langle s; p_2; \ldots; p_n \rangle Fs \text{ w. a. } a_1; \ldots; a_m; s; ((\mathcal{B}l_1?); \sigma'_1 \cup \ldots \cup (\mathcal{B}l_t?); \sigma'_t)}$$

where $s \in \mathcal{S}$ and $\mathcal{F} = \{l_1, \ldots, l_t\} = dom(s)$.

As a difference with the previous proof procedure, when a sensing action is executed, the procedure has to consider all possible outcomes of the action, so that the computation splits in more branches. If all branches lead to success, it means that the main query succeeds for all the possible results of action s. In such a case, the conditional plan σ will contain the σ'_i's as alternative sub-plans.

The following theorem states the soundness of the proof procedure for generating conditional plans (a) and the correctness of the conditional plan σ w.r.t. the conjunction of epistemic fluents Fs and the initial situation S_0 (b). In particular, (b) means that executing the plan σ (constructed by the procedure) always leads to a state in which Fs holds, for all the possible results of the sensing actions.

Theorem 2. *Let (Π, S_0) be a dynamic domain description and let $\langle p_1; p_2; \ldots; p_n \rangle Fs$ be a query. Let Δ be the unique abductive solution for (Π, S_0). If $\langle p_1; p_2; \ldots; p_n \rangle Fs$ succeeds from (Π, S_0) with answer σ, then:*

(a) $\Sigma_{(\Pi,S_0)} \cup \Delta \models \langle p_1; p_2; \ldots; p_n \rangle Fs$;
(b) $\Sigma_{(\Pi,S_0)} \cup \Delta \models [\sigma] Fs$.

5 Implementation and Applications

On the basis of the presented logic formalization, a logic programming language, named DyLOG, has been defined. An interpreter based on the proof procedure introduced in Section 4 has been implemented in Sicstus Prolog. This implementation allow DyLOG to be used as an ordinary programming language for *executing* procedures which model the behavior of an agent, but also for reasoning about them, by extracting linear or conditional plans. Moreover, the implementation deals with domain descriptions containing a simple form of *causal laws* [6] and *functional fluents* with associated finite domain, which are not explicitly treated in this paper.

In [7] it is shown how DyLOG can be used to model various kind of agents, such as goal directed or reactive agents. In particular, we experimented the use of DyLOG as an agent logic programming language to implement Adaptive Web

Applications, where a personalized dynamical site generation is guided by the user's goal and constraints [3,4]. When a user connects to a site managed by one of our agents, (s)he does not access to a fixed graph of pages and links but (s)he interacts with an agent that, starting from a knowledge base specific to the site and from the requests of the user, builds an *ad hoc* site structure. In our approach such a structure corresponds to a plan aimed at pursuing the user's goal, which is automatically generated by exploiting the *planning capabilities* of DyLOG agents. Run-time adaptation occurs at the navigation level. Indeed the agent defines the navigation possibilities available to the user and determines which page to display based on the current dynamics of the interaction.

In order to check the validity of the proposed approach, we have implemented a client-server agent system, named WLog, and we have applied it to the specific case of a virtual computer seller. In this application the users connect to the structureless web site for having a PC assembled; the assembly process is done through the interaction between the user and a software agent, the virtual seller. Similarly to what happens in a real shop, the choice of what to buy is taken thanks to a dialogue, guided by the seller, that ends up with the definition of a configuration that satisfies the user. In the current implementation the server-side of the system consists of two kinds of agents: *reasoners* and *executors*. Reasoners are programs written in DyLOG, whereas executors are Java Servlets embedded in an Apache web server. A DyLOG reasoner generates presentation plans; once built the plan is executed and the actual execution of the actions in the plan consists in showing web pages to the user. Actual execution is the task of the executors. The connection between the two kinds of agents has the form of message exchange. Technical information about the system, the Java classes that we defined, and the DyLOG programs can be found at `http://www.di.unito.it/~alice`.

6 Conclusions and Related Work

In this paper we presented a logic formalism for reasoning about actions, which combines the capabilities to handle actions affecting knowledge and to express complex actions in a modal action theory.

The problem of reasoning about actions in presence of sensing and of incomplete states has been tackled by many authors in the literature. In the Scherl and Levesque' work [25] a framework has been proposed to formalize knowledge-producing actions in classical logic, adapting the possible world model of knowledge to the situation calculus. As a difference, we describe an epistemic state by a set of epistemic literals, a simplification similar to the one considered in [8] when 0-approximate states are defined, which leads to a loss of expressivity, but to a gain in tractability.

In [21] Levesque formulates the planning task in domains including sensing. Starting from the theory of sensing actions in situation calculus presented in [25], he defines complex plans as robot programs, that may contain sensing actions, conditionals and loops, and specifies the planning task as the problem to find a robot program achieving a desired goal from a certain initial state. However the paper does not suggest how to generate automatically such robot plans, while we presented a proof procedure to deal with it (Section 4.2).

The works in [23,8] have tackled the problem of extending the Gelfond and Lifschitz' language $\mathcal{A}$ for reasoning about complex plans in presence of sensing and incomplete information. In [23] Lobo et al. introduce the language $\mathcal{A}_K$ which provides both actions to increase agent knowledge and actions to lose agent knowledge. It has a general semantics in which epistemic states are represented by sets of worlds. However, in $\mathcal{A}_K$ it is not possible for the agent to query itself about its knowledge (the agent has no introspection). Precondition laws to rule executability of actions are not provided. In particular, knowledge laws have preconditions on the effects of actions rather than on their executability. Given a domain description in $\mathcal{A}_K$, a query of the form ϕ **after** $[\alpha]$ is true if ϕ holds in every model of D after the execution of the plan α in the initial state, where α is a complex plan, possibly including conditionals and iterations. As a difference, rather than verifying the correctness of a plan, in this paper we have addressed the problem of finding a finite conditional plan (a possible execution of a procedure) which is provably correct with respect to a given condition. In [8] Baral and Son define an action description language, also called $\mathcal{A}_K$, which deals with sensing actions and distinguishes between the state of the world and the state of knowledge of an agent about the world. The semantics of the language is proved to be equivalent to the one in [23] when rational models are considered. Baral and Son [8] define several sound approximation of the language $\mathcal{A}_K$ with a smaller state space with respect to $\mathcal{A}_K$, based on three-valued interpretations. Our approach has strong similarities with the 0-Approximation. Indeed, our epistemic states are, essentially, three-valued models and, as for the 0-Approximation, our language does not provide reasoning about cases. The meaning of queries in [8] is substantially similar to the one in [23] and, therefore, it is different from ours. As a difference, conditional plans in [8] do not allow iteration.

In [13] De Giacomo and Rossati present a minimal knowledge approach to reasoning about actions and sensing, by proposing a formalism which combines the modal μ-calculus and autoepistemic logic. They have epistemic formulas $\mathcal{B}p$ (where p is a literal conjunction) and they allow precondition laws of the form $\mathcal{B}p \supset \langle a \rangle true$ and action laws of the form $\mathcal{B}p \supset [a]\mathcal{B}q$. On the other hand, their domain description does not contain formulas of the form $\mathcal{M}p \supset [a]\mathcal{M}q$, which in our case are needed for describing the possible effects of an action when there is uncertainty about its preconditions. Moreover, actions which make the agent to lose information are not provided. An algorithm is introduced to compute a transition graph from an action specification and verify properties of the possible executions through model checking. The treatment of sensing actions in the construction of the transition graph is similar to ours, in that, a sensing action is regarded as the non-deterministic choice of two atomic actions. As a difference, sensing actions do not affect fluents whose value is known before their execution.

In a recent work Thielscher [27] faces the problem of representing a robot's knowledge about its environment in the context of the Fluent Calculus, a formalism for reasoning about actions based on predicate logic. In order to account for knowledge, basic fluent calculus is extended by introducing the concept of possible world state and defining knowledge of a robot in terms of possible states. The formalism deals with sensing actions and it allows to distinguish between state of the world and state of knowledge of an agent about the world. A monotonic solution to the frame problem for knowledge is provided, by means of suitable knowledge update axioms but, as a difference with [8], indipendent specifications of state and knowledge update can be given. A concept of conditional action, denoted by $If(f, a)$, is introduced in order to deal with planning in presence of sensing. Such If-constructs allow the robot to condition its course of actions on the result of sensing actions included in its plan. However If-constructs uses only atomic conditions, while our formalism allow to express as complex actions conditional constructs with arbitrary complex conditions.

As concerns the problem of defining complex actions, there is a close relation between our language and GOLOG [22], though, from the technical point of view, it is based on a different approach. While our language makes use of modal logic, GOLOG is based on classical logic and, more precisely, on the situation calculus. In our case, procedures are defined as axioms of our modal logic, while in GOLOG they are defined by macro expansion into formulae of the situation calculus. GOLOG definition is very general, but it makes use of second order logic to define iteration and procedure definition. Hence there is a certain gap between the general theory on which GOLOG is based and its implementation in Prolog. In contrast, we have tried to keep the definition of the semantics of the language and of its proof procedure as close as possible.

References

1. M. Baldoni. *Normal Multimodal Logics: Automatic Deduction and Logic Programming Extension.* PhD thesis, Dipartimento di Informatica, Università degli Studi di Torino, Italy, 1998. Available at `http://www.di.unito.it/~baldoni/`. 415
2. M. Baldoni. Normal Multimodal Logics with Interaction Axioms. In D. Basin, M. D'Agostino, D. M. Gabbay, S. Matthews, and L. Viganò, editors, *Labelled Deduction*, volume 17 of *Applied Logic Series*, pages 33–53. Applied Logic Series, Kluwer Academic Publisher, 2000. 407, 410, 412
3. M. Baldoni, C. Baroglio, A. Chiarotto, and V. Patti. Programming Goal-driven Web Sites using an Agent Logic Language. In I. V. Ramakrishnan, editor, *Proc. of the Third International Symposium on Practical Aspects of Declarative Languages*, volume 1990 of *LNCS*, pages 60–75, Las Vegas, Nevada, USA, 2001. Springer. 422
4. M. Baldoni, C. Baroglio, and V. Patti. Structereless, Intention-guided Web Sites: Planning Based Adaptation. In *Proc. 1st International Conference on Universal Access in Human-Computer Interaction, a track of HCI International 2001*, New Orleans, LA, USA, 2001. Lawrence Erlbaum Associates. To appear. 422
5. M. Baldoni, L. Giordano, and A. Martelli. A Tableau Calculus for Multimodal Logics and Some (un)Decidability Results. In H. de Swart, editor, *Proc. of*

TABLEAUX'98, volume 1397 of *LNAI*, pages 44–59. Springer-Verlag, 1998. 407, 412

6. M. Baldoni, L. Giordano, A. Martelli, and V. Patti. An Abductive Proof Procedure for Reasoning about Actions in Modal Logic Programming. In J. Dix et al., editor, *Proc. of NMELP'96*, volume 1216 of *LNAI*, pages 132–150, 1997. 406, 415, 421
7. M. Baldoni, L. Giordano, A. Martelli, and V. Patti. Modeling agents in a logic action language. In *Proc. of Workshop on Practical Reasoning Agents, FAPR2000*, 2000. to appear. 421
8. C. Baral and T. C. Son. Formalizing Sensing Actions - A transition function based approach. *Artificial Intelligence*, 125(1-2):19–91, January 2001. 405, 406, 422, 423, 424
9. M. Castilho, O. Gasquet, and A. Herzig. Modal tableaux for reasoning about actions and plans. In S. Steel, editor, *Proc. ECP'97*, LNAI, pages 119–130, 1997. 406
10. G. De Giacomo and M. Lenzerini. PDL-based framework for reasoning about actions. In *Proc. of AI*IA '95*, volume 992 of *LNAI*, pages 103–114, 1995. 406
11. G. De Giacomo, Y. Lespérance, and H. J. Levesque. Reasoning about concurrent execution, prioritized interrupts, and exogenous actions in the situation calculus. In *Proceedings of IJCAI'97*, pages 1221–1226, Nagoya, August 1997. 405
12. G. De Giacomo and H. J. Levesque. An Incremental Interpreter for High-Level Programs with Sensing. In *Proceedings of the AAAI 1998 Fall Symposium on Cognitive Robotics*, Orlando, Florida, USA, October 1998. 405
13. G. De Giacomo and R. Rosati. Minimal knowledge approach to reasoning about actions and sensing. In *Proc. of NRAC'99*, Stockholm, Sweden, August 1999. 423
14. S. Demri. The Complexity of Regularity in Grammar Logics. *J. of Logic and Computation*, 2001. To appear, available at `http://www.lsv.ens-cachan.fr/~demri/`. 412
15. M. Denecker and D. De Schreye. Representing Incomplete Knowledge in Abduction Logic Programming. In *Proc. of ILPS '93*, Vancouver, 1993. The MIT Press. 416
16. K. Eshghi and R. Kowalski. Abduction compared with Negation by Failure. In *Proc. of ICLP '89*, Lisbon, 1989. The MIT Press. 415, 416
17. M. Gelfond and V. Lifschitz. Representing action and change by logic programs. *Journal of Logic Programming*, 17:301–321, 1993. 405
18. L. Giordano, A. Martelli, and C. Schwind. Dealing with concurrent actions in modal action logic. In *Proc. ECAI-98*, pages 537–541, 1998. 406, 408
19. L. Giordano, A. Martelli, and Camilla Schwind. Ramification and causality in a modal action logic. *Journal of Logic and Computation*, 10(5):625–662, 2000. 406, 408
20. D. Harel. Dynamic Logic. In D. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic*, volume II, pages 497–604. D. Reidel, 1984. 406, 410, 411
21. H. J. Levesque. What is planning in the presence of sensing? In *Proc. of the AAAI-96*, pages 1139–1146, 1996. 422
22. H. J. Levesque, R. Reiter, Y. Lespérance, F. Lin, and R. B. Scherl. GOLOG: A Logic Programming Language for Dynamic Domains. *J. of Logic Prog.*, 31, 1997. 406, 424
23. J. Lobo, G. Mendez, and S. R. Taylor. Adding Knowledge to the Action Description Language $\mathcal{A}$. In *Proc. of AAAI'97/IAAI'97*, pages 454–459, Menlo Park, 1997. 405, 423
24. H. Prendinger and G. Schurz. Reasoning about action and change. a dynamic logic approach. *Journal of Logic, Language, and Information*, 5(2):209–245, 1996. 406

25. R. Scherl and H. J. Levesque. The frame problem and knowledge-producing actions. In *Proc. of the AAAI-93*, pages 689–695, Washington, DC, 1993. 405, 406, 422
26. C. B. Schwind. A logic based framework for action theories. In J. Ginzburg et al., editor, *Language, Logic and Computation*, pages 275–291. CSLI, 1997. 406
27. M. Thielscher. Representing the Knowledge of a Robot. In *Proc. of the International Conference on Principles of Knowledge Representation and reasoning, KR'00*, pages 109–120. Morgan Kaufmann, 2000. 405, 424

E-unifiability via Narrowing*

Emanuele Viola

Dip. di Scienze dell'Informazione, Univ. di Roma "La Sapienza"
V. Salaria 113, 00198 Roma, Italy
viola@dsi.uniroma1.it

Abstract. We formulate a narrowing-based decision procedure for E-unifiability. Termination is obtained requiring a *narrowing bound*: a bound on the length of narrowing sequences. We study general conditions under which the method guarantees that E-unifiability is in *NP*. The procedure is also extended to narrowing *modulo AC* (associativity and commutativity). As an application of our method, we prove NP-completeness of unifiability modulo bisimulation in process algebra with proper iteration, significantly extending a result in [8]. We also give (new) proofs, under a unified point of view, of NP-decidability of I, ACI, $ACI1$-unifiability and of unifiability in quasi-groups and central groupoids.

1 Introduction

E-unification is concerned with solving term equations modulo an *equational theory E* [11,3]. It is a fundamental tool in theorem proving, logic programming and type assignment systems. *Narrowing* is a well-known technique that can be used as a *general E*-unification procedure in the presence of a *term rewriting system* (TRS) [2]. *Narrowing* a term is finding the minimal instantiation of it such that one rewrite step becomes applicable, and to apply it. If this process is applied to an equation and is iterated until finding an equation whose both terms are syntactically unifiable, then the composition of the *most general unifier* with all the substitutions computed during the narrowing sequence yields an *E-unifier* of the initial equation. The narrowing process that builds all the possible narrowing sequences starting from the equation to be solved, is an E-unification procedure that yields complete sets of unifiers, provided that E can be presented by a *convergent* (i.e. *confluent* and *terminating*) rewrite system [3]. However, in general this procedure does not terminate.

Hullot [9] gives sufficient conditions for this procedure to be terminating. His results also extend to equational narrowing. However, the TRS must satisfy strong requirements and no complexity analysis of the procedure is given. Furthermore, there is not much hope to find low complexity bounds, as long as we want complete sets of unifiers, since in simple cases their cardinality is unfeasible [14].

[*] Work carried out within the MURST project TOSCA (Theory of concurrency, higher order and types)

A. Restivo, S. Ronchi Della Rocca, L. Roversi (Eds.): ICTCS 2001, LNCS 2202, pp. 426–438, 2001.

However, constraint approaches to theorem proving [15,5] and logic programming [6], need the computation of finite complete sets of unifiers no longer for many applications. It is sufficient to decide solvability of the E-unification problems, namely *E-unifiability.*

Nieuwenhuis [18], using *basic paramodulation* techniques, shows that E-unifiability is in NP if E is *shallow* (i.e. variables at depth at most one). But there is no extension to equational paramodulation and being shallow is very restrictive.

Here we study E-unifiability via narrowing. This paper does not aim to define yet another refinement of the basic narrowing which does not destroy completeness (for a survey, see [3]). We show how optimal and new complexity results can be obtained considering a simple *bound on the length of the basic narrowing sequences.* Our method is *lazy,* in the sense that *we never compute unifiers.* Instead, we add equations to our unification problem, and only at the end we check the unifiability of all the equations we have created.

We show that if for every rule $l \to r$ in a convergent TRS we have that r is a subterm of l, then E-unifiability is in NP (where E is equivalent to the TRS). This is enough, for example, to give (new) proofs of NP-decidability of unifiability in quasi-groups, central groupoids and of I-unifiability (*idempotency*, i.e. $x + x \approx x$).

Of course not every theory can be presented as a convergent TRS. One of the main reasons is that some (sets of) axioms cannot be oriented into terminating (sets of) rewrite rules. Among these there is AC *(associativity*, i.e. $x + (y + z) \approx (x + y) + z$, and *commutativity*, i.e. $x + y \approx y + x$), satisfied by many common binary operations. Consequently, we extend our results to narrowing modulo AC. Being lazy is particularly important in this case, because the cardinality of a minimal complete set of AC-unifiers may be doubly-exponential [14,17].

The most important application we consider here is in the field of *process algebra* [4]: NP-completeness of unifiability modulo bisimulation in minimal process algebra with proper iteration [7]. This is a significant extension of a result in [8].

Again, we get new proofs of classical results: NP-decidability of ACI, $ACI1$-unifiability (1 stands for *existence of unity*, i.e. $x + 1 \approx x$).

This paper is organized as follows. In the next section we give some preliminaries. In Section 3 we define our narrowing-based decision procedure, study its complexity and compare our results with Hullot's and with Nieuwenhuis'. We also give some applications. In Section 4 we extend our results to AC-narrowing and detail an application in process algebra. We also discuss other applications. Section 5 is a conclusion.

2 Preliminaries

We assume that the reader is familiar with terms, equational theories, E-unification, term rewriting systems and related topics [2]. We denote by $Term(\sum \bigcup X)$ the set of *terms* generated by the *signature* $\sum$ and the set of *variable symbols* X. We denote the set of *positions* of a term s by $Pos(s)$. For

$p \in Pos(s)$, $s|_p$ is the *subterm of s at position p*, and $s[t]_p$ is the term obtained from s by *replacing the subterm at position p by the term t*. We denote by $\overline{Pos}(s)$ the set of non variable positions of s: $\{p : p \in Pos(s) \text{ and } s|_p \notin X\}$. The *size* of a term s is $|Pos(s)|$. We may write a *substitution* θ as $\{x_1 \leftarrow \theta x_1, ..., x_n \leftarrow \theta x_n\}$ if its *domain*, denoted by $Dom(\theta)$, is $\{x_1, ..., x_n\}$. Let E be an *equational theory.* It is convenient to see an *equation* over $\sum$ modulo E as a term with top symbol $=^?_E$ $(=^?_E \notin \sum)$, i.e. $s =^?_E t$ where $s, t \in Term(\sum \cup X)$. We write $\theta \models s =^?_E t$ for $\theta s =_E \theta t$. An *E-unification problem* over $\sum$ is a *finite* set of equations modulo E. In this paper every E-unification problem is *general*, i.e. it may contain arbitrary function symbols not occurring in E. Let Π be an E-unification problem. The *size* of Π is $\sum_{e \in \Pi} size(e)$. Π is *E-unifiable* if there is a substitution θ such that for any $e \in \Pi$, $\theta \models e$. In this case we write $\theta \models \Pi$ and θ is said to be a *E-unifier.* If θ is a substitution, we write $\theta\Pi$ for $\{\theta e : e \in \Pi\}$.

A *term rewriting system* (TRS) T is a *finite* set of *identities*, called (*rewrite*) *rules* and written $l \to r$, such that for any $l \to r \in T$, $l \notin X$ and $var(l) \supseteq var(r)$ ($var(s)$ is the set of variables occurring in s). Let T be a TRS. We write $s \to_T t$ (s *reduces* to t) iff there are $l \to r \in T$, $p \in Pos(s)$ and a substitution θ such that $s|_p = \theta l$ and $t = s[\theta r]_p$. We write $s \to_{T/AC} t$ (s *reduces* modulo AC to t) iff there are terms s', t' such that $s =_{AC} s' \to_T t' =_{AC} t$. When considering $\to_{T/AC}$ we speak of an *AC-TRS.* Let T be an (AC-)TRS. If $s \to_{T(/AC)} t$ we speak of a *reduction step.* A term s is *reducible* if there is t such that $s \to_{T(/AC)} t$; otherwise, s is in *normal form.* A substitution θ is said to be *normalized* if for any $x \in Dom(\theta)$, θx is in normal form. T is *convergent* if it is *confluent* (modulo AC) and *terminating* (modulo AC).

3 Narrowing

We start with the case where E is equivalent to a convergent TRS T. A *system* is a couple $\Pi; K$ where Π is an E-unification problem and K a set of equations (modulo $\emptyset$). We define our narrowing relation $\rightsquigarrow$:

Definition 1. *Let $\Pi \cup \{e\}; K$ be a system, $p \in \overline{Pos}(e)$ and $l \to r \in T$. Then*

$$\Pi \cup \{e[u]_p\}; K \rightsquigarrow_T \Pi \cup \{e[r]_p\}; K \cup \{l =^? u\}$$

If $\Pi; K \rightsquigarrow_T \Pi'; K'$ then we say that $\Pi; K$ *narrows* in $\Pi'; K'$. We also speak of a (*narrowing*) *sequence* $\Pi_1; K_1 \rightsquigarrow_T ... \rightsquigarrow_T \Pi_n; K_n$, and we denote by $\rightsquigarrow^*_T$ the reflexive transitive closure of $\rightsquigarrow_T$. In the following we will occasionally slightly abuse this notation, narrowing systems where we have a term in place of the E-unification problem.

Our narrowing relation is nothing but a *lazy* version of the *basic* narrowing relation introduced by Hullot [9]. Moving the subterm u in K enforces the *basic* restriction on future applications of the rule [3].

We are not interested in finding minimal complete set of E-unifiers [3], only in E-unifiability. So in our setting narrowing gives a *complete* semi-decision

Table 1. Rewrite rules for quasi-group theory

$x * (x \backslash y) \to y$	$(x * y)/y \to x$
$(x/y) * y \to x$	$(x/y) \backslash x \to y$
$x \backslash (x * y) \to y$	$x/(y \backslash z) \to y$

procedure for E-unifiability. This means that, for every E-unification problem Π, the following two conditions are equivalent:

1. Π is E-unifiable.
2. $\Pi; \emptyset \rightsquigarrow_T^* \Pi'; K$ and $\Pi' \cup K$ is syntactically unifiable.

However, this does not give rise to a terminating decision procedure for E-unifiability. In fact some narrowing sequences may not terminate, and we don't know when we can stop following them.

Hullot [9] gives sufficient conditions for the narrowing to be a terminating decision procedure for E-unifiability. He proves that, given a TRS T, if for every $l \to r \in T$ every basic narrowing sequence starting from r terminates, then every narrowing sequence starting from every term terminates. Using this fact, one can give a decision procedure for I-unifiability. In fact the TRS $\{x+x \to x\}$ is convergent and equivalent to I, and moreover it satisfies the Hullot condition for termination, because there is no basic narrowing sequence starting from x. Similarly, Hullot gets decidability of unifiability in quasi-group theory, whose theory is equivalent to the convergent TRS reported in Table 1. Hullot's approach extends to narrowing modulo equational theories. But it gives no complexity analysis of the procedure and the TRS must satisfy a quite strong requirement.

Nieuwenhuis [18] shows that E-unifiability is in NP if E is *shallow* (i.e. variables at depth at most one). As a direct application one gets a new proof that I-unifiability is in NP, but one can not prove the same for quasi-group theory because it is not shallow. In fact being shallow is rather restrictive. In addition, this approach gives no extensions to deduction modulo equational theories.

Let's consider an easy case which is not covered by any of the above approaches. Let T be the TRS $\{f(f(x)) \to f(x)\}$ convergent and equivalent to $E := \{f(f(x)) \approx f(x)\}$. Intuitively, E-unifiability looks like an easy task. But we can't apply Nieuwenhuis' results because E is not shallow. Nor can we apply Hullot's results because there is a non-terminating narrowing sequence starting from $f(x)$, namely, using the non-lazy basic narrowing introduced by Hullot [9]:

$$f(x) \rightsquigarrow_{f(f(y)) \to f(y)} f(y) \rightsquigarrow_{f(f(z)) \to f(z)} f(z) \rightsquigarrow \ldots$$

However, suppose $t \in Term(\sum \bigcup X)$. If θ is normalized, how long can a reduction sequence starting from θt be? Clearly at most the number of occurrences of f in t, which is less than $size(t)$. So, if to every reduction sequence corresponds a narrowing sequence of the same length, in order to check the E-unifiability of an equation $s =^?_E t$ it is sufficient to consider narrowing sequences as long as $\max\{size(s), size(t)\}$.

In order to formalize this idea we introduce a useful notation. We write $s \rightarrow_{T,\theta} t$ to indicate the substitution θ involved in the reduction step. We write $s \longmapsto_T t$ iff $s \rightarrow_{T,\theta} t$ and θ is normalized. Similarly, if Π and Π' are sets of equations, we write $\Pi \longmapsto_T \Pi'$ iff $\Pi = \hat{\Pi} \cup \{e\}$, $\Pi' = \hat{\Pi} \cup \{e'\}$ and $e \longmapsto_T e'$. We speak of an *inner reduction step*. Furthermore, we write $s\bar{\downarrow}^n_T$ $(\Pi\bar{\downarrow}^n_T)$ if s (Π) reaches its normal form in at most n *inner* reduction steps, i.e. if there is no sequence $s \longmapsto_T s_1 \longmapsto_T \ldots \longmapsto_T s_n \longmapsto_T s_{n+1}$ $(\Pi \longmapsto_T \Pi_1 \longmapsto_T \ldots \longmapsto_T \Pi_n \longmapsto_T \Pi_{n+1})$.

The previous considerations lead to the following definition:

Definition 2. *Let H be a computable function from the set of all E-unification problems into the positive integers. H is a* narrowing bound *for T if for any E-unification problem Π, if Π is unifiable then there is a unifier θ such that $\theta\Pi\bar{\downarrow}^{H(\Pi)}_T$.*

The reader might wonder about the rationale for considering inner reductions. In addition: why don't we use the well-known *innermost* reductions? There is a specific reason for this, explained at the end of Section 4.1.

We can now give our decision procedure for E-unifiability, when E is equivalent to a convergent TRS T and H is a narrowing bound for T. Let Π be an E-unification problem:

Decision Procedure

1. Guess $n \leq H(\Pi)$.
2. Guess a sequence $\Pi; \emptyset \rightsquigarrow_T \Pi_1; K_1 \rightsquigarrow_T \ldots \rightsquigarrow_T \Pi_n; K_n$.
3. Answer 'yes' iff $\Pi_n \cup K_n$ is syntactically unifiable.

The proof of the completeness of the above decision procedure should be substantially obvious [3,23].

3.1 Complexity

Our approach allows us to say something more than just decidability of E-unifiability: we can prove it is in NP when there is a *polynomial* narrowing bound:

Definition 3. *A narrowing bound H is* polynomial *if it is polynomially computable and there is a polynomial q such that for every E-unification problem Π, $H(\Pi) \leq q(size(\Pi))$.*

In fact, if there is a polynomial narrowing bound for T, then we can restrict to considering only narrowing sequences whose length is bounded by a (fixed) polynomial in the size of the problem. Moreover, our lazy approach guarantees that if $\Pi; K \rightsquigarrow_T \Pi'; K'$ then $size(\Pi' \cup K') = O(size(\Pi \cup K))$. So the whole narrowing sequence is polynomial in the size of the problem, and can therefore be guessed in non-deterministic polynomial time. Noticing that general syntactic unifiability is in P [19,3] concludes the proof of the following theorem.

Theorem 1. *Let E be an equational theory and T a convergent TRS equivalent to E. If there is a polynomial narrowing bound for T then E-unifiability is in NP.*

The point now is: how do we find a (polynomial) narrowing bound for a TRS? The following corollary establishes a rather general result:

Corollary 1. *Let E be an equational theory and T a convergent TRS equivalent to E. If for every $l \rightarrow r \in T$ we have that r is a subterm of l, then E-unifiability is in NP.*

Proof. We prove by induction that for every term t and for every normalized substitution θ, $\theta t \bar{\downarrow}_T^{size(t)}$. The result follows since for any $e \in \Pi$, $\theta e \bar{\downarrow}_T^{size(e)}$ and $size(\Pi) = \sum_{e \in \Pi} size(e)$, so we have $\theta \Pi \bar{\downarrow}_T^{size(\Pi)}$ and therefore we can consider $H(\Pi) := size(\Pi)$.

We proceed with the induction:

- If $t = x$ then θx is in normal form because θ is normalized.
- If $t = f(t_1, ..., t_n)$ for $n \geq 0$ then $\theta t = f(\theta t_1, ..., \theta t_n)$. By induction, one gets $\theta t_1 \bar{\downarrow}_T^{size(t_1)}, ..., \theta t_n \bar{\downarrow}_T^{size(t_n)}$. If $f(\theta t_1 \downarrow_T, ..., \theta t_n \downarrow_T) \rightarrow_T t'$ then t' is in normal form because it is a (proper) subterm of $f(\theta t_1 \downarrow_T, ..., \theta t_n \downarrow_T)$. So we perform at most $1 + \sum_{i=1}^{n} size(t_i)$ reduction steps, and we can conclude $\theta t \bar{\downarrow}_T^{size(t)}$.

□

Clearly, the existence of a polynomial narrowing bound for a TRS T does not imply that r is a subterm of l for every rule $l \rightarrow r \in T$. For example, the trivial TRS $\{a \rightarrow b\}$ has a polynomial narrowing bound. We are currently working on generalizations of the above corollary. However, it already gives interesting results, which we detail in the next section.

3.2 Applications

We immediately get a new narrowing-based proof of the following well-known result.

Theorem 2. *I-unifiability is in NP.*

Proof. The TRS $\{x + x \rightarrow x\}$ is convergent and equivalent to I. The result follows because of Corollary 1. □

Notice that this last result is optimal: I-unifiability is NP-complete [13]. As we noted before, I-unifiability can be showed to be in NP using the results in [18]. However, our technique applies to non-shallow theories as well. We give a couple of examples:

Theorem 3. *Unifiability in quasi-group theory is in NP.*

Proof. A convergent TRS equivalent to quasi-group theory is reported in Table 1. The result follows because of Corollary 1. □

The theory $\{(x * y) * (y * z) \approx y\}$ defines *central groupoids* [2].

Theorem 4. *Unifiability in central groupoids is in NP.*

Proof. The following is a convergent TRS equivalent to $\{(x * y) * (y * z) \approx y\}$ [2]:

$$\begin{aligned}(x * y) * (y * z) &\to y\\ x * ((x * y) * z) &\to x * y\\ (x * (y * z)) * z &\to y * z\end{aligned}$$

The result follows because of Corollary 1. □

4 AC-Narrowing

In this section we extend our results to narrowing modulo AC. A special attention has always been devoted to this case [9,17], as A and C are well-suited for being built-in due to their permutative nature and they often occur in practical specifications (we will give an example in Section 4.2).

For simplicity, we assume that $+$ is the only AC-symbol, i.e. interpreted as an operator satisfying AC. Cases where there are several AC-symbols can be treated analogously.

We extend our narrowing relation in a way which parallels inner equational rewriting. One extends inner rewriting to inner AC-rewriting defining

$$s \mapsto_{T/AC} t \quad \text{iff} \quad s =_{AC} s' \mapsto_T t' =_{AC} t$$

for some s', t'. We might then define our AC-narrowing relation in the following way:

$$\Pi \cup \{e\}; K \rightsquigarrow_{T/AC} \Pi'; K' \quad \text{iff} \quad e' =_{AC} e \text{ and } \Pi \cup \{e'\}; K \rightsquigarrow_T \Pi'; K'$$

This would lead to a decision procedure for E-unifiability as in Section 3, where the final system is checked for AC-unifiability instead of syntactic unifiability (now K is a set of equations modulo AC).

However, the completeness is lost. In fact, the completeness of our narrowing for E-unifiability rests on a variant of the *Hullot property* [9,11]:

Definition 4. ***(Hullot property)*** *For every normalized substitution θ:*

$$\textit{if} \quad \theta e \mapsto_{T/AC} e' \quad \textit{then} \quad e;\emptyset \rightsquigarrow_{T/AC} e'';K$$

and there is a normalized substitution $\theta' \supseteq \theta$ such that $\theta' e'' =_{AC} e'$ and $\theta' \models K$.

We now show that the introduced narrowing relation does *not* satisfy the Hullot property. For readability we write $\bar{x}_n$ for $x_1 + ... + x_n$ and $\bar{a}_n$ for $a + ... + a$ (where there are n a's).

Example 1. Let $T := \{x + x \to c\}$ and $E := AC \cup \{x + x \approx c\}$. For every even $n > 1$ consider the equation $\bar{x}_n \overset{?}{=}_E c$. The substitution $\theta := \{x_1 \leftarrow a + b, ..., x_n \leftarrow a + b\}$ is normalized and we get:

$$\theta(\bar{x}_n \overset{?}{=}_E c) \mapsto_{T/AC} c + \bar{b}_n \overset{?}{=}_E c$$

The system $\bar{x}_n \overset{?}{=}_E c;\emptyset$ could narrow in $c \overset{?}{=}_E c; \bar{x}_n \overset{?}{=}_{AC} z + z$ but now $c \neq c + \bar{b}_n$. So the Hullot property is not satisfied.

Other narrowing sequences lead to similar results.

The point is that every θx_i in the above example is only *partially* involved in the reduction step. I.e. $\theta x_i =_{AC} u_i + v_i$ where u_i is *not* involved in the reduction step.

To overcome the incompleteness, we consider *extensions* of the equations. We use them *implicitly*, that is coding them in the narrowing relation. This idea first appeared in [21]. The reader may consult [22] for a comparison between implicit and explicit [20] extensions, and for another example showing that AC-paramodulation is incomplete without extensions. In many cases, it is sufficient to consider single-variable extensions, i.e. given an equation $s = t$ one considers $s + x = t + x$ where x is a new variable [20,17]. But in our framework this is not sufficient. We will return to this later.

Recall we assume $+$ is the only AC-symbol in $\sum$. We denote by $UngPos(s)$ the set of *variable* positions *unguarded* in s, i.e. $\{p : p \in Pos(s),\ s|_p \in X$ and if $p = qq'$ with $|q'| > 0$ then $s|_q = t + t'\}$ [4].

Our AC-narrowing relation is then:

Definition 5. *Let $\Pi \cup \{e\}; K$ be a system, $e =_{AC} e'$, $p \in \overline{Pos}(e')$, $l \to r \in T$ and $y_0, ..., y_{n-1}$ new variables where $n \leq |UngPos(e'|_p)|$. Then*

$$\Pi \cup \{e\}; K \rightsquigarrow_{T/AC} \Pi \cup \{e'[r + \bar{y}_n]_p\}; K \cup \{l + \bar{y}_n \overset{?}{=}_{AC} e'|_p\}.$$

Using this relation, AC-narrowing is complete. For instance, in the previous example we get:

$$\bar{x}_n \overset{?}{=}_E c;\emptyset \rightsquigarrow_{T/AC} c + \bar{y}_n \overset{?}{=}_E c; z + z + \bar{y}_n \overset{?}{=}_{AC} \bar{x}_n$$

now considering $\theta' := \theta \cup \{z \longleftarrow \bar{a}_{n/2}, y_1 \longleftarrow b, ..., y_n \longleftarrow b\}$ we see that the Hullot property is satisfied.

This example also explains the inequality $n \leq |UngPos(e'|_p)|$: the Hullot property would not be satisfied using less than n new variables.

The notion of narrowing bound translates to the AC-case simply considering inner reductions modulo AC. Let E be equivalent to a convergent AC-TRS T and let H be a narrowing bound for T. Given an E-unification problem Π, the following procedure decides its E-unifiability:

Decision Procedure (AC case)

1. Guess $n \leq H(\Pi)$.
2. Guess a sequence $\Pi; \emptyset \rightsquigarrow_{T/AC} \Pi_1; K_1 \rightsquigarrow_{T/AC} \ldots \rightsquigarrow_{T/AC} \Pi_n; K_n$.
3. Answer 'yes' iff $\Pi_n \cup K_n$ is AC-unifiable.

Again, the proof of the completeness should be substantially obvious, but it requires some technical notations to deal with AC-symbols [23].

4.1 Complexity

Implicit extensions do not affect the complexity of our decision procedure: all we have to do is to guess the number of new variables, which is bounded by the size of the term being considered. In addition, given a term s we can guess in non-deterministic polynomial time any term s' such that $s =_{AC} s'$. Noticing that AC-unifiability is in NP [12] gives the following:

Theorem 5. *Let E be an equational theory and T a convergent AC-TRS equivalent to E. If there is a polynomial narrowing bound for T then E-unifiability is in NP.*

Polynomial narrowing bounds exist for significant AC-TRS, as we shall see in the next section. However, we can't hope in a result as Corollary 1. To see this, consider $E := AC \cup \{a + b \approx b\}$ and $T := \{a + b \rightarrow b\}$. The point is: if a term s is in normal form, how many reductions do we need to take $s + b$ to its normal form? This clearly depends on the number of a's in s, which might be exponential in the size of the unification problem. More precisely, using an argument similar to one in [19], we define for every $n \geq 1$ the unification problem $\Pi_n :=$

$$\begin{aligned} x + b &=^?_E b \\ x &=^?_E x_1 + x_1 \\ x_1 &=^?_E x_2 + x_2 \\ &\ldots \\ x_{n-1} &=^?_E x_n + x_n \\ x_n &=^?_E a \end{aligned}$$

If $\theta \models \Pi_n$, then θx contains 2^n a's. Therefore we need 2^n reduction steps to take $\theta(x + b)$ to its normal form b, since each application of the rule $a + b \rightarrow b$ removes a single a.

We conclude this section explaining why we use inner reductions instead of innermost ones. Let's consider the same set of problems $\{\Pi_n\}$ above, but now let $E := AC \cup \{x + b \approx b\}$ and $T := \{x + b \to b\}$. As before, if $\theta \models \Pi_n$ then θx contains 2^n a's. If we used innermost reductions then we would need 2^n reduction steps to take $\theta(x+b)$ to its normal form b. On the other hand, $\theta(x+b) \longmapsto b$ by just *one* inner reduction step. To sum it up: using innermost reductions there is no polynomial narrowing bound for T, while using inner reductions (one can prove) there is.

4.2 Application in Process Algebra

As mentioned before, applications of our method can be found in *Process Algebra* [4]. We can in particular prove NP-completeness of unifiability modulo bisimulation in minimal process algebra with proper iteration (MPA^+_δ) [7]. This is the so-called *compatibility checking* problem for MPA^+_δ, a significant extension of that for $BCCSP$, studied in [8]. See [8,10] for motivation and a survey of the compatibility checking.

In the following, let A be a fixed set of actions. The signature of MPA^+_δ consists of a constant δ, which represents *deadlock*, the binary *alternative composition* $x + y$, the unary prefix *sequential composition* $a(x)$ and the proper *iteration* $a^+(x)$, for $a \in A$. Often, $a(t)$ and $a^+(t)$ will be abbreviated by at and a^+t, which bind stronger than the alternative composition $+$.

The following is a complete equational axiomatization of bisimulation equivalence for MPA^+_δ [7]:

$$\begin{aligned}
(x+y)+z &\approx x+(y+z) \\
x+y &\approx y+x \\
x+x &\approx x \\
x+\delta &\approx x \\
a(a^+x+x) &\approx a^+x \qquad \forall a \in A \\
a^+(a^+x+x) &\approx a^+x \qquad \forall a \in A
\end{aligned}$$

We call this theory Bis^+. So we would like to decide Bis^+-unifiability. One could notice that Bis^+ can be seen as the union of $ACI1$ and two axioms for iteration. $ACI1$-unifiability is decidable [16] (we will give a new proof in the next section), so one may hope to use standard combination techniques for the union of equational theories. But the combination techniques currently available, see for instance [1], can not deal with this case, because these would require the whole equational theory to be equal to a union of equational theories over *disjoint* signatures, which is impossible because of the axiom $a(a^+x+x) \approx a^+x$.

We can of course try to apply our results on AC-narrowing, so we need a convergent AC-TRS equivalent to Bis^+. Such an AC-TRS is reported in Table 2 [7], and we call it T. This shifts the problem towards finding a narrowing bound for T. But $size$ is such!

Theorem 6. *Size is a polynomial narrowing bound for T.*

Table 2. Rewrite rules of the AC-TRS T

$x + x \to x$	
$x + \delta \to x$	
$a(a^+x + x) \to a^+x$	$\forall a \in A$
$a^+(a^+x + x) \to a^+x$	$\forall a \in A$
$a(a^+\delta) \to a^+\delta$	$\forall a \in A$
$a^+(a^+\delta) \to a^+\delta$	$\forall a \in A$

Proof. We prove that if $s = t + u$, where both t and u are in normal form, then $s\overset{-1}{\downarrow}_{T/AC}$. The rest of the proof proceeds exactly as that of Corollary 1.

If s is in normal form then we are done.

If $t = \delta$ ($u = \delta$) then $s \longmapsto_{T/AC} u$ (t) and we are done.

Otherwise, we reduce by the rule $x + x \to x$. Intuitively, we simply have to choose the greatest substitution that fits. More formally, we associate to every term v the multiset $M(v)$, which is defined as follows:

$$\begin{array}{ll} M(v) := M(w) \cup M(w') & \text{if } v = w + w' \\ M(v) := \{[v]_{AC}\} & \text{otherwise} \end{array}$$

where $\cup$ is the union between multisets [2] and $[v]_{AC}$ is the equivalence class of v modulo AC. Now consider the case where $t \neq \delta$, $u \neq \delta$ and $t + u$ is not in normal form. Let w be a term such that $M(w) = M(t) \cap M(u)$, and reduce applying the rule $x + x \to x$ with substitution $\theta := \{x \leftarrow w\}$. It is easily seen that we get a normal form.

In any case, we perform at most one inner reduction step, so we can conclude $s\overset{-1}{\downarrow}_{T/AC}$. □

Applying Theorem 5, we get that Bis^+-unifiability is in NP. In [8] it is proven that Bis-unifiability (Bis is Bis^+ without the two axioms for iteration) is NP-hard. The proof also works for Bis^+-unifiability. So the upper bound on the complexity of Bis^+-unifiability we have just shown is tight:

Theorem 7. *Bis^+-unifiability is NP-complete.*

To conclude, we point out that Bis^+-unifiability is (polynomially) equivalent to unifiability modulo bisimulation in minimal process algebra with prefix iteration (MPA^*_δ) [10].

4.3 Other Applications

We briefly discuss other applications of our decision procedure.

NP-decidability of both ACI-unifiability and $ACI1$-unifiability are well-known results, which can be proven via *ad hoc* decision procedures [16] or via combination techniques [1].

Our method can be used to reobtain them under a different and unified point of view: the AC-TRS $\{x{+}x \rightarrow x\}$ and $\{x{+}x \rightarrow x, x{+}1 \rightarrow x\}$ are convergent and equivalent to ACI and $ACI1$, respectively. Furthermore, *size* is a polynomial narrowing bound for them (the proof is a sub-case of the proof of Theorem 6). So our results hold and we get NP-decidability. It follows that our method can be used to prove NP-decidability of Bis-unifiability, since this is the same as $ACI1$-unifiability [8].

5 Conclusion

We have presented a narrowing-based method to decide E-unifiability when E is equivalent to a convergent $(AC$-$)TRS$. The method guarantees NP-decidability when a polynomial narrowing bound exists, and we have studied general conditions under which this existence is guaranteed.

These results have been used to provide an optimal and new result in Process Algebra, namely Bis^+-unifiability. They have also given (new) proofs under a unified point of view of NP-decidability of $I, ACI, ACI1$-unifiability and of unifiability in quasi-groups and central groupoids.

Our approach shows that sometimes E-unifiability can be shifted towards finding a narrowing bound for a convergent $(AC$-$)TRS$ equivalent to E. We are currently working on weaker conditions which guarantee the existence of a polynomial narrowing bound for a convergent $(AC$-$)TRS$.

Acknowledgment

I thank professor Marisa Venturini Zilli for having introduced me to the unification problems and for her helpful reading of this paper.

References

1. F. Baader and K. U. Schulz. Unification in the union of disjoint equational theories: combining decision procedures. *Journal of Symbolic Computation*, 21(2):211-244, 1996. 435, 436
2. F. Baader and T. Nipkow. *Term rewriting and all that*. Cambridge University Press, 1998. 426, 427, 432, 436
3. F. Baader, W. Snyder. Unification theory. In A. Robinson and A. Voronkov, editors, *Handbook of automated reasoning*, Elsevier Science Publishers B. V., 1999. 426, 427, 428, 430, 431
4. J. C. M. Baeten and W. P. Weijland. *Process Algebra*. Cambridge University Press, 1990. 427, 433, 435
5. H.-J. Bürckert. A resolution principle for a logic with restricted quantifiers. LNAI vol. 568, Springer Verlag, 1991. 427
6. A. Colmerauer. An introduction to PROLOG III. *C. ACM* 33, 69-90, 1990. 427
7. W. Fokkink. A complete equational axiomatization for prefix iteration. *Information processing letters*, 52(6):333-337, 1994. 427, 435

8. Q. Guo, P. Narendran, and S. K. Shukla. Unification and Matching in process Algebras. In T. Nipkow, editor, *Proceedings of 9th RTA*, LNCS vol. 1379, 1998. 426, 427, 435, 436, 437
9. J.-M. Hullot. Canonical forms and unification. In W. Bibel and R. Kowalski, editors, *Proceedings of the 5th International Conference on Automated Deduction*, LNCS vol. 87, 318-334, 1980. 426, 428, 429, 432
10. B. Intrigila, M. Venturini Zilli and E. Viola. Unification problems over process languages. Technical Report SI-00/01. Dipartimento di Scienze dell'Informazione, University of Rome "La Sapienza", Rome, 2000. 435, 436
11. J.-P. Jouannaud, C. Kirchner. Solving equations in abstract algebras: A rule-based survey of unification. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic: Essays in Honor of A. Robinson*, MIT Press, Cambridge, MA, 1991. 426, 432
12. D. Kapur and P. Narendran. Complexity of unification problems with associative-commutative operators. *J. Automated Reasoning* 9:261-288, 1992. 434
13. D. Kapur and P. Narendran. Matching, unification and complexity. *SIGSAM Bulletin,* 1987. 432
14. D. Kapur and P. Narendran. Double exponential complexity of computing complete sets of AC-unifiers. In *Proceedings of the 7th Annual IEEE Symposium on Logic in Computer Science*, Santa Cruz, California, 11-21, 1992. 426, 427
15. C. Kirchner, H. Kirchner and M. Rusinowitch. Deduction with symbolic constraints. *Revue d'Intelligence Artificielle* (Special issue on Automatic Deduction), 4(3):9-52, 1990. 427
16. P. Narendran. Unification modulo ACI+1+0. *Fundamenta Informaticae*, 25(1):49-57, 1996. 435, 436
17. R. Nieuwenhuis. On Narrowing, Refutation proofs and Constraints. In J. Hsiang, editor, *Proceedings of 6th RTA*, LNCS vol. 914, 1995. 427, 432, 433
18. R. Nieuwenhuis. Decidability and Complexity Analysis by Basic Paramodulation. *Information and Computation* 147:1-21, 1998. 427, 429, 432
19. M. Paterson, M. Wegman. Linear unification. *Journal of Computer and System Sciences*, 16(2):158-167, 1978. 431, 434
20. G. Peterson, M. Stickel. Complete sets of reductions for equational theories with complete unification algorithms. *Journal of the ACM* 28(2):233-264, 1981. 433
21. G. Plotkin. Building-in Equational Theories. *Machine Intelligence*, 7:73-90 1972. 433
22. L. Vigneron. Automated Deduction Techniques for Studying Rough Algebras. *Fundamenta Informaticae* 33, 85-103, 1998. 433
23. E. Viola. *E-unificabilità: decidibilità, complessità e algebre di processi.* B. S. Thesis, Dipartimento di Scienze dell'Informazione, University of Rome "La Sapienza", Rome, 2000. 430, 434

Author Index