Lecture Notes in Computer Science 1578
Edited by G. Goos, J. Hartmanis and J. van Leeuwen

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

Wolfgang Thomas (Ed.)

Foundations of Software Science and Computation Structures

Second International Conference, FOSSACS'99
Held as Part of the Joint European Conferences
on Theory and Practice of Software, ETAPS'99
Amsterdam, The Netherlands, March 22-28, 1999
Proceedings

Springer

Volume Editor

Wolfgang Thomas
RWTH Aachen, Lehrstuhl für Informatik VII
Ahornstr. 55, D-52056 Aachen, Germany
E-mail: thomas@informatik.rwth-aachen.de

Cataloging-in-Publication data applied for

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Foundations of software science and computation structures :
second international conference ; proceedings / FOSSACS '99, held
as part of the Joint European Conferences on Theory and Practice of
Software, ETAPS'99, Amsterdam, The Netherlands, March 22 - 28,
1999 / Wolfgang Thomas (ed.). - Berlin ; Heidelberg ; New York ;
Barcelona ; Hong Kong ; London ; Milan ; Paris ; Singapore ; Tokyo :
Springer, 1999
(Lecture notes in computer science ; Vol. 1578)
ISBN 3-540-65719-3

CR Subject Classification (1998): F.3, F.4.2, F.1.1, D.3.3-4, D.2.1

ISSN 0302-9743
ISBN 3-540-65719-3 Springer-Verlag Berlin Heidelberg New York

Typesetting: Camera-ready by author
SPIN 10703105 06/3142 – 5 4 3 2 1 0 Printed on acid-free paper

Preface

The International Conference "Foundations of Software Science and Computation Structures" (FOSSACS) is a constituent of the "Joint European Conferences on Theory and Practice of Software" (ETAPS). The present volume contains the contributions to FOSSACS'99, the second conference in this series, which took place in Amsterdam.

As formulated in the call for papers, FOSSACS focusses on "papers which offer progress in foundational research with a clear significance for software science. A central issue is theories and methods which support the specification, transformation, verification, and analysis of programs and software systems." The articles in this volume represent a wide spectrum of approaches to this general aim. In many papers, one finds the study of new concepts and methods which are motivated by recent trends (or problems) in the practical use of software and information technology.

The volume contains 18 contributed papers, preceded by three invited papers. The first, by M. Abadi, accompanies his "unifying invited lecture" addressed to the whole ETAPS audience. The second, by J. Esparza and J. Knoop, contains an application of the results presented by J. Esparza in his invited lecture to FOSSACS'99. The third summarizes an invited tutorial by D. Sangiorgi presented to ETAPS'99.

The selection of the contributed papers was in the hands of a programme committee consisting of R. di Cosmo (Paris), E.A. Emerson (Austin, TX), J. Engelfriet (Leiden), H. Ganzinger (Saarbrücken), D. Kozen (Ithaca, NY), B. Jonsson (Uppsala), A. Jung (Birmingham), M. Nielsen (Aarhus), T. Nipkow (Munich), D. Niwiński (Warsaw), C. Palamidessi (University Park, PA), A. Petit (Cachan), C. Stirling (Edinburgh), and W. Thomas (Aachen, chair). From 40 submissions, 18 were selected in a procedure which consisted of an e-mail discussion and a physical meeting in Aachen. Four members were present at this meeting (H. Ganzinger, B. Jonsson, A. Petit, W. Thomas); the others were contacted by e-mail in individual cases and provided with intermediate summaries of the discussion. I would like to thank all members of the programme committee and all subreferees for their diligent work and efficient cooperation. Special thanks go to Marianne Kuckertz and Jesper G. Henriksen for their excellent support regarding secretarial work and the electronic infrastructure and communication.

Aachen, January 1999

Wolfgang Thomas
FOSSACS'99 Programme Committee Chair

List of Referees

Aceto, L.
Amla, N.
Anderson, S.
Baader, F.
Banach, R.
Basin, D.
Baumeister, H.
Berard, B.
Bert, D.
Boer, F.S. de
Borovansky, P.
Bouajjani, A.
Braner, T.
Caillaud, B.
Caucal, D.
Clark, G.
Cortesi, A.
D'Argenio, P.R.
Dam, M.
David, A.
Degtyarev, A.
Delzanno, G.
Devillers, R.
Drabent, W.
Drewes, F.
Ehrig, H.
Fernandez, M.
Focardi, R.
Fournet, C.
Fribourg, L.
Gastin, P.
Ghani, N.
Gilleron, R.
Goerigk, W.
Grudzinski, G.
Hanus, M.
Harland, J.
Hasegawa, R.
Havlicek, J.
Henriksen, J.G.
Hensel, U.
Honda, K.
Jaeger, M.
Jagadeesan, R.
Jurdzinski, M.
Klaudel, H.
Klop, J.W.
Kohlhase, M.
Krishna Rao, M.R.K.
Laroussinie, F.
Lévy, J-J.
Lopez, P.E.M.
Maron, R.
Masini, A.
Matz, O.
Nyström, J.
Nyström, S-O.
Ohlebusch, E.
Pitts, A.
Podelski, A.
Pottier, F.
Power, J.
Prasad, S.
Quaglia, P.
Ramanujam, R.
Reichel, H.
Remy, D.
Roeckl, C.
Rosolini, P.
Rutten, J.J.M.M.
Salomaa, K.
Sangiorgi, D.
Schnoebelen, Ph.
Schubert, A.
Schwartzbach, M.
Sen, A.
Sewell, P.
Siegel, M.
Sistla, P.
Steffen, M.
Stencel, K.
Stevens, P.
Stolzenburg, F.
Tacchella, A.
Tommasi, M.
Trefler, R.
Victor, B.
Vogler, H.
Vogler, W.
Vorobyov, S.
Wagner, A.
Walukiewicz, I.
Ward, M.
Weidenbach, C.
Weikum, G.
Wilke, Th.

Foreword

ETAPS'99 is the second instance of the European Joint Conferences on Theory and Practice of Software. ETAPS is an annual federated conference that was established in 1998 by combining a number of existing and new conferences. This year it comprises five conferences (FOSSACS, FASE, ESOP, CC, TACAS), four satellite workshops (CMCS, AS, WAGA, CoFI), seven invited lectures, two invited tutorials, and six contributed tutorials.

The events that comprise ETAPS address various aspects of the system development process, including specification, design, implementation, analysis and improvement. The languages, methodologies and tools which support these activities are all well within its scope. Different blends of theory and practice are represented, with an inclination towards theory with a practical motivation on one hand and soundly-based practice on the other. Many of the issues involved in software design apply to systems in general, including hardware systems, and the emphasis on software is not intended to be exclusive.

ETAPS is a loose confederation in which each event retains its own identity, with a separate programme committee and independent proceedings. Its format is open-ended, allowing it to grow and evolve as time goes by. Contributed talks and system demonstrations are in synchronized parallel sessions, with invited lectures in plenary sessions. Two of the invited lectures are reserved for "unifying" talks on topics of interest to the whole range of ETAPS attendees. As an experiment, ETAPS'99 also includes two invited tutorials on topics of special interest. The aim of cramming all this activity into a single one-week meeting is to create a strong magnet for academic and industrial researchers working on topics within its scope, giving them the opportunity to learn about research in related areas, and thereby to foster new and existing links between work in areas that have hitherto been addressed in separate meetings.

ETAPS'99 has been organized by Jan Bergstra of CWI and the University of Amsterdam together with Frans Snijders of CWI. Overall planning for ETAPS'99 was the responsibility of the ETAPS Steering Committee, whose current membership is:

> André Arnold (Bordeaux), Egidio Astesiano (Genoa), Jan Bergstra (Amsterdam), Ed Brinksma (Enschede), Rance Cleaveland (Stony Brook), Pierpaolo Degano (Pisa), Hartmut Ehrig (Berlin), José Fiadeiro (Lisbon), Jean-Pierre Finance (Nancy), Marie-Claude Gaudel (Paris), Susanne Graf (Grenoble), Stefan Jähnichen (Berlin), Paul Klint (Amsterdam), Kai Koskimies (Tampere), Tom Maibaum (London), Ugo Montanari (Pisa), Hanne Riis Nielson (Aarhus), Fernando Orejas (Barcelona), Don Sannella (Edinburgh), Gert Smolka (Saarbrücken), Doaitse Swierstra (Utrecht), Wolfgang Thomas (Aachen), Jerzy Tiuryn (Warsaw), David Watt (Glasgow)

ETAPS'98 has received generous sponsorship from:

KPN Research
Philips Research
The EU programme "Training and Mobility of Researchers"
CWI
The University of Amsterdam
The European Association for Programming Languages and Systems
The European Association for Theoretical Computer Science

I would like to express my sincere gratitude to all of these people and organizations, the programme committee members of the ETAPS conferences, the organizers of the satellite events, the speakers themselves, and finally Springer-Verlag for agreeing to publish the ETAPS proceedings.

Edinburgh, January 1999

Donald Sannella
ETAPS Steering Committee Chairman

Table of Contents

Security Protocols and Specifications 1
M. Abadi

An Automata-Theoretic Approach to Interprocedural Data-Flow Analysis .. 14
J. Esparza, J. Knoop

Reasoning About Concurrent Systems Using Types 31
D. Sangiorgi

Testing Hennessy-Milner Logic with Recursion 41
L. Aceto, A. Ingólfsdóttir

A Strong Logic Programming View for Static Embedded Implications 56
R. Arruabarrena, P. Lucio, M. Navarro

Unfolding and Event Structure Semantics for Graph Grammars 73
P. Baldan, A. Corradini, U. Montanari

Expanding the Cube 90
G. Barthe

An Algebraic Characterization of Typability in ML with Subtyping 104
M. Benke

Static Analysis of Processes for No Read-Up and No Write-Down 120
C. Bodei, P. Degano, F. Nielson, H. R. Nielson

A WP-calculus for OO 135
F. S. de Boer

The Recognizability Problem for Tree Automata with Comparisons between Brothers 150
B. Bogaert, F. Seynhaeve, S. Tison

A Theory of "May" Testing for Asynchronous Languages 165
M. Boreale, R. De Nicola, R. Pugliese

A Nondeterministic Polynomial-Time Unification Algorithm for Bags, Sets and Trees 180
E. Dantsin, A. Voronkov

Categorical Models of Explicit Substitutions 197
N. Ghani, V. de Paiva, E. Ritter

Equational Properties of Mobile Ambients 212
A. D. Gordon, L. Cardelli

Model Checking Logics for Communicating Sequential Agents 227
M. Huhn, P. Niebert, F. Wallner

A Complete Coinductive Logical System for Bisimulation Equivalence on Circular Objects 243
M. Lenisa

String Languages Generated by Total Deterministic Macro Tree Transducers 258
S. Maneth

Matching Specifications for Message Sequence Charts 273
A. Muscholl

Probabilistic Temporal Logics via the Modal Mu-Calculus 288
M. Narasimha, R. Cleaveland, P. Iyer

A π-calculus Process Semantics of Concurrent Idealised ALGOL 306
C. Röckl, D. Sangiorgi

Author Index 323

Security Protocols and Specifications

Martín Abadi
ma@pa.dec.com

Systems Research Center
Compaq

Abstract. Specifications for security protocols range from informal narrations of message flows to formal assertions of protocol properties. This paper (intended to accompany a lecture at ETAPS '99) discusses those specifications and suggests some gaps and some opportunities for further work. Some of them pertain to the traditional core of the field; others appear when we examine the context in which protocols operate.

1 Introduction

The method of "security by obscurity" dictates that potential attackers to a system should be kept from knowing not only passwords and cryptographic keys but also basic information about how the system works, such as the specifications of cryptographic algorithms, communication protocols, and access-control mechanisms. It has long been argued that "security by obscurity" is usually inferior to open design [55, 28]. Of course, the value of writing and publishing specifications is greater when the specifications are clear, complete, and at an appropriate level of abstraction.

Current specifications of security mechanisms and properties vary greatly in quality, scope, purpose, and vocabulary. Some specifications are informal narrations that mix natural language and ad hoc notations. For example, the documents that describe the functioning of security protocols such as SSL [27], SSH [63], and IKE [32] often have this style. Other specifications are precise mathematical statements, sometimes expressed in formal calculi. These specifications have played a particularly significant role in cryptography and cryptographic protocols, but also appear in other areas, for example in information-flow analysis (e.g., [28, 22, 43, 48]).

Many of these specifications serve as the basis for reasoning, with various degrees of rigor and effectiveness, during system design, implementation, and analysis. In recent years, there has been much progress in the development of techniques for stating and proving properties about small but critical security components. For example, a substantial and successful body of work treats the core messages of security protocols and the underlying cryptographic functions. In this area, theory has been relevant to practice, even in cases where the theory is simplistic or incomplete. There seems to have been less progress in treating more complex systems [56], even those parts in the vicinity of familiar security

W. Thomas (Ed.): FOSSACS'99, LNCS 1578, pp. 1-13, 1999.

mechanisms. For example, we still have only a limited understanding of many of the interfaces, prologues, and epilogues of practical security protocols.

In this paper, we discuss specifications in the field of security, focusing on protocol specifications. We examine specifications of several sorts:

- In section 2, we consider specifications that concern the step-by-step behavior of a protocol. Such specifications can be largely independent of any assumptions or intended effects of the protocol.
- In section 3, we consider properties of protocols, in particular authenticity and secrecy properties, but also more exotic properties. We emphasize secrecy properties.
- In section 4, we view protocols in context by discussing their boundaries. These boundaries include programming interfaces, protocol negotiation, and error handling.

This paper is an informal, partial overview, and does not advocate any particular methods for specification and verification. Occasionally, however, the spi calculus [6] serves in explanations of formal points. In addition, the paper suggests some gaps and some opportunities for further work. The subject of this paper seems to be reaching maturity, but also expanding. There is still much scope for applying known techniques to important protocols, for developing simpler techniques, for exploring the foundations of those techniques, and also for studying protocols in context, as parts of systems.

2 Protocol narrations

The most common specifications are mere narrations of protocol executions. These narrations focus on the "bits on the wire": they say what data the various participants in a protocol should send in order to communicate. They are sometimes simple, high-level descriptions of sequences of messages, sometimes more detailed documents that permit the construction of interoperable implementations.

Following Needham and Schroeder [52], we may write a typical pair of messages of a protocol thus:

$$\begin{array}{ll} \text{Message 1} & A \to B : \{N_A\}_{K_{AB}} \\ \text{Message 2} & B \to A : \{N_A, N_B\}_{K_{AB}} \end{array}$$

Here A and B represent principals (users or computers). In Message 1, A sends to B an encrypted message, with key K_{AB} and cleartext N_A. In Message 2, B responds with a similar message, including N_B in the cleartext. The braces represent the encryption operation, in this case using a symmetric cryptosystem such as DES [48]. The subscripts on K_{AB}, N_A, and N_B are merely hints. It may be understood that A and B both know the key K_{AB} in advance and that A and B freshly generate N_A and N_B respectively, so N_A and N_B serve as nonces.

As Bob Morris has pointed out [7], the notation "Message n $X \to Y : M$" needs to be interpreted with care, because security protocols are not intended

to operate in benign environments. The network between X and Y may be unreliable and even hostile; X and Y themselves may not deserve total trust. So we may interpret "Message n $X \rightarrow Y : M$" only as "the protocol designer intended that X send M as the nth message in the protocol, and for it to be received by Y". One may want additional properties of this message, for example that only Y receive it or that Y should know that this message is part of a particular protocol execution; however, such properties cannot be taken for granted.

A sequence of messages is not a complete description of a protocol; it must be complemented with explanations of other forms. Protocol narrations often give some but not all of these explanations.

- As done above, a specification should say which pieces of data are known to principals in advance and which are freshly generated.
- A specification should also say how principals check the messages that they receive. For example, after receipt of Message 2, principal A may be expected to check that it is encrypted under K_{AB} and that the first component of its cleartext is the nonce N_A sent in Message 1. If this check fails, A may ignore the message or report an error. (Section 4 discusses errors further.) Checks are an essential part of protocols. For example, the absence of a check in the CCITT X.509 protocol [18] allowed an attack [16]; other attacks arise when principals assume that the messages that they receive have particular forms [9].
- The emission of Message n+1 follows the reception of Message n only in the simplest protocols. In general, a protocol may allow multiple messages belonging to the same session to be in flight simultaneously. The constraints on the order of messages in SSL have often been misunderstood [60]. Other complex protocols may be similarly confusing.
- As a convention, it is generally assumed that many protocol executions may happen simultaneously, and that the same principal may participate in several such executions, possibly playing different roles in each of them. This convention has exceptions, however. For example, some protocols may restrict concurrency in order to thwart attacks that exploit messages from two simultaneous executions. In addition, some roles are often reserved for fixed principals—for example, the name S may be used for a fixed authentication server. A complete specification should not rely on unclear, implicit conventions about concurrency and roles.

These limitations are widely recognized. They have been addressed in approaches based on process calculi (e.g., [41, 6, 47, 38, 57]) and other formal descriptions of processes (e.g., [53, 58]). The process calculi include established process calculi, such as CSP, and others specifically tailored for security protocols. Here we sketch how protocols are described in the spi calculus [6]; descriptions in other process calculi would have similar features.

The spi calculus is an extension of the pi calculus [50] with primitives for cryptographic operations. Spi-calculus processes can represent principals and

sets of principals. For example, the process:

$$(\nu K_{AB})(P_A \mid P_B)$$

may represent a system consisting of two principals, playing the roles of A and B as described above, in a single execution of the protocol. The construct ν is the standard restriction binder of the pi calculus; here it binds a key K_{AB}, which will occur in P_A and P_B. The construct $\mid$ is the standard parallel-composition operation of the pi calculus. Finally, P_A and P_B are two processes. The process P_B may be:

$$c(x).\mathit{case}\ x\ \mathit{of}\ \{y\}_{K_{AB}}\ \mathit{in}\ (\nu N_B)\overline{c}\langle\{y, N_B\}_{K_{AB}}\rangle$$

Informally, the components of this process have the following meanings:

- c is the name of a channel, which we use to represent the network on which the principals communicate.
- $c(x)$ awaits a message on c. When a message is received, the bound variable x is instantiated to this message. The expected message in this example is $\{N_A\}_{K_{AB}}$.
- $\mathit{case}\ x\ \mathit{of}\ \{y\}_{K_{AB}}\ \mathit{in}\ (\nu N_B)\overline{c}\langle\{y, N_B\}_{K_{AB}}\rangle$ attempts to decrypt x using the key K_{AB}. If x is a term of the form $\{M\}_{K_{AB}}$, then the bound variable y is instantiated to the contents M, and the remainder of the process $((\nu N_B)\overline{c}\langle\{y, N_B\}_{K_{AB}}\rangle)$ is executed.
- (νN_B) generates N_B.
- $\overline{c}\langle\{y, N_B\}_{K_{AB}}\rangle$ sends $\{M, N_B\}_{K_{AB}}$ on c, where M is the term to which y has been instantiated.

The syntax of the spi calculus distinguishes names (such as c, K_{AB}, and N_B) from variables (x and y), and processes (active entities) from terms (data that can be sent in messages). We refer to previous papers for the details of this syntax. We also omit a definition of P_A; it is similar in style to that of P_B.

Since the spi calculus is essentially a programming language, it is a matter of programming to specify the generation of data, checks on messages, concurrency, and replication. For these purposes, we can usually employ standard constructs from the pi calculus, but we may also add constructs when those seem inadequate (for example, for representing number-theoretic checks). In particular, we can use the ν construct for expressing the generation of keys, nonces, and other data. For example, the name N_B bound with ν in P_B represents the piece of data that B generates. On the other hand, the free names of P_B (namely c and K_{AB}) represent the data that B has before the protocol execution.

Thus, specifications in the spi calculus and other formal notations do not suffer from some of the ambiguities common in informal protocol narrations. Moreover, precise specifications need not be hard to construct: in recent work, Lowe, Millen, and others have studied how to turn sequences of messages into formal specifications [47]. To date, however, formal specifications do not seem to have played a significant role for protocol implementations. Their main use has been for reasoning about the properties of protocols; those properties are the subject of the next section.

3 Protocol properties

Although the execution of a protocol may consist in sending bits on wires, the bits have intended meanings and goals. These meanings and goals are not always explicit or evident in protocol narrations (cf. [7]).

There is no universal interpretation for protocols. Two usual objectives are to guarantee authenticity and secrecy of communications: only the intended principals can send and receive certain pieces of data. Other objectives include forward secrecy [24], non-repudiation, and availability. Some objectives contradict others. For example, some protocols aim to guarantee anonymity rather than authenticity, or plausible deniability [54] rather than non-repudiation. Moreover, many definitions have been proposed even for such basic concepts as authenticity (e.g., [11, 30, 42, 3]).

Nevertheless, there are some common themes in the treatment of protocol properties.

- The participants in security protocols do not operate in a closed world, but in communication with other principals. Some of those principals may be hostile, and even the participants may not be fully trusted. Thus, interaction with an uncertain environment is crucial.
- Security properties are relative to the resources of attackers. Moreover, it is common to attempt to guarantee some properties even if the attackers can accomplish some unlikely feats. For example, although precautions may be taken to avoid the compromise of session keys, an attacker might obtain one of those keys. A good protocol design will minimize the effect of such events. In particular, certificates for keys should expire [23]; and when one key is expiring, it should not be used for encrypting the new key that will replace it.
- It is common to separate essential security properties from other properties such as functional correctness and performance. For example, one may wish to establish that messages between a client and a server are authentic, even if one cannot prove that the server's responses contain the result of applying a certain function to the client's requests.

Protocol properties have been expressed and proved in a variety of frameworks. Some of these frameworks are simple and specialized [16], others powerful and general. A frequent, effective approach consists in formulating properties as predicates on the behaviors (sequences of states or events) of the system consisting of a protocol and its environment (e.g., [62, 11, 31, 41, 51, 53, 14, 57]). For example, in the simple dialogue between A and B shown in section 2, the authenticity of the second message may be expressed thus:

> If A receives a message encrypted under K_{AB}, and the message contains a pair N_A, N_B where N_A is a nonce that A generated, then B has sent the message sometime after the generation of N_A.

Once properly formalized, this statement is either true or false for any particular behavior. Such predicates on behaviors have been studied extensively in the literature on concurrency (e.g., [8, 36]).

A richer view of authenticity also takes into account concepts such as authority and delegation [29, 37]. Those concepts appear, for example, when we weaken the authenticity statement by allowing B to delegate the task of communicating with A and the necessary authority for this task. However, it is still unclear how to integrate those concepts with predicates on behaviors.

Furthermore, some security properties—such as noninterference—are not predicates on behaviors [44, 45]. For instance, suppose that we wish to require that a protocol preserve the secrecy of one of its parameters, x. The protocol should not leak any information about x—in other words, the value of x should not interfere with the behavior of the protocol that the environment can observe. The parameter x may denote the identity of one of the participants or the sensitive data that is sent encrypted after a key exchange. In general, we cannot express this secrecy property as a predicate on behaviors. On the other hand, representing the protocol as a process $P(x)$, we may express the secrecy property by saying that $P(M)$ and $P(N)$ are equivalent (or indistinguishable), for all possible values M and N for x (cf. [59, 33]). Here we say that two processes P_1 and P_2 are equivalent when no third process Q can distinguish running in parallel with P_1 from running in parallel in P_2. This notion of process equivalence (testing equivalence) has been applied to several classes of processes and with several concepts of distinguishability, sometimes allowing complexity-theoretic arguments (e.g., [21, 15, 6, 38]). Now focusing on the spi calculus, we obtain one definition of secrecy:

Definition 1 (One definition of secrecy). *Suppose that the process $P(x)$ has at most x as free variable. Then P* preserves the secrecy *of x if $P(M)$ and $P(N)$ are equivalent for all terms M and N without free variables.*

For example, the process $(\nu K)\overline{c}\langle\{x\}_K\rangle$, which sends x encrypted under a fresh key K on a channel c, preserves the secrecy of x. Previous papers on the spi calculus [6, 1] contain more substantial examples to which this concept of secrecy applies.

Approaches based on predicates on behaviors rely on a rather different definition of secrecy, which can be traced back to the influential work of Dolev and Yao [26] and other early work in this area [35, 49, 46]. According to that definition, a process preserves the secrecy of a piece of data M if the process never sends M in clear on the network, or anything that would permit the computation of M, even in interaction with an attacker.

Next we show one instantiation of this general definition, again resorting to the spi calculus. For this purpose, we introduce the following notation from the operational semantics of the spi calculus; throughout, P and Q are processes, M is a term, m, m_1, ..., m_k are names, and x is a variable.

- $P \xrightarrow{\tau} Q$ means that P becomes Q in one silent step (a τ step).
- $P \xrightarrow{m} (x)Q$ means that, in one step, P is ready to receive an input x on m and then to become Q.
- $P \xrightarrow{\overline{m}} (\nu m_1, \ldots, m_k)\langle M\rangle Q$ means that, in one step, P is ready to create the new names m_1, ..., m_k, to send M on m, and then to become Q.

We represent the state of knowledge of the environment of a process by a set of terms S with no free variables (intuitively, a set of terms that the environment has). Given a set S, we define $C(S)$ to be the set of all terms computable from S, with the properties that $S \subseteq C(S)$ and $C(C(S)) = C(S)$; thus, C is a closure operator. The main rules for computing $C(S)$ concern encryption and decryption:

- if $M \in C(S)$ and $N \in C(S)$ then $\{M\}_N \in C(S)$;
- if $\{M\}_N \in C(S)$ and $N \in C(S)$ then $M \in C(S)$.

Straightforward rules concern terms of other forms, for example pairs:

- if $M \in C(S)$ and $N \in C(S)$ then $(M, N) \in C(S)$;
- if $(M, N) \in C(S)$ then $M \in C(S)$ and $N \in C(S)$.

Given a set of terms S_0 and a process P_0, we let R be the least relation such that:

- $R(P_0, S_0)$.
- If $R(P, S)$ and $P \xrightarrow{\tau} Q$ then $R(Q, S)$.
- If $R(P, S)$ and $P \xrightarrow{m} (x)Q$ and $m \in C(S)$ and $M \in C(S)$ then $R(Q[M/x], S)$.
- If $R(P, S)$ and $P \xrightarrow{\overline{m}} (\nu m_1, \ldots, m_k)\langle M\rangle Q$ and $m \in C(S)$ and $m_1, \ldots, m_k$ do not occur in S then $R(Q, S \cup \{M\})$.

Intuitively, $R(P, S)$ means that, if the environment starts interacting with process P_0 knowing S_0, then the environment may know S (and all terms computable from it, $C(S)$) when P_0 evolves to P. The environment may know some names initially, but it does not create more names along the way. The first clause in this definition sets the initial state of the interaction. The second one is for silent steps. The third one deals with a message from the environment to the process; the environment must know the message's channel name m and contents M. The fourth one deals with a message in the opposite direction; assuming that the environment knows the message's channel name m, it learns the message's contents M; some new names $m_1, \ldots, m_k$ may occur in M.

We arrive at the following alternative view of secrecy:

Definition 2 (Another definition of secrecy). *Suppose that S is a set of terms with no free variables, and P a process with no free variables. Suppose that the free names of M are not bound in P or any process into which P evolves. Let R be the relation associated with P and S. Then P* may reveal *M from S if there exist P' and S' such that $R(P', S')$ and $M \in C(S')$; and P* preserves the secrecy *of M from S otherwise.*

We do not have much experience with this definition of secrecy for the spi calculus. It is a somewhat speculative translation of definitions proposed in other settings.

By presenting both definitions of secrecy in the same framework, we are in a position to compare them and understand them better. We can immediately see that, unfortunately, neither definition of secrecy implies the other: the first one

concerns a process with a free variable x, while the second one concerns a process plus a set of terms with no free variables. There are also deeper differences between them: in particular, the first definition rules out implicit information flows [22], while the second one does not. We leave for further work explaining when one definition is appropriate and when the other, and finding useful relations between them.

Both of these definitions of secrecy rely on a simple, abstract representation of cryptographic functions. More detailed accounts of cryptography may include complexity-theoretic assumptions about those functions (e.g., [43]). Another, challenging subject for further work is bridging the gap between those treatments of cryptography. For instance, we may wonder whether the complexity-theoretic assumptions justify our definitions of secrecy. Analogous questions arise for definitions of authenticity.

4 Protocol boundaries

Often the specification of a protocol and its verification focus on the core of the protocol and neglect its boundaries. However, these boundaries are far from trivial; making them explicit and analyzing them is an important part of understanding the protocol in context. These boundaries include:

(1) interfaces and rules for proper use of the protocol,
(2) interfaces and assumptions for auxiliary functions and participants, such as cryptographic algorithms and network services,
(3) traversals of machine and network boundaries,
(4) preliminary protocol negotiations,
(5) error handling.

We discuss these points in more detail next.

(1) Whereas narrations may say what data the various principals in a protocol should send, they seldom explain how the principals may generate and use that data. On the other hand, the good functioning of the protocol may require that some pieces of data be unrelated (for example, a cleartext and the key used to encrypt it). Other pieces of data (typically session keys, but sometimes also nonces) may need to remain secret for some period of time. Furthermore, as a result of an execution of the protocol, the participants may obtain some data with useful properties. For instance, the protocol may yield a key that can be used for signing application messages. Application program interfaces (or even programming languages) should allow applications to exploit those useful properties, with clear, modular semantics, and without revealing tricky low-level cryptographic details (e.g., [12, 40, 39, 61, 2, 5, 10]).
(2) Some protocols rely on fixed suites of cryptosystems. In other cases, assumptions about the properties of cryptographic operations are needed. For example, in the messages of section 2, it may be important to say whether B

can tell that A encrypted N_A using K_{AB}. This property may hold because of redundancy in N_A or in the encryption function, and would not hold if any message of the appropriate size is the result of encrypting some valid nonce with K_{AB}. It may also be important to say that B is not capable of making $\{N_A, N_B\}_{K_{AB}}$ from $\{N_A\}_{K_{AB}}$ and N_B without K_{AB}. This property is a form of non-malleability [25]. In recent years, the literature on protocols has shown an increasing awareness of subtle cryptographic issues; it may be time for some principled simplification.

Similarly, protocols often rely on network time servers, trusted third parties, and other auxiliary participants. Detailed assumptions about these servers are sometimes absent from protocol narrations, but they are essential in reasoning about protocols.

(3) Protocol messages commonly go across network interfaces, firewalls with tunnels, and administrative frontiers (e.g., [12, 61, 20, 19, 4]). In some contexts (e.g., [17]), even the protocol participants may be mobile. These traversals often require message translations (for example, marshaling and rewriting of URLs). They are subject to filtering and auditing. Furthermore, they may trigger auxiliary protocols. Some of these traversals seem to be a growing concern in protocol design.

(4) Systems often include multiple protocols, each of them with multiple versions and options. Interactions between protocols can lead to flaws; they can be avoided by distinguishing the messages that correspond to each protocol (e.g., [7, 34]). Before executing a protocol (in a particular version, with particular options) the participants sometimes agree to do so by a process of negotiation in which they may consider alternatives. The alternatives can vary in their levels of security and efficiency. In protocols such as SSL, this process of negotiation is rather elaborate and error-prone [60]. Despite clear narrations, it offers unclear guarantees.

(5) As discussed in section 2, protocol specifications often do not explain how principals react when they perceive errors. Yet proper handling of errors can be crucial to system security. For example, in describing attacks on protocols based on RSA's PKCS #1 standard [13], Bleichenbacher reported that the SSL documentation does not clearly specify error conditions and the resulting alert messages, and that SSL implementations vary in their handling of errors. He concluded that even sending out an error message may sometimes be risky and that the timing of the checks within the protocol is crucial.

The intrinsic properties of a protocol, such as the secrecy of session keys, are worthy of study. However, these intrinsic properties should eventually be translated into properties meaningful for the clients of the protocol. These clients may want security, but they may not be aware of internal protocol details (such as session keys) and may not distinguish the protocol from the sophisticated mechanisms that support it and complement it. Therefore, specification and reasoning should concern not only the core of the protocol in isolation but also its boundaries, viewing the protocol as part of a system.

Acknowledgments

Discussions with Mike Burrows, Paul Kocher, John Mitchell, Roger Needham, and Phil Rogaway contributed to the writing of this paper.

References

1. Martín Abadi. Secrecy by typing in security protocols. In *Theoretical Aspects of Computer Software*, volume 1281 of *Lecture Notes in Computer Science*, pages 611–638. Springer-Verlag, 1997.
2. Martín Abadi. Protection in programming-language translations. In *Proceedings of the 25th International Colloquium on Automata, Languages and Programming*, pages 868–883, July 1998. Also Digital Equipment Corporation Systems Research Center report No. 154, April 1998.
3. Martín Abadi. Two facets of authentication. In *Proceedings of the 11th IEEE Computer Security Foundations Workshop*, pages 27–32, 1998.
4. Martín Abadi, Andrew Birrell, Raymie Stata, and Edward Wobber. Secure web tunneling. *Computer Networks and ISDN Systems*, 30(1–7):531–539, April 1998. Proceedings of the 7th International World Wide Web Conference.
5. Martín Abadi, Cédric Fournet, and Georges Gonthier. Secure implementation of channel abstractions. In *Proceedings of the Thirteenth Annual IEEE Symposium on Logic in Computer Science*, pages 105–116, June 1998.
6. Martín Abadi and Andrew D. Gordon. A calculus for cryptographic protocols: The spi calculus. Technical Report 414, University of Cambridge Computer Laboratory, January 1997. A revised version appeared as Digital Equipment Corporation Systems Research Center report No. 149, January 1998, and an abridged version will appear in *Information and Computation*.
7. Martín Abadi and Roger Needham. Prudent engineering practice for cryptographic protocols. *IEEE Transactions on Software Engineering*, 22(1):6–15, January 1996.
8. Bowen Alpern and Fred B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, October 1985.
9. Ross Anderson and Roger Needham. Robustness principles for public key protocols. In *Proceedings of Crypto '95*, pages 236–247, 1995.
10. Mihir Bellare, Ran Canetti, and Hugo Krawczyk. A modular approach to the design and analysis of authentication and key exchange protocols. In *Proceedings of the 30th Annual ACM Symposium on Theory of Computing*, pages 419–428, May 1998.
11. Mihir Bellare and Phillip Rogaway. Entity authentication and key distribution. In *Advances in Cryptology—CRYPTO '93*, volume 773 of *Lecture Notes in Computer Science*, pages 232–249. Springer Verlag, August 1993.
12. Andrew D. Birrell. Secure communication using remote procedure calls. *ACM Transactions on Computer Systems*, 3(1):1–14, February 1985.
13. Daniel Bleichenbacher. Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS #1. In *Advances in Cryptology – CRYPTO'98*, volume 1462 of *Lecture Notes in Computer Science*, pages 1–12. Springer-Verlag, 1998.
14. Chiara Bodei, Pierpaolo Degano, Flemming Nielson, and Hanne Riis Nielson. Control flow analysis for the π-calculus. In *CONCUR'98: Concurrency Theory*, volume 1466 of *Lecture Notes in Computer Science*, pages 84–98. Springer Verlag, September 1998.

15. Michele Boreale and Rocco De Nicola. Testing equivalence for mobile processes. *Information and Computation*, 120(2):279–303, August 1995.
16. Michael Burrows, Martín Abadi, and Roger Needham. A logic of authentication. *Proceedings of the Royal Society of London A*, 426:233–271, 1989. A preliminary version appeared as Digital Equipment Corporation Systems Research Center report No. 39, February 1989.
17. L. Cardelli and A. D. Gordon. Mobile ambients. In *Foundations of Software Science and Computation Structures, First International Conference (FoSSaCS '98)*, volume 1378 of *Lecture Notes in Computer Science*, pages 140–155. Springer Verlag, 1998.
18. CCITT. *Blue Book (Recommendation X.509 and ISO 9594-8: The directory-authentication framework)*. CCITT, 1988.
19. Pau-Chen Cheng, Juan A. Garay, Amir Herzberg, and Hugo Krawczyk. Design and implementation of modular key management protocol and IP secure tunnel on AIX. In *Proceedings of the 5th USENIX UNIX Security Symposium*, pages 41–54, June 1995.
20. William Cheswick and Steven Bellovin. *Firewalls and Internet Security*. Addison-Wesley, 1994.
21. Rocco De Nicola and Matthew C. B. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34:83–133, 1984.
22. Dorothy E. Denning. *Cryptography and Data Security*. Addison-Wesley, Reading, Mass., 1982.
23. Dorothy E. Denning and Giovanni Maria Sacco. Timestamps in key distribution protocols. *Communications of the ACM*, 24(7):533–535, August 1981.
24. Whitfield Diffie, Paul C. van Oorschot, and Michael J. Wiener. Authentication and authenticated key exchanges. *Designs, Codes and Cryptography*, 2:107–125, 1992.
25. Danny Dolev, Cynthia Dwork, and Moni Naor. Non-malleable cryptography. In ACM, editor, *Proceedings of the Twenty Third Annual ACM Symposium on Theory of Computing*, pages 542–552, 1991.
26. Danny Dolev and Andrew C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, IT-29(12):198–208, March 1983.
27. Alan O. Freier, Philip Karlton, and Paul C. Kocher. The SSL protocol: Version 3.0. Available at `http://home.netscape.com/eng/ssl3/ssl-toc.html`, March 1996.
28. Morrie Gasser. *Building a Secure Computer System*. Van Nostrand Reinhold Company Inc., New York, 1988.
29. Morrie Gasser and Ellen McDermott. An architecture for practical delegation in a distributed system. In *Proceedings of the 1990 IEEE Symposium on Security and Privacy*, pages 20–30, May 1990.
30. Dieter Gollman. What do we mean by entity authentication? In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pages 46–54, May 1996.
31. James W. Gray III and John McLean. Using temporal logic to specify and verify cryptographic protocols (progress report). In *Proceedings of the 8th IEEE Computer Security Foundations Workshop*, pages 108–116, 1995.
32. D. Harkins and D. Carrel. RFC 2409: The Internet Key Exchange (IKE). Available at `ftp://ftp.isi.edu/in-notes/rfc2409.txt`, November 1998.
33. Nevin Heintze and Jon G. Riecke. The SLam calculus: programming with secrecy and integrity. In *Proceedings of the 25th ACM Symposium on Principles of Programming Languages*, pages 365–377, 1998.

34. John Kelsey, Bruce Schneier, and David Wagner. Protocol interactions and the chosen protocol attack. In *Security Protocols: 5th International Workshop*, volume 1361 of *Lecture Notes in Computer Science*, pages 91–104. Springer Verlag, 1997.
35. Richard A. Kemmerer. Analyzing encryption protocols using formal verification techniques. *IEEE Journal on Selected Areas in Communications*, 7(4):448–457, May 1989.
36. Leslie Lamport. A simple approach to specifying concurrent systems. *Communications of the ACM*, 32(1):32–45, January 1989.
37. Butler Lampson, Martín Abadi, Michael Burrows, and Edward Wobber. Authentication in distributed systems: Theory and practice. *ACM Transactions on Computer Systems*, 10(4):265–310, November 1992.
38. P. Lincoln, J. Mitchell, M. Mitchell, and A. Scedrov. A probabilistic poly-time framework for protocol analysis. In *Proceedings of the Fifth ACM Conference on Computer and Communications Security*, pages 112–121, 1998.
39. John Linn. Generic interface to security services. *Computer Communications*, 17(7):476–482, July 1994.
40. Jonn Linn. RFC 1508: Generic security service application program interface. Web page at `ftp://ds.internic.net/rfc/rfc1508.txt`, September 1993.
41. Gavin Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1055 of *Lecture Notes in Computer Science*, pages 147–166. Springer Verlag, 1996.
42. Gavin Lowe. A hierarchy of authentication specifications. In *Proceedings of the 10th IEEE Computer Security Foundations Workshop*, pages 31–43, 1997.
43. Michael Luby. *Pseudorandomness and Cryptographic Applications*. Princeton University Press, 1996.
44. John McLean. Security models. In John Marciniak, editor, *Encyclopedia of Software Engineering*. Wiley & Sons, 1994.
45. John McLean. A general theory of composition for a class of "possibilistic" properties. *IEEE Transactions on Software Engineering*, 22(1):53–66, January 1996.
46. Catherine Meadows. A system for the specification and analysis of key management protocols. In *Proceedings of the 1991 IEEE Symposium on Research in Security and Privacy*, pages 182–195, 1991.
47. Catherine Meadows. Panel on languages for formal specification of security protocols. In *Proceedings of the 10th IEEE Computer Security Foundations Workshop*, page 96, 1997.
48. Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.
49. Jonathan K. Millen, Sidney C. Clark, and Sheryl B. Freedman. The Interrogator: Protocol security analysis. *IEEE Transactions on Software Engineering*, SE-13(2):274–288, February 1987.
50. Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, parts I and II. *Information and Computation*, 100:1–40 and 41–77, September 1992.
51. John C. Mitchell, Mark Mitchell, and Ulrich Stern. Automated analysis of cryptographic protocols using Murϕ. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pages 141–151, 1997.
52. Roger M. Needham and Michael D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, December 1978.

53. L. C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6(1–2):85–128, 1998.
54. Michael Roe. *Cryptography and Evidence*. PhD thesis, University of Cambridge Computer Laboratory, 1997. Available as a technical report of the Centre for Communications Systems Research at `http://www.ccsr.cam.ac.uk/techreports/`.
55. Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer system. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975.
56. Fred B. Schneider, editor. *Trust in Cyberspace*. National Academy Press, prepublication copy edition, 1998. Report of the Committee on Information Systems Trustworthiness, Computer Science and Telecommunications Board, National Research Council.
57. Steve Schneider. Verifying authentication protocols in CSP. *IEEE Transactions on Software Engineering*, 24(9):741–758, September 1998.
58. F. Javier Thayer Fábrega, Jonathan C. Herzog, and Joshua D. Guttman. Strand spaces: Why is a security protocol correct? In *Proceedings 1998 IEEE Symposium on Security and Privacy*, pages 160–171, May 1998.
59. Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4:167–187, 1996.
60. David Wagner and Bruce Schneier. Analysis of the SSL 3.0 protocol. In *Proceedings of the Second USENIX Workshop on Electronic Commerce Proceedings*, pages 29–40, November 1996. A revised version is available at `http://www.cs.berkeley.edu/~daw/me.html`.
61. Edward Wobber, Martín Abadi, Michael Burrows, and Butler Lampson. Authentication in the Taos operating system. *ACM Transactions on Computer Systems*, 12(1):3–32, February 1994.
62. Thomas Y. C. Woo and Simon S. Lam. A semantic model for authentication protocols. In *Proceedings of the 1993 IEEE Symposium on Research on Security and Privacy*, pages 178–194, 1993.
63. Tatu Ylönen. SSH—Secure login connections over the Internet. In *Proceedings of the Sixth USENIX Security Symposium*, pages 37–42, July 1996.

An Automata-Theoretic Approach to Interprocedural Data-Flow Analysis

Javier Esparza[1] and Jens Knoop[2]

[1] Technische Universität München, Arcisstr. 21, D-80290 München, Germany
e-mail: esparza@in.tum.de
[2] Universität Dortmund, Baroper Str. 301, D-44221 Dortmund, Germany
e-mail: knoop@ls5.cs.uni-dortmund.de

Abstract. We show that recent progress in extending the automata-theoretic approach to model-checking beyond the class of finite-state processes finds a natural application in the area of interprocedural data-flow analysis.

Keywords: Interprocedural data-flow analysis, model-checking, automata theory, program optimisation.

1 Introduction

Recent work [15, 24] has shown that model-checking algorithms for abstract classes of infinite-state systems, like context-free processes [1, 5] and pushdown processes [6], find a natural application in the area of data-flow analysis (DFA) for programming languages with procedures [16], usually called interprocedural DFA. A large variety of DFA problems, whose solution is required by optimising compilers in order to apply performance improving transformations, can be solved by means of a unique model-checking technique.

The techniques of [5, 6] are based on what could be called the fixpoint approach to model-checking [24], in which the set of states satisfying a temporal property is defined and computed as a fixpoint in an adequate lattice. Some years ago, Vardi and Wolper presented in a seminal paper [25] an alternative automata-theoretic approach in which—loosely speaking—verification problems are reduced to the emptyness problem for different classes of automata. This approach has had considerable success for finite-state systems, and constitutes the theoretical basis of verification algorithms implemented in tools like SPIN [13], PROD [26], or PEP [27]. Recently, the approach has also been extended to context-free processes and pushdown processes [4, 10], and to other infinite-state classes able to model parallelism [18].

The goal of this paper is to show that the techniques derived from these recent developments can also be applied to DFA. We provide solutions for the interprocedural versions of a number of important DFA problems, starting with the class of so-called bitvector problems. On the one hand, the structural simplicity of these problems allows us a gentle way of introducing our approach. On the other hand, these problems are quite important as they are the prerequisite of

W. Thomas (Ed.): FOSSACS'99, LNCS 1578, pp. 14-30, 1999.

numerous optimisations like *partially redundant expression elimination*, *partially redundant assignment elimination*, *partially dead code elimination*, and *strength reduction* [23], which are widely used in practice. In detail, we investigate:

(a) the four problems of Hecht's taxonomy of bitvector problems [12],
(b) the computation of faint variables, and
(c) the problems of (a) for parallel languages.

In contrast to (a), for which there exist several solutions in the literature, (b) and (c) have—to the best of our knowledge—not been considered yet in an interprocedural setting; solutions for the intraprocedural case can be found in [11, 14] for (b), and in [17] for (c).

The paper is organised as follows. Section 2 contains an informal introduction to DFA, recalls the DFA problems mentioned above, and in particular presents the flow graph model. Section 3 gives flow graphs a structured operational semantics. Sections 4, 5, and 6 present the solutions to the problems (a), (b) and (c) above, respectively, and Section 7 contains our conclusions.

2 Data-flow Analysis

Intuitively, *data-flow analysis (DFA)* is concerned with deciding run-time properties of programs without actually executing them, i.e., at compile time. Basically, the properties considered can be split into two major groups (cf. [12]). Properties whose validity at a program point depends on the program's *history*, i.e., on the program paths reaching it, and properties whose validity depends on the program's *future*, i.e., on the suffixes of program paths passing it. Both groups can further be split into the subgroups of *universally* and *existentially* quantified properties, i.e., whose validity depends on *all* or on *some* paths, respectively.

Background. Using the standard machinery of DFA (cf. [12]), the validity of a property at a program point n is deduced from a data-flow fact computed for n. This fact reflects the meaning of the program at n with respect to an abstract, simpler version of the "full" program semantics, which is tailored for the property under consideration. The theory of *abstract interpretation* provides here the formal foundation (cf. [7–9, 19]). In this approach the data-flow facts are given by elements of an appropriate lattice, and the abstract semantics of statements by transformations on this lattice. The *meet-over-all-paths* (*MOP*) semantics defines then the reference solution, i.e., the data-flow fact desired for a program point n: It is the "meet" (intersection) of all data-flow facts contributed by all program paths reaching n. The *MOP*-semantics is conceptually quite close to the program property of interest, but since there can be infinitely many program paths reaching a program point it does not directly lead to algorithms for the computation of data-flow facts. Therefore, in the traditional DFA-setting the *MOP*-semantics is approximated by the so-called *maximal-fixed-point* (*MFP*) semantics. It is defined as the greatest solution of a system of equations imposing consistency constraints on an annotation of the program points with data-flow

facts. The *MFP*-semantics coincides with its *MOP*-counterpart when the functions specifying the abstract semantics of statements are distributive, a result known as the Coincidence Theorem.

Note that the *MOP*-semantics is defined in terms of possible program executions. From the point of view of temporal logic this means in a pathwise, hence *linear time* fashion. In contrast, the computation of the *MFP*-semantics proceeds by consistency checks taking the immediate neighbours of a program point simultanenously into account. From the point of view of temporal logic this means in a tree-like, hence *branching time* fashion. Thus, in the traditional DFA-setting there is a gap between the reference semantics defining the data-flow facts of the program annotation desired (the *MOP*-semantics), and the semantics the computation of the program annotation with data-flow facts relies on (the *MFP*-semantics). An important feature of the automata-theoretic approach to DFA we are going to present here is the absence of a similar separation of concerns providing in this respect a more natural and conceptually simpler access to DFA.

Flow graphs. In DFA programs are commonly represented by systems of *flow graphs*, where every flow graph represents a procedure of the underlying program. Flow graphs are directed graphs, whose nodes and edges represent the statements and the intraprocedural control flow of the procedure represented. Usually, control flow is nondeterministically interpreted in order to avoid undecidabilities. As illustrated in Figure 1(a) and (b), which show the flow graph and the flow graph system representing a single procedure and a program with procedures, we consider edge-labelled flow graphs, i.e., the edges represent both the statements and the control flow, while nodes represent program points. We assume that statements are assignments of the form $v := t$ including the empty statement, call statements of the form $call\ \Pi(t_1, \ldots, t_n)$, or output operations of the form $out(t)$, where v is a variable, Π a procedure identifier, and $t, t_1, \ldots, t_n$ are terms.

Bitvector properties and faint variables. Bitvector properties correspond to structurally particularly simple DFA-problems, which, simultaneously, are most important in practice because of the broad variety of optimisations based on them (cf. Section 1). Their most prominent representatives, the *availability* and *very busyness* of terms, the *reachability* of program points by definitions, and the *liveness* of variables, span the complete space of the taxonomy recalled above as was shown by Hecht [12]. Intuitively, a term t is *available* at a program point n, if on all program paths reaching n term t is computed without that any of its operands is assigned a new value afterwards. Thus, availability is a universally quantified history-dependent property. Very busyness is its dual counterpart. A term t is *very busy* at a program point n, if it is computed on all program paths passing n and reaching the end node before any of its operands is assigned a new value after leaving n. Hence, very busyness is a universally quantified future-dependent property. For illustration consider Figure 1(a), in which the program points where $a + b$ is very busy are greyly highlighted.

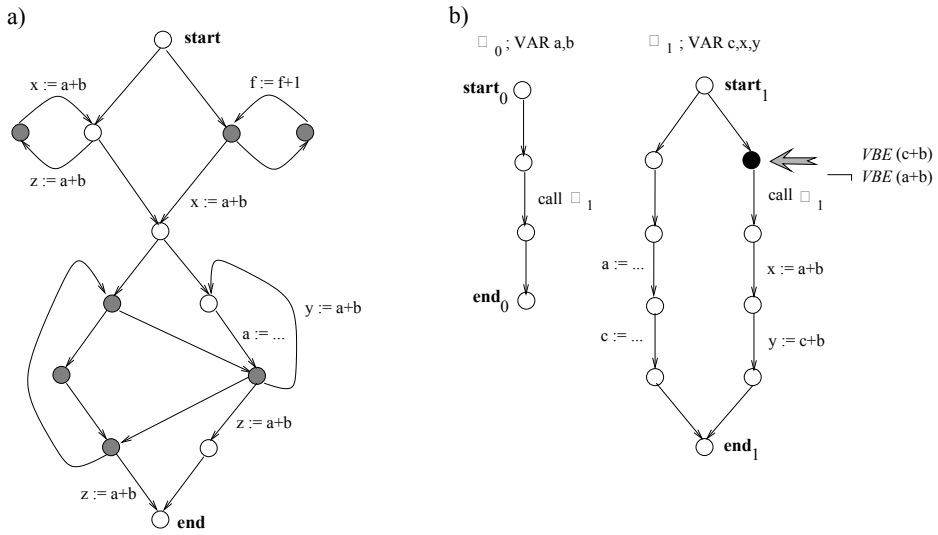


Fig. 1. Flow graphs and flow graph systems.

Reaching definitions (for convenience referred to as reachability later) and live variables are existentially quantified history- and future-dependent properties. Intuitively, a program point n is *reached* by the definition of a particular edge e if there is a program path across e reaching n which after passing e is free of definitions of the left-hand side variable of the definition of e. A variable v is *live* at a program point n, if on some program path passing n there is a reference to v, which after leaving n is not preceded by a redefinition of v. Conversely, a variable is *dead* at a program point, if it is not live at it.

This latter property is well-suited in order to illustrate how bitvector properties can be used for optimisation. Every assignment whose left-hand side variable is dead is "useless," and can be eliminated because there is no program continuation on which its left-hand side variable is referenced without a preceding redefinition of it. This is known as *dead-code elimination*.

Like the bitvector problems recalled above, DFA-problems are often concerned with sets of program items like terms, variables, or definitions. Characteristic for bitvector problems, however, is that they are "separable (decomposable):" The validity of a bitvector property for a specific item is independent of that of any other item. This leads to particularly simple formulations of bitvector problems on sets of items (and implementations in terms of bitvectors).

Faintness is an example of a program property which lacks the decomposability property. Intuitively, faintness generalizes the notion of dead variables. A variable f is faint at a program point if on all program continuations any right-hand side occurrence of f is preceded by a modification of f, or occurs in an assignment whose left-hand side variable is faint, too. A simple example of a faint but not dead variable is the variable f in the assignment $f := f + 1$, assuming that the assignment occurs in a loop without any other occurrence

of f elsewhere in the program (cf. Figure 1(a)). Assignments to faint variables can be eliminated as useless like those to dead ones. Whereas deadness, however, is a bitvector property, faintness is not. It is not separable preventing the computation of faintness for a variable in isolation.

DFA in the interprocedural and parallel setting. DFA is particularly challenging in the presence of recursive procedures—because of the existence of potentially infinitely many copies of local variables of recursive procedures at run-time—and parallel statements—because of the phenomena of interference and synchronisation. In the following we illustrate this by means of the programs of Figure 1(b), 2, and 3 using very busyness of the terms $a+b$ and $c+b$ as example.

In the example of Figure 1(b), the term $c+b$ is very busy at the program point preceding the recursive call of Π_1 in procedure Π_1, while $a+b$ is not. The difference lies in the fact that in the case of $a+b$ a global operand is modified within the recursive call, while it is a local one in the case of $c+b$. Thus, very busyness of $c+b$ is not affected because the assignment $c := \ldots$ modifies a new incarnation of c. In fact, after returning from the procedure call, the incarnation of c which has been valid when calling Π_1 is valid again, and, of course, it has not been modified. In contrast, the modification of a affects a global variable, and hence, the modification survives the return from the call. Thus, computing $a+b$ after the recursive call will yield a different value than computing it before this call. Similar phenomena can be observed for the other bitvector problems. This is illustrated in Figure 2 and Table 1, which summarizes for specific pairs of program points and program items the availability, reachability and liveness information. In the framework of [16] these obstacles of interprocedural DFA are overcome by mimicking the behaviour of the run-time stack by a corresponding DFA-stack, and additionally, by keeping track if modifications of operands affect a global or a local variable. So-called return functions, which are additionally introduced in this setting enlarging the specification of an abstract semantics, extract this information from the DFA-informations valid at call time and valid immediately before returning from the called procedure, which allows a proper

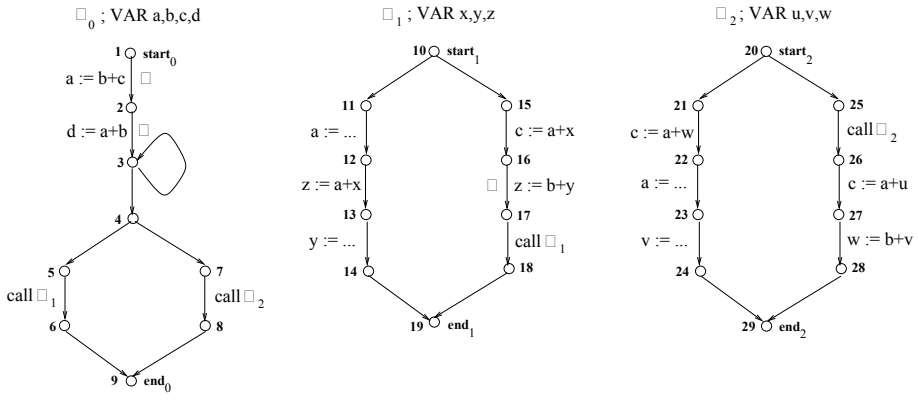


Fig. 2. The sequential interprocedural setting.

Program Point	Availability				Reaching Definitions		
	$a+b$	$b+c$	$a+x$	$b+y$	$\alpha: a:=$	$\beta: d:=$	$\gamma: z:=$
5	tt	tt	—	—	tt	tt	—
6	ff	ff	—	—	ff	tt	—
17	tt	ff	tt	tt	tt	tt	tt
18	ff	ff	ff	tt	ff	tt	tt

Program Point	Liveness			
	b	c	v	w
7	tt	ff	—	—
8	ff	ff	—	—
25	tt	ff	tt	ff
26	tt	ff	tt	ff

Table 1. Values of some bitvector problems.

treatment of programs with both local and global variables. In this paper we present a different approach to this problem.

Consider now Figure 3, and imagine that the three procedures shown in (a) are embedded either into a sequential (b) or parallel (c) program context, respectively. The pattern of very busy program points is different because of the effects of interference and synchronisation. While in (b) the term $a+b$ is very busy at nodes **1**, **7**, and **8**, in (c) it is very busy at nodes **1** and **10**. DFA of programs with explicit parallelism have attracted so far little attention, possibly because naive adaptations of the sequential techniques typically fail [21], and the costs of rigorous straightforward adaptations are prohibitive because of the number of interleavings expressing the possible executions of a parallel program. Though for an intraprocedural setting it could recently be shown that bitvector problems are an exception, which can be solved as easily and as efficiently as their sequential counterparts [17], a corresponding result for a setting with procedures has not yet been presented.

Conventions. Without loss of generality we make the following assumptions on flow graphs, which allow us a simpler notation during the presentation of the automata-theoretic approach. Flow graphs have a unique start node and end node without incoming and outgoing edges, respectively. Each node of a flow graph lies on a path connecting its start node and end node. The main procedure of a program cannot be called by any procedure of the program. Procedures are not statically nested. Edges leaving (approaching) a node with more than one successor (predecessor) are labelled by the empty statement. And, finally, the left-hand-side variable of an assignment does not occur in its right-hand-side term.

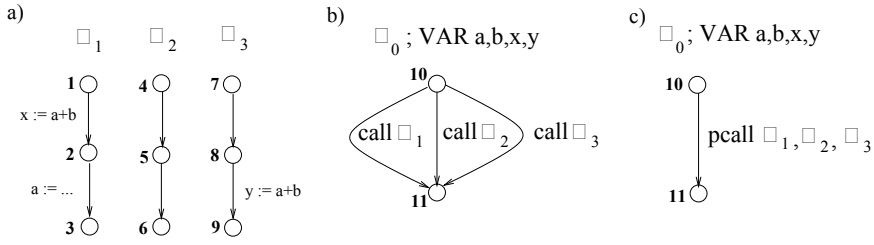


Fig. 3. The parallel interprocedural setting.

3 A Structured Operational Semantics for Flow Graph Systems

In this section we give flow graph systems a formal semantics very much in the style of the operational semantics of process algebras (see for instance [3]). Intuitively, we interpret a node of a flow graph as an *agent* that can execute some *actions*, namely the labels of the edges leaving it. The execution of an action transforms an agent into a new one. The actions that an agent can execute and the result of their execution are determined by *transition rules*. So, for instance, for a flow graph edge of the form $n \xrightarrow{v:=3} n'$, interpreted as "the agent n can execute the action $v := 3$, and become the agent n'," we introduce the rule $N \xrightarrow{v:=3} N'$ (uppercase letters are used to avoid confusion). In order to model procedure calls we introduce a sequential composition operator on agents with the following intended meaning: The sequential composition of N_1 and N_2, denoted by the concatenation $N_1 \cdot N_2$, is the agent that behaves like N_1 until it terminates, and then behaves like N_2. It is now natural to assign to an edge $n \xrightarrow{call\ \Pi_i(T)} n'$, where T is a vector $(t_1, \ldots, t_k)$ of terms, the rule $N \xrightarrow{\Pi(T)} START_i \cdot N'$ (the name of the action is shortened for readability)[1].

Let us now formally define the semantics. We associate to a flow graph system a triple (Con, Act, Δ), called a *process system*[2], where Con is a set of *agent constants*, Act is a set of *actions*, and $\Delta \subseteq Con^+ \times Act \times Con^*$ is a set of *transition rules*. An *agent* over Con is a sequential composition of agent constants, seen as an element of Con^*. In particular, the empty sequence ϵ is an agent. The set Δ induces a reachability relation $\xrightarrow{a} \subseteq Con^* \times Con^*$ for each $a \in Act$, defined as the smallest relation satisfying the following inference rules:

- if $(P_1, a, P_2) \in \Delta$, then $P_1 \xrightarrow{a} P_2$;
- if $P_1 \xrightarrow{a} P_1'$ then $P_1 \cdot P_2 \xrightarrow{a} P_1' \cdot P_2$ for every $P_2 \in Con^*$.

The second rule captures the essence of sequential composition: Only the first constant of a sequence can perform an action. Since the left-hand side of a rule cannot be empty, the agent ϵ cannot execute any action, and so it corresponds to the terminated agent.

In the sequel we overload the $\xrightarrow{a}$ symbol and write $P_1 \xrightarrow{a} P_2$ instead of $(P_1, a, P_2) \in \Delta$.

We associate to a flow graph system the process system (Con, Act, Δ) where Con is the set of program nodes plus a special agent constant $START$, Act is the set of edge labels plus special actions τ, $start$, end, and end_i for each procedure Π_i, and Δ contains the rules shown in Table 2. Observe that the left-hand sides of the rules of Δ have length 1, and that all terminating executions of the flow graph system begin with the action $start_0$ and end with end_0.

[1] Recall that $start_i$ is the start node of the flow graph Π_i.

[2] Process systems are very close the Basic Process Algebra of [2] or the context-free processes of [5]. We use another name due to small syntactic differences.

Flow graph	Process rule
$n \longrightarrow n'$	$N \xrightarrow{\tau} N'$
$n \xrightarrow{v := t} n'$	$N \xrightarrow{v := t} N'$
$n \xrightarrow{out(t)} n'$	$N \xrightarrow{out(t)} N'$
$n \xrightarrow{call\ \Pi_i(T)} n'$	$N \xrightarrow{\Pi_i(T)} START_i \cdot N'$
start node $start_0$	$START \xrightarrow{start_0} START_0$
end nodes end_i	$END_i \xrightarrow{end_i} \epsilon$

Table 2. Rules of the process system.

4 Interprocedural Bitvector Problems

In this section we provide solutions to the basic four bitvector problems using a language-theoretic formalism. We show how to compute the set of program points satisfying the existentially quantified properties. For universally quantified properties we first compute the set of points satisfying the *negation* of the property, which is existentially quantified, and then take the complement with respect to the set of all program points.

We first consider the case in which all variables are global, and subsequently move to the general case with both local and global variables.

4.1 Global Variables

We introduce some notations:

- Def_v denotes the set of actions of the form $v := t$;
- Ref_v denotes the set of actions of the form $u := t$ such that v appears in t;
- $Comp_t$ denotes the set of actions of the form $v := t'$ such that t' contains t as subterm;
- Mod_t denotes the set of actions of the form $v := t'$ such that v appears in t.[3]

Let us start by formalising the liveness problem. A global variable v is live at a program point n if there exists a sequence $START \xrightarrow{\sigma_1} P_1 \xrightarrow{\sigma_2} P_2$ satisfying the following constraints:

1. $P_1 = N \cdot P_1'$ (so σ_1 corresponds to a program path ending at the program point n);
2. $\sigma_2 \in (Act - Def_v)^* Ref_v$ (so in σ_2 the variable v is referenced before it is defined).

The other problems (or their negations) can be formalised following the same pattern. In fact, in the case of very busyness and availability what we directly compute in our approach, as mentioned above, is the set of program points at which v is *not* very busy or t is *not* available, respectively. Table 3 lists the constraints on P_1, σ_2, P_2 that must be satisfied in each case (there are no constraints on σ_1).

[3] Notice that, due to the conventions at the end of Section 2, the sets $Comp_t$ and Mod_t are disjoint.

Property	P_1	σ_2	P_2
v is live at n	$N \cdot P_1'$	$LI_v = (Act - Def_v)^* Ref_v$	Con^*
$m \xrightarrow{v := t} m'$ reaches n	$M \cdot P_1'$	$RE_v = (v := t)(Act - Def_v)^*$	$N \cdot P_2'$
t is *not* very busy at n	$N \cdot P_1'$	$NVB_t = (Act - Comp_t)^*(Mod_t + end_0)$	Con^*
t is *not* available at n	Con^*	$NA_t = (start_0 + Mod_t)(Act - Comp_t)^*$	$N \cdot P_2'$

Table 3. Constraints on P_1, σ_2, and P_2.

Solving the Problems. Given a process system (Con, Act, Δ), we make the following straightforward but crucial observation: A set of agents is just a language over the alphabet Con, and so it makes sense to speak of a regular set of agents. Automata can be used to finitely represent infinite regular sets of agents.

This observation has been exploited by Finkel, Williams and Wolper in [10] and by Bouajjani, Maler and the first author in [4] to develop efficient algorithms for reachability problems in process systems. We present some of the results of [10, 4], and apply them to the bitvector problems for flow graph systems.

Let $L \in Con^*$ and $C \in Act^*$ be regular languages. We call C a *constraint*, and define

$$post^*[C](L) = \{P \in Con^* \mid P' \xrightarrow{\sigma} P \text{ for some } P' \in L \text{ and some } \sigma \in C\}$$

In words, $post^*[C](L)$ is the set of agents that can be reached from L by means of sequences satisfying the constraint C. Analogously, we define

$$pre^*[C](L) = \{P \in Con^* \mid P \xrightarrow{\sigma} P' \text{ for some } P' \in L \text{ and some } \sigma \in C\}$$

So $pre^*[C](L)$ is the set of agents from which it is possible to reach an agent in L by means of a sequence in C. We abbreviate $post^*[Act^*](L)$ and $pre^*[Act^*](L)$ to $post^*(L)$ and $pre^*(L)$, respectively. We have the following result:

Theorem 1 ([10, 4]). *Let (Con, Act, Δ) be a process system such that each rule $P_1 \xrightarrow{a} P_2$ satisfies $|P_1| \leq 2$, and let $L \subseteq Con^*$ and $C \subseteq Act^*$ be regular sets. Then $pre^*[C](L)$ and $post^*[C](L)$ are regular sets of agents. Moreover, automata accepting these sets can be computed in $O(n_\Delta \cdot n_L^2 \cdot n_C)$ time, where n_Δ is the size of Δ, and n_L, n_C are the sizes of two automata accepting L and C, respectively.*

Let us use this result to compute the set of program points at which the variable v is live. This is by definition the set of program points n for which there is a sequence $START \xrightarrow{\sigma_1} P_1 \xrightarrow{\sigma_2} P_2$ satisfying $P_1 = N \cdot P_1'$ and $\sigma_2 \in LI_v$. Observe that $pre^*[LI_v](Con^*)$ is the set of agents from which a sequence $\sigma_2 \in LI_v$ can be executed. Notice however that not all these agents are necessarily reachable from $START$. Since the set of agents reachable from $START$ is $post^*(START)$, we compute an automaton accepting

$$pre^*[LI_v](Con^*) \cap post^*(START)$$

Now, in order to know if v is live at n it suffices to check if this automaton accepts some word starting by N.

Problem	Set of agents
liveness	$pre^*[LI_v](Con^*) \cap post^*(START)$
reachability	$post^*[RE_v](post^*(START) \cap M \cdot Con^*)$
very busyness	$pre^*[NVB_t](Con^*) \cap post^*(START)$
availability	$post^*[Act^*NA_t](START)$

Table 4. Agents corresponding to the four bitvector problems.

Table 4 shows the sets of agents that have to be computed to solve all four bitvector problems. For the complexity, notice that the number of states of the automata for L and C depends only on the bitvector problem, and not on the process system. So the liveness and reaching definition problems for a given variable v and the very busyness and availability problems for a given term t can be solved in $O(n_\Delta)$ time.

4.2 Local and Global Variables

The reader has possibly noticed that we have not exploited all the power of Theorem 1 so far. While the theorem holds for rules with a left-hand side of length 1 or 2, we have only applied it to rules with left-hand sides of length 1. We use now full power in order to solve the bitvector problems in a setting with global and local variables.

A local variable v is live at a program point n if and only if there exists a sequence $START \xrightarrow{\sigma_1} N \cdot P \xrightarrow{\sigma_2 d} N' \cdot P$ for some agent P such that

(1) $d \in Ref_v$,
(2) all the agents reached along the execution of $\sigma_2 d$ are of the form $P' \cdot P$, and
(3) for every transition $P_1 \cdot P \xrightarrow{a} P_2 \cdot P$ of σ_2, if $a \in Def_v$, then $|P_1| \geq 2$.

Here, condition (2) guarantees that the incarnation of v referenced by d and the incarnation of $N \cdot P$ are the same. Condition (3) guarantees that this incarnation is not modified along the execution of σ_2.

We now apply a general strategy of our automata approach, which will be used again in the next section: Instead of checking conditions (1) to (3) on the simple process system coresponding to the flow graph, we check a simpler condition on a more complicated process system. Intuitively, this new system behaves like the old one, but at any procedure call in a computation (or at the beginning of the program) it can nondeterministically decide to push a new variable M onto the stack—used to Mark the procedure call—and enter a new mode of operation, called the *local mode*. In local mode the process distinguishes between actions occurring at the current procedure call (the *marked call* in the sequel), and actions occurring outside it, i.e., actions occurring after encountering further procedure calls before finishing the marked call, or actions occurring after finishing the marked call.

We extend the process system with new agents, actions, and rules. The additional new agents are M (the Marker) and O (used to signal that we are Outside

the marked call). There is also a new action a_m for each old action a, plus extra actions $mark, return, exit$.[4] In the new process system a_m can only occur at the marked call, and a only at other levels. For each old rule we add one or more new ones as shown in Table 5. Notice that once the marker is introduced, all

Old rule	New additional rule(s)	
$N \xrightarrow{a} N'$	$M \cdot N \xrightarrow{a_m} M \cdot N'$	(a_m in the marked call)
	$O \cdot N \xrightarrow{a} O \cdot N'$	(a outside the marked call)
$START \xrightarrow{start} START_0$	$START \xrightarrow{mark} M \cdot START_0$	
$N \xrightarrow{\Pi_i(T)} START_i \cdot N'$	$N \xrightarrow{\Pi_i(T)} M \cdot START_i \cdot N'$	(marking the current call)
	$M \cdot N \xrightarrow{\Pi_i(T)} O \cdot START_i \cdot M \cdot N'$	(entering a deeper level)
	$O \cdot N \xrightarrow{\Pi_i(T)} O \cdot START_i \cdot N'$	
$N \xrightarrow{end_i} \epsilon$	$M \cdot N \xrightarrow{exit} O$	(end of the marked call)
	$O \cdot N \xrightarrow{end_i} O$	
	$O \cdot M \xrightarrow{return} M$	(return to the marked call)

Table 5. Rules of the extended process system.

reachable agents have either M or O in front, and that once the marked call terminates no agent ever has M in front again.

Given a local variable v, we define $Rel\_Def_v = \{a_m \mid a \in Def_v\}$ and $Rel\_Ref_v = \{a_m \mid a \in Ref_v\}$, where *Rel* stands for "relevant." If an agent moves into local mode at a procedure call in which a local variable v is incarnated, then Rel_Def_v and Rel_Ref_v are the actions concerning this same incarnation of v.

If we let Ext_Act be the extended set of actions of the new process system, then a local variable v is live at a program point n if and only if there exists a sequence $START \xrightarrow{\sigma_1} P_1 \xrightarrow{\sigma_2} P_2$ satisfying the following constraints:

- $P_1 = M \cdot N \cdot P_1'$, and
- $\sigma_2 \in (Ext\_Act - Rel\_Def_v)^* Rel\_Ref_v$.

So the constraint on σ_2 when the program has both local and global variables is obtained from the constraint for global variables by substituting Ext_Act for Act, Rel_Def_v for Def_v, and Rel_Ref_v for Ref_v. The reaching definitions problem can be solved analogously.

For the very busyness and the availability problems we have to take into account that the term t may contain local and global variables. Let $LocId(t)$ and $GlobId(t)$ be the set of local and global variables that appear in t. We define

$$Rel\_Mod_t = \bigcup_{v \in GlobId(t)} Def_v \cup \bigcup_{v \in LocId(t)} Rel\_Def_v$$

$$Rel\_Comp_t = \{a_m \mid a \in Comp_t\}$$

A term is not very busy at a program point n if and only if there exists a sequence $START \xrightarrow{\sigma_1} P_1 \xrightarrow{\sigma_2} P_2$ satisfying the following constraints:

[4] These actions are not strictly necessary, they are only included for clarity.

- $P_1 = M \cdot N \cdot P_1'$, and
- $\sigma_2 \in (Ext\_Act - Rel\_Comp_t)^*(Rel\_Mod_t + end_0)$.

Since the number of transition rules of the new process system increases only by a constant factor, the algorithm still runs in $O(n_\Delta)$ time.

5 Interprocedural Faint Variables

Recall that a variable v is *faint* at a program point n if on every program path from n every right-hand side occurrence of v is either preceded by a modification of v or is in an assignment whose left-hand side variable is faint as well. We show how to compute the set of program points at which a variable is faint in an interprocedural setting with global and local variables. This requires to split the set of references of a variable into the subset of references occurring in an output statement, and its complement set. To this end we introduce the notation:

- $RefOut_v$ denotes the set of actions of the form $out(t)$, such that v appears in t.

Faintness is a universal property, i.e., one that holds only if *all* program paths from a point n satisfy some condition. We formalise its negation, which is an existential property. A variable v is not faint at a program point n if there exists a sequence $START \xrightarrow{\sigma_1} P_1 \xrightarrow{\sigma_2} P_2$ satisfying the following constraints:

- $P_1 = N \cdot P_1'$ (so σ_1 corresponds to a program path ending at the program point n), and
- v is not faint at $P_1 \xrightarrow{\sigma_2} P_2$.

It remains to define the set of paths at which v is not faint. In comparison to the related bitvector property of deadness, the only new difficulty is to deal adequately with the recursive nature of the definition of faintness. The set is recursively defined as the smallest set of finite paths $P_1 \xrightarrow{a_1} P_2 \xrightarrow{a_2} P_3 \cdots P_n$, where $P_1 = N \cdot P_1'$, such that

(1) $a_1 \in RefOut_v$, or
(2) v is global, $a_1 \notin Def_v$ and v is not faint at $P_2 \xrightarrow{a_2} P_3 \cdots P_n$, or
(3) v is local for N, a_1 is an assignment not in Def_v, and v is not faint at $P_2 \xrightarrow{a_2} P_3 \cdots P_n$, or
(4) $a_1 \equiv u := t$, where v appears in t, and u is not faint at $P_2 \xrightarrow{a_2} P_3 \cdots P_n$.

Notice the difference between (2) and (3). If v is local and a_1 is a call action, then after the execution of a_1 the new program point is out of the scope of v, and so v is faint at $P_1 \xrightarrow{a_1} P_2 \xrightarrow{a_2} P_3 \cdots P_n$.[5] If v is global, then we remain within the scope of v, and so we do not know yet whether v is faint or not.

[5] With our definition v may be faint at $N \cdot P_1' \xrightarrow{a_1} P_2 \xrightarrow{a_2} P_3 \ldots P_n$ but not faint at some other path $N \cdot P_1'' \xrightarrow{a_1'} P_2' \xrightarrow{a_2'} P_3' \cdots P_{n'}'$, and so not faint at N.

As we did in the last section, we do not check the complicated property "v is faint at n" on a simple process system, but a simpler property on a more complicated process system obtained by adding new agents and rules (although this time no actions) to the old one. Intuitively, the new process system behaves like the old one, but at any point in a computation it can nondeterministically decide to push a program variable v into the stack—represented by an agent constant V—and enter a *faint mode* of operation. From this moment on, the system updates this variable according to the definition of non-faintness. The (updated) variable stays in the stack as long as the path executed by the system can be possibly extended to a path at which the variable is not faint. So the variable is removed from the stack only when the process system knows that (a) v is not faint for the path executed so far, or (b) v is faint at all paths extending the path executed so far. In case (a) the process system pushes a new agent $\bot$ into the stack, and in case (b) it pushes an agent $\top$.

Formally, the extended process system is obtained by adding the two agents $\top$ and $\bot$, and new rules as shown in Table 6.

Old rule	New additional rule(s)
$N \xrightarrow{a} N'$	$N \xrightarrow{a} V \cdot N'$ for each variable v (the faint mode can be entered anytime) $V \cdot N \xrightarrow{a} \bot$ if $a \in RefOut_v$ (condition (1), $\bot$ signals that v is not faint) $V \cdot N \xrightarrow{a} V \cdot N'$ if v global and $a \notin Def_v$ (condition (2)) $V \cdot N \xrightarrow{a} U \cdot N'$ if $a \equiv u := t$ and v appears in t (condition (4))
$N \xrightarrow{\Pi_i(T)} START_i \cdot N'$	$V \cdot N \xrightarrow{\Pi_i(T)} V \cdot START_i \cdot N'$ if v global (condition (2)) $V \cdot N \xrightarrow{\Pi_i(T)} START_i \cdot V \cdot N'$ if v local at n (out of the scope of v)
$N \xrightarrow{end_i} \epsilon$	$V \cdot N \xrightarrow{end_i} V$ if v global (condition (2)) $V \cdot N \xrightarrow{end_i} \top$ if v local at n (condition (3), this incarnation of v can no longer be defined or referenced)

Table 6. Rules of the extended process system.

In order to obtain the set of program points at which the variable v is faint, we compute the set of agents N for which there is a sequence $START \xrightarrow{\sigma_1} P_1 \xrightarrow{\sigma_2} P_2$ satisfying $P_1 = V \cdot N \cdot P_1'$ and $P_2 = \bot \cdot P_2'$. It suffices to compute an automaton for

$$(pre^*(\bot \cdot Con^*) \cap V \cdot Con^*) \cap post^*(START)$$

6 Interprocedural Bitvector Problems with Parallelism

In flow graph systems with parallelism, which we call in the sequel *parallel flow graph systems*, we allow edge-labels of the form $n \xrightarrow{pcall\ \Pi_{i_1}(T_1),\ldots,\Pi_{i_k}(T_k)} n'$. The procedures $\Pi_{i_1},\ldots,\Pi_{i_k}$ are called in parallel; if and when they all terminate, execution is continued at n'. Notice that parallel calls can be nested, and so the number of procedures running in parallel is unbounded. Notice also that flow graphs without parallelism are the special case in which $k = 1$ for all *pcall* instructions.

We show that the four bitvector problems can be solved for parallel flow graph systems in polynomial time when all variables are global. For this it will suffice to apply a beautiful extension of Theorem 1 recently proved by Lugiez and Schnoebelen in [18].

In order to model parallel flow graph systems by process systems we need to extend these to *parallel process systems* (also called process rewrite systems in [20]). An *agent* of a parallel process system is a tree whose root and internal nodes are labelled with either $\cdot$ or $\|$, representing sequential and parallel composition, and whose leaves are labelled with agent variables. So, for instance, the intended meaning of the tree $(X\|Y)\cdot Z$ is that X and Y are first executed in parallel, and if and when they terminate Z is executed. The empty tree γ, which satisfies

$$\gamma\cdot P = P\cdot\gamma = \gamma\|P = P\|\gamma = P$$

plays now the rôle of the terminated process. We denote the set of agents by $T(Con)$ (trees over Con). Rules are now elements of $(T(Con)\setminus\{\gamma\})\times Act\times T(Con)$. A set Δ of rules induces a reachability relation $\xrightarrow{a}\subseteq T(Con)\times T(Con)$ for each $a\in Act$, defined as the smallest relation satisfying the following inference rules:

- if $(P_1,a,P_2)\in\Delta$, then $P_1\xrightarrow{a}P_2$;
- if $P_1\xrightarrow{a}P_1'$ then $P_1\cdot P_2\xrightarrow{a}P_1'\cdot P_2$ for every $P_2\in T(Con)$;
- if $P_1\xrightarrow{a}P_1'$ then $P_1\|P_2\xrightarrow{a}P_1'\|P_2$ and $P_2\|P_1\xrightarrow{a}P_2\|P_1'$ for every $P_2\in T(Con)$.

We make free use of the fact that parallel composition is associative and commutative with respect to any reasonable behavioural equivalence between agents, such as bisimulation equivalence [22].

We associate to a parallel flow graph system the process system (Con, Act, Δ) as in the sequential case, the only difference being the rule corresponding to parallel calls, which is shown in Table 7. Observe that the left-hand side of all

Parallel flow graph	Process rule
$n \xrightarrow{pcall\ \Pi_{i_1}(T_1),\ldots,\Pi_{i_k}(T_k)} n'$	$N \xrightarrow{\Pi_1(T_1),\ldots,\Pi_k(T_k)} (START_{i_1}\|\ldots\|START_{i_k})\cdot N'$

Table 7. Rules of the parallel process system

rules consists of just one variable. Parallel process systems with this property are closely related to the PA-algebra studied in [18].

The four bitvector problems are defined almost as in the non-parallel case. The only difference is that in the parallel setting a program path can begin or end at *many* nodes due to the existence of parallel computation threads. In the non-parallel case, the agents corresponding to "being at node n" are those of the form $N \cdot P$. In the parallel case, they are the agents (trees) satisfying the following property: there is a leaf N which does not belong to the right subtree of any node labelled by $\cdot$. We call the set of these agents At_n.

Solving the Problems. An agent is no longer a word, but a tree—and so a set of agents is now a tree language. Tree automata can be used to finitely represent infinite regular sets of agents. We briefly introduce the tree automata we need. They are tuples (Q, A, δ, F) where:

- Q is a finite set of states and $F \subseteq Q$ is a set of final states.
- A is a finite alphabet containing the set Con and two binary infix operators $\cdot$ and $\|$. The automaton accepts terms over this alphabet, which we just call trees.
- δ is a finite set of transition rules of the form $N \to q$, $q_1 \cdot q_2 \to q$, or $q_1 \| q_2 \to q$. The rules define a rewrite relation on terms over $A \cup Q$.

The automaton accepts the trees that can be rewritten into a final state using the transition rules. As an example, we present a tree automaton accepting the set At_n. It has two states $\{q_1, q_2\}$, with q_1 as final state, rules $N \to q_1$ and $N' \to q_2$ for every program point $n' \neq n$, and rules

$$q_i \cdot q_j \to \begin{cases} q_1 & \text{if } i = 1 \\ q_2 & \text{otherwise} \end{cases} \qquad q_i \| q_j \to \begin{cases} q_1 & \text{if } i = 1 \text{ or } j = 1 \\ q_2 & \text{otherwise} \end{cases}$$

for $i, j \in \{1, 2\}$.

The question arises whether Theorem 1 can be extended to the tree case, i.e., the case in which Con^* is replaced by $T(Con)$. The answer is unfortunately negative. For instance, it is not difficult to see that the problem of deciding whether $post^*[C](L)$ is nonempty is undecidable even for the special case in which each rule $P_1 \xrightarrow{a} P_2$ satisfies $P_1 \in Con$ and L contains only one agent [18]. However, Lugiez and Schnoebelen show in [18] that it is possible to save part of the theorem. In particular, they prove the following result:

Theorem 2 ([18]). *Let (Con, Act, Δ) be a parallel process system such that each rule $P_1 \xrightarrow{a} P_2$ satisfies $P_1 \in Con$, let $L \in T(Con)$ be a regular set of agents, and let $A \subseteq Act$. Then $pre^*[A](L)$, $post^*[A](L)$, $pre^*[A^*](L)$, and $post^*[A^*](L)$ are regular sets of agents, and tree automata accepting them can be effectively computed in polynomial time.*

In order to check if the variable v is live at a program point n, we have to decide if there is a sequence $START \xrightarrow{\sigma_1} P_1 \xrightarrow{\sigma_2} P_2$ satisfying $P_1 \in At_n$ and $\sigma_2 \in LI_v$. Fortunately, LI_v is the concatenation of two languages of the form

A and A^* for which Theorem 2 holds, namely $(Act - Def_v)^*$ and Ref_v. So it suffices to compute a tree automaton accepting

$$pre^*[(Act - Def_v)^*](pre^*[Ref_v](T(Con))) \cap post^*(START) \cap At_n$$

and check if it is empty. The other three bitvector problems are solved analogously. If we now wish to extend this result to the case with both local and global variables, we can proceed as in Section 4.2. However, the process system so obtained contains rules whose left-hand side is a sequential composition of two agents. Since Theorem 2 has only been proved for the case $P_1 \in Con$, we cannot directly apply it. The question whether the bitvector problems can also be efficiently computed for local and global variables is still open.

7 Conclusions

We have shown that recent progress in extending the automata-theoretic approach to classes of processes with an infinite state space finds interesting applications in interprocedural data-flow analysis. Even though research in this area is at its very beginning, it is already possible to envisage some advantages of automata techniques. First of all, data-flow problems are expressed in terms of the possible executions of a program, and so it is very natural to formalise them in language terms; from the point of view of temporal logic, data-flow problems correspond to linear-time properties, and so the automata-theoretic approach, which is particularly suitable for linear-time logics, seems to be very adequate. Secondly, the approach profits from the very well studied area of automata theory. For instance, Lugiez and Schnoebelen obtained their results [18] by generalising constructions of [4, 10] for word automata to tree automata, and we could immediately apply them to bitvector problems in the interprocedural parallel case.

References

1. J. Baeten, J.A. Bergstra, and J.W. Klop. Decidability of bisimulation equivalence for processes generating context-free languages. *Journal of the ACM*, 40(3):653–682, 1993.
2. J. C. M. Baeten and W. P. Weijland. Process Algebra. *Cambridge Tracts in Theoretical Computer Science*, 1990.
3. B. Bloom. Structured operational semantics as an specification language. In *Proceedings of POPL '95*, pages 107–117, 1995.
4. A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model-checking. In *Proceedings of CONCUR '97*, volume 1243 of *Lecture Notes in Computer Science*, pages 135–150, 1997.
5. O. Burkart and B. Steffen. Model checking for context-free processes. In *Proceedings of CONCUR '92*, volume 630 of *Lecture Notes in Computer Science*, pages 123–137, 1992.
6. O. Burkart and B. Steffen. Composition, decomposition and model checking of pushdown processes. *Nordic Journal of Computing*, 2(2):89–125, 1995.

7. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conf. Rec. 4th Symp. Principles of Prog. Lang.* (*POPL'77*), pages 238 – 252. ACM, NY, 1977.
8. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Conf. Rec. 4th Symp. Principles of Prog. Lang.* (*POPL'79*), pages 269 – 282. ACM, New York, 1979.
9. P. Cousot and R. Cousot. Abstract interpretation frameworks. *J. of Logic and Computation*, 2(4):511 – 547, 1992.
10. A. Finkel, B. Willems, and P. Wolper. A direct symbolic approach to model checking pushdown systems. *Electronic Notes in Theoretical Computer Science*, 9, 1997.
11. R. Giegerich, U. Möncke, and R. Wilhelm. Invariance of approximative semantics with respect to program transformations. In *Proc. 3rd Conf. Europ. Co-operation in Informatics*, Informatik-Fachberichte 50, pages 1 – 10. Springer-V., 1981.
12. M. S. Hecht. *Flow Analysis of Computer Programs.* Elsevier, North-Holland, 1977.
13. G. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.
14. S. Horwitz, A. Demers, and T. Teitelbaum. An efficient general iterative algorithm for data flow analysis. *Acta Informatica*, 24:679 – 694, 1987.
15. M. Klein, J. Knoop, D. Koschützki, and B. Steffen. DFA&OPT-METAFrame: A tool kit for program analysis and optimization. In *Proc. 2nd Int. Workshop on Tools and Algorithms for Constr. and Analysis of Syst.* (*TACAS'96*), LNCS 1055, pages 422 – 426. Springer-V., 1996.
16. J. Knoop. *Optimal Interprocedural Program Optimization: A new Framework and its Application.* PhD thesis, Univ. of Kiel, Germany, 1993. LNCS Tutorial 1428, Springer-V., 1998.
17. J. Knoop, B. Steffen, and J. Vollmer. Parallelism for free: Efficient and optimal bitvector analyses for parallel programs. *ACM Trans. Prog. Lang. Syst.*, 18(3):268 – 299, 1996.
18. D. Lugiez and P. Schnoebelen. The regular viewpoint on PA-processes. In *Proceedings of CONCUR '98*, volume 1466 of *Lecture Notes in Computer Science*, pages 50–66, 1998.
19. K. Marriot. Frameworks for abstract interpretation. *Acta Informatica*, 30:103 – 129, 1993.
20. R. Mayr. *Decidability and Complexity of Model Checking Problems for Infinite-State Systems.* Ph.D. thesis, Technische Universität München, 1998.
21. S. P. Midkiff and D. A. Padua. Issues in the optimization of parallel programs. In *Proc. Int. Conf. on Parallel Processing, Volume II*, pages 105 – 113, 1990.
22. R. Milner. *Communication and Concurrency.* Prentice Hall, 1990.
23. S. S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, San Francisco, CA, 1997.
24. B. Steffen, A. Claßen, M. Klein, J. Knoop, and T. Margaria. The fixpoint-analysis machine. In *Proc. 6th Int. Conf. on Concurrency Theory* (*CONCUR'95*), LNCS 962, pages 72 – 87. Springer-V., 1995. Invited contribution.
25. M. Y. Vardi and P. Wolper. Automata Theoretic Techniques for Modal Logics of Programs. *Journal of Computer and System Sciences*, 32:183–221, 1986.
26. K. Varpaaniemi. PROD 3.3.02. An advanced tool for efficient reachability analysis. Technical report, Department of Computer Science and Engineering, Helsinky University of Technology, 1998. Available at http://www.tcs.hut.fi/pub/prod/.
27. F. Wallner. Model checking LTL using net unfoldings. In *Proceedings of CAV '98*, volume 1427 of *Lecture Notes in Computer Science*, pages 207–218, 1998.

Reasoning about Concurrent Systems Using Types

Davide Sangiorgi

INRIA – Sophia Antipolis, France

The purpose of these notes is to discuss some examples of the importance of types for reasoning about concurrent systems, and to list some relevant references. The list is surely not meant to be exhaustive, as the area is broad and very active. The examples are presented in the *π-calculus* [29], a paradigmatical process calculus for message-passing concurrency. We will not describe the proof techniques based on types with which the equalities in the examples are actually proved; for this, the interested reader can follow the references.

Acknowledgements. I would like to thank Benjamin Pierce, for our collaborations and the numerous discussions on the topic of these notes.

The π-calculus. As the λ-calculus, so the π-calculus language consists of a small set of primitive constructs. In the λ-calculus, they are constructs for building functions. In the π-calculus, they are constructs for building processes, notably: composition $P \mid Q$ to run two processes in parallel; restriction $\boldsymbol{\nu} x\, P$ to localise the scope of name x to process P (name is a synonymous for channels); input $x(y).\, P$ to receive a name z at x and then to continue as $P\{^z/_y\}$; output $\overline{x}\langle y\rangle.\, P$ to emit name y at x and then to continue as P; replication $!P$ to express processes with an infinite behaviour ($!P$ stands for a countable infinite number of copies of P in parallel); the inactive process $\mathbf{0}$. In the pure (i.e., untyped) calculus, all values transmitted are names.

We will find it convenient to present some of the examples on the *polyadic π-calculus*, an extension of the pure calculus in which tuples of names may be transmitted. A polyadic input process $x(y_1, \ldots, y_n).\, P$ waits for an n-uple of names $z_1, \ldots, z_n$ at x and then continues as $P\{^{z_1, \ldots, z_n}/_{y_1, \ldots, y_n}\}$ (that is, P with the y_i's replaced by the z_i's); a polyadic ouput process $\overline{x}\langle y_1, \ldots, y_n\rangle.\, P$ emits names $y_1, \ldots, y_n$ at x and then continues as P. We will abbreviate processes of the form $\overline{x}\langle y_1, \ldots, y_n\rangle.\, \mathbf{0}$ as $\overline{x}\langle y_1, \ldots, y_n\rangle$.

The most important predecessor of the π-calculus is CCS. The main novelty of the π-calculus over CCS is that names themselves may be communicated. This gives π-calculus a much greater expressiveness. We can encode, for instance: data values, the λ-calculus, higher-order process calculi (i.e., calculi where terms of the language can be exchanged) [27, 28, 42], which indicates that the π-calculus can be a model of languages incorporating functional and concurrent features, and that it may be a foundation for the design of new programming languages; the spatial dependencies among processes [39], which indicates that the π-calculus can be a model of languages for distributed computing; (some) object-oriented languages [21, 52, 22, 20].

Types. A type system is, roughly, a mechanism for classifying the expressions of a program. Type systems are useful for several reasons: to perform optimisa-

W. Thomas (Ed.): FOSSACS'99, LNCS 1578, pp. 31-40, 1999.

tions in compilers; to detect simple kinds of programming errors at compilation time; to aid the structure and design of systems; to extract behavioral information that can be used for *reasoning* about programs. In sequential programming languages, type systems are widely used and generally well-understood. In concurrent programming languages, by contrast, the tradition of type systems is much less established.

In the π-calculus world, types have quickly emerged as an important part of its theory and of its applications, and as one of the most important differences with respect to CCS-like languages. The types that have been proposed for the π-calculus are often inspired by well-known type systems of sequential languages, especially λ-calculi. Also type systems specific to processes have been (and are being) investigated, for instance for preventing certain forms of interferences among processes or certain forms of deadlocks.

One of the main reasons for which types are important for reasoning on π-calculus processes is the following. Although well-developed, the theory of the pure π-calculus is often insufficient to prove "expected" properties of processes. This because a π-calculus programmer normally uses names according to some precise logical discipline (the same happens for the λ-calculus, which is hardly-ever used untyped since each variable has usually an 'intended' functionality). This discipline on names does not appear anywhere in the terms of the pure calculus, and therefore cannot be taken into account in proofs. *Types* can bring this structure back into light. Below we illustrate this point with two examples that have to do with *encapsulation*.

Encapsulation. Desirable features in both sequential and concurrent languages are facilities for encapsulation, that is for constraining the access to components such as data and resources. The need of encapsulation has led to the deveolpment of abstract data types and is a key feature of objects in object-oriented languages.

In CCS, encapsulation is given by the *restriction* operator. Restricting a channel x on a process P, written (using π-calculus notation) $\boldsymbol{\nu} x\, P$, guarantees that interactions along x between subcomponents of P occur without interference from outside. For instance, suppose we have two 1-place buffers, `Buf1` and `Buf2`, the first of which receives values along a channel x and resends them along y, whereas the second receives at y and resends at z. They can be composed into a 2-place buffer which receives at x and resends at z thus: $\boldsymbol{\nu} y\, (\texttt{Buf1} \mid \texttt{Buf2})$. Here, the restriction ensures us that actions at y from `Buf1` and `Buf2` are not stolen by processes in the external environment. With the formal definitions of `Buf1` and `Buf2` at hand, one can indeed prove that the system $\boldsymbol{\nu} y\, (\texttt{Buf1} \mid \texttt{Buf2})$ is behaviourally equivalent to a 2-place buffer.

The restriction operator provides quite a satisfactory level of protection in CCS, where the visibility of channels in processes is fixed. By contrast, restriction alone is often not satisfactory in the π-calculus, where the visibility of channels may change dynamically. Here are two examples.

Example 1 (A printer with mobile ownership [34]). Consider the situation in which several client processes cooperate in the use of a shared resource such as

a printer. Data are sent for printing by the client processes along a channel p. Clients may also communicate channel p so that new clients can get access to the printer. Suppose that initially there are two clients

$$\begin{aligned} \texttt{C1} &= \overline{p}\langle j_1\rangle.\,\overline{p}\langle j_2\rangle\ldots \\ \texttt{C2} &= \overline{b}\langle p\rangle \end{aligned}$$

and therefore, writing P for the printer process, the initial system is

$$\nu p\ \ (P\ \mid\ \texttt{C1}\ \mid\ \texttt{C2}).$$

One might wish to prove that C1's print jobs represented by j_1 and j_2 are eventually received and processed in that order by the printer, possibly under some fairness condition on the printer scheduling policy. Unfortunately this is false: a misbehaving new client C3 which has obtained p from C2 can disrupt the protocol expected by P and C1 just by reading print requests from p and throwing them away:

$$\texttt{C3} = p(j).\,p(j').\,\mathbf{0}.$$

□

Example 2 (A boolean package implementation [35]). For an even more dramatic example, consider a π-calculus representation of a simple boolean package:

$$\begin{aligned} \texttt{BoolPack1} = (\nu t, f, \texttt{if}\,)\Big(& \\ & \overline{\texttt{getBool}}\langle t, f, if\rangle \\ & \mid\ !t(x,y).\,\overline{x}\langle\rangle \\ & \mid\ !f(x,y).\,\overline{y}\langle\rangle \\ & \mid\ !\texttt{if}(b,x,y).\,\overline{b}\langle x,y\rangle\Big) \end{aligned}$$

The package provides implementation of the `true` and `false` values and of an `if-true` function. In the π-calculus, a boolean value is implemented as a process located at a certain name; above the name is t for the value `true` and f for the value `false`. This process receives two return channels, above called x and y, and produces an answer at the first or at the second depending on whether the value `true` or `false` is implemented. The `if-true` function is located at `if`, where it receives three arguments: the location b of a boolean value and two return channels x and y; the function interacts with the boolean located at b and, depending on whether this is `true` or `false`, an answer at x or y is produced. Both the boolean values and the `if-true` function are replicated so that they may be used more than once. Other functionalities, like `and`, `or` and `not` functions, can be added to the package in a similar way.

A client can use the package by reading at `getBool` the channels t, f and `if`. After this, what remains of the package is

$$\begin{aligned} & !t(x,y).\,\overline{x}\langle\rangle \\ \mid\ & !f(x,y).\,\overline{y}\langle\rangle \\ \mid\ & !\texttt{if}(b,x,y).\,\overline{b}\langle x,y\rangle \end{aligned}$$

But now the implementation of the package is completely uncovered! A misbehaving client has free access to the internal representation of the components. It may interfere with these components, by attempting to read from t, f or `if`. It may also send at `if` a tuple of names the first of which is not the location of a boolean value. If multiple processes get to know the access channels t, f and `if` (which may happen because these channel may be communicated), then a client has no guarantee about the correctness of the answers obtained from querying the package. □

Using types to obtain encapsulation. In the two examples, the protection of a resource fails if the access to the resource is transmitted, because no assumptions on the use of that access by a recipient can be made. Simple and powerful encapsulation barriers against the mobility of names can be created using *type* concepts familiar from the literature of typed λ-calculi. We discuss the two examples above.

The misbehaving printer client `C3` of Example 1 can be prevented by separating between the input and the output capabilities of a channel. It suffices to assign the input capability on channel p to the printer and the output capability to the initial clients `C1` and `C2`. In this way, new clients which receive p from existing clients will only receive the output capability on p. The misbehaving `C3` is thus ruled out as ill-typed, as it uses p in input. This idea of "directionality in channels" was introduced in [34] and formalised by means of type constructs, the i/o *types*. They give rise to a natural subtyping relation, similar to those used for reference types in imperative languages like Forsythe (cf: Reynolds [38]). In the case of the π-calculus encodings of the λ-calculus [27], this subtyping validates the standard subtyping rules for function types [42]. This subtyping is also important when modeling object-oriented languages, whose type systems usually incorporate some powerful form of subtyping.

A common concept in typed λ-calculi is *polymorphism*. It is rather straightforward to add it onto a π-calculus type system by allowing channels to carry a tuple of both types and values. Forms of polymorphic type systems for the π-calculus are presented in [12, 50, 48, 47, 35, 11]. Polymorphic types can be used in Example 2 of the boolean package `BoolPack1` to hide the implementation details of the package components, in a way similar to Mitchell and Plotkin's representation of abstract data types in the λ-calculus [30]. We can make channel `getBool` polymorphic by abstracting away the type of the boolean channels t and f. This forces a well-typed observer to use t and f only as arguments of the `if-true` function. Indeed, using polymorphism this way the package `BoolPack1` is undistinguishable from the package

$$\begin{aligned}\texttt{BoolPack2} = (\nu t, f, \texttt{if}\,)\Big(\;&\overline{\texttt{getBool}}\langle t, f, if\rangle\\ &\mid\; !t(x,y).\,\overline{y}\langle\rangle\\ &\mid\; !f(x,y).\,\overline{x}\langle\rangle\\ &\mid\; !\texttt{if}(b,x,y).\,\overline{b}\langle y,x\rangle\Big)\end{aligned}$$

The latter has a different internal representations of the boolean values (a value `true` responds on the second of the two return channels, rather than on the first, and similarly for the value `false`) and of the `if-true` function. By "undistinguishable", we mean that no well-typed observer can tell the difference between the two packages by interacting with them.

The packages `BoolPack1` and `BoolPack2` are not behavioural equivalent in the standard theories of behavioural equivalences for process calculi. Trace equivalence is considered the coarsest behavioural equivalence; the packages are not trace equivalent because they have several different traces of actions, e.g.,

$$\overline{getBools}\langle t, f, \texttt{if}\rangle \texttt{if}(t, x, y).\, \overline{t}\langle x, y\rangle$$

is a trace of `BoolPack1` but not of `BoolPack2`.

Similarly, suppose we have, as in some versions of the π-calculus, a mismatch construct $[x \neq y]P$ that behaves as P if names x and y are different, as $\mathbf{0}$ if they are equal. With polymorphism we can make `BoolPack1` equivalent to the package `BoolPack3` obtained from `BoolPack1` by replacing the line implementing the conditional test with:

$$\texttt{if}(b, x, y).\ (\overline{b}\langle x, y\rangle \mid [b \neq t][b \neq f]\ \texttt{BAD}).$$

where `BAD` can be any process. The new package is equivalent to `BoolPack1` because the value received at `if` for b is always either t or f. This example shows that a client of the boolean package is not authorized to make up new values of the same type as the boolean channels t and f, since the client knows nothing about this type. Again, the equivalence betweeb `BoolPack1` and `BoolPack3` is not valid in the standard theories of behavioural equivalences for process calculi.

Types for reasoning. Types are important for reasoning on π-calculus processes. First, types reduce the number of legal contexts in which a given process may be tested. The consequence is that *more behavioural equalities between processes are true than in the untyped calculus.* Examples of this have been given above. The equalities considered in these examples fail in the untyped π-calculus, even with respect to the very coarse notion of trace equivalence. That is, there are contexts of the untyped π-calculus that are able to detect the difference between the processes of the equalities. By imposing type systems, these contexts are ruled out as ill-typed. On the remaining legal contexts the processes being compared are undistinguishable. Useful algebraic laws, such as laws for copying or distributing resources whose effect is to localise computation or laws for enhancing the parallelism in a process, can thus become valid.

Secondly, types facilitate the reasoning, by allowing the use of some proof techniques or simplifying their application. For instance type system for linearity, confluence, and receptiveness (see below) guarantee that certain communications are not preemptive. This is a partial confluence property, in the presence of which only parts of process behaviours need to be explored. Types can also allow more efficient implementations of communications between channels, or optimisations in compilers such as tail-call optimisation.

Another situation where types are useful is in limiting the explosion of the number of the derivatives of a process. To see why this can be a problem, consider a process $a(x).P$. In the untyped π-calculus, its behaviour is determined by the set of all derivatives $P\{^b/_x\}$, where b ranges over the free names of P plus a fresh one. In case of a cascade of inputs, this gives rise to state explosion, which at present is a serious obstacle to the development of tools for mechanical analysis of processes. The number of legal derivatives of processes can be reduced using types. For instance, in the example of the boolean package `BoolPack1`, having polymorphic types we know that the only possible names that can be received for the parameter b of the `if-true` function are t and f.

Types in π-calculi: some references. In the λ-calculus, where functions are the unit of interaction, the key type construct is arrow type. In the π-calculus names are the unit of interaction and therefore the key type construct is the *channel* (or *name*) type $\sharp\, T$. A type assignment $a : \sharp\, T$ means that a can be used as a channel to carry values of type T. As names can carry names, T itself can be a channel type. If we add a set of *basic types*, such as integer or boolean types, we obtain the analogous of simply-typed λ-calculus, which we may therefore call the *simply-typed π-calculus*. Type constructs familiar from sequential languages, such as those for products, unions, records, variants, recursive types, polymorphism, subtyping, linearity, can be adapted to the π-calculus [12, 50, 51, 18, 48, 47, 24, 19, 32, 11, 35, 3].

If we have recursive types, then we may avoid basic types as initial elements for defining types. The calculus with channel, product and recursive types is the *polyadic π-calculus*, mentioned at the beginning of these notes. Its type system is, essentially, Milner's *sorting systems* [27], historically the first form of type system for the π-calculus (in the sorting system type equality is syntactic, i.e., 'by-name'; more flexible notions of type equality are adopted in later systems).

The following type systems are development of those above but go beyond traditional type systems for sequential languages. Sewell [43] and Hennessy and Riely [17, 16] extend the i/o type system with richer sets of capabilities for distributed versions of the π-calculus (also [9] extends i/o types, on a Linda-based distributed language). Steffen and Nestmann [45] use types to obtain confluent processes. *Receptive types* [40] guarantee that the input end of a name is "functional", in the sense that it is always available (hence messages sent along that names can be immediately processed) and with the same continuation. Yoshida [53], Boudol [7] and Kobayashi and Sumii [23, 46], Ravara and Vasconcelos [37] put forward type systems that prevent certain forms of deadlocks. Abadi [1] uses types for guaranteeing secrecy properties in security protocols. The typing rules guarantee that a protocol that typechecks does not leak its secret information. Typing rules and protocols are presented on the spi-calculus, an extension of the π-calculus with shared-key cryptographic primitives. Honda [19] proposes a general framework for the the above-mentioned types, as well as other type systems.

Experimental typed programming languages, or proposals for typed programming languages, inspired by the π-calculus include Pict [36], Join [10], Blue [6], and Tyco [49].

Reasoning techniques for typed behavioural equivalences are presented in [34, 5, 3, 26, 17] for i/o or related types, in [35] for polymorphic types, in [24] for linear types, in [40] for receptive types. One of the most important application areas for the π-calculus is object-oriented languages. The reason is that naming is a central notion both for these languages and for the π-calculus. Proof techniques based on types have been used to prove the validity of algebraic laws and programm transformations on object-oriented languages [22, 41, 8, 20].

Other type sytems for concurrent calculi. Type systems can be used to guarantee safety properties, such as the absence of run-time errors. Examples 1 and 2 above show more refined properties, in which types prevent undesirable interactions among processes (even if these interactions would not produce run-time errors) thus guaranteeing that certain security constraints are not violated. In the printer Example 1, i/o types prevent malicious adversary from stealing jobs sent to the printer. In the boolean package Example 2, polymorphism prevents free access to the implementation details of the package.

Here are other works that apply types to security, on calculi or languages that are not based on the π-calculus. Smith and Volpano [44] use type systems to control information flow and to guarantee that private information is not improperly disclosed. Program variables are separated into high security and low security variables; the type system prevents information from flowing from high variables to low variables, so that the final values of the low variables are independent of the initial values of the high variables. On the use of type systems for controlling the flow of secure information, see also Heintze and Riecke [15]. Leroy and Rouaix [25] show how types can guarantee certain security properties on applets. Necula and Lee's proof-carrying code [31] is an elegant technique for ensuring safety of mobile code; mobile code is equipped with a proof attesting the conformity of the code to some safety policy. Defining and checking the validity of proofs exploits the type theory of the Edinburgh Logical Framework.

Applications of type theories to process reasoning include the use of theorem provers to verify the correctness of process protocols and process transformations [4, 33, 14].

We conclude mentioning a denotational approach to types for reasoning on processes. Abramsky, Gay and Nagarajan [2] have proposed *Interaction Categories* as a semantic foundation for typed concurrent languages, based on category theory and linear logic. In Interaction Categories, objects are types, morphisms are processes respecting those types, and composition is process interaction. Interaction Categories have been used to give the semantics to data-flow languages such as Lustre and Signal, and to define classes of processes that are deadlock-free in a compositional way. [13] presents a typed process calculus whose design follows the structure of Interaction Categories. It is not clear at present how Interaction Categories can handle process mobility and distribution.

References

1. M. Abadi. Secrecy by typing in security protocols. In M. Abadi and T. Ito, editors, *Proc. TACS '97*, volume 1281 of *Lecture Notes in Computer Science*. Springer Verlag, 1997.
2. S. Abramsky, S. J. Gay, and R. Nagarajan. Interaction categories and the foundations of typed concurrent programming. In *Deductive Program Design: Proceedings of the 1994 Marktoberdorf Summer School*, NATO ASI Series F, 1995.
3. R. Amadio. An asynchronous model of locality, failure, and process mobility. In *Proc. Coordination'97*, volume 1282 of *Lecture Notes in Computer Science*. Springer Verlag, 1997.
4. D. Bolignano and V. Ménissier-Morain. Formal verification of cryptographic protocols using coq. Submitted to a journal, 1997.
5. M. Boreale and D. Sangiorgi. Bisimulation in name-passing calculi without matching. In *13th LICS Conf.* IEEE Computer Society Press, 1998.
6. G. Boudol. The pi-calculus in direct style. In *Proc. 24th POPL*. ACM Press, 1997.
7. G. Boudol. Typing the use of resources in a concurrent calculus. In *Proc. ASIAN '97*, volume 1345 of *Lecture Notes in Computer Science*. Springer Verlag, 1997.
8. S. Dal-Zilio. Concurrent objects in the blue calculus. Submitted, 1998.
9. R. De Nicola, G. Ferrari, and R. Pugliese. KLAIM: A kernel Language for Agents Interaction and Mobility. *IEEE Transactions on Software Engineering*, 24(5):315–330, 1998.
10. C. Fournet and G. Gonthier. The Reflexive Chemical Abstract Machine and the Join calculus. In *Proc. 23th POPL*. ACM Press, 1996.
11. C. Fournet, C. Laneve, L. Maranget, and D. Rémy. Implict typing à la ML for the join-calculus. In *CONCUR'97*. Springer Verlag, 1997.
12. S. Gay. A sort inference algorithm for the polyadic π-calculus. In *Proc. 20th POPL*. ACM, 1993.
13. S. J. Gay and R. R. Nagarajan. A typed calculus of synchronous processes. In *10th LICS Conf.* IEEE Computer Society Press, 1995.
14. J. F. Groote, F. Monin, and J. C. Van de Pol. Checking verifications of protocols and distributed systems by computer. In *Proc. CONCUR '98*, volume 1466 of *Lecture Notes in Computer Science*. Springer Verlag, 1998.
15. N. Heintze and Riecke J.G. The SLam calculus: Programming with security and integrity. In *Proc. 25th POPL*. ACM Press, 1998.
16. M. Hennessy and J. Riely. Resource access control in systems of mobile agents. In U. Nestmann and B. C. Pierce, editors, *HLCL '98: High-Level Concurrent Languages*, volume 16.3 of *ENTCS*. Elsevier Science Publishers, 1998.
17. M. Hennessy and J. Riely. A typed language for distributed mobile processes. In *Proc. 25th POPL*. ACM Press, 1998.
18. K. Honda. Types for dyadic interaction. In E. Best, editor, *Proc. CONCUR '93*, volume 715 of *Lecture Notes in Computer Science*, pages 509–523. Springer Verlag, 1993.
19. K. Honda. Composing processes. In *Proc. 23th POPL*. ACM Press, 1996.
20. H. Hüttel, J. Kleist, M. Merro, and U. Nestmann. Migration = cloning ; aliasing. To be presented at Sixth Workshop on Foundations of Object-Oriented Languages (FOOL 6), 1999.
21. C.B. Jones. A π-calculus semantics for an object-based design notation. In E. Best, editor, *Proc. CONCUR '93*, volume 715 of *Lecture Notes in Computer Science*, pages 158–172. Springer Verlag, 1993.

22. J. Kleist and D. Sangiorgi. Imperative objects and mobile processes. In *Proc. IFIP Working Conference on Programming Concepts and Methods (PROCOMET'98)*. North-Holland, 1998.
23. N. Kobayashi. A partially deadlock-free typed process calculus. *Transactions on Programming Languages and Systems*, 20(2):436–482, 1998. A preliminary version in *12th Lics Conf.* IEEE Computer Society Press 128–139, 1997.
24. N. Kobayashi, B.C. Pierce, and D.N. Turner. Linearity and the pi-calculus. In *Proc. 23th POPL*. ACM Press, 1996.
25. X. Leroy and F. Rouaix. Security properties of typed applets. In *Proc. 25th POPL*. ACM Press, 1998.
26. M. Merro and D. Sangiorgi. On asynchrony in name-passing calculi. In *25th ICALP*, volume 1443 of *Lecture Notes in Computer Science*. Springer Verlag, 1998.
27. R. Milner. The polyadic π-calculus: a tutorial. Technical Report ECS–LFCS–91–180, LFCS, Dept. of Comp. Sci., Edinburgh Univ., October 1991. Also in *Logic and Algebra of Specification*, ed. F.L. Bauer, W. Brauer and H. Schwichtenberg, Springer Verlag, 1993.
28. R. Milner. Functions as processes. *Journal of Mathematical Structures in Computer Science*, 2(2):119–141, 1992.
29. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, (Parts I and II). *Information and Computation*, 100:1–77, 1992.
30. J. C. Mitchell and G. D. Plotkin. Abstract types have existential type. *ACM Trans. Prog. Lang. and Sys.*, 10(3):470–502, 1988.
31. G.C. Necula. Proof-carrying code. In *Proc. 24th POPL*. ACM Press, 1997.
32. J. Niehren. Functional computation as concurrent computation. In *Proc. 23th POPL*. ACM Press, 1996.
33. L.C. Paulson. The inductive approach to verifying cryptographic protocols. *J. Computer Security*, 6:85–128, 1998.
34. B. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes. *Journal of Mathematical Structures in Computer Science*, 6(5):409–454, 1996. An extended abstract in *Proc. LICS 93*, IEEE Computer Society Press.
35. B. Pierce and D. Sangiorgi. Behavioral equivalence in the polymorphic pi-calculus. In *24th POPL*. ACM Press, 1997.
36. B. C. Pierce and D. N. Turner. Pict: A programming language based on the pi-calculus. Technical Report CSCI 476, Indiana University, 1997. To appear in *Proof, Language and Interaction: Essays in Honour of Robin Milner*, Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, MIT Press.
37. A. Ravara and V.T. Vasconcelos. Behavioural types for a calculus of concurrent objects. In *3rd International Euro-Par Conference*, volume 1300 of *Lecture Notes in Computer Science*, pages 554–561. Springer Verlag, 1997.
38. J. C. Reynolds. Preliminary design of the programming language Forsythe. Tech. rept. CMU-CS-88-159. Carnegie Mellon University., 1988.
39. D. Sangiorgi. Locality and non-interleaving semantics in calculi for mobile processes. *Theoretical Computer Science*, 155:39–83, 1996.
40. D. Sangiorgi. The name discipline of receptiveness. In *24th ICALP*, volume 1256 of *Lecture Notes in Computer Science*. Springer Verlag, 1997. To appear in TCS.
41. D. Sangiorgi. Typed π-calculus at work: a proof of Jones's parallelisation transformation on concurrent objects. Presented at the Fourth Workshop on Foundations of Object-Oriented Languages (FOOL 4). To appear in *Theory and Practice of Object-oriented Systems*, 1997.
42. D. Sangiorgi. Interpreting functions as pi-calculus processes: a tutorial. Technical Report RR-3470, INRIA-Sophia Antipolis, 1998.

43. P. Sewell. Global/local subtyping and capability inference for a distributed π-calculus. In *Proc. 25th ICALP*, volume 1443 of *Lecture Notes in Computer Science*. Springer Verlag, 1998.
44. G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *Proc. 25th POPL*. ACM Press, 1998.
45. M. Steffen and U. Nestmann. Typing confluence. Interner Bericht IMMD7-xx/95, Informatik VII, Universität Erlangen-Nürnberg, 1995.
46. E. Sumii and N. Kobayashi. A generalized deadlock-free process calculus. In U. Nestmann and B.C. Pierce, editors, *HLCL '98: High-Level Concurrent Languages*, volume 16.3 of *ENTCS*. Elsevier Science Publishers, 1998.
47. N.D. Turner. *The polymorphic pi-calculus: Theory and Implementation*. PhD thesis, Department of Computer Science, University of Edinburgh, 1996.
48. V.T. Vasconcelos. Predicative polymorphism in -calculus. In *Proc. 6th Parallel Architectures and Languages Europe*, volume 817 of *Lecture Notes in Computer Science*. Springer Verlag, 1994.
49. V.T. Vasconcelos and R. Bastos. Core-TyCO, the language definition, version 0.1. DI/FCUL TR 98–3, Department of Computer Science, University of Lisbon, March 1998.
50. V.T. Vasconcelos and K. Honda. Principal typing schemes in a polyadic π-calculus. In E. Best, editor, *Proc. CONCUR '93*, volume 715 of *Lecture Notes in Computer Science*. Springer Verlag, 1993.
51. V.T. Vasconcelos and M. Tokoro. A typing system for a calculus of objects. In *Proc. Object Technologies for Advanced Software '93*, volume 742 of *Lecture Notes in Computer Science*, pages 460–474. Springer Verlag, 1993.
52. D. Walker. Objects in the π-calculus. *Information and Computation*, 116(2):253–271, 1995.
53. N. Yoshida. Graph types for monadic mobile processes. In *Proc. FST & TCS*, volume 1180 of *Lecture Notes in Computer Science*, pages 371–386. Springer Verlag, 1996.

Testing Hennessy-Milner Logic with Recursion*

Luca Aceto** and Anna Ingólfsdóttir***

BRICS†, Department of Computer Science, Aalborg University,
Fredrik Bajers Vej 7-E, DK-9220 Aalborg Ø, Denmark.

Abstract. This study offers a characterization of the collection of properties expressible in Hennessy-Milner Logic (HML) with recursion that can be tested using finite LTSs. In addition to actions used to probe the behaviour of the tested system, the LTSs that we use as tests will be able to perform a distinguished action `nok` to signal their dissatisfaction during the interaction with the tested process. A process s *passes the test* T iff T does not perform the action `nok` when it interacts with s. A test T tests for a property ϕ in HML with recursion iff it is passed by exactly the states that satisfy ϕ. The paper gives an expressive completeness result offering a characterization of the collection of properties in HML with recursion that are testable in the above sense.

1 Introduction

Observational semantics for concurrent processes are based upon the general idea that two processes should be equated, unless they behave differently, in some precise sense, when they are made to interact with some distinguishing environment. Such an idea is, in arguably its purest form, the foundation of the theory of the *testing equivalences* of De Nicola and Hennessy [4, 6]. In the theory of testing equivalence, two processes, described abstractly as *labelled transition systems* (LTSs) [8], are deemed to be equivalent iff they pass exactly the same tests. A *test* is itself an LTS — i.e., a process — which may perform a distinguished action to signal that it is (un)happy with the outcome of its interaction with the tested process. Intuitively, the purpose of submitting a process to a test is to discover whether it enjoys some distinguished property or not. Testing equivalence then stipulates that two processes that enjoy the same properties for which tests can be devised are to be considered equivalent. The main aim of this study is to present a characterization of the collection of properties of concurrent processes that can be tested using LTSs. Of course, in order to be able to even attempt such a characterization (let alone provide

* The work reported in this paper was mostly carried out during the authors' stay at the Dipartimento di Sistemi ed Informatica, Università di Firenze, Italy.

** Partially supported by a grant from the CNR, Gruppo Nazionale per l'Informatica Matematica (GNIM). Email:`luca@cs.auc.dk`.

*** Supported by the Danish Research Council. Email: `annai@cs.auc.dk`.

† **B**asic **R**esearch **i**n **C**omputer **S**cience, Centre of the Danish National Research Foundation.

W. Thomas (Ed.): FOSSACS'99, LNCS 1578, pp. 41-55, 1999.

it), we need to precisely define a formalism for the description of properties of LTSs, single out a collection of LTSs as tests, and describe the testing process and when an LTS passes or fails a test.

As our specification formalism for properties of processes, we use Hennessy-Milner Logic (HML) with recursion [10]. This is a very expressive property language which results from the addition of least and greatest fixed points to the logic considered by Hennessy and Milner in their seminal study [7]. The resulting property language is indeed just a reformulation of the modal μ-calculus [10]. Following the idea of using test automata to check whether processes enjoy properties described by formulae in such a language [2, 1], we use finite LTSs as property testers. In addition to actions used to probe the behaviour of the tested system, the LTSs that we use as tests will be able to perform a distinguished action `nok` (read 'not okay') to signal their dissatisfaction during the interaction with the tested process. As in the approach underlying the testing equivalences, a test interacts with a process by communicating with it, and, in keeping with the aforementioned references, the interaction between processes and tests will be described using the (derived) operation of restricted parallel composition from CCS [13].

We say that a process s *fails the test* T iff T can perform the action `nok` when it interacts with s. Otherwise s *passes* T. A test T tests for a property ϕ in HML with recursion iff it is passed by exactly the states that satisfy ϕ. The main result of the paper is an expressive completeness result offering a characterization of the collection of properties in HML with recursion that are testable in the above sense. We refer to this language as SHML (for 'safety HML'). More precisely we show that:

- every property ϕ of SHML is testable, in the sense that there exists a test T_ϕ such that s satisfies ϕ if and only if s passes T_ϕ, for every process s; and
- every test T is expressible in SHML, in the sense that there exists a formula ϕ_T of SHML such that, for every process s, the agent s passes T if and only if s satisfies ϕ_T.

This expressive completeness result will be obtained as a corollary of a stronger result pertaining to the compositionality of the property language SHML. A property language is *compositional* if checking whether a composite system $s\|T$ satisfies a property ϕ can be reduced to deciding whether the component s has a corresponding property ϕ/T. As the property ϕ/T is required to be expressible in the property language under consideration, compositionality clearly puts a demand on its expressive power. Let $\mathcal{L}_{\mathtt{nok}}$ be the property language that only contains the simple safety property $[\mathtt{nok}]\mathtt{ff}$, expressing that the `nok` action cannot be performed. We prove that SHML is the least expressive, compositional extension of the language $\mathcal{L}_{\mathtt{nok}}$ (Thm. 3.19). This yields the desired expressive completeness result because any compositional property language that can express the property $[\mathtt{nok}]\mathtt{ff}$ is expressive complete with respect to tests (Propn. 3.13). Any increase in expressiveness for the language SHML can only be obtained at the loss of testability.

The paper is organized as follows. After reviewing the model of labelled transition systems and HML with recursion (Sect. 2), we introduce tests and describe how they can be used to test for properties of processes (Sect. 3). We then proceed to argue that not every formula in HML with recursion is testable (Propn. 3.4), but that its sub-language SHML is (Sect. 3.1). Our main results on the compositionality and completeness of SHML are presented in Sect. 3.2.

2 Preliminaries

We begin by briefly reviewing the basic notions from process theory that will be needed in this study. The interested reader is referred to, e.g., [7, 10, 13] for more details.

Labelled Transition Systems Let Act be a set of *actions*, and let a, b range over it. We assume that Act comes equipped with a mapping $\bar{\cdot} : \mathsf{Act} \to \mathsf{Act}$ such that $\bar{\bar{a}} = a$, for every $a \in \mathsf{Act}$. Action $\bar{a}$ is said to be the *complement* of a. We let Act_τ (ranged over by μ) stand for $\mathsf{Act} \cup \{\tau\}$, where τ is a symbol not occurring in Act. Following Milner [13], the symbol τ will stand for an internal action of a system; such actions will typically arise from the synchronization of complementary actions (cf. the rules for the operation of parallel composition in Defn. 2.2).

Definition 2.1. A *labelled transition system (LTS)* over the set of actions Act_τ is a triple $\mathcal{T} = \langle \mathcal{S}, \mathsf{Act}_\tau, \longrightarrow \rangle$ where $\mathcal{S}$ is a set of *states*, and $\longrightarrow \subseteq \mathcal{S} \times \mathsf{Act}_\tau \times \mathcal{S}$ is a *transition relation*. An LTS is *finite* iff its set of states and its transition relation are both finite. It is *rooted* if a distinguished state $\mathsf{root}(\mathcal{T}) \in \mathcal{S}$ is singled out as its start state.

As it is standard practice in process theory, we use the more suggestive notation $s \xrightarrow{\mu} s'$ in lieu of $(s, \mu, s') \in \longrightarrow$. We also write $s \xrightarrow{\mu}$ if there is no state s' such that $s \xrightarrow{\mu} s'$. Following [13], we now proceed to define versions of the transition relations that abstract from the internal evolution of states as follows:

$$s \stackrel{\varepsilon}{\Rightarrow} s' \quad \text{iff} \quad s \stackrel{\tau}{\rightarrow}{}^{*} s'$$

$$s \stackrel{\mu}{\Rightarrow} s' \quad \text{iff} \quad \exists s_1, s_2. \quad s \stackrel{\varepsilon}{\Rightarrow} s_1 \xrightarrow{\mu} s_2 \stackrel{\varepsilon}{\Rightarrow} s'$$

where we use $\stackrel{\tau}{\rightarrow}{}^{*}$ to stand for the reflexive, transitive closure of $\stackrel{\tau}{\rightarrow}$.

Definition 2.2 (Operations on LTSs).

- Let $\mathcal{T}_i = \langle \mathcal{S}_i, \mathsf{Act}_\tau, \longrightarrow_i \rangle$ $(i \in \{1,2\})$ be two LTSs. The parallel composition of $\mathcal{T}_1$ and $\mathcal{T}_2$ is the LTS $\mathcal{T}_1 \parallel \mathcal{T}_2 = \langle \mathcal{S}_1 \times \mathcal{S}_2, \mathsf{Act}_\tau, \longrightarrow \rangle$, where the transition relation $\longrightarrow$ is defined by the rules $(\mu \in \mathsf{Act}_\tau, a \in \mathsf{Act})$:

$$\frac{s_1 \xrightarrow{\mu}_1 s_1'}{s_1 \| s_2 \xrightarrow{\mu} s_1' \| s_2} \qquad \frac{s_2 \xrightarrow{\mu}_2 s_2'}{s_1 \| s_2 \xrightarrow{\mu} s_1 \| s_2'} \qquad \frac{s_1 \xrightarrow{a}_1 s_1' \qquad s_2 \xrightarrow{\overline{a}}_2 s_2'}{s_1 \| s_2 \xrightarrow{\tau} s_1' \| s_2'}$$

  In the rules above, and in the remainder of the paper, we use the more suggestive notation $s \parallel s'$ in lieu of (s, s').

– Let $\mathcal{T} = \langle \mathcal{S}, \mathsf{Act}_\tau, \rightarrow \rangle$ be an LTS and let $L \subseteq \mathsf{Act}$ be a set of actions. The restriction of $\mathcal{T}$ over L is the LTS $\mathcal{T}\backslash L = \langle \mathcal{S}\backslash L, \mathsf{Act}_\tau, \rightsquigarrow \rangle$, where $\mathcal{S}\backslash L = \{s\backslash L \mid s \in \mathcal{S}\}$ and the transition relation $\rightsquigarrow$ is defined by the rules:

$$\frac{s \stackrel{\tau}{\rightarrow} s'}{s\backslash L \stackrel{\tau}{\rightsquigarrow} s'\backslash L} \qquad \frac{s \stackrel{a}{\rightarrow} s'}{s\backslash L \stackrel{a}{\rightsquigarrow} s'\backslash L}$$

where $a, \overline{a} \notin L$.

The reader familiar with [13] may have noticed that the above definitions of parallel composition and restriction are precisely those of CCS. We refer the interested reader to *op. cit.* for more details on these operations.

Hennessy-Milner Logic with Recursion In their seminal study [7], Hennessy and Milner gave a logical characterization of bisimulation equivalence [14] (over states of image-finite LTSs) in terms of a (multi-)modal logic which has since then been referred to as *Hennessy-Milner Logic* (HML). For the sake of completeness and clarity, we now briefly review a variation of this property language for concurrent processes which contains operations for the recursive definition of formulae — a feature that dramatically increases its expressive power. The interested reader is referred to, e.g., [10] for more details.

Definition 2.3. Let Var be a countably infinite set of formula variables, and let $\mathtt{nok}$ denote an action symbol not contained in Act. The collection $\mathsf{HML}(\mathsf{Var})$ of formulae over Var and $\mathsf{Act} \cup \{\mathtt{nok}\}$ is given by the following grammar:

$$\phi ::= \mathtt{tt} \mid \mathtt{ff} \mid \phi \vee \phi \mid \phi \wedge \phi \mid \langle \alpha \rangle \phi \mid [\alpha]\phi \mid X \mid \mathtt{min}(X, \phi) \mid \mathtt{max}(X, \phi)$$

where $\alpha \in \mathsf{Act} \cup \{\mathtt{nok}\}$, X is a formula variable and $\mathtt{min}(X, \phi)$ (respectively, $\mathtt{max}(X, \varphi)$) stands for the least (respectively, largest) solution of the recursion equation $X = \varphi$.

We use $\mathsf{SHML}(\mathsf{Var})$ (for 'safety HML') to stand for the collection of formulae in $\mathsf{HML}(\mathsf{Var})$ that do not contain occurrences of $\vee$, $\langle \alpha \rangle$ and $\mathtt{min}(X, \phi)$.

A *closed recursive formula* of $\mathsf{HML}(\mathsf{Var})$ is a formula in which every formula variable X is *bound*, i.e., every occurrence of X appears within the scope of some $\mathtt{min}(X, \phi)$ or $\mathtt{max}(X, \phi)$ construct. A variable X is *free* in the formula ϕ if some occurrence of it in ϕ is not bound. For example, the formula $\mathtt{max}(X, X)$ is closed, but $\mathtt{min}(X, [a]Y)$ is not because Y is free in it. The collection of closed formulae contained in $\mathsf{HML}(\mathsf{Var})$ (respectively, $\mathsf{SHML}(\mathsf{Var})$) will be written HML (resp. SHML). In the remainder of this paper, every formula will be closed, unless specified otherwise, and we shall identify formulae that only differ in the names of their bound variables. For formulae ϕ and ψ, and a variable X, we write $\phi\{\psi/X\}$ for the formula obtained by replacing every free occurrence of X in ϕ with ψ. The details of such an operation in the presence of binders are standard (see, e.g., [15]), and are omitted here.

Given an LTS $\mathcal{T} = \langle \mathcal{S}, \mathsf{Act}_\tau, \longrightarrow \rangle$, an *environment* is a mapping $\rho : \mathsf{Var} \rightarrow 2^{\mathcal{S}}$. For an environment ρ, variable X and subset of states S, we write $\rho[X \mapsto S]$ for the environment mapping X to S, and acting like ρ on all the other variables.

Definition 2.4 (Satisfaction Relation). Let $\mathcal{T} = \langle \mathcal{S}, \mathsf{Act}_\tau, \longrightarrow \rangle$ be an LTS. For every environment ρ and formula φ contained in $\mathsf{HML}(\mathsf{Var})$, the collection $[\![\varphi]\!]\rho$ of states in $\mathcal{S}$ satisfying the formula φ with respect to ρ is defined by structural recursion on φ thus:

$$
\begin{aligned}
[\![\mathtt{tt}]\!]\rho &\stackrel{\text{def}}{=} \mathcal{S} \\
[\![\mathtt{ff}]\!]\rho &\stackrel{\text{def}}{=} \varnothing \\
[\![\varphi_1 \vee \varphi_2]\!] &\stackrel{\text{def}}{=} [\![\varphi_1]\!]\rho \cup [\![\varphi_2]\!]\rho \\
[\![\varphi_1 \wedge \varphi_2]\!] &\stackrel{\text{def}}{=} [\![\varphi_1]\!]\rho \cap [\![\varphi_2]\!]\rho \\
[\![\langle\alpha\rangle\varphi]\!] &\stackrel{\text{def}}{=} \left\{ s \mid s \stackrel{\alpha}{\Rightarrow} s' \text{ for some } s' \in [\![\varphi]\!]\rho \right\} \\
[\![[\alpha]\varphi]\!] &\stackrel{\text{def}}{=} \left\{ s \mid \text{for every } s',\ s \stackrel{\alpha}{\Rightarrow} s' \text{ implies } s' \in [\![\varphi]\!]\rho \right\} \\
[\![X]\!]\rho &\stackrel{\text{def}}{=} \rho(X) \\
[\![\mathtt{min}(X,\varphi)]\!]\rho &\stackrel{\text{def}}{=} \bigcap \{ S \mid [\![\varphi]\!]\rho[X \mapsto S] \subseteq S \} \\
[\![\mathtt{max}(X,\varphi)]\!]\rho &\stackrel{\text{def}}{=} \bigcup \{ S \mid S \subseteq [\![\varphi]\!]\rho[X \mapsto S] \} \ .
\end{aligned}
$$

The interested reader will find more details on this definition in, e.g., [10]. Here we just confine ourselves to remarking that, as the interpretation of each formula ϕ containing at most X free induces a monotone mapping $[\![\phi]\!] : 2^{\mathcal{S}} \to 2^{\mathcal{S}}$, the closed formulae $\mathtt{min}(X,\phi)$ and $\mathtt{max}(X,\phi)$ are indeed interpreted as the least and largest solutions, respectively, of the equation $X = \phi$. If φ is a closed formula, then the collection of states satisfying it is independent of the environment ρ, and will be written $[\![\varphi]\!]$. In the sequel, for every state s and closed formula φ, we shall write $s \models \varphi$ (read 's satisfies φ') in lieu of $s \in [\![\varphi]\!]$.

When restricted to SHML, the satisfaction relation $\models$ is the largest relation included in $\mathcal{S} \times \mathsf{SHML}$ satisfying the implications in Table 1. A relation satisfying the defining implications for $\models$ will be called a *satisfiability relation*. It follows from standard fixed-point theory [16] that, over $\mathcal{S} \times \mathsf{HML}$, the relation $\models$ is the union of all satisfiability relations and that the above implications are in fact biimplications for $\models$.

Remark. Since $\mathtt{nok}$ is not contained in Act, every state of an LTS trivially satisfies formulae of the form $[\mathtt{nok}]\phi$. The role played by these formulae in the developments of this paper will become clear in Sect. 3.2. Dually, no state of an LTS satisfies formulae of the form $\langle\mathtt{nok}\rangle\varphi$.

Formulae ϕ and ψ are *logically equivalent* (with respect to $\models$) iff they are satisfied by the same states. We say that a formula is *satisfiable* iff it is satisfied by at least one state in some LTS, otherwise we say that it is *unsatisfiable*.

3 Testing Formulae

As mentioned in Sect. 1, the main aim of this paper is to present a complete characterization of the class of testable properties of states of LTSs that can

$$
\begin{array}{rcl}
s \models \mathtt{tt} & \Rightarrow & true \\
s \models \mathtt{ff} & \Rightarrow & false \\
s \models \varphi_1 \wedge \varphi_2 & \Rightarrow & s \models \varphi_1 \text{ and } s \models \varphi_2 \\
s \models [\alpha]\varphi & \Rightarrow & \forall s'.\ s \stackrel{\alpha}{\Rightarrow} s' \text{ implies } s' \models \varphi \\
s \models \mathtt{max}(X, \varphi) & \Rightarrow & s \models \varphi\{\mathtt{max}(X, \varphi)/X\}
\end{array}
$$

Table 1. Satisfaction implications

be expressed in the language HML. In this section we define the collection of *tests* and the notion of *property testing* used in this study. Informally, testing involves the parallel composition of the tested state with a test. Following the spirit of the classic approach of De Nicola and Hennessy [4, 6], we say that the tested state fails a test if the distinguished reject action `nok` can be performed by the test while it interacts with it, and passes otherwise. The formal definition of testing then involves the definition of what a test is, how interaction takes place and when the test has failed or succeeded. We now proceed to make these notions precise.

Definition 3.1 (Tests). A *test* is a finite, rooted LTS over the set of actions $\mathsf{Act}_\tau \cup \{\mathtt{nok}\}$.

In the remainder of this study, tests will often be concisely described using the regular fragment of Milner's CCS [13] given by the following grammar:

$$T ::= \mathbf{0} \mid \alpha.T \mid T + T \mid X \mid \mathrm{fix}(X = T)$$

where $\alpha \in \mathsf{Act}_\tau \cup \{\mathtt{nok}\}$, and X ranges over Var. As usual, we shall only be concerned with the closed expressions generated by the above grammar, with $\mathrm{fix}(X = T)$ as the binding construct, and we shall identify expressions that only differ in the names of their bound variables. In the sequel, the symbol $\equiv$ will be used to denote syntactic equality up to renaming of bound variables. The operation of substitution over the set of expressions given above is defined exactly as for formulae in HML(Var). The operational semantics of the expressions generated by the above grammar is given by the classic rules for CCS. These are reported below for the sake of clarity:

$$\frac{}{\alpha.T \stackrel{\alpha}{\to} T} \qquad \frac{T_1 \stackrel{\alpha}{\to} T_1'}{T_1 + T_2 \stackrel{\alpha}{\to} T_1'} \qquad \frac{T_2 \stackrel{\alpha}{\to} T_2'}{T_1 + T_2 \stackrel{\alpha}{\to} T_2'} \qquad \frac{T\{\mathrm{fix}(X = T)/X\} \stackrel{\alpha}{\to} T'}{\mathrm{fix}(X = T) \stackrel{\alpha}{\to} T'}$$

where α is either `nok` or an action in Act_τ. The intention is that the term T stands for the test whose start state is T itself, whose transitions are precisely those that are provable using the above inference rules, and whose set of states is the collection of expressions reachable from T by performing zero or more transitions. We refer the reader to [13] for more information on the operational semantics of CCS.

Definition 3.2 (Testing Properties). Let φ be a formula in HML, and let T be a test.

- A state s of an LTS passes the test T iff $(s \| \mathsf{root}(T)) \backslash \mathsf{Act} \stackrel{\mathtt{nok}}{\not\Rightarrow}$. Otherwise we say that s fails the test T.
- We say that the test T *tests* for the formula φ (and that φ is *testable*) iff for every LTS $\mathcal{T}$ and every state s of $\mathcal{T}$, $s \models \varphi$ iff s passes the test T.
- Let $\mathcal{L}$ be a collection of formulae in HML. We say that $\mathcal{L}$ is testable iff each of the formulae in $\mathcal{L}$ is.

Example 3.3. The formula $[a]\mathtt{ff}$ states that a process does not afford a $\stackrel{a}{\Rightarrow}$-transition. We therefore expect that a suitable test for such a property is $T \equiv \bar{a}.\mathtt{nok}.\mathbf{0}$. Indeed, the reader will easily realize that $(s \| T) \backslash \mathsf{Act} \stackrel{\mathtt{nok}}{\not\Rightarrow}$ iff $s \stackrel{a}{\not\Rightarrow}$, for every state s. The formula $[a]\mathtt{ff}$ is thus testable, in the sense of this paper.

The formula $\mathtt{max}(X, [a]\mathtt{ff} \wedge [b]X)$ is satisfied by those states which cannot perform a $\stackrel{a}{\Rightarrow}$-transition, no matter how they engage in a sequence of $\stackrel{b}{\Rightarrow}$-transitions. A suitable test for such a property is $\mathrm{fix}(X = \bar{a}.\mathtt{nok}.\mathbf{0} + \bar{b}.X)$, and the formula $\mathtt{max}(X, [a]\mathtt{ff} \wedge [b]X)$ is thus testable.

As already stated, our main aim in this paper is to present a characterization of the collection of HML-properties that are testable in the sense of Defn. 3.2. To this end, we begin by providing evidence to the effect that not every property expressible in HML is testable.

Proposition 3.4 (Two Negative Results).

1. *Let ϕ be a formula in* HML*. Suppose that ϕ is satisfiable. Then, for every action a in* Act*, the formula $\langle a \rangle \phi$ is not testable.*
2. *Let a and b be two distinct actions in* Act*. Then the formula $[a]\mathtt{ff} \vee [b]\mathtt{ff}$ is not testable.*

Remark. If φ is unsatisfiable, then the formula $\langle a \rangle \varphi$ is logically equivalent to $\mathtt{ff}$. Since $\mathtt{ff}$ is testable using the test $\mathtt{nok}.\mathbf{0}$, the requirement on φ is necessary for Propn. 3.4(1) to hold. Note moreover that, as previously remarked, both the formulae $[a]\mathtt{ff}$ and $[b]\mathtt{ff}$ are testable, but their disjunction is not (Propn. 3.4(2)).

Our aim in the remainder of this paper is to show that the collection of testable properties is precisely SHML. This is formalized by the following result.

Theorem 3.5. *The collection of formulae* SHML *is testable. Moreover, every testable property in* HML *can be expressed in* SHML*.*

The remainder of this paper will be devoted to a proof of the above theorem. In the process of developing such a proof, we shall also establish some results pertaining to the expressive power of SHML which may be of independent interest.

3.1 Testability of SHML

We begin our proof of Thm. 3.5 by showing that the language SHML is testable. To this end, we define, for every open formula ϕ in the language SHML(Var), a regular CCS expression T_ϕ by structural recursion thus:

$$\begin{array}{rlrl} T_{\mathtt{tt}} & \stackrel{\text{def}}{=} \mathbf{0} & T_{[a]\phi} & \stackrel{\text{def}}{=} \bar{a}.T_\phi \\ T_{\mathtt{ff}} & \stackrel{\text{def}}{=} \mathtt{nok}.\mathbf{0} & T_X & \stackrel{\text{def}}{=} X \\ T_{\phi_1 \wedge \phi_2} & \stackrel{\text{def}}{=} \tau.T_{\phi_1} + \tau.T_{\phi_2} & T_{\mathtt{max}(X,\phi)} & \stackrel{\text{def}}{=} \mathrm{fix}(X = T_\phi) \ . \end{array}$$

For example, if $\phi \equiv \mathtt{max}(X, [a]\mathtt{ff} \wedge [b]X)$ then T_ϕ is the test $\mathrm{fix}(X = \tau.\bar{a}.\mathtt{nok}.\mathbf{0} + \tau.\bar{b}.X)$. We recall that we identify CCS descriptions of tests that only differ in the name of their bound variables since they give rise to isomorphic LTSs. Our order of business in this section will be to show the following result:

Theorem 3.6. *Let ϕ be a closed formula contained in* SHML*. Then the test T_ϕ tests for it.*

In the proof of this theorem, it will be convenient to have an alternative, novel characterization of the satisfaction relation for formulae in the language SHML. This we now proceed to present.

Definition 3.7. Let $\mathcal{T} = \langle \mathcal{S}, \mathsf{Act}_\tau, \longrightarrow \rangle$ be an LTS. The satisfaction relation $\models_\varepsilon$ is the largest relation included in $\mathcal{S} \times$ SHML satisfying the following implications:

$$\begin{array}{rcl} s \models_\varepsilon \mathtt{tt} & \Rightarrow & \mathit{true} \\ s \models_\varepsilon \mathtt{ff} & \Rightarrow & \mathit{false} \\ s \models_\varepsilon \varphi_1 \wedge \varphi_2 & \Rightarrow & s' \models_\varepsilon \varphi_1 \text{ and } s' \models_\varepsilon \varphi_2, \text{ for every } s' \text{ such that } s \stackrel{\varepsilon}{\Rightarrow} s' \\ s \models_\varepsilon [a]\varphi & \Rightarrow & s \stackrel{a}{\Rightarrow} s' \text{ implies } s' \models_\varepsilon \varphi, \text{ for every } s' \\ s \models_\varepsilon \mathtt{max}(X, \varphi) & \Rightarrow & s' \models_\varepsilon \varphi\{\mathtt{max}(X,\varphi)/X\}, \text{ for every } s' \text{ such that } s \stackrel{\varepsilon}{\Rightarrow} s' \end{array}$$

A relation satisfying the above implications will be called a *weak satisfiability relation.*

The satisfaction relation $\models_\varepsilon$ is closed with respect to the relation $\stackrel{\varepsilon}{\Rightarrow}$, in the sense of the following proposition.

Proposition 3.8. *Let $\mathcal{T} = \langle \mathcal{S}, \mathsf{Act}_\tau, \longrightarrow \rangle$ be an LTS. Then, for every $s \in \mathcal{S}$ and $\varphi \in$ SHML, $s \models_\varepsilon \varphi$ iff $s' \models_\varepsilon \varphi$, for every s' such that $s \stackrel{\varepsilon}{\Rightarrow} s'$.*

Proof. The only interesting thing to check is that if $s \models_\varepsilon \varphi$ and $s \stackrel{\varepsilon}{\Rightarrow} s'$, then $s' \models_\varepsilon \varphi$. To this end, it is sufficient to prove that the relation $\mathcal{R}$ defined thus:

$$\mathcal{R} \stackrel{\text{def}}{=} \{(s, \varphi) \mid \exists t.\ t \models_\varepsilon \varphi \text{ and } t \stackrel{\varepsilon}{\Rightarrow} s\}$$

is a weak satisfiability relation. The straightforward verification is left to the reader. □

We now proceed to establish that the relations $\models_\varepsilon$ and $\models$ coincide for formulae in SHML.

Proposition 3.9. *Let ϕ be a formula contained in* SHML. *Then, for every state s of an LTS, $s \models \phi$ iff $s \models_\varepsilon \phi$.*

In the proof of Thm. 3.6, it will be convenient to have at our disposal some further auxiliary results. For ease of reference, these are collected in the following lemma.

Lemma 3.10.

1. *Let ϕ be a formula in* SHML. *Assume that $T_\phi \stackrel{\mathtt{nok}}{\rightarrow}$. Then ϕ is logically equivalent to $\mathtt{ff}$.*
2. *Let ϕ be a formula in* SHML. *Assume that $T_\phi \stackrel{\tau}{\rightarrow} T$. Then there are formulae ϕ_1 and ϕ_2 in* SHML *such that $T \equiv T_{\phi_1}$, and ϕ is logically equivalent to $\phi_1 \wedge \phi_2$.*
3. *Let ϕ be a formula in* SHML. *Assume that $T_\phi \stackrel{\bar{a}}{\rightarrow} T$. Then there is a formula ψ in* SHML *such that $T \equiv T_\psi$, and ϕ is logically equivalent to $[a]\psi$.*

Using these results, we are now in a position to prove Thm. 3.6.

Proof of Thm. 3.6: In light of Propn. 3.9, it is sufficient to show that, for every state s of an LTS and closed formula $\phi \in \mathsf{SHML}$,

$$s \models_\varepsilon \phi \text{ iff } (s \| T_\phi) \backslash \mathsf{Act} \stackrel{\mathtt{nok}}{\not\Rightarrow} \quad .$$

We prove the two implications separately.

- 'If Implication'. It is sufficient to show that the relation

  $$\mathcal{R} \stackrel{\text{def}}{=} \left\{ (s, \phi) \mid (s \| T_\phi) \backslash \mathsf{Act} \stackrel{\mathtt{nok}}{\not\Rightarrow} \text{ and } \phi \in \mathsf{SHML} \right\}$$

  is a weak satisfiability relation. The details of the proof are left to the reader.
- 'Only If Implication'. We prove the contrapositive statement. To this end, assume that

  $$(s \| T_\phi) \backslash \mathsf{Act} \stackrel{\varepsilon}{\Rightarrow} (s' \| T') \backslash \mathsf{Act} \stackrel{\mathtt{nok}}{\rightarrow}$$

  for some state s' and test T'. We show that $s \not\models_\varepsilon \phi$ holds by induction on the length of the computation $(s \| T_\phi) \backslash \mathsf{Act} \stackrel{\varepsilon}{\Rightarrow} (s' \| T') \backslash \mathsf{Act}$.

  - Base Case: $(s \| T_\phi) \backslash \mathsf{Act} \equiv (s' \| T') \backslash \mathsf{Act} \stackrel{\mathtt{nok}}{\rightarrow}$. In this case, we may infer that $T_\phi \stackrel{\mathtt{nok}}{\rightarrow}$. By Lemma 3.10(1), it follows that ϕ is unsatisfiable. Propn. 3.9 now yields that $s \not\models_\varepsilon \phi$, which was to be shown.
  - Inductive Step: $(s \| T_\phi) \backslash \mathsf{Act} \stackrel{\tau}{\rightarrow} (s'' \| T'') \backslash \mathsf{Act} \stackrel{\varepsilon}{\Rightarrow} (s' \| T') \backslash \mathsf{Act} \stackrel{\mathtt{nok}}{\rightarrow}$, for some state s'' and test T''. We proceed by a case analysis on the form the transition

    $$(s \| T_\phi) \backslash \mathsf{Act} \stackrel{\tau}{\rightarrow} (s'' \| T'') \backslash \mathsf{Act}$$

    may take.

- CASE: $s \xrightarrow{\tau} s''$ and $T'' \equiv T_\phi$. In this case, we may apply the inductive hypothesis to infer that $s'' \not\models_\varepsilon \phi$. By Propn. 3.8, it follows that $s \not\models_\varepsilon \phi$, which was to be shown.
- CASE: $T_\phi \xrightarrow{\tau} T''$ and $s = s''$. By Lemma 3.10(2), it follows that ϕ is logically equivalent to $\phi_1 \wedge \phi_2$ for some formulae ϕ_1 and ϕ_2 in SHML, and that $T'' \equiv T_{\phi_1}$. By induction, we may now infer that $s \not\models_\varepsilon \phi_1$. Since ϕ is logically equivalent to $\phi_1 \wedge \phi_2$, this implies that $s \not\models_\varepsilon \phi$ (Propn. 3.9), which was to be shown.
- CASE: $s \xrightarrow{a} s''$ and $T_\phi \xrightarrow{\bar{a}} T''$, for some action $a \in \mathsf{Act}$. By Lemma 3.10(3), it follows that ϕ is logically equivalent to $[a]\psi$ for some formula ψ in SHML, and that $T'' \equiv T_\psi$. By induction, we may now infer that $s'' \not\models_\varepsilon \psi$. Since ϕ is logically equivalent to $[a]\psi$ and $s \xrightarrow{a} s'' \not\models_\varepsilon \psi$, this implies that $s \not\models_\varepsilon \phi$ (Propn. 3.9), which was to be shown.

This completes the inductive argument, and the proof of the 'only if' implication.

The proof of the theorem is now complete. □

3.2 Expressive Completeness of SHML

We have just shown that every property φ which can be expressed in the language SHML is testable, in the sense of Defn. 3.2. We now address the problem of the expressive completeness of this property language with respect to tests. More precisely, we study whether all properties that are testable can be expressed in the property language SHML — in the sense that, for every test T, there exists a formula ψ_T in SHML such that every state of an LTS passes the test T if, and only if, it satisfies ψ_T. Our aim in this section is to complete the proof of Thm. 3.5 by arguing that the language SHML *is* expressive complete, in the sense that every test T may be expressed as a property in the language SHML in the precise technical sense outlined above. This amounts to establishing an expressive completeness result for SHML akin to classic ones presented in, e.g., [9, 5, 17]. In the proof of this expressive completeness result, we shall follow an indirect approach by focusing on the compositionality of a property language $\mathcal{L}$ with respect to tests and the parallel composition operator $\|$. As we shall see (cf. Propn. 3.13), if a property language $\mathcal{L}$, that contains the property $[\mathtt{nok}]\mathtt{ff}$, is compositional with respect to tests and $\|$ (cf. Defn. 3.12) then it is expressive complete (cf. Defn. 3.11). We shall show that SHML is compositional with respect to tests and $\|$, and obtain the expressive completeness of such a language as a corollary of this stronger result.

We begin with some preliminary definitions, introducing the key concepts of compositionality and (expressive) completeness.

Definition 3.11 (Expressive completeness). Let $\mathcal{L}$ be a collection of formulae in HML. We say that $\mathcal{L}$ is *(expressive) complete* (with respect to tests) if for every test T there exists a formula $\varphi_T \in \mathcal{L}$ such that, for every state s of an LTS, $s \models \varphi_T$ iff s passes the test T.

Compositionality, on the other hand, is formally defined as follows:

Definition 3.12 (Compositionality). Let $\mathcal{L}$ be a collection of formulae in HML. We say that $\mathcal{L}$ is *compositional* (with respect to tests and $\|$) if, for every $\varphi \in \mathcal{L}$ and every test T, there exists a formula $\varphi/T \in \mathcal{L}$ such that, for every state s of an LTS, $s \parallel \mathsf{root}(T) \models \varphi$ iff $s \models \varphi/T$.

Intuitively, the formula φ/T states a necessary and sufficient condition for state s to satisfy φ when it is made to interact with the test T.

Our interest in compositionality stems from the following result that links it to the notion of completeness. In the sequel, we use $\mathcal{L}_{\mathtt{nok}}$ to denote the property language that only consists of the formula $[\mathtt{nok}]\mathtt{ff}$. (Recall that nok is a fresh action not contained in Act.)

Proposition 3.13. *Let $\mathcal{L}$ be a collection of formulae in* HML *that includes $\mathcal{L}_{\mathtt{nok}}$. Suppose that $\mathcal{L}$ is compositional. Then $\mathcal{L}$ is complete with respect to tests.*

Proof. Consider an arbitrary test T. We aim at exhibiting a formula $\phi_T \in \mathcal{L}$ meeting the requirements in Defn. 3.11. Since $\mathcal{L}$ is compositional and contains the formula $[\mathtt{nok}]\mathtt{ff}$, we may define φ_T to be the formula $([\mathtt{nok}]\mathtt{ff})/T$. Let s be an arbitrary state of an LTS. We can now argue that s passes T iff it satisfies ϕ_T thus:

$$
\begin{aligned}
s \text{ passes the test } T \text{ iff } & (s \| \mathsf{root}(T)) \backslash \mathsf{Act} \stackrel{\mathtt{nok}}{\not\Rightarrow} \\
\text{iff } & (s \| \mathsf{root}(T)) \backslash \mathsf{Act} \models [\mathtt{nok}]\mathtt{ff} \\
\text{iff } & (s \| \mathsf{root}(T)) \models [\mathtt{nok}]\mathtt{ff} \\
& \qquad (\text{As } \mathtt{nok} \notin \mathsf{Act}) \\
\text{iff } & s \models ([\mathtt{nok}]\mathtt{ff})/T \\
& \qquad (\text{As } \mathcal{L} \text{ is compositional}) \\
\text{iff } & s \models \varphi_T \quad .
\end{aligned}
$$

This completes the proof. □

As we shall now show, SHML is compositional with respect to tests and $\|$, and thus expressive complete with respect to tests. We begin by defining a quotient construction for formulae of SHML, in the spirit of those given for different property languages and over different models in, e.g., [12, 3, 11].

Definition 3.14 (Quotient Construction). Let T be a test, and let t be one of its states. For every formula φ SHML, we define the formula φ/t (read 'φ quotiented by t') as shown in Table 2.

Some remarks about the definition presented in Table 2 are now in order. The definition of the quotient formula φ/t presented *ibidem* should be read as yielding a finite list of recursion equations, over variables of the form ψ/t', for every formula φ and state t of a test. The quotient formula φ/t itself is the component associated with φ/t in the largest solution of the system of equations having φ/t

$$\begin{aligned}
\mathtt{ff}/t &\stackrel{\text{def}}{=} \mathtt{ff}\\
\mathtt{tt}/t &\stackrel{\text{def}}{=} \mathtt{tt}\\
(\phi_1 \wedge \phi_2)/t &\stackrel{\text{def}}{=} \phi_1/t \wedge \phi_2/t\\
([\alpha]\phi)/t &\stackrel{\text{def}}{=} [\alpha](\phi/t) \wedge \bigwedge_{\left\{t' \mid t \stackrel{\alpha}{\Rightarrow} t'\right\}} (\phi/t') \wedge \bigwedge_{\left\{(b,t') \mid t \stackrel{b}{\Rightarrow} t'\right\}} [\bar{b}](([\alpha]\phi)/t')\\
\mathtt{max}(X,\phi)/t &\stackrel{\text{def}}{=} (\phi\{\mathtt{max}(X,\phi)/X\})/t
\end{aligned}$$

Table 2. Quotient construct for SHML

as leading variable. For instance, if φ is the formula $[a]\mathtt{ff}$ and t is a node of a test whose only transition is $t \xrightarrow{\bar{b}} t$, then, as the reader can easily verify, φ/t is the largest solution of the recursion equation:

$$\varphi/t \stackrel{\text{def}}{=} [a]\mathtt{ff} \wedge [b](\varphi/t)$$

which corresponds to the formula $\mathtt{max}(X, [a]\mathtt{ff} \wedge [b]X)$ in the property language SHML. This formula states the, intuitively clear, fact that a state of the form $s \parallel t$ cannot perform a $\stackrel{a}{\Rightarrow}$-transition iff s cannot execute such a step no matter how it engages in a sequence of synchronizations on b with t. Note that the quotient of a recursion-free formula may be a formula involving recursion. It can be shown that this is inevitable, because the recursion-free fragment of SHML is *not* compositional. Finally, we remark that, because of our finiteness restrictions on tests, the right-hand side of the defining equation for $([\alpha]\phi)/t$ is a finite conjunction of formulae.

The following key result states the correctness of the quotient construction.

Theorem 3.15. *Let φ be a closed formula in* SHML. *Suppose that s is a state of an LTS, and t is a state of a test. Then $s \parallel t \models \varphi$ iff $s \models \varphi/t$.*

Proof. We prove the two implications separately.

- 'Only If Implication'. Consider the environment ρ mapping each variable φ/t in the list of equations in Table 2 to the set of states $\{s \mid s\|t \models \varphi\}$. We prove that ρ is a post-fixed point of the monotonic functional on environments associated with the equations in Table 2, i.e., that if $s \in \rho(\phi/t)$ then $s \in [\![\psi]\!]\rho$, where ψ is the right-hand side of the defining equation for ϕ/t. This we now proceed to do by a case analysis on the form the formula φ may take. We only present the details for the most interesting case in the proof.
  - Case: $\varphi \equiv [\alpha]\psi$. Assume that $s\|t \models [\alpha]\psi$. We show that state s is contained in $[\![\xi]\!]\rho$ for every conjunct ξ in the right-hand side of the defining equation for $([\alpha]\psi)/t$.

* Case: $\xi \equiv [\alpha](\psi/t)$. To show that $s \in [\![\xi]\!]\rho$, it is sufficient to prove that $s' \in [\![\psi/t]\!]\rho$, for every s' such that $s \stackrel{\alpha}{\Rightarrow} s'$. To this end, we reason as follows:

$$\begin{array}{rl} s \stackrel{\alpha}{\Rightarrow} s' \text{ implies } s\|t \stackrel{\alpha}{\Rightarrow} s'\|t & \\ \text{implies } s'\|t \models \psi & \\ & \text{(As } s\|t \models [\alpha]\psi\text{)} \\ \text{iff} \quad s' \in \rho(\psi/t) & \\ & \text{(By the definition of } \rho\text{)} \\ \text{iff} \quad s' \in [\![\psi/t]\!]\rho \ . & \end{array}$$

* Case: $\xi \equiv \psi/t'$ with $t \stackrel{\alpha}{\Rightarrow} t'$. To show that $s \in [\![\xi]\!]\rho$, it is sufficient to prove that $s \in [\![\psi/t']\!]\rho$, for every t' such that $t \stackrel{\alpha}{\Rightarrow} t'$. To this end, we reason as follows:

$$\begin{array}{rl} t \stackrel{\alpha}{\Rightarrow} t' \text{ implies } s\|t \stackrel{\alpha}{\Rightarrow} s\|t' & \\ \text{implies } s\|t' \models \psi & \\ & \text{(As } s\|t \models [\alpha]\psi\text{)} \\ \text{iff} \quad s \in \rho(\psi/t') & \\ & \text{(By the definition of } \rho\text{)} \\ \text{iff} \quad s \in [\![\psi/t']\!]\rho \ . & \end{array}$$

* Case: $\xi \equiv [\bar{b}]\big(([\alpha]\psi)/t'\big)$ with $t \stackrel{b}{\Rightarrow} t'$. To show that $s \in [\![\xi]\!]\rho$, it is sufficient to prove that $s' \in [\![([\alpha]\psi)/t']\!]\rho$, for every s' such that $s \stackrel{\bar{b}}{\Rightarrow} s'$. To this end, we reason as follows:

$$\begin{array}{rl} s \stackrel{\bar{b}}{\Rightarrow} s' \text{ and } t \stackrel{b}{\Rightarrow} t' \text{ imply } s\|t \stackrel{\tau}{\Rightarrow} s'\|t' & \\ \text{implies } s'\|t' \models [\alpha]\psi & \\ & \text{(By Propns. 3.8 and 3.9, as } s\|t \models [\alpha]\psi\text{)} \\ \text{iff} \quad s' \in \rho(([\alpha]\psi)/t') & \\ & \text{(By the definition of } \rho\text{)} \\ \text{iff} \quad s' \in [\![([\alpha]\psi)/t']\!]\rho \ . & \end{array}$$

The proof for the case $\phi \equiv [\alpha]\psi$ is now complete.

– 'If Implication'. Consider the relation $\mathcal{R}$ defined thus:

$$\mathcal{R} \stackrel{\text{def}}{=} \{(s\|t, \varphi) \mid s \models \varphi/t\} \ .$$

It is not hard to show that $\mathcal{R}$ is a satisfiability relation.

The proof of the theorem is now complete. □

Corollary 3.16. *The property language* SHML *is compositional with respect to tests and the parallel composition operator* $\|$.

Proof. Given a property $\varphi \in \mathsf{SHML}$ and a test T, define φ/T to be the formula $\varphi/\mathsf{root}(T)$ given by the quotient construction. The claim is now an immediate consequence of Thm. 3.15. □

Theorem 3.17. *The property language* SHML *is expressive complete.*

Example 3.18. Applying the construction in the proof of Propn. 3.13, and the definition of the quotient formula to the tests

$$T_1 \equiv \mathrm{fix}(X = \bar{a}.\mathtt{nok}.\mathbf{0} + \bar{b}.X) \quad \text{and}$$
$$T_2 \equiv \mathrm{fix}(X = \tau.\bar{a}.\mathtt{nok}.\mathbf{0} + \tau.\bar{b}.X)$$

yields that the formula tested by both T_1 and T_2 is $\mathtt{max}(X, [a]\mathtt{ff} \wedge [b]X)$.

Collecting the results in Thms. 3.6 and 3.17, we have now finally completed the proof of Thm. 3.5. Thus, as claimed, the collection of testable properties coincides with that of the properties expressible in SHML. The following result gives another characterization of the expressive power of SHML which has some independent interest.

Theorem 3.19. *The property language* SHML *is the least expressive extension of* $\mathcal{L}_{\mathtt{nok}}$ *that is compositional with respect to tests and* $\|$.

Proof. Assume that $\mathcal{L}$ is a property language that extends $\mathcal{L}_{\mathtt{nok}}$ and is compositional. We show that every property in SHML is logically equivalent to one in $\mathcal{L}$, i.e., that $\mathcal{L}$ is at least as expressive as SHML. To this end, let φ be a property in SHML. By Thm. 3.6, there is a test T_φ such that $s \models \varphi$ iff s passes the test T_φ, for every state s. Since $\mathcal{L}$ is an extension of $\mathcal{L}_{\mathtt{nok}}$ that is compositional, Propn. 3.13 yields that $\mathcal{L}$ is complete. Thus there is a formula $\psi \in \mathcal{L}$ such that $s \models \psi$ iff s passes the test T_φ, for every state s. It follows that ψ and φ are satisfied by precisely the same states, and are therefore logically equivalent. □

Acknowledgements: We thank Kim Guldstrand Larsen for previous joint work and discussions that gave us the inspiration for this study. The anonymous referees provided useful comments.

References

1. L. Aceto, P. Bouyer, A. Burgueño, and K. G. Larsen, *The power of reachability testing for timed automata*, in Proceedings of the Eighteenth Conference on the Foundations of Software Technology and Theoretical Computer Science, V. Arvind and R. Ramanujam, eds., Lecture Notes in Computer Science, Springer-Verlag, December 1998.
2. L. Aceto, A. Burgueño, and K. G. Larsen, *Model checking via reachability testing for timed automata*, in Proceedings of TACAS '98, Lisbon, B. Steffen, ed., vol. 1384 of Lecture Notes in Computer Science, Springer-Verlag, 1998, pp. 263–280.

3. H. R. Andersen, *Partial model checking (extended abstract)*, in Proceedings of the Tenth Annual IEEE Symposium on Logic in Computer Science, San Diego, California, 26–29 June 1995, IEEE Computer Society Press, pp. 398–407.
4. R. De Nicola and M. Hennessy, *Testing equivalences for processes*, Theoretical Comput. Sci., 34 (1984), pp. 83–133.
5. D. M. Gabbay, A. Pnueli, S. Shelah, and J. Stavi, *On the temporal basis of fairness*, in Conference Record of the Seventh Annual ACM Symposium on Principles of Programming Languages, Las Vegas, Nevada, Jan. 1980, pp. 163–173.
6. M. Hennessy, *Algebraic Theory of Processes*, MIT Press, Cambridge, Massachusetts, 1988.
7. M. Hennessy and R. Milner, *Algebraic laws for nondeterminism and concurrency*, J. Assoc. Comput. Mach., 32 (1985), pp. 137–161.
8. R. Keller, *Formal verification of parallel programs*, Comm. ACM, 19 (1976), pp. 371–384.
9. S. Kleene, *Representation of events in nerve nets and finite automata*, in Automata Studies, C. Shannon and J. McCarthy, eds., Princeton University Press, 1956, pp. 3–41.
10. D. Kozen, *Results on the propositional mu-calculus*, Theoretical Comput. Sci., 27 (1983), pp. 333–354.
11. F. Laroussinie, K. G. Larsen, and C. Weise, *From timed automata to logic - and back*, in Mathematical Foundations of Computer Science 1995, 20th International Symposium, J. Wiedermann and P. Hájek, eds., vol. 969 of Lecture Notes in Computer Science, Prague, Czech Republic, 28 Aug.–1 Sept. 1995, Springer-Verlag, pp. 529–539.
12. K. G. Larsen and L. Xinxin, *Compositionality through an operational semantics of contexts*, Journal of Logic and Computation, 1 (1991), pp. 761–795.
13. R. Milner, *Communication and Concurrency*, Prentice-Hall International, Englewood Cliffs, 1989.
14. D. Park, *Concurrency and automata on infinite sequences*, in 5^{th} GI Conference, Karlsruhe, Germany, P. Deussen, ed., vol. 104 of Lecture Notes in Computer Science, Springer-Verlag, 1981, pp. 167–183.
15. A. Stoughton, *Substitution revisited*, Theoretical Comput. Sci., 59 (1988), pp. 317–325.
16. A. Tarski, *A lattice-theoretical fixpoint theorem and its applications*, Pacific Journal of Mathematics, 5 (1955), pp. 285–309.
17. M. Y. Vardi and P. Wolper, *Reasoning about infinite computations*, Information and Computation, 115 (1994), pp. 1–37.

A Strong Logic Programming View for Static Embedded Implications *

R. Arruabarrena, P. Lucio, and M. Navarro

Dpto de L.S.I., Facultad de Informática, Paseo Manuel de Lardizabal, 1, Apdo 649, 20080-San Sebastián, SPAIN. Tel: +34 (9)43 448000, Fax: +34 (9)43 219306, e-mail: marisa@si.ehu.es.

Abstract. A *strong ($\mathcal{L}$) logic programming language* ([14, 15]) is given by two subclasses of formulas (*programs* and *goals*) of the underlying logic $\mathcal{L}$, provided that: firstly, any program P (viewed as a $\mathcal{L}$-theory) has a canonical model M_P which is initial in the category of all its $\mathcal{L}$-models; secondly, the $\mathcal{L}$-satisfaction of a goal G in M_P is equivalent to the $\mathcal{L}$-derivability of G from P, and finally, there exists an effective (computable) proof-subcalculus of the $\mathcal{L}$-calculus which works out for derivation of goals from programs. In this sense, Horn clauses constitute a strong (first-order) logic programming language. Following the methodology suggested in [15] for designing logic programming languages, an extension of Horn clauses should be made by extending its underlying first-order logic to a richer logic which supports a strong axiomatization of the extended logic programming language. A well-known approach for extending Horn clauses with *embedded implications* is the static scope programming language presented in [8]. In this paper we show that such language can be seen as a strong $\mathcal{FO}^{\supset}$ logic programming language, where $\mathcal{FO}^{\supset}$ is a very natural extension of first-order logic with intuitionistic implication. That is, we present a new characterization of the language in [8] which shows that Horn clauses extended with embedded implications, viewed as $\mathcal{FO}^{\supset}$-theories, preserves all the attractive mathematical and computational properties that Horn clauses satisfy as first-order-theories.

1 Introduction

Horn clause programs are theories in first-order logic (namely $\mathcal{FO}$) whose computation relation (between programs and goals) is equivalent to the following relations of $\mathcal{FO}$: logical consequence, derivability and satisfaction in the least Herbrand model of the program. Moreover, the least Herbrand model of a program is initial in the category of all first-order models of the program and it exactly satisfies the goals which are satisfied in every model in this category. In other words, Horn clauses can be seen as a $\mathcal{FO}$ logic programming language, in the strong sense of [14, 15], because its underlying logic $\mathcal{FO}$ has attractive (model-theoretic) mathematical and (proof-theoretic) computational properties (for programs and goals). This idea was formalized in [14, 15] where the notion of a *strong logic programming language* is defined as a restriction of an underlying logic satisfying good properties. This means, once fixed an underlying logic, setting which subclasses of its formulas correspond to the classes of *programs* and *queries or goals*, respectively. The underlying logic, for these subclasses, must satisfy three properties: mathematical semantics, goal completeness and operational semantics. The mathematical semantics property requires that any program has a canonical

* This work has been partially supported by the CICYT-project TIC95-1016-C02-02.

W. Thomas (Ed.): FOSSACS'99, LNCS 1578, pp. 56-72, 1999.

model, which is initial in the class of all models of the program (seen as a theory in the underlying logic). Goal completeness means that logical satisfaction of goals in the initial model is equivalent to the derivability relation (of the logic) restricted to programs and goals. The operational semantics property means the existence of an effective (computable) proof-subcalculus of the calculus (of the logic) for deriving goals from programs. We believe that this view of axiomatizing a logic programming language inside an underlying logic has many advantages. On one hand, it allows one to separate general logical features from programming language features. On the other hand, a useful way to analyse, compare and integrate different programming features is to axiomatize them into a common underlying logic.

Attempts to extend Horn clause logic programming (e.g. with modules, higher-order, data abstraction, etc.) should be done by preserving (as much as possible) the above-mentioned mathematical and computational properties. Since Horn clause logic is the greatest fragment of $\mathcal{FO}$ admitting initial models, concrete extensions could require to change (by restricting or enriching) the underlying logic $\mathcal{FO}$. Many approaches are concerned with extending Horn clauses with some features for program structuring that can be seen as a form of modularity in logic programming (see for instance [2] for a survey). Some of them consider the extension of Horn clauses with implication goals of the form $D \supset G$, called *blocks*, where D can be seen as a *local set of clauses* (or *module*) for proving the goal G. This approach yields to different extensions of Horn clause programming depending on the given semantics to such blocks. A first basic distinction is between *closed blocks*: G can be proved only using local clauses from D, and *open blocks*: G can be proved using D and also the external environment. Therefore, open blocks require *scope rules* to fix the interplay between the predicate definitions inside a module D and those in the environment. In general, dealing with open blocks, a module can extend the definition of a predicate already defined in the enviroment. Hence, different definitions of the same predicate could have to be considered, depending on the collection of modules corresponding to different goals. There are mainly two scope rules, named *static* and *dynamic*, allowing this kind of extension of predicate definitions. In the dynamic approach the set of modules taking part in the resolution of a goal G can only be determined from the sequence of goals generated until G. However, in the static case this set of modules can be determined (for each goal) statically from the block structure of the program. Different proposals of logic programming languages for open blocks with dynamic scope have been presented and studied in several papers (e.g.[4–6, 16–18]). The static scope approach has been mainly studied in [8, 7]. In [2, 7] both different approaches are compared. Some other works (e.g. [19, 20]) treat open blocks with different scope rules avoiding this kind of predicate extension.

In [16] Miller proves that the proof-theoretic semantics for its dynamic scope programming language is based on intuitionistic logic, and in [2] it is shown that the Miller's canonical model for a program is indeed an intuitionistic model of this program. However, for the static scope programming language introduced in [8], neither first-order logic nor intuitionistic logic can be used for this purpose. Following the methodology suggested in [15] for designing logic programming languages, the extension of Horn clauses with intuitionistic implication should be strongly axiomatized in a logic which integrates $\mathcal{FO}$ and intuitionistic implication. In this paper we introduce a complete logic called $\mathcal{FO}^{\supset}$, which is a very natural extension of $\mathcal{FO}$ with intuitionistic implication. We give a new characterization of the well-known semantics for the static scope programming language presented in [8]. This characterization strongly axiomatizes the

logic programming language inside $\mathcal{FO}^{\supset}$ logic, showing that it satisfies all the desirable properties.
The paper is organized as follows: In Section 2 we introduce the formalization of [14, 15] for a strong logic programming language which is the methodological basis for our work. In Section 3 we give a short introduction to the underlying logic $\mathcal{FO}^{\supset}$ giving the necessary notions and results for the rest of the paper. In Section 4 we develop the $\mathcal{FO}^{\supset}$ strong axiomatization of the static scope programming language. We conclude, in Section 5, by summarizing the presented results and related work.

2 Preliminaries

In this section, we introduce the notions of *logic* and *strong logic programming language*, following [14, 15].
The notion of a *logic* is obtained by combining an *entailment system* (formalizing the proof-theoretical component of a logic) with an *institution* (formalizing the model-theoretical component) such that a *soundness condition* relating provability and satisfaction holds. An *entailment system* is a triple $(\underline{\mathsf{Sign}}, \mathsf{sen}, \vdash)$ with $\underline{\mathsf{Sign}}$ a category of *signatures*, sen a functor associating to each $\Sigma \in \underline{\mathsf{Sign}}$ a set $\mathsf{sen}(\Sigma)$ of Σ-sentences and $\vdash$ a function associating to each $\Sigma \in \underline{\mathsf{Sign}}$ a binary relation $\vdash_\Sigma \subseteq \mathcal{P}(\mathsf{sen}(\Sigma)) \times \mathsf{sen}(\Sigma)$, called Σ-entailment or Σ-derivability, which satisfies the properties of reflexivity, monotonicity, transitivity and $\vdash$-translation (i.e. preservation by signature morphisms). An *institution* is a 4-tuple $(\underline{\mathsf{Sign}}, \mathsf{sen}, \underline{\mathsf{Mod}}, \models)$ with $\underline{\mathsf{Sign}}$ and sen as above; $\underline{\mathsf{Mod}}$ is a functor associating to each $\Sigma \in \underline{\mathsf{Sign}}$ a corresponding category $\underline{\mathsf{Mod}}(\Sigma)$ whose objects are called Σ-structures (or Σ-models) and whose morphisms preserve the interpretation given to signature symbols; and $\models$ is a function associating to each $\Sigma \in \underline{\mathsf{Sign}}$ a binary relation $\models_\Sigma \subseteq \underline{\mathsf{Mod}}(\Sigma) \times \mathsf{sen}(\Sigma)$, called Σ-satisfaction, which satisfies the $\models$-invariance property (i.e. for any $M_2 \in \underline{\mathsf{Mod}}(\Sigma_2)$, $H : \Sigma_1 \to \Sigma_2$, $\varphi \in \mathsf{sen}(\Sigma_1)$: $\underline{\mathsf{Mod}}(H)(M_2) \models_{\Sigma_1} \varphi$ iff $M_2 \models_{\Sigma_2} H(\varphi)$). Given $\Gamma \subseteq \mathsf{sen}(\Sigma)$, $\underline{\mathsf{Mod}}(\Gamma)$ denotes the full subcategory of $\underline{\mathsf{Mod}}(\Sigma)$ determined by the structures $M \in \underline{\mathsf{Mod}}(\Sigma)$ such that $M \models_\Sigma \varphi$ for each $\varphi \in \Gamma$. The satisfaction relation induces a logical consequence relation between sets of sentences and sentences, also denoted $\models$, as follows: $\Gamma \models_\Sigma \varphi$ iff $M \models_\Sigma \varphi$ for each $M \in \underline{\mathsf{Mod}}(\Gamma)$.
A *logic* is given by an entailment system and an institution sharing the same signatures and sentences, such that it holds *soundness* of the derivability relation w.r.t. the logical consequence relation. A *logic* is a 5-tuple $\mathcal{L}=(\underline{\mathsf{Sign}}, \mathsf{sen}, \underline{\mathsf{Mod}}, \vdash, \models)$ such that:

- $(\underline{\mathsf{Sign}}, \mathsf{sen}, \vdash)$ is an entailment system
- $(\underline{\mathsf{Sign}}, \mathsf{sen}, \underline{\mathsf{Mod}}, \models)$ is an institution
- For any $\Sigma \in \underline{\mathsf{Sign}}$, $\Gamma \subseteq \mathsf{sen}(\Sigma)$ and $\varphi \in \mathsf{sen}(\Sigma)$, $\Gamma \vdash_\Sigma \varphi \implies \Gamma \models_\Sigma \varphi$ (*Soundness*).

In addition, there are some other useful properties that a logic could satisfy, like completeness, compactness, etc.

From the axiomatic point of view, a *strong logic programming language* is a 4-tuple $\mathcal{LPL}= (\mathcal{L}, \underline{\mathsf{Sign}}', prog, goal)$ with:

- $\mathcal{L}=(\underline{\mathsf{Sign}}, \mathsf{sen}, \underline{\mathsf{Mod}}, \vdash, \models)$ a logic, namely the *underlying logic* of $\mathcal{LPL}$
- $\underline{\mathsf{Sign}}'$ a subcategory of $\underline{\mathsf{Sign}}$
- $prog$ is a functor associating to each $\Sigma \in \underline{\mathsf{Sign}}'$ a set $prog(\Sigma)$ (of Σ-programs) included in $\mathcal{P}_{fin}(\mathsf{sen}(\Sigma))$
- $goal$ is a functor associating to each $\Sigma \in \underline{\mathsf{Sign}}'$ a set of Σ-goals, $goal(\Sigma) \subseteq \mathsf{sen}(\Sigma)$

such that the following properties are satisfied:

1. *Mathematical Semantics*: Each program $P \in prog(\Sigma)$ has a model M_P which is initial in the category $\underline{\mathsf{Mod}}(P)$ of all models in $\underline{\mathsf{Mod}}(\Sigma)$ satisfying P
2. *Goal Completeness (w.r.t. the initial model)*: For any program $P \in prog(\Sigma)$ and any goal $G \in goal(\Sigma)$, $P \vdash_\Sigma G \iff M_P \models_\Sigma G$
3. *Operational Semantics*: Existence of an effective proof subcalculus for the derivability relation $\vdash_\Sigma$ restricted to $prog(\Sigma) \times goal(\Sigma)$.

3 The Logic $\mathcal{FO}^{\supset}$

In this section we introduce the sound and complete logic $\mathcal{FO}^{\supset}$ which extends classical first-order logic with intuitionistic implication. We present its language, semantical structures, logical consequence relation, derivability relation and some other details which are relevant to understand the rest of the paper. A more detailed presentation of this logic is out of the scope of this paper and it can be found in [11], in particular there it is proved soundness and completeness of $\mathcal{FO}^{\supset}$ logic.

A signature $\Sigma \in \underline{\mathsf{Sign}}$ consists of countable sets FS_Σ of function symbols, and PS_Σ of predicate symbols, with some specific arity for each function and predicate symbol. We also assume a countable set VS_Σ of variable symbols. We denote by T_Σ the set of all well-formed first-order Σ-terms. A term is closed if no variable symbol does occur on it. Well-formed Σ-formulas are built, from atomic ones, using classical connectives $(\neg, \wedge, \vee, \rightarrow)$, intuitionistic implication $(\supset)$, and classical quantifiers $(\forall, \exists)$. Free and bound variables and substitution have the usual definitions. $\mathsf{sen}(\Sigma)$ is the set of Σ-sentences, that is, Σ-formulas with no free variables. We will denote formulas by lowercase Greek letters $\varphi, \psi, \gamma, \chi, \ldots$. The uppercase Greek letters Γ and Φ (probably with sub- and superscripts) will be used as metavariables for *sets* of formulas. Model theory is based on Kripke structures ([21]).

Definition 1. A *Kripke Σ-structure* is a triple $K = (W(K), \preceq, \langle \mathcal{A}_w \rangle_{w \in W(K)})$ where $(W(K), \preceq)$ is a non-empty partially ordered set (of worlds) and each $\mathcal{A}_w$ is a first-order Σ-structure (with universe A_w, over which predicate and function symbols are interpreted) such that for any pair of worlds $v \preceq w$ in $W(K)$:

- $A_v \subseteq A_w$,
- $p^{\mathcal{A}_v} \subseteq p^{\mathcal{A}_w}$, for all $p \in PS_\Sigma$
- $f^{\mathcal{A}_w}(a_1, \ldots, a_n) = f^{\mathcal{A}_v}(a_1, \ldots, a_n)$, for all $a_1, \ldots, a_n \in A_v$ and $f \in FS_\Sigma$. ■

$\underline{\mathsf{Mod}}(\Sigma)$ will denote the category whose objects are Kripke Σ-structures. The morphisms in this category will be given in Definition 6.
We denote by t^w the classical first-order interpretation $t^{\mathcal{A}_w}$ of $t \in T_\Sigma$. Terms interpretation behaves monotonically, that is, for any Kripke-structure K and any pair of worlds $v, w \in W(K)$ such that $v \preceq w$: $t^w = t^v \in A_v \subseteq A_w$. The satisfaction of sentences in worlds is handled by the following *forcing relation*:

Definition 2. Let $K \in \underline{\mathsf{Mod}}(\Sigma)$, the binary *forcing relation* $\Vdash \subseteq W(K) \times \mathsf{sen}(\Sigma)$ is inductively defined as follows:

$w \nVdash_F$
$w \Vdash p(t_1, \ldots, t_n)$ iff $(t_1^w, \ldots, t_n^w) \in p^{\mathcal{A}_w}$
$w \Vdash \neg\varphi$ iff $w \nVdash \varphi$
$w \Vdash \varphi \wedge \psi$ iff $w \Vdash \varphi$ and $w \Vdash \psi$

$w \Vdash \varphi \vee \psi$ iff $w \Vdash \varphi$ or $w \Vdash \psi$
$w \Vdash \varphi \rightarrow \psi$ iff if $w \Vdash \varphi$ then $w \Vdash \psi$
$w \Vdash \varphi \supset \psi$ iff for all $v \in W(K)$ such that $v \succeq w$: if $v \Vdash \varphi$ then $v \Vdash \psi$
$w \Vdash \exists x\varphi$ iff $w \Vdash \varphi(\hat{a}/x)$ for some $a \in A_w$ [1]
$w \Vdash \forall x\varphi$ iff $w \Vdash \varphi(\hat{a}/x)$ for all $a \in A_w$. ■

We will write $w, K \Vdash \varphi$ (instead of $w \Vdash \varphi$) whenever confusion on the structure K may occur. This forcing relation gives a non-intuitionistic semantics to negation, classical implication ($\rightarrow$) and universal quantification; as a consequence, the forcing relation on sentences does not behave monotonically w.r.t. the world ordering. We say that a sentence is *persistent* whenever the forcing relation behaves monotonically for it.

Definition 3. A Σ-sentence φ is *persistent* iff for any $K \in \underline{\mathsf{Mod}}(\Sigma)$ and $w \in W(K)$: if $w \Vdash \varphi$ then $v \Vdash \varphi$ for any $v \in W(K)$ such that $v \succeq w$. ■

Persistent sentences play an important role in the $\mathcal{FO}^{\supset}$-axiomatization of logic programming languages with embedded implications, since there is a subclass of persistent sentences (that can be syntactically delimited) which includes the class of goals.

Proposition 4. *Any atomic sentence is persistent. Any sentence $\varphi \supset \psi$ is persistent. If φ and ψ are persistent sentences, then $\varphi \vee \psi$ and $\varphi \wedge \psi$ are persistent. If $\varphi(\hat{a})$ is a persistent sentence, then $\exists x\varphi$ is persistent.*

Proof. For atoms the property is a trivial consequence of the Kripke structure definition. For intuitionistic implication it is also trivial from forcing relation definition. The other two cases are easily proved, by induction, using the forcing relation definition for $\vee, \wedge$ and $\exists$. ■

The satisfaction relation $\models_{\Sigma} \subseteq \underline{\mathsf{Mod}}(\Sigma) \times \mathsf{sen}(\Sigma)$ requires the sentence to be forced (only) in the minimal worlds of the structure. This satisfaction relation induces the logical consequence relation, denoted by the same symbol $\models_{\Sigma}$.

Definition 5. Let $K \in \underline{\mathsf{Mod}}(\Sigma)$ and $\Gamma \cup \{\varphi\} \subseteq \mathsf{sen}(\Sigma)$. We say that
(a) A world $w \in W(K)$ is *minimal* iff there does not exist $v \in W(K)$ such that $v \preceq w$ and $v \neq w$.
(b) $K \models_{\Sigma} \varphi$ (K satisfies φ) iff $w \Vdash \varphi$ for each minimal world $w \in W(K)$.
(c) $\Gamma \models_{\Sigma} \varphi$ (φ is logical consequence of Γ) iff $K \models_{\Sigma} \Gamma \Rightarrow K \models_{\Sigma} \varphi$, for each $K \in \underline{\mathsf{Mod}}(\Sigma)$. ■

Morphisms in $\underline{\mathsf{Mod}}(\Sigma)$ relate only minimal worlds, with the idea of preserving the satisfaction relation for ground atoms, in the following way:

Definition 6. For $i = 1, 2$, let $K_i = (W(K_i), \preceq_{K_i}, \langle \mathcal{A}^i_w \rangle_{w \in W(K_i)}) \in \underline{\mathsf{Mod}}(\Sigma)$ and let $W(K_i)^{min}$ be the set of minimal worlds in $W(K_i)$. A *morphism* $H : K_1 \rightarrow K_2$ is given by a mapping $\sigma_H : W(K_2)^{min} \rightarrow W(K_1)^{min}$ together with a collection of first-order Σ-homomorphisms $\langle H_w : \mathcal{A}^1_{\sigma_H(w)} \rightarrow \mathcal{A}^2_w \rangle_{w \in W(K_2)^{min}}$. If the mapping σ_H is unique (for instance when K_1 has only one minimal world) then we will identify H directly with its collection of first-order Σ-homomorphisms. ■

[1] The constant symbol $\hat{a}$ stands for the syntactic denotation of a (see e.g.[21]).

Remark 7. We recall that first-order Σ-homomorphisms are mappings that preserve the operations and relations which (respectively) interpret function and predicate symbols. In particular they preserve ground atoms. ■

Actually, the above-defined 4-tuple $\mathcal{FO}^{\supset} = (\underline{\mathsf{Sign}}, \mathsf{sen}, \underline{\mathsf{Mod}}, \models)$ forms an institution. The satisfaction relation is preserved: for each signature morphism $H : \Sigma \to \Sigma'$, each $K' \in \underline{\mathsf{Mod}}(\Sigma')$ and each $\varphi \in \mathsf{sen}(\Sigma)$, it is the case that $\underline{\mathsf{Mod}}(H)(K') \models_{\Sigma} \varphi$ iff $K' \models_{\Sigma'} \mathsf{sen}(H)(\varphi)$, where $\mathsf{sen}(H) : \mathsf{sen}(\Sigma) \to \mathsf{sen}(\Sigma')$ is the translation of sentences induced by H and where $\underline{\mathsf{Mod}}(H) : \underline{\mathsf{Mod}}(\Sigma') \to \underline{\mathsf{Mod}}(\Sigma)$ is the forgetful functor associated to H. This functor applies each Σ'-structure K' into a Σ-structure K with the same ordered set of worlds and it associates each first-order structure ${\mathcal{A}'}_w$ into its forgetful first-order structure $V_H({\mathcal{A}'}_w)$.

Structural Rules

$(Init)\ \Delta \rhd A$ if A is atomic and $A \in \Delta$ $\qquad (_F L)\ \Delta; \Gamma, {}_F; \Delta' \rhd \chi$

$$(Cut)\ \frac{\Delta; \Gamma \rhd \varphi \quad \Delta; \Gamma, \varphi; \Delta' \rhd \chi}{\Delta; \Gamma; \Delta' \rhd \chi} \qquad (R_F)\ \Delta; \Gamma, \varphi, \neg\varphi; \Delta' \rhd {}_F$$

Connective Rules

$$(\neg L)\ \frac{\Delta; \Gamma; \neg\chi \rhd \varphi}{\Delta; \Gamma, \neg\varphi \rhd \chi} \qquad (R\neg)\ \frac{\Delta; \Gamma, \varphi \rhd {}_F}{\Delta; \Gamma \rhd \neg\varphi}$$

$$(\vee L)\ \frac{\Delta; \Gamma, \varphi; \Delta' \rhd \chi \quad \Delta; \Gamma, \psi; \Delta' \rhd \chi}{\Delta; \Gamma, \varphi \vee \psi; \Delta' \rhd \chi} \qquad (R\vee)\ \frac{\Delta \rhd \varphi}{\Delta \rhd \varphi \vee \psi} \quad \frac{\Delta \rhd \psi}{\Delta \rhd \varphi \vee \psi}$$

$$(\wedge L)\ \frac{\Delta; \Gamma, \varphi, \psi; \Delta' \rhd \chi}{\Delta; \Gamma, \varphi \wedge \psi; \Delta' \rhd \chi} \qquad (R\wedge)\ \frac{\Delta \rhd \varphi \quad \Delta \rhd \psi}{\Delta \rhd \varphi \wedge \psi}$$

$$(\to L)\ \frac{\Delta; \Gamma \rhd \varphi \quad \Delta; \Gamma, \psi; \Delta' \rhd \chi}{\Delta; \Gamma, \varphi \to \psi; \Delta' \rhd \chi} \qquad (R\to)\ \frac{\Delta; \Gamma, \varphi \rhd \psi}{\Delta; \Gamma \rhd \varphi \to \psi}$$

$$(\supset L)\ \frac{\Delta; \Gamma; \Delta'; \Gamma' \rhd \varphi \quad \Delta; \Gamma; \Delta'; \Gamma', \psi; \Delta'' \rhd \chi}{\Delta; \Gamma, \varphi \supset \psi; \Delta'; \Gamma'; \Delta'' \rhd \chi} \qquad (R\supset)\ \frac{\Delta; \{\varphi\} \rhd \psi}{\Delta \rhd \varphi \supset \psi}$$

Quantifier Rules

$$(\exists L)\ \frac{\Delta; \Gamma, \varphi(c/x); \Delta' \rhd \chi}{\Delta; \Gamma, \exists x\varphi; \Delta' \rhd \chi} \qquad (R\exists)\ \frac{\Delta \rhd \varphi(t/x)}{\Delta \rhd \exists x\varphi}$$

$$(\forall L)\ \frac{\Delta; \Gamma, \varphi(t/x); \Delta' \rhd \chi}{\Delta; \Gamma, \forall x\varphi; \Delta' \rhd \chi} \qquad (R\forall)\ \frac{\Delta \rhd \varphi(c/x)}{\Delta \rhd \forall x\varphi}$$

Fig. 1. A sound and complete sequent calculus for $\mathcal{FO}^{\supset}$.

We will complete the definition of $\mathcal{FO}^{\supset}$ logic by giving a derivability relation $\vdash_{\Sigma} \subseteq \mathcal{P}(\mathsf{sen}(\Sigma)) \times \mathsf{sen}(\Sigma)$ in terms of *sequent calculus* proofs. The original Gentzen's notion considers sequents $\Gamma \rhd \Phi$ whose *antecedent* Γ and *consequent* Φ are both finite (possibly empty) sequences of formulas. In $\mathcal{FO}^{\supset}$ logic, to deal with classical and intuitionistic implications inside the same logic, it is essential to introduce extra structure in sequent antecedents. That is, to achieve soundness and completeness for $\mathcal{FO}^{\supset}$ logic, we consider sequents consisting of pairs $\Delta \rhd \varphi$ where the antecedent Δ is a (finite) sequence of (finite) sets of formulas, and the consequent φ is (like in intuitionistic logic) a single formula. Uppercase Greek letters Δ, Δ', Δ'', ... will be used as metavariables for *sequences of sets* of formulas. In order to simplify sequent notation: the semicolon sign $(;)$ will represent the infix operation for concatenation of sequences, $\Gamma \cup \{\varphi\}$ will be abbreviated by Γ, φ; and a set Γ will be identified with the sequence consisting of this

unique set. On these bases, we present a sound and complete sequent calculus for the logic $\mathcal{FO}^{\supset}$ in Figure 1 where (in quantifier rules) c stands for a new fresh constant symbol and t stands for a closed term.
Notice that every rule in the calculus of Fig.1 is a natural generalization (to sequences of sets in the antecedent) of some classical first-order sequent rule. Moreover, by viewing the antecedent as a single set of formulas, the rules for both implication connectives would coincide. It is also easy to see that $(R\supset)$ is the unique rule creating a new set in the antecedent.

Definition 8. For any (possibly infinite) set $\Gamma \cup \{\varphi\} \subseteq \mathsf{sen}(\Sigma)$ we say that $\Gamma \vdash_\Sigma \varphi$ iff for some finite $\Gamma' \subseteq \Gamma$ there exists a proof of the sequent $\Gamma' \rhd \varphi$ using the calculus in Figure 1. ■

In general, a *proof* for the sequent $\Delta \rhd \varphi$ is a finite tree constructed using inference rules of the calculus, such that the root is the sequent $\Delta \rhd \varphi$ and whose leaves are labeled with initial sequents (in our case, these are $(Init)$, $(_FL)$, (R_F)). In particular, the antecedent Δ may be a unitary sequence of one finite set Γ. We recall that $\vdash_\Sigma$ is the relation induced (by the calculus in Fig.1) on the set $\mathcal{P}(\mathsf{sen}(\Sigma)) \times \mathsf{sen}(\Sigma)$. It is worthwhile noting that this relation satisfies reflexivity, monotonicity and transitivity, although any rule in the calculus (Fig.1) does not directly correspond with them. Besides, the $\vdash$-translation property is also satisfied. However, the extension to a relation between sequences of sets of formulas and formulas lacks to satisfy the former three properties.

4 The Logic Programming Language $Horn^{\supset}$

In this section we give the strong $\mathcal{FO}^{\supset}$ axiomatization for the static scope programming language introduced in [8]. Its syntax is an extension of the Horn clause language, by adding the intuitionistic implication $\supset$ in goals. We define this language as the following 4-tuple $Horn^{\supset} = (\mathcal{FO}^{\supset}, \underline{\mathsf{Sign}}', prog, goal)$, where $\underline{\mathsf{Sign}}'$ is the class of finite signatures in $\underline{\mathsf{Sign}}$ and, for each Σ in $\underline{\mathsf{Sign}}'$, $prog(\Sigma)$ is the set of all Σ-programs, which are finite sets of closed D-clauses (called Σ-*clauses*), and $goal(\Sigma)$ is the set of all closed G-clauses (called Σ-*goals*). D- and G-clauses are recursively defined as follows (where A stands for an atomic formula):

$$G := A \mid G_1 \wedge G_2 \mid D \supset G \mid \exists x G \qquad\qquad D := A \mid G \to A \mid D_1 \wedge D_2 \mid \forall x D$$

Following [8], we use a simple definition of the operational semantics of $Horn^{\supset}$, given by a nondeterministic set of rules which define when a Σ-goal G is operationally derivable from a program sequence $\Delta = P_0;\ldots;P_n$, in symbols $\Delta \vdash_s G$. Moreover, to deal with clauses in $P \in prog(\Sigma)$ of the form $D_1 \wedge D_2$ and $\forall x D$, we utilize the closure (w.r.t. conjunction and instantiation) set $[P]$ of all clauses in P. This abstract definition of the operational semantics is more suitable to be compared with the mathematical semantics of $Horn^{\supset}$.

Definition 9. $[P]$ is defined as the set $\cup\{[D] \mid D \in P\}$ where $[D]$ is recursively defined as follows: $[A] = \{A\}$, $[G \to A] = \{G \to A\}$, $[D_1 \wedge D_2] = [D_1] \cup [D_2]$, $[\forall x D] = \cup\{[D(t/x)] \mid t \in T_\Sigma \text{ and } t \text{ is closed}\}$. ■

(1) $\Delta \vdash_s A$ if A is atomic and $A \in [\Delta]$

(2) $\dfrac{P_0;\ldots;P_i \vdash_s G}{P_0;\ldots;P_i;\ldots;P_n \vdash_s A}$ if $G \to A \in [P_i]$ and $0 \leq i \leq n$

(3) $\dfrac{\Delta \vdash_s G_1 \quad \Delta \vdash_s G_2}{\Delta \vdash_s G_1 \wedge G_2}$ (4) $\dfrac{\Delta \vdash_s G(t/x)}{\Delta \vdash_s \exists x G}$ (5) $\dfrac{\Delta;\{D\} \vdash_s G}{\Delta \vdash_s D \supset G}$

Fig. 2. Operational Semantics for $Horn^{\supset}$.

Notice that $w \Vdash P \Leftrightarrow w \Vdash [P]$ and also that all clauses in $[P]$ match the pattern $G \to A$ (with G possibly empty for handling the case A). We extend the notation $[P]$ to $[\Delta]$ by $[P_0;\ldots;P_n] = \bigcup_{i=0}^{n}[P_i]$. Now, we define $\Delta \vdash_s G$ by means of the rules given in Figure 2. In order to illustrate the operational behaviour of this language we give the Example 10.

Example 10. Let the program with two clauses $P = \{((b \to c) \supset c) \to a,\ b\}$ and let the goal $G1 = a$. A proof of $P \vdash_s G1$ is given by the following steps (applying rules in Figure 2):

$$\begin{array}{rl} P \vdash_s a & \text{by Rule (2)} \\ \text{if } P \vdash_s (b \to c) \supset c & \text{by Rule (5)} \\ \text{if } P;\{b \to c\} \vdash_s c & \text{by Rule (2)} \\ \text{if } P;\{b \to c\} \vdash_s b & \text{by Rule (1) since } b \in P;\{b \to c\} \end{array}$$

However, let now the program with a unique clause $Q = \{((b \to c) \supset c) \to a\}$ and let the goal $G2 = b \supset a$. The only way to obtain a proof of $Q \vdash_s G2$ would make the following steps:

$$\begin{array}{rl} Q \vdash_s b \supset a & \text{by Rule (5)} \\ \text{if } Q;\{b\} \vdash_s a & \text{by Rule (2)} \\ \text{if } Q \vdash_s (b \to c) \supset c & \text{by Rule (5)} \\ \text{if } Q;\{b \to c\} \vdash_s c & \text{by Rule (2)} \\ \text{if } Q;\{b \to c\} \vdash_s b & \end{array}$$

Since the last sequent can not be proved then $Q \nvdash_s G2$. ■

This example shows the "static scope rule" meaning: the set of clauses which can be used to solve a goal depends on the program block's structure. Whereas $G1 = a$ can be proved from the program P because b was defined in P, in the case of $G2 = b \supset a$ and the program Q the "external" definition of b is not permitted for proving the body of the clause in Q. This is a mayor difference with the "dynamic scope rule" used in [16].

In the Appendix A we prove that the proof-subcalculus $\vdash_s$ is sound with respect to the $\mathcal{FO}^{\supset}$-calculus when restricted to the programming language $Horn^{\supset}$.

In the rest of this section we show that $Horn^{\supset}$ satisfies all the desirable properties to be a strong $\mathcal{FO}^{\supset}$ logic programming language. In Subsection 4.1 we present the mathematical (or model) semantics and we prove the goal completeness property. The operational semantics is studied in Subsection 4.2, showing the equivalence between mathematical and operational semantics. Also completeness of $\vdash_s$ w.r.t. the $\mathcal{FO}^{\supset}$-calculus will be proved there as a consequence of previous results. Along the whole section $\models$ (respectively $\vdash$) stands for the satisfaction and the logical consequence relations $\models_{\Sigma}$ (respectively the derivability relation $\vdash_{\Sigma}$) of $\mathcal{FO}^{\supset}$.

4.1 Mathematical Semantics and Goal Completeness

In this subsection we first define the subcategory $\underline{\mathsf{FMod}}(\Sigma)$ of $\underline{\mathsf{Mod}}(\Sigma)$. Its objects are Kripke structures with Herbrand interpretations associated to worlds, with a unique minimal world and closed w.r.t. superset. Then, we show that to deal with $Horn^{\supset}$ programs (as particular $\mathcal{FO}^{\supset}$-theories) the category $\underline{\mathsf{Mod}}(P)$ of Kripke Σ-structures satisfying P can be restricted to the subcategory $\underline{\mathsf{FMod}}(P)$. Notice that, for Horn clauses, the Herbrand models constitute the corresponding subcategory of the general first-order structures. We will prove the existence of a model in $\underline{\mathsf{FMod}}(P)$ which is initial in the whole category $\underline{\mathsf{Mod}}(P)$. Again, one can observe the parallelism with the least Herbrand model of Horn clauses. Finally, we will prove the goal completeness property w.r.t. this initial model.

Given a signature Σ, U_Σ and B_Σ will denote the Herbrand universe and the Herbrand base, respectively. Consider the complete lattice $\mathcal{P}(B_\Sigma)$ of all Herbrand (first-order) Σ-interpretations over the universe U_Σ. Any subset K of $\mathcal{P}(B_\Sigma)$, ordered by set inclusion, can be viewed as a Kripke Σ-structure. On these structures, $I, K \Vdash \varphi$ (or simply $I \Vdash \varphi$) will denote $w, K \Vdash \varphi$ for the world w whose first-order associated Σ-structure is I.

Definition 11. $\underline{\mathsf{FMod}}(\Sigma)$ is the full subcategory of $\underline{\mathsf{Mod}}(\Sigma)$ whose objects are the Kripke Σ-structures $\{Fil(I) \mid I \subseteq B_\Sigma\}$ where $Fil(I)$ denotes the *filter* $\{J \subseteq B_\Sigma \mid J \supseteq I\}$. $(\underline{\mathsf{FMod}}(\Sigma), \sqsubseteq)$ is the partial order given by $Fil(I_1) \sqsubseteq Fil(I_2)$ iff $I_1 \subseteq I_2$. The morphisms in $\underline{\mathsf{FMod}}(\Sigma)$ can be seen as these inclusions, that is $Fil(I_1) \sqsubseteq Fil(I_2)$ is the morphism $H \in \underline{\mathsf{Mod}}(\Sigma)$ defined by $\sigma_H(I_2) = I_1$ and the singleton $\{\subseteq : I_1 \to I_2\}$. ■

Remark 12. Note that the morphisms $H : K_1 \to K_2$ with $K_1 \in \underline{\mathsf{FMod}}(\Sigma)$ are unique since: (i) K_1 has only one minimal world and (ii) if A and B are first-order Σ-structures and A is finitely generated then the Σ-homomorphism $A \to B$ is unique. ■

Hence, for formulas $\varphi \supset \psi$, the forcing relation restricted to the class $\underline{\mathsf{FMod}}(\Sigma)$ satisfies: $I \Vdash \varphi \supset \psi$ iff for all $J \subseteq B_\Sigma$ such that $I \subseteq J$, if $J \Vdash \varphi$ then $J \Vdash \psi$.

Proposition 13. *Let I_1,I_2 be two Σ-interpretations, $\{I_j\}_{j \in J}$ a (possibly infinite) set of Σ-interpretations, D a Σ-clause and G a Σ-goal.*
(a) *If $I_1 \Vdash G$ then for all I_2 such that $I_1 \subseteq I_2$, $I_2 \Vdash G$*
(b) *If $I_j \Vdash D$ for each $j \in J$ then $\cap_j I_j \Vdash D$*

Proof. (a) is a direct consequence of persistence of goals (see Proposition 4). The proof of (b) can be made by structural induction on D: For $D = A$ it is trivial, since $I \Vdash A$ iff $A \in I$. Cases $D = D_1 \wedge D_2$ and $D = \forall x D_1$ can be easily proved by applying the induction hypothesis. For $D = G \to A$, the case $\cap_j I_j \Vdash A$ is trivial. Now suppose that $\cap_j I_j \nVdash A$, then there exists $j \in J$ such that $I_j \nVdash A$ and $I_j \nVdash G$. Hence $\cap_j I_j \nVdash G$ holds by (a), and therefore $\cap_j I_j \Vdash G \to A$. ■

Proposition 14. $(\underline{\mathsf{FMod}}(\Sigma), \sqsubseteq)$ *is a complete lattice with bottom $Fil(\emptyset) = \mathcal{P}(B_\Sigma)$.*

Proof. It is enough to define the operations $\sqcup$ and $\sqcap$ for any (possibly infinite) collection $\{Fil(I_i)\}_i$ as follows: $\sqcup_i Fil(I_i) = Fil(\cup_i I_i)$ and $\sqcap_i Fil(I_i) = Fil(\cap_i I_i)$. ■

The notion of satisfaction between elements in $\underline{\mathsf{FMod}}(\Sigma)$ and Σ-clauses (respectively -goals), borrowed from the underlying logic, is given by $Fil(I) \models D$ iff $I \Vdash D$ (respectively for G).
The class of models of a Σ-program P, denoted $\underline{\mathsf{FMod}}(P)$, is defined as $\underline{\mathsf{FMod}}(P) = \{K \in \underline{\mathsf{FMod}}(\Sigma) \mid K \models P\}$ or equivalently as $\{Fil(I) \mid I \subseteq B_\Sigma, I \Vdash P\}$. $\underline{\mathsf{FMod}}(P)$ is a full subcategory of $\underline{\mathsf{FMod}}(\Sigma)$.

Proposition 15. *There exists a least element M_P in $\underline{\mathsf{FMod}}(P)$ with respect to $\sqsubseteq$.*

Proof. $\underline{\mathsf{FMod}}(P)$ is not empty since $Fil(B_\Sigma) = \{B_\Sigma\}$ satisfies P. As a consequence of Proposition 13(b), the intersection ($\sqcap$) of elements in $\underline{\mathsf{FMod}}(P)$ is an element of $\underline{\mathsf{FMod}}(P)$. Then $M_P = \sqcap\{K \in \underline{\mathsf{FMod}}(\Sigma) \mid K \models P\}$ belongs to $\underline{\mathsf{FMod}}(P)$ and it is the least element w.r.t. $\sqsubseteq$. Moreover, $M_P = Fil(I_P)$ with $I_P = \cap\{I \subseteq B_\Sigma \mid I \Vdash P\}$. ■

Then, M_P is the initial object in the category $\underline{\mathsf{FMod}}(P)$. Now, we will prove the initiality of M_P in the (more general) category $\underline{\mathsf{Mod}}(P)$. Then, following [15], the denotation function $P \mapsto M_P$ is called the *mathematical semantics* of $Horn^{\supset}$.

Definition 16. A Σ-program P is *satisfiable* (respectively *F-satisfiable*) iff there exists $K \in \underline{\mathsf{Mod}}(\Sigma)$ (respectively $K \in \underline{\mathsf{FMod}}(\Sigma)$) such that $K \models P$. ■

Lemma 17. *For each $K \in \underline{\mathsf{Mod}}(\Sigma)$ there exists $I_K \in \mathcal{P}(B_\Sigma)$ (therefore $Fil(I_K) \in \underline{\mathsf{FMod}}(\Sigma)$) such that, for every Σ-clause D and every Σ-goal G:*
(a) *If $K \models D$ then $Fil(I_K) \models D$*
(b) *If $Fil(I_K) \models G$ then $K \models G$*
Moreover, there exists a unique morphism $H_K : Fil(I_K) \to K$.

Proof. Let $K = (W(K), \preceq, \langle \mathcal{A}_w \rangle_{w \in W(K)})$. We consider, for each $w \in W(K)$, the H-interpretation $I_w = \{p(t_1, ..., t_n) \in B_\Sigma \mid w, K \Vdash p(t_1, ..., t_n)\}$ and let $I_K = \cap\{I_w \mid w \in W(K)\}$. That is, $I_K = \{p(t_1, ..., t_n) \in B_\Sigma \mid K \models p(t_1, ..., t_n)\}$. Then, for each Σ-clause D and each Σ-goal G:

(i) If $w, K \Vdash D$ then $I_w \Vdash D$
(ii) If $I_w \Vdash G$ then $w, K \Vdash G$

The proof of above facts (i) and (ii) is made by simultaneous induction on D and G. (i) and (ii) for an atom A: $w, K \Vdash A$ iff $A \in I_w$ iff $I_w \Vdash A$. (i) for $D_1 \wedge D_2$, $\forall x D$ and (ii) for $G_1 \wedge G_2$, $\exists x G$, can be easily proved by applying the induction hypothesis. To prove (i) for $G \to A$, let us suppose that $w, K \Vdash G \to A$, then $w, K \Vdash A$ or $w, K \not\Vdash G$. By the induction hypothesis, $I_w \Vdash A$ or $I_w \not\Vdash G$ holds. Therefore $I_w \Vdash G \to A$. To prove (ii) for $D \supset G$, suppose that $w, K \not\Vdash D \supset G$, then there exists $v \in W(K)$ such that $w \preceq v$, $v, K \Vdash D$ and $v, K \not\Vdash G$. By induction, $I_v \Vdash D$ and $I_v \not\Vdash G$ hold. Then $I_w \not\Vdash D \supset G$, since $w \preceq v$ implies $I_w \subseteq I_v$.
Now, to prove (a), let us suppose that $K \models D$, then for all minimal $w \in W(K)$: $w, K \Vdash D$. Hence, by (i), for all minimal $w \in W(K)$: $I_w \Vdash D$. Then, by Proposition 13(b), $I_K \Vdash D$ holds. Therefore $Fil(I_K) \models D$. The proof for (b) is symmetric, suppose that $Fil(I_K) \models G$, this means that $I_K \Vdash G$. Then, by Proposition 13(a), $I_w \Vdash G$ holds for all minimal $w \in W(K)$. Therefore by (ii), $w, K \Vdash G$ for all minimal $w \in W(K)$. Hence $K \models G$.
The unique morphism $H_K : Fil(I_K) \to K$ is given by the collection of unique first-order Σ-homomorphisms $\{H_w : I_K \to \mathcal{A}_w \mid w \text{ minimal in } W(K)\}$. ■

Theorem 18. *M_P is initial in the category $\underline{\mathsf{Mod}}(P)$.*

Proof. Given $K \in \underline{\mathsf{Mod}}(P)$, the unique morphism from M_P into K is $H = H_K \circ \sqsubseteq$ obtained by composing the two morphisms $\sqsubseteq$: $M_P \to Fil(I_K)$ and $H_K : Fil(I_K) \to K$ of the previous lemma. ■

Corollary 19. *A Σ-program P is satisfiable iff it is F-satisfiable.* ■

Now, we will show that M_P is typical in $\underline{\mathsf{Mod}}(P)$ (and also in $\underline{\mathsf{FMod}}(P)$) w.r.t. goal satisfaction.

Proposition 20. *For each Σ-program P and each Σ-goal G: $P \models G$ iff $Fil(I) \models G$ for all $Fil(I) \in \underline{\mathsf{FMod}}(P)$.*

Proof. The only-if part is trivial. For the if part let $K \in \underline{\mathsf{Mod}}(P)$, that means $K \models P$. Then by Lemma 17 $Fil(I_K) \models P$. Then $Fil(I_K) \models G$ and, again by Lemma 17, $K \models G$. ■

Theorem 21. *For each Σ-program P and each Σ-goal G: $P \models G$ iff $M_P \models G$.*

Proof. The only-if part is trivial. Conversely, $M_P \models G$ is equivalent to $\cap\{I \subseteq B(\Sigma) \mid I \Vdash P\} \Vdash G$. Therefore $I \Vdash G$ for all $I \subseteq B(\Sigma)$ such that $I \Vdash P$, hence $Fil(I) \models G$ for all $Fil(I) \in \underline{\mathsf{FMod}}(P)$. Then by Proposition 20, $P \models G$. ■

From this result and the fact of that $\mathcal{FO}^{\supset}$ is a complete logic, the goal completeness property is obtained:

Theorem 22. *For each Σ-program P and each Σ-goal G, $P \vdash G$ iff $M_P \models G$.* ■

Remark 23. It is worthwhile noting that: $Fil(I) \models G$ (or $I \Vdash G$) iff G is logical consequence of I. This can be proved by Proposition 20, by seeing I as a (posibly infinite) program of ground atoms, and by persistency of G. ■

4.2 Operational Semantics

In this subsection we first define, for each Σ-program P, an immediate consequence operator T_P (on $\underline{\mathsf{FMod}}(\Sigma)$). The monotonicity and continuity of T_P in the lattice $(\underline{\mathsf{FMod}}(\Sigma), \sqsubseteq)$ allow us to use the fixpoint semantics as a bridge between the mathematical and the operational semantics. First we prove the equivalence between mathematical and fixpoint semantics and then between fixpoint and operational semantics (given by $\vdash_s$). Specifically, given a Σ-program P, we will use the fixpoint characterization of the least model M_P of P in terms of T_P, to prove that for every Σ-goal G, $M_P \models G$ if and only if $P \vdash_s G$. We will also show that the proof-subcalculus $\vdash_s$ is sound and complete with respect to the $\mathcal{FO}^{\supset}$ derivability relation, restricted to $Horn^{\supset}$-programs and -goals.

Definition 24. The immediate consequence operator $T_P : \underline{\mathsf{FMod}}(\Sigma) \to \underline{\mathsf{FMod}}(\Sigma)$ is given by $T_P(Fil(I)) = Fil(\{A \mid \text{there exists } G \to A \in [P] \text{ such that } Fil(I) \models G\})$. ■

The operator T_P has been defined in terms of the satisfaction relation of $\mathcal{FO}^{\supset}$. That is, given a filter (generated by a set of ground atoms), it generates the head of the clauses whose bodies are satisfied by this filter. We want to remark that T_P is indeed a $\mathcal{FO}^{\supset}$ logical consequence operator because we can replace (see Remark 23) the satisfaction of G in the model $Fil(I)$ (or equivalently the forcing relation of G in the minimal world I) by the logical consequence of G from I. Unlikely for Horn clauses, logical consequence can not be replaced by set membership since goals are not just conjunction of atoms. It is well-known that the least fixpoint and the least pre-fixpoint of a continuous operator in a complete lattice is $T^{\omega}(\bot)$ where $\bot$ is the bottom in the lattice. In the Appendix B we prove that the above-defined operator T_P is monotone and continuous in the complete lattice $(\underline{\mathsf{FMod}}(\Sigma), \sqsubseteq)$ and also that the models of P are the pre-fixpoints of T_P . Therefore, the least fixpoint of T_P is $T_P^w(\mathcal{P}(B_\Sigma))$ which will be simply denoted T_P^w. Then the correspondence between mathematical and fixpoint semantics is a direct consequence of these results.

Theorem 25. *For all Σ-program P, $T_P^w = M_P$.* ■

Now we will prove the equivalence between mathematical, fixpoint and operational semantics. We need the following lemma to complete such equivalences. This result was proved in [8] and our proof is an adaptation (for our operator T_P) of the proof given there. For that reason we will give a sketch of this proof detailing only the main differences.

Lemma 26. *Given a Σ-program P and a Σ-goal G, if $T_P^w \models G$ then $P \vdash_s G$.*

Sketch of the proof. Let I_n denote the minimal world in $T_P^n(\mathcal{P}(B_\Sigma))$, for each $n \geq 0$. Since $T_P^w = \sqcup_{n<w} T_P^n(\mathcal{P}(B_\Sigma))$, the minimal world in T_P^w is $\cup_{n<w} I_n$. Then by continuity of T_P it suffices to prove that $I_n \Vdash G \Longrightarrow P \vdash_s G$ holds for each $n \geq 0$. The proof is made by induction on the highest number m of $(\supset)$-nesting levels in P and G. If $m = 0$ (there are no occurrences of $\supset$ either in P or in G), then the proof can be done by double induction on n and G. The induction hypothesis holds for at most $m-1$ $(\supset)$-nesting levels in P and G. For the case $m > 0$, let us develop in detail only the subcase $n > 0$ and $G = D_1 \supset G_1$. Let D_1' be the program $I_n \cup D_1$ (seen the atoms in I_n as clauses with empty bodies) and let $I_{D_1'}$ be the minimal world in $T_{D_1'}^{\omega}$. Then $I_{D_1'} \Vdash D_1$ and $I_n \subseteq I_{D_1'}$. Therefore $I_{D_1'} \Vdash G_1$. By induction on D_1' and G_1 (note that the highest number of $(\supset)$-nesting levels in D_1' and G_1 is less than m), $I_n \cup D_1 \vdash_s G_1$ holds. Finally, some $\vdash_s$-properties easy to prove (see [8] for details) are used to obtain the following implications: $I_n \cup D_1 \vdash_s G_1 \Longrightarrow I_n; D_1 \vdash_s G_1 \Longrightarrow I_n \vdash_s (D_1 \supset G_1) \Longrightarrow \{A \mid P \vdash_s A\} \vdash_s (D_1 \supset G_1) \Longrightarrow P \vdash_s (D_1 \supset G_1)$. ■

The following Theorem summarizes all the obtained results. In particular, the equivalence between mathematical and operational semantics is given by $(c) \Leftrightarrow (e)$.

Theorem 27. *For each Σ-program P and each Σ-goal G, the following sentences are equivalent:*

(a) $P \models G$
(b) $P \vdash G$
(c) $M_P \models G$
(d) $T_P^w \models G$
(e) $P \vdash_s G$

Proof. $(a) \Leftrightarrow (b)$ by the soundness and completeness of $\mathcal{FO}^{\supset}$
$(b) \Leftrightarrow (c)$ by the goal completeness property (Th. 22)
$(c) \Leftrightarrow (d)$ by the equivalence between mathematical and fixpoint semantics (Th. 25)
$(d) \Rightarrow (e)$ by the Lemma 26
$(e) \Rightarrow (b)$ by the soundness of $\vdash_s$ w.r.t. $\vdash$ (Th. 30 in Appendix A) ■

Corollary 28. *The proof-subcalculus $\vdash_s$ is complete with respect to the $\mathcal{FO}^{\supset}$-calculus when restricted to the programming language $Horn^{\supset}$.*

We have used an abstract formulation of the operational semantics, given by the proof-subcalculus $\vdash_s$. The effectiveness of such subcalculus means the capability for implementing it. This task is out of the scope of this paper, however we would like to mention here some works giving the main ideas towards such implementation. In [8] a less abstract operational semantics is given by using notions of substitution, unification and variable renaming for the notation $[P]$. Whereas this semantics is equivalent to the given one, it provides an abstract interpreter for the language $Horn^{\supset}$. In [1] are also shown, by means of examples, some of the most relevant points taken into account to make a concrete implementation.

5 Conclusions and Related Work

We have presented a new characterization for the language $Horn^{\supset}$ of Horn clauses extended with static embedded implication (introduced in [8]). Our characterization is based on the methodology proposed in [14, 15] for define logic programming languages. Hence, we have enriched the underlying logic ($\mathcal{FO}$) of the original language (Horn clauses) with intuitionistic implication, in a very natural way, obtaining the complete logic $\mathcal{FO}^{\supset}$. Then we have given a $\mathcal{FO}^{\supset}$-axiomatization of $Horn^{\supset}$, showing that it satisfies all the desirable mathematical and computational properties. The fact of fixing the underlying logic $\mathcal{FO}^{\supset}$ allows us to deal with $Horn^{\supset}$-programs as special $\mathcal{FO}^{\supset}$-theories. Therefore, metalogical properties of programs and goals can be studied in a clean and sound way relative to fixed notions (as model, satisfaction, morphism, derivability, etc.) in the underlying framework. Following this methodology, we have obtained a subclass ($\underline{\mathsf{FMod}}(\Sigma)$) of logical structures powerful enough for dealing with $Horn^{\supset}$-programs, like the subclass of Herbrand interpretations is for Horn clauses in the first-order case. Indeed, we show that a program (as a theory) has a (general) model iff it has a model in the subclass $\underline{\mathsf{FMod}}(\Sigma)$. We believe that this is an important result about the model-theoretic semantics of $Horn^{\supset}$. Actually, the equivalence between the two model-theoretic semantics presented in [8] is a direct consequence of the definition of $\underline{\mathsf{FMod}}(\Sigma)$. Moreover, $\underline{\mathsf{FMod}}(\Sigma)$ is crucial for both: the initial and the fixpoint semantics. On one hand, for any program P, $\underline{\mathsf{FMod}}(P)$ has a least element M_P which can be obtained by intersection of all models of P and also as the ω-iteration of a continuous immediate consequence operator T_P defined on $\underline{\mathsf{FMod}}(\Sigma)$. Our fixpoint semantics is essentially equivalent to the fixpoint semantics of [8], although it is obtained in a very different way. As we pointed out in Subsection 4.2, the operator T_P is indeed based on the logical consequence of the underlying logic (or equivalently on its satisfaction relation). However, the immediate consequence operator of [8] is based on the notion of environment and it requires an ad-hoc satisfaction relation between Herbrand interpretations and goals. Moreover, we prove that the operational semantics of $Horn^{\supset}$ is equivalent to the underlying logical derivability relation. In fact, this derivability

relation is induced by a calculus (Figure 1) designed as an extension of the operational semantics of $Horn^{\supset}$. On the other hand, we have showed that $Horn^{\supset}$-programs are $\mathcal{FO}^{\supset}$-theories with initial semantics: M_P (or equivalently T_P^{ω}) is the initial object in the class $\underline{\mathsf{Mod}}(P)$ of all (general) models of the program P. Hence, our characterization of $Horn^{\supset}$, firstly, places some well-known results into the logical framework given by $\mathcal{FO}^{\supset}$ and, secondly, it extends these results to a strong axiomatization providing a well-established model-theoretic semantics and an initial semantics.

We believe that further extensions of this logic programming language, for example with some kind of negation, could be better developed using the logical foundation provided by this strong $\mathcal{FO}^{\supset}$-axiomatization. With respect to this matter, there are several papers dealing with dynamic intuitionistic implication and some kind of negation, e.g. [3, 5, 9, 10, 12, 13]. We plan to investigate also a possible $\mathcal{FO}^{\supset}$-axiomatization of the dynamic scope language of [16] in order to place both languages (from [8] and [16]) into the common underlying logic $\mathcal{FO}^{\supset}$. $\mathcal{FO}^{\supset}$ and intuitionistic logic are essentially equivalent to deal with the latter language. We mean, although these two logics differ in the universal quantifier interpretation, both coincide in clause interpretation over structures with constant universe, and it is well-known (cf. [2, 7]) that these structures are powerful enough. In [16] it is proved that the operational semantics of its language corresponds to intuitionistic derivability. In [2] it is shown that the canonical model (of a program), obtained in [16] by a fixpoint construction, is indeed an intuitionistic model of the program. They also give an intuitionistic (Kripke's based) model-theory for this language. Apart from the difference in the considered programming language, there are three most remarkable differences with our Kripke's based approach: their logical structures are generated by terms, our notions of satisfaction and logical consequence are different, and the worlds of their canonical model are indexed by programs.

A different approach to give logical foundations to this kind of logic programming languages (or in general to Horn clause extensions) is the transformational one which consists in translating programs to the language of some well-known logic. In [7] the language defined in [8] is translated to $S4$-modal logic. They also translate the language defined in [16] in order to set both languages into a common logical framework.

The transformational approach is also taken in [19, 20] where logic programs with embedded implications are translated to Horn clause programs. In [20] the definition of a predicate in a new module overrides its definition in previous modules, therefore nested definitions are independent of definitions in outer modules. The semantics of such languages can be defined by a direct mapping from programs in the extended language to Horn clause programs. Then, Horn clause theory can be used to give logical and computational foundation to the extended language. However, as it is pointed in [2, 20], when predicate extension is allowed, the translation of each predicate definition (inside a module) raises different predicate definitions, each one depending on the collection of modules that have to be used. In dynamic scoped languages this collection can only be determined in run-time, forcing to add new arguments to the translated predicates to represent the modules currently in use. This makes the transformational approach inadequate for both semantics and implementation issues. For static scoped languages, such as the language studied in this paper, this approach could be still useful for implementation issues, since there is a lexical way to determine such collection of modules (for each goal). However, the translation would not be so direct because of the multiple transformation of each original predicate. Therefore, in our opinion, for semantical foundation it is more adequate the model-theoretic approach started in [8], whose results we have enriched by setting a well-stablished logical framework.

Acknowledgment: The authors are greatly indebted to Fernando Orejas for fruitful discussions and suggestions.

References

1. Arruabarrena, R. and Navarro, M. On Extended Logic Languages Supporting Program Structuring, In: *Proc. of APPIA-GULP-PRODE'96*, 191-203, (1996).
2. Bugliesi, M., Lamma, E. and Mello, P., Modularity in Logic Programming, *Journal of Logic Programming*, (19-20): 443-502, (1994).
3. Bonner, A. J., and McCarty, L. T., Adding Negation-as-Failure to Intuitionistic Logic Programming, In: *Proc. of the North American Conf. on Logic Programming*, MIT Press, 681-703, (1990).
4. Bonner, A. J., McCarty, L. T., and Vadaparty, K., Expresing Database Queries with Intuitionistic Logic. In: *Proc. of the North American Conf. on Logic Programming*, MIT Press, 831-850, (1989).
5. Gabbay, D. M., N-Prolog: An Extension of Prolog with Hypothetical Implications. II. Logical Foundations and Negation as Failure, *Journal of Logic Programming* 2(4):251-283 (1985).
6. Gabbay, D. M. and Reyle, U., N-Prolog: An Extension of Prolog with Hypothetical Implications. I., *Journal of Logic Programming* 1(4):319-355 (1984).
7. Giordano, L., and Martelli, A.; Structuring Logic Programs: A Modal Approach, *Journal of Logic Programming* 21:59-94 (1994).
8. Giordano, L., Martelli, A., and Rossi, G., Extending Horn Clause Logic with Implication Goals, *Theoretical Computer Sscience*, 95:43-74, (1992).
9. Giordano, L., and Olivetti, N.; Combining Negation as Failure and Embedded Implications in Logic Programs, *Journal of Logic Programming* 36:91-147 (1998).
10. Harland., J. Succecs and Failure for Hereditary Harrop Formulae, *Journal of Logic Programming*, 17:1-29, (1993).
11. Lucio, P. $\mathcal{FO}^{\supset}$: A Complete Extension of First-order Logic with Intuitionistic Implication, Technical Research Report UPV-EHU/LSI/TR-6-98, URL address: http://www.sc.ehu.es/paqui, Submitted to a journal for publication.
12. McCarty, L. T., Clausal Intuitionistic Logic I. Fixed-Point Semantics, *Journal of Logic Programming*, 5:1-31, (1988).
13. McCarty, L. T., Clausal Intuitionistic Logic II. Tableau Proof Procedures, *Journal of Logic Programming*, 5:93-132, (1988).
14. Meseguer, J., General Logics, In: Ebbinghaus H.-D. et al. (eds), *Logic Colloquium'87*, North-Holland, 275-329, (1989).
15. Meseguer, J., Multiparadigm Logic Programming, In: *Proccedings of ALP'92*, L.N.C.S. 632. Springer-Verlag, 158-200, (1992).
16. Miller, D., A Logical Analysis of Modules in Logic Programming, In: *Journal of Logic Programming*, 6:79-108, (1989).
17. Miller, D., Abstraction in Logic Programs. In: Odifreddi, P. (ed), *Logic and Computer Science*, Academic Press, 329-359, (1990).
18. Miller, D., Nadathur, G., Pfenning, F. and Scedrov, A., Uniform Proofs as a Foundation for Logic Programming, *Annals of Pure and App. Logic*, 51:125-157, (1991).
19. Monteiro, L., Porto, A., Contextual Logic Programmming, In: *Proc. 6th International Conf. on Logic Programming*, 284-299, (1989).
20. Moscowitz, Y., and Shapiro, E., Lexical logic programs, In: *Proc. 8th International Conf. on Logic Programming*, 349-363, (1991).
21. van Dalen, D., and Troelstra, *Constructivism in Mathematics: An Introduction* Vol.1 and Vol.2, Elsevier Science, North-Holland, (1988).

A Appendix: Soundness of the proof-subcalculus

We prove here that the proof-subcalculus $\vdash_s$ is sound with respect to the $\mathcal{FO}^{\supset}$-calculus when restricted to the programming language $Horn^{\supset}$. In the following, the rules in the $\mathcal{FO}^{\supset}$-calculus and the rules in $\vdash_s$ will be respectively called the *logical* and *operational* rules.

Lemma 29. *Let Δ be a sequence of Σ-programs $P_0; ...; P_n$ $(n \geq 0)$ and G be a Σ-goal. If $\Delta \vdash_s G$ then $\Delta \vdash G$.*

Proof. Soundness of $\vdash_s$ w.r.t. $\vdash$ would be obvious if each operational rule was a logical rule, but there is a slight difference: the use of $(\forall L)$ and $(\wedge L)$ logical rules is compensated by the use of notation $[\Delta]$ in operational rules (1) and (2). So that, each of the operational rules (1) through (5) is derivable in the $\mathcal{FO}^{\supset}$-calculus in the following way: Rule (1) is derivable using a number of steps of $(\forall L)$ and $(\wedge L)$ and one step of $(Init)$. Rule (2) can be seen as a particular case of $(\rightarrow L)$ when $\chi = \psi$. For this reason Rule (2) does not need a second premise which holds by $(Init)$. Therefore, Rule (2) is a combination of $(\forall L)$, $(\wedge L)$, $(\rightarrow L)$ and $(Init)$. Rule (3) is $(R\wedge)$, Rule (4) is $(R\exists)$ and Rule (5) is $(R \supset)$.
Now, a proof of the sequent $\Delta \rhd G$ can be made by substituting the corresponding step(s) in the $\mathcal{FO}^{\supset}$-calculus for each step in the proof of $\Delta \vdash_s G$. ■

As a particular case of this lemma, for Δ being a single program P, the following result is obtained:

Theorem 30. *Given a Σ-program P and Σ-goal G, if $P \vdash_s G$ then $P \vdash G$.* ■

B Appendix: Fixpoint Semantics

In this part, we prove the results that are sufficient to establish that T_P^{ω} is the least fixpoint of the operator T_P defined in Subsection 4.2 and that T_P^{ω} is the least model of P.

Proposition 31. *T_P is monotone.*

Proof. Suppose that $Fil(I_1) \sqsubseteq Fil(I_2)$, that is $I_1 \subseteq I_2$. Then (by Proposition 13(a)) $\{A \mid G \rightarrow A \in [P],\ I_1 \Vdash G\} \subseteq \{A \mid G \rightarrow A \in [P],\ I_2 \Vdash G\}$ holds. Therefore $T_P(Fil(I_1)) \sqsubseteq T_P(Fil(I_2))$. ■

In order to prove the continuity of T_P, we first establish the following key lemma:

Lemma 32. *For every chain $I_1 \subseteq I_2 \subseteq ... \subseteq I_j \subseteq ...$, of Herbrand Σ-interpretations, every Σ-clause D and every Σ-goal G,*
(a) $\cup_j I_j \Vdash G \Longrightarrow$ *there exists j_0 such that $I_{j_0} \Vdash G$*
(b) $\cup_j I_j \nVdash D \Longrightarrow$ *there exists j_0 such that $I_{j_0} \nVdash D$*

Proof. We proceed by simultaneous induction. For atoms (a) and (b) are trivial since $I \Vdash A$ iff $A \in I$. (a) for $G = \exists x G_1$ and (b) for $D = D_1 \wedge D_2$, $D = \forall x D_1$ can be easily proved by the induction hypothesis.
To prove (a) for $G = G_1 \wedge G_2$ suppose that $\cup_j I_j \Vdash G_1 \wedge G_2$. Then for some indices j_1,j_2: $I_{j_1} \Vdash G_1$ and $I_{j_2} \Vdash G_2$. Hence $I_j \Vdash G_1 \wedge G_2$ holds for $j = max(j_1, j_2)$.
Now, consider (b) for $D = G_1 \rightarrow A_1$. If $\cup_j I_j \nVdash D$ then $\cup_j I_j \Vdash G_1$ and $A_1 \notin \cup_j I_j$. By induction, there exists j_0 such that $I_{j_0} \Vdash G_1$ and $A_1 \notin I_{j_0}$. Therefore $I_{j_0} \nVdash D$.
In order to prove (a) for $G = D_1 \supset G_1$, we proceed by contradiction. Let us suppose that for all index j: $I_j \nVdash D_1 \supset G_1$. Then, for each j, there exists I'_j such that $I_j \subseteq I'_j$, $I'_j \Vdash D_1$ and $I'_j \nVdash G_1$. Considering, for each j, the non-empty set of interpretations $C_j = \{I \mid I_j \subseteq I,\ I \Vdash D_1,\ I \nVdash G_1\}$ and taking, for each j, the interpretation $I'_j = \cap\{I \mid I \in C_j\}$, the following facts are verified:

(i) $I_j \subseteq I'_j$, for all j
(ii) $I'_j \Vdash D_1$ and $I'_j \nVdash G_1$, for all j
(iii) $\{I'_j\}_j$ form the chain $I'_1 \subseteq I'_2 \subseteq ... \subseteq I'_j \subseteq ...$

By applying the induction hypothesis on D_1, G_1 and the chain $\{I'_j\}_j$, we have $\cup_j I'_j \Vdash D_1$ and $\cup_j I'_j \nVdash G_1$. Since $\cup_j I_j \subseteq \cup_j I'_j$, then $\cup_j I_j \nVdash D_1 \supset G_1$, in contradiction with the hypothesis. ■

Theorem 33. *Let* $Fil(I_1) \sqsubseteq Fil(I_2) \sqsubseteq ... \sqsubseteq Fil(I_j) \sqsubseteq ...$ *be a chain of elements in* $\underline{\mathsf{FMod}}(\Sigma)$. *Then* $T_P(\sqcup_j Fil(I_j)) = \sqcup_j T_P(Fil(I_j))$.

Proof. $\sqcup_j T_P(Fil(I_j)) \sqsubseteq T_P(\sqcup_j Fil(I_j))$ holds by monotonicity. The reverse inclusion is equivalent to prove that $\{A \mid G \rightarrow A \in [P],\ \cup_j I_j \Vdash G\} \subseteq \cup_j \{A \mid G \rightarrow A \in [P],\ I_j \Vdash G\}$. Let $A \in \{A \mid G \rightarrow A \in [P],\ \cup_j I_j \Vdash G\}$. Then, for some G: $G \rightarrow A \in [P]$ and $\cup_j I_j \Vdash G$. Since $I_1 \subseteq I_2 \subseteq ... \subseteq I_j \subseteq ...$, there exists an index j_0 such that $I_{j_0} \Vdash G$. Then $A \in \{A \mid G \rightarrow A \in [P],\ I_{j_0} \Vdash G\} \subseteq \cup_j \{A \mid G \rightarrow A \in [P],\ I_j \Vdash G\}$. ■

The following lemma states that the models of P are the pre-fixpoints of T_P.

Lemma 34. *Let* P *be a* Σ*-program and* $Fil(I) \in \underline{\mathsf{FMod}}(\Sigma)$. *Then* $Fil(I) \in \underline{\mathsf{FMod}}(P)$ *iff* $T_P(Fil(I)) \sqsubseteq Fil(I)$.

Proof. Let $Fil(I) \in \underline{\mathsf{FMod}}(P)$ and let us show that $\{A \mid G \rightarrow A \in [P],\ I \Vdash G\} \subseteq I$. If $A \in \{A \mid G \rightarrow A \in [P],\ I \Vdash G\}$, then there exists some G such that $G \rightarrow A \in [P]$ and $I \Vdash G$. Therefore $A \in I$, since $I \Vdash [P]$. Conversely, let $\{A \mid G \rightarrow A \in [P],\ I \Vdash G\} \subseteq I$. We have to show that $Fil(I) \models G \rightarrow A$, for each $G \rightarrow A \in [P]$. Suppose $Fil(I) \models G$. Then $A \in \{A \mid G \rightarrow A \in [P],\ I \Vdash G\} \subseteq I$. Hence $Fil(I) \models A$. ■

Unfolding and Event Structure Semantics for Graph Grammars*

Paolo Baldan, Andrea Corradini, and Ugo Montanari

Dipartimento di Informatica - University of Pisa
Corso Italia, 40, 56125 Pisa, Italy
E-mail: {baldan, andrea, ugo}@di.unipi.it

Abstract. We propose an unfolding semantics for graph transformation systems in the double-pushout (DPO) approach. Mimicking Winskel's construction for Petri nets, a graph grammar is unfolded into an acyclic branching structure, that is itself a (nondeterministic occurrence) graph grammar describing all the possible computations of the original grammar. The unfolding can be abstracted naturally to a prime algebraic domain and then to an event structure semantics. We show that such event structure coincides both with the one defined by Corradini et al. [3] via a comma category construction on the category of concatenable derivation traces, and with the one proposed by Schied [13], based on a deterministic variant of the DPO approach. This results, besides confirming the appropriateness of our unfolding construction, unify the various event structure semantics for the DPO approach to graph transformation.

1 Introduction

Since many (natural or artificial) distributed structures can be represented (at a suitable level of abstraction) by graphs, and graph productions act on those graphs with local transformations, it is quite obvious that graph transformation systems are potentially interesting for the study of the concurrent transformation of structures. In particular, Petri nets [11], the first formal tool proposed for the specification of the behaviour of concurrent systems, can be regarded as graph transformation systems that act on a restricted kind of graphs, namely discrete, labelled graphs (to be interpreted as sets of tokens labelled by place names).

In recent years, various concurrent semantics for graph rewriting systems have been proposed in the literature, some of which are inspired by the mentioned correspondence with Petri nets (see [2] for a tutorial introduction to the topic and for relevant references). A classical result in the theory of concurrency for Petri nets, due to Winskel [15], shows that the event structure semantics of *safe* nets can be given via a chain of adjunctions starting from the category **Safe** of safe nets, through category **Occ** of occurrence nets (this result has been generalized to arbitrary P/T nets in [9]). In particular, the event structure associated with

* Research partially supported by MURST project Tecniche Formali per Sistemi Software, by TMR Network GETGRATS and by Esprit WG APPLIGRAPH.

W. Thomas (Ed.): FOSSACS'99, LNCS 1578, pp. 73-89, 1999.

a net is obtained by first constructing a "nondeterministic unfolding" of the net, and then by extracting from it the events (which correspond to the transitions) and the causal and conflict relations among them.

In the present paper we propose a similar unfolding construction for DPO graph grammars, which can be considered as a first contribution to a functorial semantics in Winskel's style. After recalling in Section 2 the basics of typed graph transformation systems and their correspondence with Petri nets, we introduce, in Section 3, the notion of *nondeterministic occurrence grammar*, a generalization of the deterministic occurrence grammars of [4], representing in a unique "branching" structure several possible "acyclic" computations. Interestingly, unlike the case of Petri nets, the relationships among productions of an occurrence graph grammar cannot be captured completely by two binary relations representing causality and symmetric conflict. Firstly, due to the possibility of preserving some items in a rewriting step an asymmetric notion of conflict has to be considered. The way we face the problem is borrowed from [1], where we addressed an analogous situation arising in the treatment of *contextual nets*. Secondly, further dependencies among productions are induced by the application conditions, which constrain the applicability of the rewrite rules in order to preserve the consistency of the graphical structure of the state.

Next in Section 4 we present an unfolding construction that, when applied to a given grammar $\mathcal{G}$, yields a nondeterministic occurrence grammar $\mathcal{U}_{\mathcal{G}}$, which describes its behaviour. The idea consists of starting from the initial graph of the grammar, applying in all possible ways its productions, and recording in the unfolding each occurrence of production and each new graph item generated by the rewriting process. Our unfolding construction is conceptually similar to the unfolding semantics proposed for graph rewriting in the *single-pushout approach* by Ribeiro in [12]. However, here the situation is more involved and the two approaches are not directly comparable, due to the absence of the application conditions (dangling and identification) in the single-pushout approach.

In Section 5 we show how a prime algebraic domain (and therefore a prime event structure) can be extracted naturally from a nondeterministic occurrence grammar. Then the event structure semantics $ES(\mathcal{G})$ of a grammar $\mathcal{G}$ is defined as the event structure associated to its unfolding $\mathcal{U}_{\mathcal{G}}$. In Section 6 such semantics is shown to coincide with two other event structure semantics for graph rewriting in the literature: the one by Corradini et al. [3], built on top of the *abstract, truly concurrent model of computation* of a grammar (a category having abstract graphs as objects and concatenable derivation traces as arrows), and the one by Schied [13], based on a deterministic variation of the DPO approach. Finally, in Section 7 we conclude and present some possible directions of future work.

2 Typed Graph Grammars

This section briefly summarizes the basic definitions about typed graph grammars [4], a variation of classical DPO graph grammars [6, 5] where the rewriting takes place on the so-called *typed graphs*, namely graphs labelled over a structure

(the *graph of types*) that is itself a graph. Besides being strictly more general than usual labelled graphs, typed graphs will also allow us to have a clearer correspondence between graph grammars and Petri nets.

Formally, a *(directed, unlabelled) graph* is a tuple $G = \langle N, E, s, t\rangle$, where N is a set of *nodes*, E is a set of *arcs*, and $s, t : E \to N$ are the *source* and *target* functions. A *graph morphism* $f : G \to G'$ is a pair of functions $f = \langle f_N : N \to N', f_E : E \to E'\rangle$ preserving sources and targets, i.e., such that $f_N \circ s = s' \circ f_E$ and $f_N \circ t = t' \circ f_E$. Given a *graph of types* TG, a *typed graph* is a pair $\langle G, t_G\rangle$, where G is a graph and $t_G : G \to TG$ is a morphism. A morphism between typed graphs $f : \langle G_1, t_{G_1}\rangle \to \langle G_2, t_{G_2}\rangle$ is a graph morphisms $f : G_1 \to G_2$ consistent with the typing, i.e., such that $t_{G_1} = t_{G_2} \circ f$. The category of TG-typed graphs and typed graph morphisms is denoted by **TG-Graph**.

Fixed a graph TG of types, a *(TG-typed graph) production* $(L \stackrel{l}{\leftarrow} K \stackrel{r}{\rightarrow} R)$ is a pair of injective typed graph morphisms $l : K \to L$ and $r : K \to R$. It is called *consuming* if morphism $l : K \to L$ is not surjective. The typed graphs L, K, and R are called the *left-hand side*, the *interface*, and the *right-hand side* of the production, respectively. A *(TG-typed) graph grammar* $\mathcal{G}$ is a tuple $\langle TG, G_{in}, P, \pi\rangle$, where G_{in} is the *initial (typed) graph*, P is a set of *production names*, and π a function which associates a graph production to each production name in P. We denote by $Elem(\mathcal{G})$ the set $N_{TG} \cup E_{TG} \cup P$. Moreover, sometimes we shall write $q : (L \stackrel{l}{\leftarrow} K \stackrel{r}{\rightarrow} R)$ for $\pi(q) = (L \stackrel{l}{\leftarrow} K \stackrel{r}{\rightarrow} R)$.

Since in this paper we work only with typed notions, we will usually omit the qualification "typed", and we will not indicate explicitly the typing morphisms. Moreover, we will consider only *consuming* grammars, namely grammars where all productions are consuming: this corresponds, in the theory of Petri nets, to the common requirement that transitions must have non-empty preconditions.

Given a typed graph G, a production $q : (L \stackrel{l}{\leftarrow} K \stackrel{r}{\rightarrow} R)$, and a *match* (i.e., a graph morphism) $g : L \to G$, a *direct derivation δ from G to H using q (based on g)* exists, written $\delta : G \Rightarrow_q H$, if and only if the diagram

$$\begin{array}{cccccc} q: & L & \xleftarrow{l} & K & \xrightarrow{r} & R \\ & {\scriptstyle g}\downarrow & & \downarrow{\scriptstyle k} & & \downarrow{\scriptstyle h} \\ & G & \xleftarrow[b]{} & D & \xrightarrow[d]{} & H \end{array}$$

can be constructed, where both squares have to be pushouts in **TG-Graph**.

Roughly speaking, the rewriting step removes from the graph G the items of the left-hand side which are not in the image of the interface, namely $L - l(K)$, producing in this way the graph D. Then the items in the right-hand side which are not in the image of the interface, namely $R - r(K)$, are added to D, obtaining the final graph H. Notice that the interface graph K (common part of L and R) specifies both what is preserved and how the added subgraph has to be connected to the remaining part.

It is worth recalling here that given an injective morphism $l : K \to L$ and a match $g : L \to G$ as in the above diagram, their *pushout complement* (i.e., a

graph D with morphisms k and b such that the left square is a pushout) only exists if the *gluing conditions* are satisfied. These consist of two parts:

- the *identification condition*, requiring that if two distinct nodes or arcs of L are mapped by g to the same image, then both must be in the image of l;
- the *dangling condition*, stating that no arc in $G - g(L)$ should be incident to a node in $g(L - l(K))$ (because otherwise the application of the production would leave such an arc "dangling").

Notice that the identification condition does not forbid the match to be non-injective on preserved items. Intuitively this means that preserved (read-only) resources can be used with multiplicity greater than one.

A *derivation* over a grammar $\mathcal{G}$ is a sequence of direct derivations (over $\mathcal{G}$) $\rho = \{G_{i-1} \Rightarrow_{q_{i-1}} G_i\}_{i \in \{1,\ldots,n\}}$. The derivation is written as $\rho : G_0 \Rightarrow^*_{\{q_0,\ldots,q_{n-1}\}} G_n$ or simply as $\rho : G_0 \Rightarrow^* G_n$. The graphs G_0 and G_n are called the *starting* and the *ending graph* of ρ, and are denoted by $\sigma(\rho)$ and $\tau(\rho)$, respectively.

Relation with Petri nets. To conclude this section it is worth explaining the relation between Petri nets and DPO graph grammars. The fact that graph transformation systems can model the behaviour of Petri nets has been first formalized by Kreowski in [8]. The proposed encoding of nets into grammars represents the topological structure of a marked net as a graph, in such a way that the firing of transitions is modelled by direct derivations.

Here we use a slightly simpler modelling, discussed, among others, in [2]. The basic observation is that a P/T Petri net is essentially a rewriting system on multisets, and that, given a set A, a multiset of A can be represented as a discrete graph typed over A. In this view a P/T Petri net can be seen as a graph grammar acting on discrete graphs typed over the set of places, the productions being (some encoding of) the net transitions: a marking is represented by a set of nodes (tokens) labelled by the place where they are, and, for example, the unique transition t of the net in Fig. 1.(a) is represented by the graph production in the top row of Fig. 1.(b). Notice that the interface is empty since nothing is explicitly preserved by a net transition.

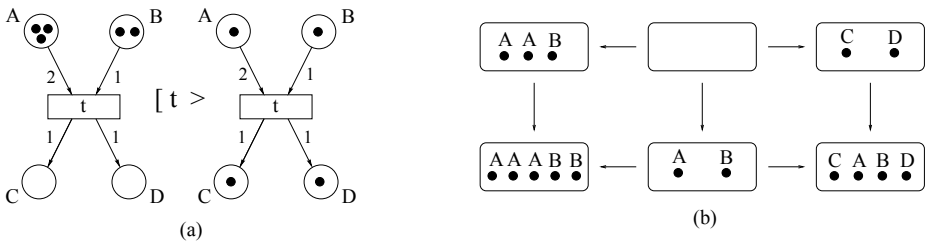


Fig. 1. Firing of a transition and corresponding DPO direct derivation.

It is easy to check that this representation satisfies the properties one would expect: a production can be applied to a given marking if and only if the corre-

sponding transition is enabled and, in this case, the double pushout construction produces the same marking as the firing of the transition. For instance, the firing of transition t, leading from the marking $3A+2B$ to the marking $A+B+C+D$ in Fig. 1.(a), becomes the double pushout diagram of Fig. 1.(b).

The considered encoding of nets into grammars enlightens the dimensions in which graph grammars properly extend nets. First of all grammars allow for a more structured state, that is a general graph rather than a multiset (discrete graph). Perhaps more interestingly, graph grammars allow for productions where the interface graph may not be empty, thus specifying a "context" consisting of items that have to be present for the productions to be applied, but are not affected by the application. In this respect, graph grammars are closer to some generalizations of nets in the literature, called nets with read (test) arcs or contextual nets (see e.g. [7, 10, 14]), which generalize classical nets by adding the possibility of checking for the presence of tokens which are not consumed.

3 Nondeterministic occurrence grammars

The notion of derivation introduced in the previous section formalizes how a single computation of a grammar can evolve. Nondeterministic occurrence grammars are intended to represent the computations of graph grammars in a more static way, by recording the events (production applications) which can appear in all possible derivations and the dependency relations between them.

Analogously to what happens for nets, occurrence grammars are "safe" grammars, where the dependency relations between productions satisfy suitable acyclicity and well-foundedness requirements. However, while for nets it suffices to take into account only the causal dependency and the conflict relations, the greater complexity of grammars makes the situation much more involved. On the one hand, the fact that a production application not only consumes and produces, but also preserves a part of the state leads to a form of asymmetric conflict (or weak dependency) between productions. On the other hand, because of the dangling condition, also the graphical structure of the state imposes some precedences between productions.

A first step towards a definition of occurrence grammar is a suitable notion of safeness for grammars [4], generalizing the usual one for P/T nets, which requires that each place contains at most one token in any reachable marking.

Definition 1 ((strongly) safe grammar). *A grammar* $\mathcal{G} = \langle TG, G_{in}, P, \pi \rangle$ *is* (strongly) safe *if, for all* H *such that* $G_{in} \Rightarrow^* H$, H *has an injective typing morphism.*

Strongly safe graph grammars (hereinafter called just safe grammars) admit a natural net-like pictorial representation, where items of the type graph and productions play, respectively, the rôle of places and transitions of Petri nets. The basic observation is that typed graphs having an injective typing morphism can be safely identified with the corresponding subgraphs of the type graph (just thinking of injective morphisms as inclusions). Therefore, in particular,

each graph $\langle G, t_G \rangle$ reachable in a safe grammar can be identified with the subgraph $t_G(G)$ of the type graph TG, and thus it can be represented by suitably decorating the nodes and arcs of TG. Concretely, a node is drawn as a filled circle if it belongs to $t_G(G)$ and as an empty circle otherwise, while an arc is drawn as a continuous line if it is in $t_G(G)$ and as a dotted line otherwise (see Fig. 2). This is analogous to the usual technique of representing the marking of a safe net by putting one token in each place which belongs to the marking.

With the above identification, in each computation of a safe grammar starting from the initial graph a production can only be applied to the subgraph of the type graph which is the image via the typing morphism of its left-hand side. Therefore according to its typing, we can safely think that a production *produces*, *preserves* or *consumes* items of the type graph. This is expressed by drawing productions as arrow-shaped boxes, connected to the consumed and produced resources by incoming and outcoming arrows, respectively, and to the preserved resources by undirected lines. Fig. 2 presents two examples of safe grammars, with their pictorial representation. Notice that the typing morphisms for the initial graph and the productions are represented by suitably labelling the involved graphs with items of the type graph.

Using a net-like language, we speak of *pre-set* ${}^\bullet q$, *context* $\underline{q}$ and *post-set* $q^\bullet$ of a production q, defined in the obvious way. Similarly for a node or arc x in TG we write ${}^\bullet x$, $\underline{x}$ and $x^\bullet$ to denote the sets of productions which produce, preserve and consume x. For instance, for grammar $\mathcal{G}_2$ in Fig. 2, the pre-set, context and post-set of production q_1 are ${}^\bullet q_1 = \{C\}$, $\underline{q_1} = \{B\}$ and $q_1{}^\bullet = \{A, L\}$, while for the node B, ${}^\bullet B = \emptyset$, $\underline{B} = \{q_1, q_2, q_3\}$ and $B^\bullet = \{q_4\}$.

Although the notion of causal relation is meaningful only for safe grammars, it is technically convenient to define it for general grammars. The same holds for the asymmetric conflict relation introduced below.

Definition 2 (causal relation). *The* causal relation *of a grammar $\mathcal{G}$ is the binary relation $<$ over $Elem(\mathcal{G})$ defined as the least transitive relation satisfying: for any node or arc x in the type graph TG, and for productions $q_1, q_2 \in P$*

1. *if $x \in {}^\bullet q_1$ then $x < q_1$;*
2. *if $x \in q_1{}^\bullet$ then $q_1 < x$;*
3. *if $q_1{}^\bullet \cap \underline{q_2} \neq \emptyset$ then $q_1 < q_2$;*

As usual $\leq$ is the reflexive closure of $<$. Moreover, for $x \in Elem(\mathcal{G})$ we denote by $\lfloor x \rfloor$ the set of causes of x in P, namely $\{q \in P : q \leq x\}$.

The first two clauses of the definition of relation $<$ are obvious. The third one formalizes the fact that if an item is generated by q_1 and it is preserved by q_2, then q_2, to be applied, requires that q_1 had already been applied.

Notice that the fact that an item is preserved by q_1 and consumed by q_2, i.e., $\underline{q_1} \cap {}^\bullet q_2 \neq \emptyset$ (e.g., the node B in grammar $\mathcal{G}_1$ of Fig. 2), does not imply $q_1 < q_2$. Actually, since q_1 must precede q_2 in any computation where both appear, in such computations q_1 acts as a cause of q_2. However, differently from a true cause, q_1 is not necessary for q_2 to be applied. Therefore we can think of

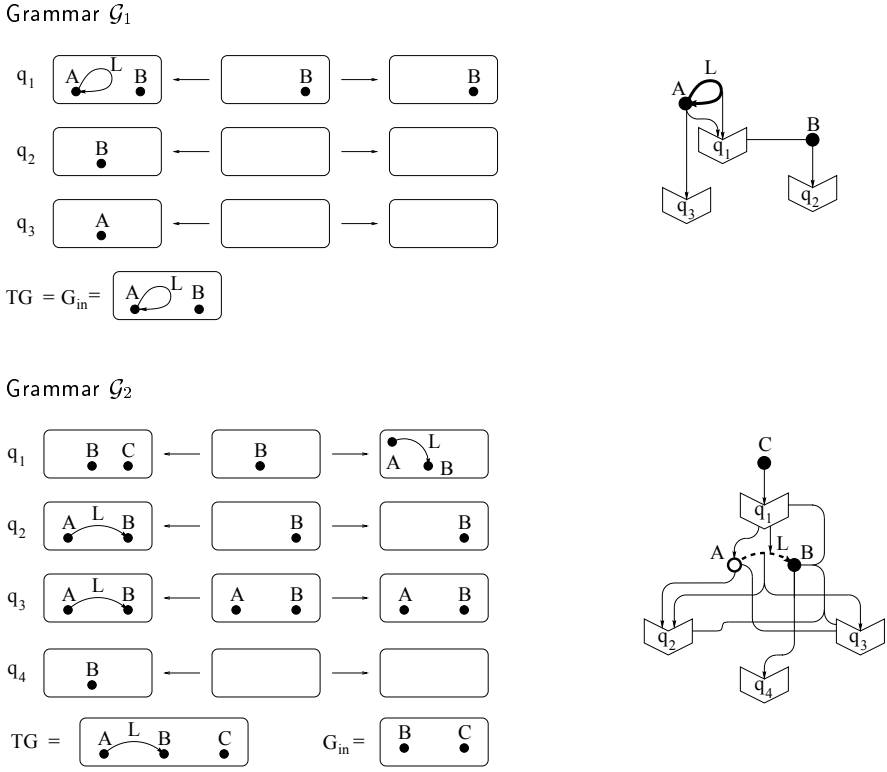


Fig. 2. Two safe grammars and their net-like representation.

the relation between the two productions as a *weak* form of *causal dependency*. Equivalently, we can observe that the application of q_2 prevents q_1 to be applied, so that q_1 can never follow q_2 in a derivation. But the converse is not true, since q_1 *can* be applied before q_2. Thus this situation can also be interpreted naturally as an *asymmetric conflict* between the two productions (see [1]).

Definition 3 (asymmetric conflict). *The* asymmetric conflict relation *of a grammar $\mathcal{G}$ is the binary relation $\nearrow$ over the set of productions, defined by:*

1. *if $\underline{q_1} \cap {}^\bullet q_2 \neq \emptyset$ then $q_1 \nearrow q_2$;*
2. *if ${}^\bullet q_1 \cap {}^\bullet q_2 \neq \emptyset$ and $q_1 \neq q_2$ then $q_1 \nearrow q_2$;*
3. *if $q_1 < q_2$ then $q_1 \nearrow q_2$.*

Condition 1 is justified by the discussion above. Condition 2 essentially expresses the fact that a situation of "classical" symmetric conflict is coded, in this setting, as an asymmetric conflict in both directions. Finally, since $<$ represents a global order of execution, while $\nearrow$ determines an order of execution only locally to each computation, it is natural to impose $\nearrow$ to be an extension of $<$ (Condition 3).

A *nondeterministic occurrence grammar* is an acyclic grammar which represents, in a branching structure, several possible computations starting from its initial graph and using each production at most once.

Definition 4 ((nondeterministic) occurrence grammar). *A* (nondeterministic) occurrence grammar *is a graph grammar* $\mathcal{O} = \langle TG, G_{in}, P, \pi \rangle$ *such that*

1. *its causal relation* $\leq$ *is a partial order, and for any* $q \in P$, *the set* $\lfloor q \rfloor$ *is finite and asymmetric conflict* $\nearrow$ *is acyclic on* $\lfloor q \rfloor$*;*
2. *the initial graph* G_{in} *coincides with the set* $Min(\mathcal{O})$ *of minimal elements of* $\langle Elem(\mathcal{O}), \leq \rangle$ *(with the graphical structure inherited from* TG *and typed by the inclusion);*
3. *each arc or node* x *in* TG *is created by at most one production in* P, *namely* $|\ {}^\bullet x\ | \leq 1$.
4. *for each production* $q : \langle L, t_L \rangle \stackrel{l}{\leftarrow} \langle K, t_K \rangle \stackrel{r}{\rightarrow} \langle R, t_R \rangle$, *the typing* t_L *is injective on the "consumed part"* $L - l(K)$, *and similarly* t_R *is injective on the "produced part"* $R - r(K)$.

Since the initial graph of an occurrence grammar $\mathcal{O}$ *is determined by* $Min(\mathcal{O})$, *we often do not mention it explicitly.*

One can show that, by the defining conditions, each occurrence grammar is *safe*.

Intuitively, conditions (1)–(3) recast in the framework of graph grammars the analogous conditions of occurrence nets (actually of occurrence contextual nets [1]). In particular condition (1) requires causality to be acyclic and each production q to have a finite set of causes $\lfloor q \rfloor$. Acyclicity of asymmetric conflict on $\lfloor q \rfloor$ corresponds to the requirement of irreflexivity for the conflict relation in occurrence nets. In fact, notice that if a set of productions forms an asymmetric conflict cycle $q_0 \nearrow q_1 \nearrow \ldots \nearrow q_n \nearrow q_0$, then such productions cannot appear in the same computation, otherwise the application of each production should precede the application of the production itself; this fact can be naturally interpreted as a form of n-ary conflict. Condition (2) forces the set of minimal items of the type graph to be a graph, coinciding with the initial graph of the grammar and Condition (3) requires the absence of backward conflicts. Condition (4), instead, is closely related to safeness and requires that each production consumes and produces items with multiplicity one. Together with acyclicity of $\nearrow$, it disallows the presence of some productions which surely could never be applied, because they fail to satisfy the identification condition with respect to the typing morphism.

On the contrary, the definition does not imply that every production of an occurrence grammar will ever satisfy the *dangling condition*. This fact deserves some comments since the dangling condition, which requires the *absence* of arcs pointing to nodes which are removed by the production, induces precedence relations on productions. For example, in the grammar $\mathcal{G}_2$ of Fig. 2 the application of production q_1 "disables" q_4, since q_4 would remove the node B, leaving the arc L dangling. The production q_4 becomes enabled again only after q_2 or q_3 has been applied. The reason why we defined occurrence grammars in this way is that the dangling condition is not purely syntactical and cannot be checked "locally" by looking only at the causes of the considered production. Checking such negative (non monotonic) condition on a production, would require to find a possible computation allowing for the execution of productions which remove

the potentially dangling arcs, and to verify the consistency of such computation with the production at hand. It can be shown that such verification is, in general, exponential in the size of the occurrence grammar in the finite case. Even worse, for infinite occurrence grammars (which can be obtained as unfolding of finite grammars), the problem is undecidable, as it can be shown by using the Turing completeness of DPO graph grammars.

Disregarding the dangling condition has as a consequence the fact that, differently from what happens for occurrence nets, not every production in an occurrence grammar is guaranteed to be applicable at least in one derivation starting from the initial graph. The restrictions to the behaviour imposed by the dangling condition are considered when defining the configurations of an occurrence grammar, which represent exactly, in a sense formalized later, all the possible deterministic runs of the grammar.

Definition 5 (configuration). *A* configuration *of an occurrence grammar* $\mathcal{O} = \langle TG, P, \pi \rangle$ *is a subset* $C \subseteq P$ *such that*

1. *if* $\nearrow_C$ *denotes the restriction of the asymmetric conflict relation to* C*, then* $(\nearrow_C)^*$ *is a partial order, and* $\{q' \in C : q'(\nearrow_C)^* q\}$ *is finite for all* $q \in C$*;*[1]
2. C *is left-closed w.r.t.* $\leq$*, i.e. for all* $q \in C$*,* $q' \in P$*,* $q' \leq q$ *implies* $q' \in C$*;*
3. *for all* $e \in TG$ *and* $n \in \{s(e), t(e)\}$*, if* $n^\bullet \cap C \neq \emptyset$ *and* ${}^\bullet e \subseteq C$ *then* $e^\bullet \cap C \neq \emptyset$*.*

If C *satisfies conditions (1) and (2), then it is called a* pre-configuration.

The notion is reminiscent of that of configuration of asymmetric event structures and thus of occurrence contextual nets [1]. The first part of Condition 1 ensures that in C there are no $\nearrow$-cycles, and thus excludes the possibility of having in C a subset of productions in conflict. The second part guarantees that each production has to be preceded only by finitely many other productions in the computation represented by the configuration. Condition 2 requires the presence of all the causes of each production, while Condition 3 formalizes the dangling condition. If a configuration contains a production q consuming a node n and a production q' producing an arc e (i.e. ${}^\bullet e = \{q'\}$) with source (or target) n, then a production q'' removing such an arc must be present as well, otherwise, due to the dangling condition, q could not be executed. Notice that in this situation the production q'' can coincide with q itself; otherwise it surely preserves the node n and thus $q'' \nearrow q$, i.e. q'' correctly precedes q in the computation represented by the configuration. Similar considerations apply if the arc e is present in the initial graph, i.e., ${}^\bullet e = \emptyset$. For example the set of configurations of the grammar $\mathcal{G}_2$ in Fig. 2 is $Conf(\mathcal{G}_2) = \{\emptyset, \{q_1\}, \{q_1, q_2\}, \{q_1, q_3\}, \{q_1, q_2, q_4\}, \{q_1, q_3, q_4\}, \{q_4\}\}$. The set $S = \{q_1, q_4\}$, is instead only a pre-configuration, since for the node B we have $B = t(L)$, $q_4 \in B^\bullet$, ${}^\bullet L = \{q_1\} \subseteq S$, but the intersection of S with $L^\bullet = \{q_2, q_3\}$ is empty.

The fact that configurations represent all and only the deterministic runs of an occurrence grammar is formalized by the following result.

[1] As usual, for a binary relation r, with r^* we denote its transitive and reflexive closure.

Proposition 1 (configurations and derivations). *For any configuration C of an occurrence grammar $\mathcal{O}$, there exists a subgraph G_C of the type graph TG such that $Min(\mathcal{O}) \Rightarrow^*_C G_C$ with a derivation which applies exactly once every production in C, in any order consistent with $(\nearrow_C)^*$. Viceversa for each derivation $Min(\mathcal{O}) \Rightarrow^*_S G$ in $\mathcal{O}$, the set of productions S it applies is a configuration.*

As an immediate consequence of the previous result, a production which does not satisfy the dangling condition in any graph reachable from the initial graph is not part of any configuration. For example, q_3 does not appear in the set of configurations of $\mathcal{G}_1$, $Conf(\mathcal{G}_1) = \{\emptyset, \{q_1\}, \{q_2\}, \{q_1, q_2\}\}$.

In the theory of Petri nets the notion of occurrence grammar is strictly related to that of process. A (non)deterministic net process is a (non)deterministic occurrence net with a morphism to the original net. Similarly, nondeterministic occurrence grammars can be used to define a suitable notion of *nondeterministic graph processes*, generalizing the deterministic graph processes of [4]. Then, the unfolding of a grammar, introduced in the next section, could be seen as a "complete" nondeterministic process of the grammar. Unfortunately, these notions cannot be discussed here because of space limitations.

4 Unfolding

This section introduces the unfolding construction which, applied to a consuming grammar $\mathcal{G}$, produces a nondeterministic occurrence grammar $\mathcal{U}_\mathcal{G}$ describing the behaviour of $\mathcal{G}$. The unfolding is equipped with a mapping $\phi_\mathcal{G}$ to the original grammar $\mathcal{G}$ which allows to see productions in $\mathcal{U}_\mathcal{G}$ as instances of production applications in $\mathcal{G}$, and items of the type graph of $\mathcal{U}_\mathcal{G}$ as instances of items of the type graph of $\mathcal{G}$.

The idea consists of starting from the initial graph of the grammar, then applying in all possible ways its productions, and recording in the unfolding each occurrence of production and each new graph item generated in the rewriting process, both enriched with the corresponding causal history. According to the discussion in the previous section, during the unfolding process productions are applied without considering the dangling condition. Moreover we adopt a notion of concurrency which is "approximated", again in the sense that it does not take care of the precedences between productions induced by the dangling condition.

Definition 6 (quasi-concurrent graph). *Let $\mathcal{O} = \langle TG, P, \pi \rangle$ be an occurrence grammar. A subgraph G of TG is called* quasi-concurrent *if*

1. $\bigcup_{x \in G} \lfloor x \rfloor$ *is a pre-configuration;*
2. $\neg(x < y)$ *for all* $x, y \in G$.

The intuitive idea is that each quasi-concurrent graph is contained in a graph reachable in a "lax version" of the DPO rewriting, where the dangling condition is not tested.

Another basic ingredient of the unfolding is the gluing operation. It can be interpreted as a "partial application" of a rule to a given match, in the sense that

it generates the new items as specified by the production (i.e., items of right-hand side not in the interface), but items that should have been deleted are not affected: intuitively, this is because such items may still be used by another production in the nondeterministic unfolding. In the following we assume that for each production name q its associated production is $L_q \leftarrow K_q \rightarrow R_q$, where the injections l_q and r_q are inclusions (and not generic injective morphisms).

Definition 7 (gluing). *Let q be a production, G a graph and $m : L_q \rightarrow G$ a graph morphism. We define, for any symbol $*$, the* gluing of G and R_q along K_q, according to m and marked by $*$, *denoted by $glue_*(q, m, G)$ as the graph $\langle N, E, s, t\rangle$, where:*

$$N = N_G \cup m_*(N_{R_q}) \qquad E = E_G \cup m_*(E_{R_q})$$

with m_* defined by: $m_*(x) = m(x)$ if $x \in K_q$ and $m_*(x) = \langle x, *\rangle$ otherwise. The source and target functions s and t, and the typing are inherited from G and R_q.

The gluing operation keeps unchanged the identity of the items already in G, and records in each newly added item from R_q the given symbol $*$. We remark that the gluing, as just defined, is a concrete deterministic definition of the pushout of the arrows $G \overset{m}{\leftarrow} L_q \overset{l_q}{\hookleftarrow} K_q$ and $K_q \overset{r_q}{\hookrightarrow} R_q$.

Now the unfolding of a grammar $\mathcal{G} = \langle TG, G_{in}, P, \pi\rangle$ can be as follows. For each n, we construct a partial unfolding $\mathcal{U}(\mathcal{G})^{(n)} = \langle \mathcal{U}^{(n)}, \phi^{(n)}\rangle$, where $\mathcal{U}^{(n)} = \langle TG^{(n)}, P^{(n)}, \pi^{(n)}\rangle$ is an occurrence grammar and the mapping $\phi^{(n)} = \langle fp^{(n)}, fg^{(n)}\rangle$ consists of two components: a function $fp^{(n)} : P^{(n)} \rightarrow P$ mapping the productions of the unfolding into productions of $\mathcal{G}$, and a morphism $fg^{(n)} : TG^{(n)} \rightarrow TG$ from the type graph of $\mathcal{U}^{(n)}$ to TG. Intuitively, the occurrence grammar generated at level n contains all possible computations of the grammar with "causal depth" at most n.

- $(\mathbf{n = 0})$ $\langle TG^{(0)}, fg^{(0)}\rangle = G_{in}$, while $P^{(0)}$, $\pi^{(0)}$ and $fp^{(0)}$ are empty.
- $(\mathbf{n \rightarrow n + 1})$ Given $\mathcal{U}(\mathcal{G})^{(n)}$, the partial unfolding $\mathcal{U}(\mathcal{G})^{(n+1)}$ is obtained by extending it with all the possible production applications to quasi-concurrent subgraphs of the type graph of $\mathcal{U}^{(n)}$. More precisely, for each production $q \in P$ and match $m : L_q \rightarrow \langle TG^{(n)}, fg^{(n)}\rangle$ satisfying the identification condition, with $m(L_q)$ *quasi-concurrent* subgraph of $TG^{(n)}$:
  - Add to $P^{(n)}$ an occurrence of the production q, with name $q' = \langle q, m\rangle$. The match m is needed to record the "history" of q'. Now let $P^{(n)} := P^{(n)} \cup \{q'\}$, and extend $fp^{(n)}$ so that $fp^{(n)}(q') = q$. The production $\pi^{(n)}(q)$ coincides with $\pi(q)$ except for the typing.
  - Glue the type graph $TG^{(n)}$, typed over TG by $fg^{(n)}$, with the right-hand side R_q of q along K_q, according to the mapping m and marked by q'; in this way the new items generated by the production contain the name q' of the occurrence of the production and thus their history. The morphism $fg^{(n)}$ is updated consequently.

  After all the applicable productions have been considered we obtain $\mathcal{U}(\mathcal{G})^{(n+1)}$.

The deterministic gluing construction ensures that, at each step, the order in which productions are applied does not influence the final result of the step. Moreover if a production is applied twice (also in different steps) at the same match, the generated items are always the same and thus they appear only once in the unfolding.

Definition 8 (unfolding). *The unfolding $\mathcal{U}(\mathcal{G}) = \langle \mathcal{U}_\mathcal{G}, \phi_\mathcal{G} \rangle$ of the grammar $\mathcal{G}$ is defined as $\bigcup_n \mathcal{U}(\mathcal{G})^{(n)}$, where union is applied componentwise.*

It is not difficult to verify that for each n, $\mathcal{U}^{(n)}$ is a (finite) nondeterministic occurrence grammar, and $\mathcal{U}(\mathcal{G})^{(n)} \subseteq \mathcal{U}(\mathcal{G})^{(n+1)}$, componentwise. Therefore $\mathcal{U}_\mathcal{G}$ is an occurrence grammar. Moreover the unfolding process applied to an occurrence grammar yields a grammar which is isomorphic to the original one.

Finally, we notice that, as already remarked, not all productions in the unfolding are executable in some computation and thus correspond to occurrences of production of the original grammar. This is due to the fact that only the identification condition is tested and an "approximated notion" of concurrent subgraph is used. We stress that this is needed to have a decidable unfolding, a fact which, besides being pleasant from a purely theoretical point of view, is essential if one wants to use the unfolding in practice to prove properties of the modelled system.

5 Domain and event structure semantics

In the seminal work of Winskel on (safe) Petri nets, the unfolding semantics of a net, given in terms of a nondeterministic occurrence net, is further abstracted to an event structure semantics, by forgetting the "real structure" of the unfolding and recording only the relationships induced by such structure on the transitions of the unfolding itself. In this section we show that a similar construction can be carried out for graph grammars.

Recall that a *prime event structure with binary conflict (PES),* consists of a set of events endowed with two binary relations: a partial order relation $\leq$, modelling *causality*, and a symmetric and irreflexive relation $\#$, hereditary w.r.t. causality, modelling *conflict*. A *configuration* of a PES is a subset of events left-closed w.r.t. $\leq$ and conflict free, representing a possible computation of the system modelled by the event structure. The set of configurations of a PES, ordered by subset inclusion, is a finitary prime algebraic domain, i.e. a coherent, prime algebraic, finitary partial order, briefly a *domain*, and the set of prime elements of a domain (with the induced partial order as causality and the inconsistency relation as conflict) is a PES.

We already observed that the notion of configuration of an occurrence grammar allows us to recover exactly the different possible deterministic computations of the grammar. Following the ideas suggested for asymmetric event structures and contextual nets in [1], an order can be defined on configurations which captures the idea of computational extension. The main point is that, differently

from what happens for classical event structures and Petri nets, due to the presence of the asymmetric conflict such an order is not simply set-inclusion: in fact, a configuration C cannot be extended with a production inhibited by some of the productions already present in C.

Definition 9 (poset of configurations). *Given an occurrence grammar $\mathcal{O}$, we denote by $Conf(\mathcal{O})$ the set of its configurations, ordered by the relation $\sqsubseteq$ defined as $C \sqsubseteq C'$ if $C \subseteq C'$ and $\neg(q' \nearrow q)$, for all $q \in C$ and $q' \in C' - C$.*

The partial order of configurations of an occurrence grammar exhibits a very nice algebraic structure, i.e., it is a domain. The proof (that we skip here) follows the same outline as in [1], but more effort is needed to take care of the additional requirement in the definition of configuration, related to the dangling condition.

Theorem 1 (from occurrence grammars to domains). *Given an occurrence grammar $\mathcal{O}$, the partial order of configurations $Conf(\mathcal{O})$ is a domain.*

By the relation between domains and event structures sketched above, $Conf(\mathcal{O})$ determines indirectly an event structure $ES(\mathcal{O})$, namely, the unique (up to isomorphisms) PES having $Conf(\mathcal{O})$ as domain of configurations. Differently from what happens for Petri nets, there is not a one to one correspondence between events of $ES(\mathcal{O})$ and productions in $\mathcal{O}$. Instead, a different event is generated for any possible "history" of each production of $\mathcal{O}$. This phenomenon of "duplication of events" is related to the fact that the new precedence relations arising between productions in graph grammars are represented via causality and conflict in classical PES's. Basically, a situation of asymmetric conflict like $q_1 \nearrow q_2$ in grammar $\mathcal{G}_1$ of Fig. 2, is coded in the PES by the insertion of a single event e_1 corresponding to q_1, and two "copies" e_2' ad e_2'' of q_2, the first one in conflict with e_1 and the second one caused by e_1 (see Fig. 3.(a)). For what concerns the dangling condition, consider the grammar $\mathcal{G}_2$ in Fig. 2. In this case three conflicting events are generated corresponding to q_4: e_4 representing the execution of q_4 from the initial graph, which inhibits all other productions, and e_4', e_4'' representing the execution of q_4 after q_2 and q_3, respectively.

Fig. 3. Coding asymmetric conflict and dangling condition in prime event structures.

As a final simple step, a domain and an event structure semantics for a graph grammar are readily defined via the unfolding construction.

Definition 10 (event structure semantics). *For any grammar $\mathcal{G}$, we denote by $Conf(\mathcal{G})$ the domain of configurations of the unfolding of $\mathcal{G}$, namely $Conf(\mathcal{U}_\mathcal{G})$, and by $ES(\mathcal{G})$ the corresponding event structure $ES(\mathcal{U}_\mathcal{G})$.*

6 Relation with other event structure semantics

This section briefly reviews two other event structure semantics proposed in the literature for DPO graph transformation systems. The first one [3] is built on top of the "abstract truly concurrent model of computation" of a grammar. The other one [13] is based on a deterministic variation of the DPO approach. Nicely, these two alternative event structures turn out to coincide with the one obtained from the unfolding, which thus can be claimed to give "the" event structure semantics of DPO graph transformation.

Event structure semantics from abstract derivations. The derivations of a grammar $\mathcal{G}$ are easily equipped with a simple algebraic structure which turns them into a category, called the *concrete model of computation for* $\mathcal{G}$ and denoted $\mathbf{Der}[\mathcal{G}]$. Objects in $\mathbf{Der}[\mathcal{G}]$ are graphs, and each derivation ρ is seen as an arrow from $\sigma(\rho)$ to $\tau(\rho)$. Given two derivations ρ and ρ' such that the ending graph of ρ and the starting graph of ρ' coincide, i.e., $\tau(\rho) = \sigma(\rho')$, their sequential composition $\rho\,;\,\rho'$ is the derivation obtained by identifying $\tau(\rho)$ with $\sigma(\rho')$.

The concrete model contains a lot of redundant information and it is far from representing what one has in mind as truly concurrent behaviour of the system modeled by the grammar. A more reasonable model, called the *abstract, truly concurrent model of computation* of a grammar $\mathcal{G}$, and denoted by $\mathbf{Tr}[\mathcal{G}]$, is the category obtained by imposing a suitable equivalence on objects and arrows of the concrete model. In particular, the objects of $\mathbf{Tr}[\mathcal{G}]$ are abstract graphs (i.e., isomorphism classes of graphs), while its arrows are *concatenable derivation traces*, i.e., equivalence classes of derivations with respect to the *concatenable truly concurrent equivalence* [3]. This equivalence is the least equivalence on derivations containing both the *abstraction equivalence*, a refinement of the obvious notion of derivation isomorphism compatible with sequential composition, and the *shift equivalence*, which equates two derivations if one can be obtained from the other by repeatedly shifting independent derivation steps.

The category $\mathbf{Tr}[\mathcal{G}]$ is used in [3] to define a domain and a prime event structure semantics for graph transformation systems. More precisely, for any consuming graph grammar $\mathcal{G} = \langle TG, G_{in}, P, \pi\rangle$, one considers the comma category $([G_{in}] \downarrow \mathbf{Tr}[\mathcal{G}])$, where objects are concatenable derivation traces of $\mathbf{Tr}[\mathcal{G}]$ with source in $[G_{in}]$, and given two such traces δ_0 and δ_1, an arrow from δ_0 to δ_1 is a concatenable derivation trace δ satisfying $\delta_0;\delta = \delta_1$. Such category can be shown to be a preorder $\mathbf{PreDom}[\mathcal{G}]$, i.e., there is at most one arrow between any pair of objects. Moreover the ideal completion of $\mathbf{PreDom}[\mathcal{G}]$ is a domain, denoted by $\mathbf{Dom}[\mathcal{G}]$ and proposed as truly concurrent semantics of the grammar.

As announced such domain semantics can be proved to coincide with the one obtained from the unfolding $\mathcal{U}(\mathcal{G})$. In fact, we know from [3] that the finite

elements of the domain $\mathbf{Dom}[\mathcal{G}]$ are one-to-one with derivation traces of $\mathcal{G}$ having the initial graph as source. Then the result is proved by showing that a bijection can be defined between finite elements of $Conf(\mathcal{G})$ and such derivation traces. In one direction, given a finite configuration $C \in Conf(\mathcal{G})$, consider any derivation $Min(\mathcal{U}_\mathcal{G}) \Rightarrow^*_C G_C$ in $\mathcal{U}_\mathcal{G}$ which applies exactly once every production in C, in any order consistent with the asymmetric conflict $\nearrow$. Such derivation, typed over the type graph of $\mathcal{G}$ via the mapping $\phi_\mathcal{G}$, gives a derivation d in $\mathcal{G}$, which determines the derivation trace associated to C. Viceversa, given a derivation trace $[d]$ of $\mathcal{G}$, the corresponding configuration of $\mathcal{U}_\mathcal{G}$ is determined as the set of productions needed to "simulate" d in the unfolding of $\mathcal{G}$. The fact that the ordering on configurations is not simply set-inclusion then plays a key rôle in the proof that such bijection is an isomorphism of partial orders.

Theorem 2. *For any (consuming) graph grammar $\mathcal{G}$, the domains $Conf(\mathcal{G})$ and $\mathbf{Dom}[\mathcal{G}]$ are isomorphic.*

Event structure semantics from deterministic derivations. Schied in [13] proposes a construction for defining an event structure semantics for *distributed rewriting systems*, an abstract unified model where several kind of rewriting systems, such as graph grammars and term rewriting systems, naturally fit. He shows that, given a distributed rewriting system $\mathcal{R}$, a domain $\mathcal{T}_\mathcal{R}$ can be obtained as the quotient, with respect to shift equivalence, of the collection of derivations starting from the initial state, ordered by the prefix relation. To prove the algebraic properties of $\mathcal{T}_\mathcal{R}$ he constructs, as an intermediate step, a *trace language* based on the shift equivalence, and applies general results to extract an event structure $\mathcal{E}_\mathcal{R}$ from the trace language. Finally he shows that $\mathcal{T}_\mathcal{R}$ is isomorphic to the domain of configurations of $\mathcal{E}_\mathcal{R}$.

The main interest in Schied's paper is for the application to graph grammars. Let us sketch how, according to Schied, the above construction instantiates to the case of grammars. Graph grammars are modeled as distributed rewriting systems by considering a *deterministic* variation of the DPO approach, where at each direct derivation the derived graph is uniquely determined by the host graph, the applied production and the match. The idea consists of working on concrete graphs, where each item records his causal history. Formally the definition of *deterministic direct derivation* (adapted to the typed case) is as follows.

Definition 11 (deterministic derivation). *Let $q : L_q \leftarrow K_q \rightarrow R_q$ be a production and let $m : L_q \rightarrow G$ be a match. Then a* deterministic direct derivation *$G \rightsquigarrow_{q,m} H$ exists if m satisfies the gluing conditions and*

$$H = glue_{\langle q,m \rangle}(q, m, G) - m(L_q - l(K_q)).$$

Let $\mathcal{G} = \langle TG, G_{in}, P, \pi \rangle$ be a typed graph grammar. A deterministic derivation *in $\mathcal{G}$ is a sequence of deterministic direct derivations $G_{in} \rightsquigarrow_{q_1,m_1} G_1 \rightsquigarrow_{q_2,m_2} \ldots \rightsquigarrow_{q_n,m_n} G_n$, starting from the initial graph and applying productions of $\mathcal{G}$.*

The construction of the domain of a grammar is based on the partial order of deterministic derivations with the prefix relation, and on shift equivalence.

Definition 12 (Schied's domain). *The Schied's domain for a consuming grammar $\mathcal{G}$, denoted by $\mathcal{T}_{\mathcal{G}}$, is defined as the quotient, w.r.t. shift equivalence, of the partial order of deterministic derivations of a grammar $\mathcal{G}$.*

It is not difficult to see that the (ideal completion of) Schied's domain for a grammar coincides with the domain of configurations of its unfolding $Conf(\mathcal{G})$, and thus with the domain $\mathbf{Dom}[\mathcal{G}]$ of [3]. The bijection between $\mathcal{T}_{\mathcal{G}}$ and the finite elements of $Conf(\mathcal{G})$ associates to the class of shift equivalent deterministic derivations containing $d : G_{in} \rightsquigarrow_{q_1,m_1} G_1 \rightsquigarrow_{q_2,m_2} \ldots \rightsquigarrow_{q_n,m_n} G_n$ the set $\{\langle q_i, m_i\rangle : i \in \underline{n}\}$, which can be shown to be a configuration in the unfolding of $\mathcal{G}$, independently of the particular derivation picked up in the class.

Theorem 3. *For any graph grammar $\mathcal{G}$, the ideal completion of $\mathcal{T}_{\mathcal{G}}$ and the domain $Conf(\mathcal{G})$ are isomorphic.*

7 Conclusions and future work

This paper introduces a notion of *nondeterministic* occurrence grammar for graph transformation systems in the algebraic DPO approach, by extending the work developed in [4] for the deterministic case. The phenomenon of asymmetric conflict between productions, caused by the possibility of performing "context sensitive" rewritings, cannot be ignored in this nondeterministic setting, and comes into play as an essential ingredient. A new kind of dependency between productions is also induced by the dangling condition, which imposes precedences among productions finalized at preserving the consistency of the graphical structure of the state.

Following the classical idea proposed by Winskel [15] for Petri nets, an *unfolding semantics* for DPO graph rewriting systems has been defined as a nondeterministic occurrence grammar, representing, in a single "branching" structure, all the possible computations of the grammar. The dangling condition, being a negative (non monotone) condition, can hardly be verified during the unfolding process. As a consequence the generated unfolding contains some garbage of which we get rid only when considering the set of configurations.

Interestingly, the set of configurations $Conf(\mathcal{G})$ of (the unfolding of) a grammar $\mathcal{G}$, suitably ordered using the asymmetric conflict relation, turns out to be a (finitary pairwise coherent) prime algebraic domain, one of the most widely used mathematical structures in the semantics of concurrency, equivalent to prime event structures (with binary conflict). Such domain is shown to coincide both with the domain $\mathbf{Dom}[\mathcal{G}]$, built from the category of concatenable derivation traces and proposed as semantics of a grammar in [3], and with the domain defined by Schied [13] and based on a concrete formulation of the DPO rewriting.

Finally, it is worth mentioning that the original work of Winskel shows that the unfolding construction extends to a coreflection from the category of safe nets to the category of domains, while our construction has been defined, up to now, only at "object level". We are working to obtain a full correspondence with Winskel's construction for nets, by extending the results presented in this paper

to a categorical "in the large" level. Some suggestions can surely come from [1], where Winskel's construction has already been fully extended to contextual nets.

Acknowledgements

The authors wish to thank Roberto Bruni and Fabio Gadducci for helpful discussions and the anonymous referees for their comments on the submitted version of this paper.

References

1. P. Baldan, A. Corradini, and U. Montanari. An event structure semantics for P/T contextual nets: Asymmetric event structures. *Proceedings of FoSSaCS '98*, volume 1378, pages 63–80. Springer, 1998.
2. A. Corradini. Concurrent Graph and Term Graph Rewriting. *Proceedings CONCUR'96*, volume 1119 of *LNCS*, pages 438–464. Springer, 1996.
3. A. Corradini, H. Ehrig, M. Löwe, U. Montanari, and F. Rossi. An Event Structure Semantics for Graph Grammars with Parallel Productions. *Proceedings of the 5th International Workshop on Graph Grammars and their Application to Computer Science*, volume 1073 of *LNCS*. Springer, 1996.
4. A. Corradini, U. Montanari, and F. Rossi. Graph processes. *Fundamenta Informaticae*, 26:241–265, 1996.
5. A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Löwe. Algebraic Approaches to Graph Transformation I: Basic Concepts and Double Pushout Approach. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation. Volume 1: Foundations*. World Scientific, 1997.
6. H. Ehrig. Tutorial introduction to the algebraic approach of graph-grammars. *Proceedings of the 3rd International Workshop on Graph-Grammars and Their Application to Computer Science*, volume 291 of *LNCS*, pages 3–14. Springer, 1987.
7. R. Janicki and M. Koutny. Semantics of inhibitor nets. *Information and Computation*, 123:1–16, 1995.
8. H.-J. Kreowski. A comparison between Petri nets and graph grammars. *Proceedings of the Workshop on Graphtheoretic Concepts in Computer Science*, volume 100 of *LNCS*, pages 306–317. Springer, 1981.
9. J. Meseguer, U. Montanari, and V. Sassone. Process versus unfolding semantics for Place/Transition Petri nets. *Theoret. Comput. Sci.*, 153(1-2):171–210, 1996.
10. U. Montanari and F. Rossi. Contextual nets. *Acta Informatica*, 32, 1995.
11. W. Reisig. *Petri Nets: An Introduction*. EACTS Monographs on Theoretical Computer Science. Springer, 1985.
12. L. Ribeiro. *Parallel Composition and Unfolding Semantics of Graph Grammars*. PhD thesis, Technische Universität Berlin, 1996.
13. G. Schied. On relating Rewriting Systems and Graph Grammars to Event Structures. *Proceedings of the Dagstuhl Seminar 9301 on Graph Transformations in Computer Science*, volume 776 of *LNCS*, pages 326–340. Springer, 1994.
14. W. Vogler. Efficiency of asynchronous systems and read arcs in Petri nets. Technical Report 352, Institüt für Mathematik, Augsburg University, 1996.
15. G. Winskel. Event Structures. In *Petri Nets: Applications and Relationships to Other Models of Concurrency*, volume 255 of *LNCS*, pages 325–392. Springer, 1987.

Expanding the Cube

Gilles Barthe

Institutionen för Datavetenskap, Chalmers Tekniska Högskola, Göteborg, Sweden
Departamento de Informática, Universidade do Minho, Braga, Portugal
gilles@di.uminho.pt

Abstract. We prove strong normalization of β-reduction+η-expansion for the Calculus of Constructions, thus providing the first strong normalization result for β-reduction+η-expansion in calculi of dependent types and answering in the affirmative a conjecture by Di Cosmo and Ghani. In addition, we prove strong normalization of β-reduction+η-expansion+algebraic reduction for the Algebraic Calculus of Constructions, which extends the Calculus of Constructions with first-order term-rewriting systems. The latter result, which requires the term-rewriting system to be non-duplicating, partially answers in the affirmative another conjecture by Di Cosmo and Ghani.

1 Introduction

Extensionality, as embodied in *η-conversion*

$$\lambda x{:}A.\ M\ x = M \qquad \text{if } x \notin \mathrm{FV}(M)$$

is a basic notion in λ-calculus and type theory. Traditionally, η-conversion has been oriented from left to right, thus leading to *η-reduction*. Recently, several authors have advocated a different computational interpretation, in which η-conversion is oriented from right to left, thus leading to *η-expansion*. The latter, which originates from proof-theoretical considerations [23], has found an increasing number of applications in computer science. The current body of results on η-expansion and its applications is too large to be presented here in any detail but we refer to [9] for a recent survey of the field, including a summary of the applications of η-expansion, an historical account of the subject and pointers to the literature.

The first part of this paper is concerned with normalization of β-reduction+η-expansion for the λ-cube [4, 5, 14].[1] While weak normalization of β-reduction+η-expansion is relatively easy to establish [6, 17], even for systems of dependent types, strong normalization of β-reduction+η-expansion has remained unaddressed thus far. In fact, it was conjectured by Di Cosmo and Ghani [10, 17] that β-reduction+η-expansion is strongly normalizing for the legal terms of the Calculus of Constructions. In the first part of the paper, we prove their conjecture by means of a model construction inspired from [25] and based on saturated sets.

The second part of this paper is concerned with normalization of β-reduction+η-expansion+algebraic reduction for the algebraic λ-cube, see e.g. [3, 8]. In contrast to the λ-cube, neither weak normalization results nor strong normalization results are known for systems of dependent types. Again, it was conjectured by Di Cosmo and Ghani [10] that strong normalization is a modular property of the algebraic λ-cube, i.e. that a system of the algebraic λ-cube is strongly normalizing w.r.t. β-reduction+η-expansion+algebraic reduction if its underlying term-rewriting system is strongly normalizing. In the second part of this paper, we solve their conjecture partially, under the extra assumption that the underlying term-rewriting system is non-duplicating. The proof is obtained by using ideas from [7] and modifying the model construction of the first part of the paper.

Related work

Much work has been devoted to extensionality in type systems so we shall only focus on systems of dependent types.

[1] Throughout the paper, we shall be concerned with the extensional versions of the λ-cube, in which the conversion rule uses $=_{\beta\eta}$. These versions are presented in [14] but differ from Barendregt's original presentation [4, 5], in which the conversion rule uses $=_\beta$. In order to avoid confusion, we refer to the latter presentation as the usual λ-cube, the usual Calculus of Constructions...

W. Thomas (Ed.): FOSSACS'99, LNCS 1578, pp. 90-104, 1999.

η-reduction The study of β-reduction+η-reduction in dependent type theories dates back to the early 70's, with Nederpelt's thesis [21]. Nederpelt showed that β-reduction+η-reduction is not confluent on the pseudo-terms of a dependently typed language *à la* Church, i.e. with typed λ-abstractions. Later, van Daalen [12] proved confluence of β-reduction+η-reduction for (the typed terms of) a language of the Automath family. More recently, Geuvers [14] and Salvesen [24] proved confluence of β-reduction+η-reduction for functional, normalizing Pure Type Systems [4, 5, 14]. Finally, the author [6] recently generalized Geuvers and Salvesen's results by showing that all Pure Type Systems have unique normal forms with respect to β-reduction+η-reduction. As for strong normalization, Geuvers [14] seems to be the standard proof of strong normalization for the extensional version of the Calculus of Constructions.

$\beta\eta$-long normal forms The existence of $\beta\eta$-long normal forms for the systems of the λ-cube was first shown by Dowek, Huet and Werner [13], who defined a non-standard induction principle to this end. The induction principle was later simplified and generalized by the author [6] who showed the uniqueness of $\beta\eta$-long normal forms for all Pure Type Systems and the existence of $\beta\eta$-long normal forms for most Pure Type Systems of interest.

η-expansion for dependent types For systems of dependent types, the notion of η-expansion was first studied by Ghani [17], who showed that unrestricted η-expansion is not normalizing for the Calculus of Constructions, introduced a restricted notion of η-expansion, and showed that β-reduction+η-expansion is confluent and weakly normalizing for the Calculus of Constructions. More recently, the author [6] generalized Ghani's results by showing that all Pure Type Systems have unique normal forms with respect to β-reduction+η-expansion. As for strong normalization, Joachimski [18] has shown that a Pure Type System is strongly normalizing for β-reduction+η-expansion if it is strongly normalizing for β-reduction. Unfortunately, Joachimski's notion of η-expansion is weaker than the notions of η-expansion that appear in the literature [6, 13, 17], and subsequently its underlying notion of $\beta\eta$-long normal form does not correspond to the well-established notion of [6, 13, 17]. We consider it a severe drawback.

Finally, Di Cosmo and Ghani [10] have considered η-expansion in the context of the algebraic λ-cube, in particular for the Algebraic Calculus of Constructions, and have shown that confluence is a modular property of the Algebraic Calculus of Constructions.

Organization of the paper

The paper is organized as follows: in Section 2, we review the definition of the λ-cube. In Section 3, we prove strong normalization of β-reduction+η-expansion for the systems of the λ-cube. In Section 4, we define the algebraic λ-cube with $\beta\eta$-conversion and prove strong normalization of β-reduction+η-expansion+algebraic reduction for the systems of the algebraic λ-cube. Finally, we conclude in Section 5.

Notation

We use standard notation and terminology from Abstract Rewriting Systems [19]. In particular, $\to_{ij}$ denotes the union of two relations $\to_i$ and $\to_j$, $\to_i^+$ denotes the transitive closure of $\to_i$, $\twoheadrightarrow_i$ denotes the reflexive-transitive closure of $\to_i$ and $=_i$ denotes the reflexive-symmetric-transitive closure of $\to_i$. Finally, the relation $\downarrow_i$ is defined by $a \downarrow_i b$ if there exists c such that $a \twoheadrightarrow_i c$ and $b \twoheadrightarrow_i c$.

An object a is a *i-normal form* if there is no b such that $a \to_i b$; the set of i-normal forms is denoted by $\mathsf{NF}(i)$. An object a is *i-normalizing* if there is some $b \in \mathsf{NF}(i)$ such that $a \twoheadrightarrow_i b$; the set of i-normalizing objects is denoted by $\mathsf{WN}(i)$. An object a is *i-strongly normalizing* if all reduction sequences starting from a are finite; the set of i-strongly normalizing objects is denoted by $\mathsf{SN}(i)$.

2 η-expansion in the λ-cube

2.1 The λ-cube

Throughout this paper, we let $\mathcal{S} = \{*, \Box\}$. Elements of $\mathcal{S}$ are called *sorts*. For technical convenience, we distinguish between object variables and constructor variables. This distinction, which originates from [14], yields a neat formulation of the Classification Lemma (Lemma 7).

Definition 1. *A* rule set *is a set* $\mathbf{S}$ *such that* $\mathbf{S} \subseteq \mathcal{S} \times \mathcal{S}$. *Elements of sets are called* rules.

Every rule set $\mathbf{S}$ yields a Pure Type System $\lambda\mathbf{S}$ as specified below.

Definition 2 (Pure Type Systems of the λ-cube).

1. *The set* $\mathcal{T}$ *of* pseudo-terms *is given by the abstract syntax*
$$\mathcal{T} = V^* \mid V^\Box \mid * \mid \Box \mid \mathcal{T}\mathcal{T} \mid \lambda V : \mathcal{T}.\mathcal{T} \mid \Pi V : \mathcal{T}.\mathcal{T}$$
where V^* *and* $V^\Box$ *are fixed, pairwise disjoint and countably infinite sets of* variables.
2. β-reduction $\to_\beta$ *is defined as the compatible closure of the contraction*
$$(\lambda x{:}A.\ M)\ N \to_\beta M\{x := N\}$$
where $.\{. := .\}$ *is the standard* substitution *operator.*
3. η-reduction $\to_\eta$ *is defined as the compatible closure of the contraction*
$$\lambda x{:}A.\ (M\ x) \to_\eta M$$
provided $x \notin \mathrm{FV}(M)$ *where* $\mathrm{FV}(M)$ *is the standard set of* free variables *of* M.
4. *The set* $\mathcal{G}$ *of* pseudo-contexts *is given by the abstract syntax*
$$\mathcal{G} = \langle\rangle \mid \mathcal{G}, V : \mathcal{T}$$
The domain of a context Γ *is* $\mathsf{dom}(\Gamma) = \{x \mid \exists t \in \mathcal{T}.\ x : t \in \Gamma\}$.
5. *A* judgment *is a triple* $\Gamma \vdash M : A$ *where* $\Gamma \in \mathcal{G}$ *and* $M, A \in \mathcal{T}$.
6. *The* derivability *relation* $\vdash$ *is given by the rules of Figure 1. If* $\Gamma \vdash M : A$ *is derivable, then* Γ, M *and* A *are* legal. *The set of legal contexts is denoted by* $\mathcal{H}$.

(axiom)	$\langle\rangle \vdash * : \Box$	
(start)	$\dfrac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A}$	if $x \in V^s \setminus \mathsf{dom}(\Gamma)$
(weakening)	$\dfrac{\Gamma \vdash A : B \quad \Gamma \vdash C : s}{\Gamma, x : C \vdash A : B}$	if $x \in V^s \setminus \mathsf{dom}(\Gamma)$ and $A \in V \cup \mathcal{S}$
(product)	$\dfrac{\Gamma \vdash A : s_1 \quad \Gamma, x : A \vdash B : s_2}{\Gamma \vdash (\Pi x{:}A.\ B) : s_2}$	if $(s_1, s_2) \in \mathcal{R}$
(application)	$\dfrac{\Gamma \vdash F : (\Pi x{:}A.\ B) \quad \Gamma \vdash a : A}{\Gamma \vdash F\ a : B\{x := a\}}$	
(abstraction)	$\dfrac{\Gamma, x : A \vdash b : B \quad \Gamma \vdash (\Pi x{:}A.\ B) : s}{\Gamma \vdash \lambda x{:}A.\ b : \Pi x{:}A.\ B}$	
(conversion)	$\dfrac{\Gamma \vdash A : B \quad \Gamma \vdash B' : s}{\Gamma \vdash A : B'}$	if $B =_{\beta\eta} B'$

Fig. 1. Rules for the λ-cube

The eight Pure Type Systems $\lambda\mathbf{S}$ depicted in Figure 2 are collectively known as the λ-*cube* and relate to well-known systems:

System	$\mathbf{S}$	Rule set			
Simply typed λ-calculus	$\to$	$(*,*)$			
Polymorphic typed λ-calculus	2	$(*,*)$	$(\Box,*)$		
Logical Frameworks	P	$(*,*)$		$(*,\Box)$	
	$P2$	$(*,*)$	$(\Box,*)$	$(*,\Box)$	
	$\underline{\omega}$	$(*,*)$			$(\Box,\Box)$
Higher-order typed λ-calculus	ω	$(*,*)$	$(\Box,*)$		$(\Box,\Box)$
	$P\underline{\omega}$	$(*,*)$		$(*,\Box)$	$(\Box,\Box)$
Calculus of Constructions	$P\omega = C$	$(*,*)$	$(\Box,*)$	$(*,\Box)$	$(\Box,\Box)$

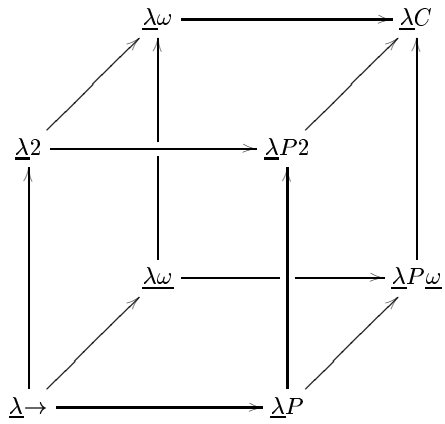


Fig. 2. The λ-cube

2.2 η-expansion

Defining η-expansion for systems of dependent types is not straightforward. As pointed out by Ghani [17], full η-expansion $\to_{\overline{\eta}_f(\Gamma)}$, as defined by the rule

$$M \to_{\overline{\eta}_f(\Gamma)} \lambda x{:}A.\ (M\ x) \qquad \text{if } \Gamma \vdash M : \Pi x{:}A.\ B \text{ and } M \text{ not a } \lambda\text{-abstraction}$$

leads to infinite reduction sequences. Indeed, let $\Gamma = X : *, x : X \to X$ and let $B(y) = (\lambda z{:}X \to X.\ X)\ y$. Then

$$\begin{aligned} x &\to_{\overline{\eta}_f(\Gamma)} \lambda z{:}B(x).\ (x\ z) \\ &\to_{\overline{\eta}_f(\Gamma)} \lambda z{:}B(\lambda z{:}B(x).\ (x\ z)).\ (x\ z) \\ &\to_{\overline{\eta}_f(\Gamma)} \cdots \end{aligned}$$

In order to rule out such infinite reduction sequences, Ghani [17] proposes to impose the extra constraint that A is in $\beta\eta$-long normal form, where the notion of being $\beta\eta$-long normal form is defined by induction on the structure of terms.

Definition 3 (Ghani [17]). *Let $\Gamma \in \mathcal{H}$. A term $M \in \mathcal{N}$ is a* $\beta\eta$-long normal form in context Γ, *written $\xi_\Gamma(M)$, if M is legal in Γ and one of the following conditions holds:*

1. $M \in \mathcal{S}$;
2. $M = \Pi x{:}B.\ C$, $\xi_\Gamma(B)$ *and* $\xi_{\Gamma,x:B}(C)$;
3. $M = \lambda x{:}B.\ N$, $\xi_\Gamma(B)$ *and* $\xi_{\Gamma,x:B}(N)$;
4. $M = x\ P_1\ \ldots\ P_n$, $\Gamma \vdash M : A$ *for some* $A \in \mathcal{C} \cup \mathcal{S}$ *and* $\xi_\Gamma(P_i)$ *for* $i = 1, \ldots, n$.

The set of $\beta\eta$-long normal form in context Γ is denoted Ξ_Γ.

The above definition is also implicitly present in [13].

Definition 4 (Ghani [17]). Restricted η-expansion (in context Γ) $\to_{\overline{\eta}_r(\Gamma)}$ *is defined by the rule*

$$M \to_{\overline{\eta}_r(\Gamma)} \lambda x{:}A.\ (M\ x) \qquad \textit{if } \Gamma \vdash M : \Pi x{:}A.\ B \textit{ and } M \textit{ not a } \lambda\textit{-abstraction and } \xi_\Gamma(A)$$

and by the compatibility clauses of Figure 3.

As argued in [6], this definition is somewhat contrived because:

1. one needs to define $\beta\eta$-long normal forms before defining η-expansion;
2. one can only deduce that x is not in $\to_{\overline{\eta}(\Gamma,x:A\to A)}$-normal form (where A is an arbitrary type) if one knows that $\xi_\Gamma(B)$ for some B such that $A =_{\beta\eta} B$.

Based on the above observations, the author [6] suggested that the definition of η-expansion may be simplified by requiring that $A \in \mathsf{NF}(\beta)$. Note how Ghani's counter-example does not apply to such a notion of η-expansion.

Definition 5 ([6]). *η-expansion $\to_{\overline{\eta}(\Gamma)}$ is defined by the rule*

$$M \to_{\overline{\eta}(\Gamma)} \lambda x{:}A.\ (M\ x) \qquad \text{if } \Gamma \vdash M : \Pi x{:}A.\ B \text{ and } M \text{ not a } \lambda\text{-abstraction and } A \in \mathsf{NF}(\beta)$$

and by the compatibility clauses of Figure 3.

$$\begin{array}{rl}
M \to_{\overline{\eta}(\Gamma)} N \Rightarrow M\ P \to_{\overline{\eta}(\Gamma)} N\ P & \text{if } N \neq \lambda x{:}B.\ (M\ x) \\
M \to_{\overline{\eta}(\Gamma)} N \Rightarrow P\ M \to_{\overline{\eta}(\Gamma)} P\ N & \\
M \to_{\overline{\eta}(\Gamma)} N \Rightarrow \lambda x{:}M.\ P \to_{\overline{\eta}(\Gamma)} \lambda x{:}N.\ P & \\
M \to_{\overline{\eta}(\Gamma)} N \Rightarrow \Pi x{:}M.\ P \to_{\overline{\eta}(\Gamma)} \Pi x{:}N.\ P & \\
M \to_{\overline{\eta}(\Gamma, x:A)} N \Rightarrow \lambda x{:}P.\ M \to_{\overline{\eta}(\Gamma)} \lambda x{:}P.\ N & \\
M \to_{\overline{\eta}(\Gamma, x:A)} N \Rightarrow \Pi x{:}P.\ M \to_{\overline{\eta}(\Gamma)} \Pi x{:}P.\ N &
\end{array}$$

Fig. 3. Compatibility rules for η-expansion

A few words are in order to justify our definition:

1. In the definition of η-expansion it is implicitly assumed that x is fresh;
2. η-expansion is taken to be a compatible relation as it is folklore that η-expanding the first argument of an application would lead to the loop

$$M\ N \to_{\overline{\eta}(\Gamma)} (\lambda x{:}A.\ M\ x)\ N \to_{\beta} M\ N$$

Similarly one cannot η-expand λ-abstractions as it would lead to the loop

$$\lambda x{:}A.\ M \to_{\overline{\eta}(\Gamma)} \lambda y{:}A.\ (\lambda x{:}A.\ M)\ y \to_{\beta} \lambda x{:}A.\ M$$

It is immediate to see that $\to_{\overline{\eta}}$ is a generalization of $\to_{\overline{\eta}_r}$.

Lemma 1. *If $M \to_{\overline{\eta}_r(\Gamma)} N$ then $M \to_{\overline{\eta}(\Gamma)} N$.*

3 Strong normalization of β-reduction+η-expansion for the λ-cube

In this section, we prove that β-reduction+η-expansion is strongly normalizing on the legal terms of the λ-cube.

Analysis Many proofs of strong normalization for the Calculus of Constructions (see e.g. [1, 15, 20, 25]) are achieved through the definition of a suitable model construction, in which types are interpreted as specific sets of strongly normalizing λ-terms, called *saturated sets*. Our proof proceeds in a similar way and is largely inspired from [25]. There are some minor differences however:

1. the notion of saturated set is modified so as to account for the peculiarities of β-reduction+η-expansion, typically of not being substitutive;
2. in the proof of the soundness theorem, the induction step for the abstraction rule is modified so as to take into account the characterization of strongly normalizing terms w.r.t. β-reduction+η-expansion.

The proof is not fully satisfactory, as one would hope to have a modular proof of strong normalization of β-reduction+η-expansion for the λ-cube. In other words, one would like to reduce (in a weak system of arithmetic) strong normalization of β-reduction+η-expansion for the λ-cube to strong normalization of β-reduction for the usual λ-cube (without having to prove the latter). One possibility, which needs to be further explored, is to extend the technique of simulating-expansions-without-expansions [11, 26] to the Calculus of Constructions. However, this is a challenging problem which remains unresolved so far.

Prerequisites The proof is self-contained, except that we make use of the following results.

Theorem 1 (Geuvers [14]). *For the systems of the λ-cube:*

1. *$\to_{\beta\eta}$ is confluent on legal terms:* $(\Gamma \vdash M : A \wedge \Gamma \vdash N : A \wedge M =_{\beta\eta} N) \quad \Rightarrow \quad M \downarrow_{\beta\eta} N$
2. *$\to_{\beta\eta}$ enjoys subject reduction:* $(\Gamma \vdash M : A \wedge M \to_{\beta\eta} N) \quad \Rightarrow \quad \Gamma \vdash N : A$

For the point of view of the proof, the most important consequence of confluence and subject reduction is that the systems of the λ-cube are *sound w.r.t. convertibility* [16], i.e. that every two convertible types or kinds are convertible through well-typed terms. Soundness w.r.t. convertibility is required to prove that the interpretation of types $\langle\!\langle . \rangle\!\rangle$ preserves convertibility.

Theorem 2 ([6]). *For the systems of the λ-cube:*

1. *$\to_{\overline{\eta}(\Gamma)}$ is strongly normalizing on Γ-legal terms;*
2. *$\to_{\overline{\eta}(\Gamma)}$ preserves β-normal forms.*

It follows:

Lemma 2. *If $\Gamma \vdash M : A$ and $M \in \mathsf{NF}(\beta)$ then $M \in \mathsf{SN}(\beta\overline{\eta})$.*

Proof. By the above theorem, every reduction sequence starting from $M \in \mathsf{NF}(\beta)$ is a $\overline{\eta}$-reduction sequence and hence terminates.

Finally, we need the following characterization of strongly normalizing terms.

Lemma 3. *Assume $(\lambda x{:}A.\, M)\ N\ P_1\ \ldots\ P_n \in \mathsf{SN}(\beta\overline{\eta}(\Gamma))$. If $A, N \in \mathsf{SN}(\beta\overline{\eta}(\Gamma))$ and for every $N' \in \mathcal{T}$ such that $N \twoheadrightarrow_{\beta\overline{\eta}(\Gamma)} N'$, $M\{x := N'\}\ P_1\ \ldots\ P_n \in \mathsf{SN}(\beta\overline{\eta}(\Gamma))$ then $(\lambda x{:}A.\, M)\ N\ P_1\ \ldots\ P_n \in \mathsf{SN}(\beta\overline{\eta}(\Gamma))$.*

Proof. By an analysis of the possible reduction sequences starting from $(\lambda x{:}A.\, M)\ N\ P_1\ \ldots\ P_n$. Necessarily an infinite sequence will be of the form

$$\begin{aligned}(\lambda x{:}A.\, M)\ N\ P_1\ \ldots\ P_n &\to_{\beta\overline{\eta}} \lambda z_1 : B_1.\ \ldots . \lambda z_m : B_m.\ (\lambda x{:}A'.\, M')\ N'\ P'_1\ \ldots\ P'_n\ z'_1\ \ldots\ z'_m \\ &\to_{\beta\overline{\eta}} \lambda z_1 : B_1.\ \ldots . \lambda z_m : B_m.\ (M'\{x := N'\})\ P'_1\ \ldots\ P'_n\ z'_1\ \ldots\ z'_m \\ &\twoheadrightarrow_{\beta\overline{\eta}} \ldots\end{aligned}$$

with $B_1, \ldots, B_m, z'_1, \ldots, z'_m \in \mathsf{NF}(\beta) \subseteq \mathsf{SN}(\beta\overline{\eta}(\Gamma))$ and $M', N', P'_1, \ldots, P'_n \in \mathsf{SN}(\beta\overline{\eta}(\Gamma))$. By assumption $(M\{x := N'\})\ P_1\ \ldots\ P_n \in \mathsf{SN}(\beta\overline{\eta}(\Gamma))$. Now

$$(M\{x := N'\})\ P_1\ \ldots\ P_n \twoheadrightarrow_{\beta\overline{\eta}} \lambda z_1 : B_1.\ \ldots . \lambda z_n : B_n.\ (M'\{x := N'\})\ P'_1\ \ldots\ P'_n\ z'_1\ \ldots\ z'_n$$

so we conclude that the reduction sequence cannot be infinite, a contradiction.

Lemma 3 is used crucially in proving the soundness of the model construction, more specifically in the case of the (abstraction) rule.

3.1 Environments

The notion of environment conveniently captures the notion of infinite context.

Definition 6 (Pottinger [22]). *An* environment *$\mathcal{E}$ is an infinite sequence of variable declarations $\mathcal{E} = x_1 : E_1, x_2 : E_2, \ldots$ such that for every $i > 0$*

- *$\mathcal{E}^i = x_1 : E_1, \ldots, x_i : E_i$ is a legal context;*
- *if $\mathcal{E}^i \vdash A : s$ and $s \in S$ then there exists infinitely many k s.t. $E_k = A$.*

We write $\mathcal{E} \vdash M : A$ whenever there exists $i \geq 0$ s.t. $\mathcal{E}^i \vdash M : A$.

Note that the definition of environment implicitly embeds our variable convention, i.e. for every $(x : A) \in \mathcal{E}$, $\mathcal{E} \vdash A : s$ iff $x \in V^s$.

Lemma 4 (Pottinger [22]). *Every context can be extended to an environment.*

Environments are convenient for the purpose of strong normalization proofs, because (up to convertibility) an expression has at most one type in an environment and because η-expansion is stable under thinning, i.e. for $i \leq j$, we have $M \to_{\overline{\eta}(\mathcal{E}^i)} N \quad \Rightarrow \quad M \to_{\overline{\eta}(\mathcal{E}^j)} N$. In the sequel, we therefore omit $\mathcal{E}^i$ and write $M \to_{\overline{\eta}} N$ when there is no risk of confusion.

3.2 Saturated sets

Throughout this section, we let $\mathcal{E}$ be a fixed environment. We say A is an *$\mathcal{E}$-type* (or simply a type) if $\mathcal{E} \vdash A : s$ with $s \in \mathcal{S}$. Moreover, we let $\mathsf{SN}(A)$ denote the set of $\beta\overline{\eta}$-strongly normalizing terms of type A in $\mathcal{E}$. Finally we let $V^A = \{x \in V \mid \mathcal{E} \vdash x : A\}$.

Definition 7 (Base terms). *The set* $\mathsf{Base}(A)$ *of* A-base terms *is defined inductively as follows:*

1. $V^A \subseteq \mathsf{Base}(A)$,
2. *if* $a \in \mathsf{Base}(\Pi x{:}B.\ A)$ *and* $b \in \mathsf{SN}(B)$, *then* $a\ b \in \mathsf{Base}(A\{x := b\})$.

Key reduction, as defined below, is a specific strategy reducing terms to weak-head normal form.

Definition 8 (Key reduction). Key-reduction $\to_k$ *is the smallest reduction relation satisfying*

$$(\lambda x{:}B.\ a)\ b\ c_1\ \ldots\ c_n \to_k a\{x := b\}\ \ c_1\ \ldots\ c_n$$

Note that we do not *take the compatible closure of the above rule.*

Note that, for every $b \in \mathcal{T}$ there exists at most one $b' \in \mathcal{T}$ such that $b \to_k b'$.

Definition 9 (Saturated set). *Let A be a type. A set $X \subseteq \mathsf{SN}(A)$ is* A-saturated *if* $\mathsf{Base}(A) \subseteq X$ *and X is closed under the rules:*

$$\frac{b \in X \qquad b \to_{\beta\overline{\eta}} b'}{b' \in X} \qquad\qquad \frac{b' \in X \qquad b \in \mathsf{SN}(A) \qquad b \to_k b'}{b \in X}$$

The collection of A-saturated sets is denoted by $\mathbf{SAT}(A)$.

The extra clause in the definition of saturated sets is due to Lemma 3.[2] In order for the definition of saturated sets to make sense, we need to prove base terms are strongly normalizing; otherwise there would not be any saturated set.

Lemma 5. *For every type A,* $\mathsf{Base}(A) \subseteq \mathsf{SN}(A)$.

Proof. Similar to that of Lemma 3.

We conclude this subsection by stating some fundamental closure properties of saturated sets.

Lemma 6.

1. $\mathsf{SN}(A) \in \mathbf{SAT}(A)$.
2. *If* $X_i \in \mathbf{SAT}(A)$ *for every* $i \in I$ *and* $I \neq \emptyset$ *then* $\bigcap_{i\in I} X_i \in \mathbf{SAT}(A)$.
3. *If* $X \in \mathbf{SAT}(A)$ *and for every* $x \in X$, $Y_x \in \mathbf{SAT}(B)$ *then* $\prod x \in X.\ Y_x \in \mathbf{SAT}(\Pi x{:}A.\ B)$, *where*

$$\prod x \in X.\ Y_x = \{M \in \mathsf{SN}(\Pi x{:}A.\ B) \mid \forall N \in X.\ M\ N \in Y_N\}$$

3.3 The Classification Lemma

It is convenient to base the model construction on a classification lemma, which stratifies the different classes of pseudo-terms. The formulation used below is a variant of Geuvers' Classification Lemma, see e.g. [14].

Definition 10 (Pseudo-objects, pseudo-constructors and pseudo-kinds). *The classes $\mathcal{O}$, $\mathcal{C}$ and $\mathcal{K}$ of* pseudo-objects, pseudo-constructors *and* pseudo-kinds *are given by the abstract syntaxes*

$$\begin{aligned}
\mathcal{O} &= V^* \mid \lambda V^*{:}\mathcal{C}.\ \mathcal{O} \mid \lambda V^\Box{:}\mathcal{K}.\ \mathcal{O} \mid \mathcal{O}\ \mathcal{O} \mid \mathcal{O}\ \mathcal{C} \\
\mathcal{C} &= V^\Box \mid \Pi V{:}\mathcal{C}.\ \mathcal{C} \mid \Pi V{:}\mathcal{K}.\ \mathcal{C} \mid \lambda V^*{:}\mathcal{C}.\ \mathcal{C} \mid \lambda V^\Box{:}\mathcal{K}.\ \mathcal{C} \mid \mathcal{C}\ \mathcal{C} \mid \mathcal{C}\ \mathcal{O} \\
\mathcal{K} &= * \mid \Pi V{:}\mathcal{C}.\ \mathcal{K} \mid \Pi V{:}\mathcal{K}.\ \mathcal{K}
\end{aligned}$$

We write $M \sim N$ if $M \in \mathcal{D}$ and $N \in \mathcal{D}$ for some $\mathcal{D} \in \{\mathcal{O}, \mathcal{C}, \mathcal{K}\}$.

We have:

Lemma 7 (Classification Lemma). *If $\mathcal{E} \vdash M : A$ then exactly one of the three conditions hold: (1) $M \in \mathcal{O}$ and $A \in \mathcal{C}$ (2) $M \in \mathcal{C}$ and $A \in \mathcal{K}$ (3) $M \in \mathcal{K}$ and $A = \Box$.*

[2] Compare with the similar result for β-reduction: if $A, N, M\{x := N\}\ P_1\ \ldots\ P_n \in \mathsf{SN}(\beta)$ then $(\lambda x : A.\ M)\ N\ P_1\ \ldots\ P_n \in \mathsf{SN}(\beta)$. Here we need to consider all reducts of N and hence the first clause.

3.4 The model construction

The first step of the model construction is to define an interpretation which assigns to every pseudo-term the set of possible interpretations for its inhabitants. Below we let $\mathbf{1}$ denote an arbitrary one element set and write $B : K$ instead of $B \in \{M \in \mathcal{T} \mid \mathcal{E} \vdash M : K\}$.

Definition 11 (Possible values). *The map $a : \mathcal{T} \to \mathbf{Set}$ and the relation $\doteq_A \subseteq a(A) \times a(A)$ (where $A \in \mathcal{T}$) are defined by*

$$a(A) = \begin{cases} \mathbf{SAT}(A) & \text{if } \mathcal{E} \vdash A : * \text{ or } \mathcal{E} \vdash A : \Box \\ \{(f_B)_{B:K_1} \mid \forall B : K_1. f_B \in a(B) \to a(A\ B) \\ \qquad \wedge\ (\forall B, B' : K_1.\ B =_{\beta\eta} B' \Rightarrow f_B =_A f_{B'})\} & \text{if } \mathcal{E} \vdash A : \Pi\alpha{:}K_1.\ K_2 \text{ and } K_2 \in \mathcal{K} \\ \{\mathbf{1}\} & \text{otherwise} \end{cases}$$

and

$$x \doteq_A y = \begin{cases} x = y & \text{if } \mathcal{E} \vdash A : * \text{ or } \mathcal{E} \vdash A : \Box \\ \forall B : K_1.\ \forall z \in a(B).\ x_B\ z\ \doteq_{K_2\{x:=B\}}\ y_B\ z & \text{if } \mathcal{E} \vdash A : \Pi x{:}K_1.\ K_2 \text{ and } K_2 \in \mathcal{K} \\ \mathsf{true} & \text{otherwise} \end{cases}$$

In the sequel, we let $\mathbf{A} = \bigcup_{M \in \mathcal{T}} a(M)$.

The interpretation preserves convertibility.

Lemma 8. *If A and B are legal terms in $\mathcal{E}$ and $A =_{\beta\eta} B$ then $a(A) = a(B)$.*

Proof. By induction on the type of A and B.

The next interpretation maps types and kinds to saturated sets.

Definition 12.

1. *A* valuation *is a pair (ρ, ζ) with $\rho : V \to \mathcal{T}$ and $\zeta : V \to \mathbf{A}$.*
2. *The extension $(\!|.|\!)_\rho : \mathcal{T} \to \mathcal{T}$ of ρ is defined as the unique capture-avoiding substitution extending ρ.*
3. *The extension $\langle\!\langle . \rangle\!\rangle_{(\rho,\zeta)} : \mathcal{T} \to \mathbf{A}$ of ζ is defined as follows:*

$$\begin{array}{rll} \langle\!\langle x \rangle\!\rangle_{(\rho,\zeta)} &= \zeta(x) & \text{if } x \in V \\ \langle\!\langle s \rangle\!\rangle_{(\rho,\zeta)} &= \mathsf{SN}(s) & \text{if } s \in \mathcal{S} \\ \langle\!\langle \Pi x : A.B \rangle\!\rangle_{(\rho,\zeta)} &= \prod P \in \langle\!\langle A \rangle\!\rangle_{(\rho,\zeta)}.\mathsf{int}_{(\rho,\zeta)}(B, P) & \\ \langle\!\langle M\ N \rangle\!\rangle_{(\rho,\zeta)} &= (\langle\!\langle M \rangle\!\rangle_{(\rho,\zeta)})_{(\!|N|\!)_\rho}\ \langle\!\langle N \rangle\!\rangle_{(\rho,\zeta)} & \text{if } (\!|MN|\!)_\rho \in \mathcal{C} \\ \langle\!\langle \lambda x : A.M \rangle\!\rangle_{(\rho,\zeta)} &= (\mathsf{fun}_{(\rho,\zeta)}(M, P))_{P:(\!|A|\!)_\rho} & \text{if } (\!|\lambda x : A.M|\!)_\rho \in \mathcal{C} \\ \langle\!\langle M \rangle\!\rangle_{(\rho,\zeta)} &= \mathbf{1} & \text{otherwise} \end{array}$$

where

$$\mathsf{int}_{(\rho,\zeta)}(B, P) = \bigcap_{c \in a(P)} \langle\!\langle B \rangle\!\rangle_{(\rho(x:=P),\zeta(x:=c))}$$

$$\mathsf{fun}_{(\rho,\zeta)}(M, P) = \lambda\!\!\lambda c \in a(P).\langle\!\langle M \rangle\!\rangle_{(\rho(x:=P),\zeta(x:=c))}$$

and $\lambda\!\!\lambda$ denotes the set-theoretic abstraction.

In order to prove strong normalization one first needs to show that, under suitable conditions, valuations preserve typability and that the $\langle\!\langle . \rangle\!\rangle$-interpretation of a term is an element of the possible values of its $(\!|.|\!)$-interpretation.

Definition 13.

1. *A valuation (ρ, ζ)* satisfies *$M : A$, written $(\rho, \zeta) \models M : A$ if*
   (a) $\mathcal{E} \vdash (\!|M|\!)_\rho : (\!|A|\!)_\rho$
   (b) $\langle\!\langle M \rangle\!\rangle_{(\rho,\zeta)} \in a((\!|M|\!)_\rho)$.

2. *A valuation* (ρ, ζ) satisfies Γ, *written* $(\rho, \zeta) \models \Gamma$, *where* Γ *is a pseudo-context, if* $(\rho, \zeta) \models x : A$ *for every* $(x : A) \in \Gamma$.
3. *A judgment* $\Gamma \vdash M : A$ *is* valid, *written* $\Gamma \models M : A$, *if* $(\rho, \zeta) \models M : A$ *for every valuation* (ρ, ζ) *s.t.* $(\rho, \zeta) \models \Gamma$.

We need to start with a few technical lemmas. First of all, we show that the interpretation is well-behaved w.r.t. substitution.

Lemma 9. *Assume that* $\Gamma_1, x : B, \Gamma_2 \vdash M : A$ *and* $\Gamma_1 \vdash N : B$. *If* $(\rho, \zeta) \models \Gamma_1, x : B, \Gamma_2$ *then* $\langle\!\langle M\{x := N\} \rangle\!\rangle_{(\rho,\zeta)} = \langle\!\langle M \rangle\!\rangle_{(\rho(x:=(\!|N|\!)_\rho), \zeta(x:=\langle\!\langle N \rangle\!\rangle_{(\rho,\zeta)}))}$.

Proof. By induction on the structure of M.

We also need to show that valuations can always be extended in such a way that satisfaction is preserved.

Lemma 10. *Let* $\Gamma, x : A$ *be a legal context and let* (ρ, ζ) *be a valuation s.t.* $(\rho, \zeta) \models \Gamma, A : s$.

1. $(\rho(x := N), \zeta(x := c)) \models \Gamma, x : A$ *for every* $N \in \mathcal{T}$ *s.t.* $\mathcal{E} \vdash N : (\!|A|\!)_\rho$ *and* $c \in a(N)$.
2. *There exists* $z \in V$ *s.t.* $(\rho(x := z), \zeta(x := c)) \models \Gamma, x : A$ *for every* $c \in a(z)$.

Proof. By definition of $\models$.

The interpretation is preserved under conversion.

Lemma 11.

1. *Assume* $\Gamma \vdash M, N : A$ *and* $M =_{\beta\eta} N$. *Then* $\langle\!\langle M \rangle\!\rangle_{(\rho,\zeta)} = \langle\!\langle N \rangle\!\rangle_{(\rho,\zeta)}$ *for every valuation* (ρ, ζ) *such that* $(\rho, \zeta) \models \Gamma$.
2. *Assume* $\Gamma \vdash M : A$. *Then* $\langle\!\langle M \rangle\!\rangle_{(\rho,\zeta)} = \langle\!\langle M \rangle\!\rangle_{(\rho,\xi)}$ *for every valuations* (ρ, ζ) *and* (ρ', ζ) *such that* $(\rho, \zeta) \models \Gamma$, $(\rho', \zeta) \models \Gamma$ *and* $\rho =_{\beta\eta} \rho'$.[3]

Proof. (1) By confluence and subject reduction of $\to_{\beta\eta}$, it is sufficient to consider the case where $M \to_{\beta\eta} N$. The proof proceeds by induction on the structure of M. (2) Similarly, it is sufficient to consider the case where $\rho \to_{\beta\eta} \rho'$. The proof proceeds by induction on the structure of M.

We can now prove the following result.

Proposition 1. *If* $\Gamma \vdash M : A$ *then* $\Gamma \models M : A$.

Proof. By induction on the structure of derivations. We treat three cases:

1. assume that $\Gamma \vdash M : A$ is derived using (product). Then $A = s_2$ and $M = \Pi x{:}B.\, C$. Moreover the last rule of the derivation is of the form

$$\frac{\Gamma \vdash B : s_1 \qquad \Gamma, x : B \vdash C : s_2}{\Gamma \vdash \Pi x{:}B.\, C : s_2}$$

   Assume $(\rho, \zeta) \models \Gamma$. To show $(\rho, \zeta) \models \Pi x{:}B.\, C : s_2$.
   (a) To prove $\mathcal{E} \vdash (\!|\Pi x{:}B.\, C|\!)_\rho : s_2$. By induction hypothesis, $(\rho, \zeta) \models B : s_1$. In particular $\mathcal{E} \vdash (\!|B|\!)_\rho : s_1$. Moreover there exists by Lemma 10 a variable z s.t. for every $c \in a(z)$,

$$(\rho(x := z), \zeta(x := c)) \models \Gamma, x : B$$

   By induction hypothesis, $\mathcal{E} \vdash (\!|C|\!)_{\rho(x:=z)} : s_2$. We conclude by (product).

[3] $\to_{\beta\eta}$ is extended to valuations in the obvious way, i.e. $\rho \to_{\beta\eta} \rho'$ if there exists $x \in V$ such that $\rho(x) \to_{\beta\eta} \rho'(x)$ and $\rho(y) = \rho'(y)$ if $x \neq y$. Then $=_{\beta\eta}$ is the reflexive-symmetric-transitive closure of $\to_{\beta\eta}$.

(b) To prove $\langle\!\langle \Pi x{:}B.\ C \rangle\!\rangle_{(\rho,\zeta)} \in a((\!| \Pi x{:}B.\ C |\!)_\rho)$. Note that it is enough to show that

$$\langle\!\langle \Pi x{:}B.\ C \rangle\!\rangle_{(\rho,\zeta)} \in \mathbf{SAT}((\!| \Pi x{:}B.\ C |\!)_\rho)$$

By induction hypothesis, $\langle\!\langle B \rangle\!\rangle_{(\rho,\zeta)} \in \mathbf{SAT}((\!| B |\!)_\rho)$ and

$$\forall N \in \mathcal{T}.\ \forall c \in a(N).\ (\rho(x := N), \zeta(x := c)) \models \Gamma, x : B \ \Rightarrow\ \langle\!\langle C \rangle\!\rangle_{(\rho(x:=N),\zeta(x:=c))} \in \mathbf{SAT}((\!| C |\!)_{\rho(x:=N)})$$

By Lemma 10, $\forall N \in \mathcal{T}.\ \forall c \in a(N).\ \mathcal{E} \vdash N : (\!| B |\!)_\rho \ \Rightarrow\ (\rho(x := N), \zeta(x := c)) \models \Gamma, x : B$ and hence $\forall N \in \mathcal{T}.\ \forall c \in a(N).\ \mathcal{E} \vdash N : (\!| B |\!)_\rho \ \Rightarrow\ \langle\!\langle C \rangle\!\rangle_{(\rho(x:=N),\zeta(x:=c))} \in \mathbf{SAT}((\!| C |\!)_{\rho(x:=N)})$. By part 2 of Lemma 6,

$$\forall N \in \mathcal{T}.\ \mathcal{E} \vdash N : (\!| B |\!)_\rho \ \Rightarrow\ \bigcap_{c \in a(N)} \langle\!\langle C \rangle\!\rangle_{\zeta(x:=c)\rho(x:=N)} \in \mathbf{SAT}((\!| C |\!)_{\rho(x:=N)})$$

We conclude by part 3 of Lemma 6.

2. assume that $\Gamma \vdash M : A$ is derived using (application). Then $M = M_1\, M_2$, $A = C\{x := M_2\}$ and the last rule of the derivation is

$$\frac{\Gamma \vdash M_1 : (\Pi x{:}B.\ C) \qquad \Gamma \vdash M_2 : B}{\Gamma \vdash M_1\ M_2 : C\{x := M_2\}}$$

Assume $(\rho, \zeta) \models \Gamma$. To show $(\rho, \zeta) \models M_1\, M_2 : C\{x := M_2\}$.

(a) to prove $\mathcal{E} \vdash (\!| M_1\ M_2 |\!)_\rho : (\!| C\{x := M_2\} |\!)_\rho$. It follows directly from the induction hypothesis.

(b) to prove that $\langle\!\langle M_1\ M_2 \rangle\!\rangle_{(\rho,\zeta)} \in a((\!| M_1\ M_2 |\!)_\rho)$. There are two cases to distinguish:

i. If $(\!| M_1\ M_2 |\!)_\rho \notin \mathcal{C}$, then $a((\!| M_1\ M_2 |\!)_\rho) = \{\mathbf{1}\}$ and $\langle\!\langle M_1\ M_2 \rangle\!\rangle_{(\rho,\zeta)} = \mathbf{1}$, so we are done.

ii. If $(\!| M |\!)_\rho \in \mathcal{C}$ then by induction hypothesis, $\langle\!\langle M_1 \rangle\!\rangle_{(\rho,\zeta)} \in a((\!| M_1 |\!)_\rho)$ and $\langle\!\langle M_2 \rangle\!\rangle_{(\rho,\zeta)} \in a((\!| M_2 |\!)_\rho)$. Now $(\!| M_1 |\!)_\rho \in \mathcal{C}$ and hence $(\langle\!\langle M_1 \rangle\!\rangle_{(\rho,\zeta)})_{(\!| M_2 |\!)_\rho} \langle\!\langle M_2 \rangle\!\rangle_{(\rho,\zeta)} \in a((\!| M_1\ M_2 |\!)_\rho)$ and we are done.

3. assume that $\Gamma \vdash M : A$ is derived using (abstraction). Necessarily $M = \lambda x{:}B.\ N$, $A = \Pi x{:}B.\ C$ and the last rule of the derivation is

$$\frac{\Gamma, x : B \vdash N : C \qquad \Gamma \vdash (\Pi x{:}B.\ C) : s}{\Gamma \vdash \lambda x{:}B.\ N : \Pi x{:}B.\ C}$$

Assume $(\rho, \zeta) \models \Gamma$. To show $(\rho, \zeta) \models \lambda x{:}B.\ N : \Pi x{:}B.\ C$.

(a) To prove $\mathcal{E} \vdash (\!| \lambda x{:}B.\ N |\!)_\rho : (\!| \Pi x{:}B.\ C |\!)_\rho$. By Lemma 10, there exists $z \in V$ s.t. for every $c \in a(z)$,

$$(\rho(x := z), \zeta(x := c)) \models \Gamma, x : B$$

By induction hypothesis,

$$\mathcal{E} \vdash (\!| N |\!)_{\rho(x:=z)} : (\!| C |\!)_{\rho(x:=z)}$$
$$\mathcal{E} \vdash (\!| \Pi x{:}B.\ C |\!)_\rho : s$$

We conclude by (abstraction).

(b) To prove that $\langle\!\langle \lambda x{:}B.\ N \rangle\!\rangle_{(\rho,\zeta)} \in a((\!| \lambda x{:}B.\ N |\!)_\rho)$. There are two cases to distinguish:

i. If $(\!| \lambda x{:}B.\ N |\!)_\rho \notin \mathcal{C}$, then $a((\!| \lambda x{:}B.\ N |\!)_\rho) = \{\mathbf{1}\}$ and $\langle\!\langle \lambda x{:}B.\ N \rangle\!\rangle_{(\rho,\zeta)} = \mathbf{1}$, so we are done.

ii. If $(\!| \lambda x{:}B.\ N |\!)_\rho \in \mathcal{C}$, then we have to show that for every P such that $\mathcal{E} \vdash P : (\!| B |\!)_\rho$:

(α) $(\langle\!\langle \lambda x{:}B.\ N \rangle\!\rangle_{(\rho,\zeta)})_P\ Q \in a((\!| \lambda x{:}B.\ N |\!)_\rho\ P)$ for every $Q \in a(P)$;

(β) $(\langle\!\langle \lambda x{:}B.\ N \rangle\!\rangle_{(\rho,\zeta)})_P = (\langle\!\langle \lambda x{:}B.\ N \rangle\!\rangle_{(\rho,\zeta)})_{P'}$ for every $P' \in \mathcal{T}$ legal in $\mathcal{E}$ and s.t. $P =_{\beta\eta} P'$.

By definition, $(\langle\!\langle \lambda x{:}B.\ N \rangle\!\rangle_{(\rho,\zeta)})_P\ Q = \langle\!\langle N \rangle\!\rangle_{(\rho(x:=P),\zeta(x:=Q))}$. We conclude the proof of (α) by Lemma 10 and induction hypothesis. As for (β), we conclude by Lemma 11.

We now turn to soundness, which states that under suitable conditions, $(\!| M |\!)_\rho \in \langle\!\langle A \rangle\!\rangle_{(\rho,\zeta)}$ whenever $\Gamma \vdash M : A$. We begin with some definitions and preliminary results.

Definition 14.

1. *A valuation* (ρ, ζ) semantically entails $M : A$, *written* $(\rho, \zeta) \models^s M : A$, *if* $(\rho, \zeta) \models M : A$ *and* $(\!| M |\!)_\rho \in a(\langle\!\langle A \rangle\!\rangle_{(\rho,\zeta)})$.

2. *A valuation* (ρ,ζ) semantically entails Γ, *written* $(\rho,\zeta) \models^s \Gamma$, *where* Γ *is a pseudo-context, if* $(\rho,\zeta) \models^s x : A$ *for every* $(x : A) \in \Gamma$.
3. *A judgment* $\Gamma \vdash M : A$ *is* sound, *written* $\Gamma \models M : A$, *if* $(\rho,\zeta) \models^s M : A$ *for every valuation* (ρ,ζ) *s.t.* $(\rho,\zeta) \models^s \Gamma$.

The following lemma shows that valuations can always be extended in such a way that satisfaction is preserved.

Lemma 12. *Let* $\Gamma, x : A$ *be a legal context and let* (ρ,ζ) *be a valuation s.t.* $(\rho,\zeta) \models^s \Gamma, A : s$.

1. $(\rho(x := N), \zeta(x := c)) \models^s \Gamma, x : A$ *for every* $N \in \mathcal{T}$ *s.t.* $\mathcal{E} \vdash N : (\!|A|\!)_\rho$ *and* $c \in a(N)$ *and* $N \in \langle\!\langle A \rangle\!\rangle_{(\rho,\zeta)}$.
2. *There exists* $z \in V$ *s.t.* $(\rho(x := z), \zeta(x := c)) \models^s \Gamma, x : A$ *for every* $c \in a(z)$.

Proof. By definition of $\models^s$.

Finally, we prove that the model construction is sound.

Proposition 2 (Soundness). *If* $\Gamma \vdash M : A$ *then* $\Gamma \models^s M : A$.

Proof. Note that we only have to show that $(\!|M|\!)_\rho \in \langle\!\langle A \rangle\!\rangle_{(\rho,\zeta)}$ whenever $(\zeta,\rho) \models^s \Gamma$. We treat three cases:

1. assume that $\Gamma \vdash M : A$ is derived using (product). Then $A = s_2$ and $M = \Pi x{:}B.\, C$. Moreover the last rule of the derivation is of the form
$$\frac{\Gamma \vdash B : s_1 \qquad \Gamma, x : B \vdash C : s_2}{\Gamma \vdash \Pi x{:}B.\, C : s_2}$$
Assume $(\rho,\zeta) \models^s \Gamma$. To show $(\!|\Pi x{:}B.\, C|\!)_\rho \in \langle\!\langle s_2 \rangle\!\rangle_{(\rho,\zeta)}$. Note that $\langle\!\langle s_2 \rangle\!\rangle_{(\rho,\zeta)} = \mathsf{SN}(s_2)$ so we need to prove that $(\!|\Pi x{:}B.\, C|\!)_\rho$ is strongly normalizing. By induction hypothesis, both $(\!|B|\!)_\rho$ and $(\!|C|\!)_{\rho'}$ (where $(\rho',\zeta') \models^s \Gamma, x : B$) are strongly normalizing. By Lemma 12, there exists $z \in V$ and $c \in a(z)$ s.t. $(\rho(x := z), \zeta(x := c)) \models^s \Gamma, x : B$. Hence $(\!|C|\!)_{\rho(x:=z)}$ is strongly normalizing. Thus $(\!|\Pi x{:}B.\, C|\!)_\rho$ is strongly normalizing.
2. assume that $\Gamma \vdash M : A$ is derived using (application). Then $M = M_1\, M_2$, $A = C\{x := M_2\}$ and the last rule of the derivation is
$$\frac{\Gamma \vdash M_1 : (\Pi x{:}B.\, C) \qquad \Gamma \vdash M_2 : B}{\Gamma \vdash M_1\, M_2 : C\{x := M_2\}}$$
Assume $(\rho,\zeta) \models^s \Gamma$. To show $(\!|M_1\, M_2|\!)_\rho \in \langle\!\langle C\{x := M_2\} \rangle\!\rangle_{(\rho,\zeta)}$. By induction hypothesis,
$$(\!|M_1|\!)_\rho \in \langle\!\langle \Pi x{:}B.\, C \rangle\!\rangle_{(\rho,\zeta)} \qquad (\!|M_2|\!)_\rho \in \langle\!\langle B \rangle\!\rangle_{(\rho,\zeta)} \qquad \langle\!\langle M_2 \rangle\!\rangle_{(\rho,\zeta)} \in a((\!|M_2|\!)_\rho)$$
By definition of $\langle\!\langle \Pi x{:}B.\, C \rangle\!\rangle_{(\rho,\zeta)}$, $(\!|M_1\, M_2|\!)_\rho \in \bigcap_{c \in a((\!|M_2|\!)_\rho)} \langle\!\langle C \rangle\!\rangle_{(\rho(x:=(\!|M_2|\!)_\rho), \zeta(x:=c))}$. A fortiori
$$(\!|M_1\, M_2|\!)_\rho \in \langle\!\langle C \rangle\!\rangle_{(\rho(x:=(\!|M_2|\!)_\rho), \zeta(x:=\langle\!\langle M_2 \rangle\!\rangle_{(\rho,\zeta)}))}$$
By Lemma 9, $(\!|M_1\, M_2|\!)_\rho \in \langle\!\langle C\{x := M_2\} \rangle\!\rangle_{(\rho,\zeta)}$.
3. assume that $\Gamma \vdash M : A$ is derived using (abstraction). Necessarily $M = \lambda x{:}B.\, N$, $A = \Pi x{:}B.\, C$ and the last rule of the derivation is
$$\frac{\Gamma, x : B \vdash N : C \qquad \Gamma \vdash (\Pi x{:}B.\, C) : s}{\Gamma \vdash \lambda x{:}B.\, N : \Pi x{:}B.\, C}$$
Assume $(\rho,\zeta) \models^s \Gamma$. To show that $(\!|\lambda x{:}B.\, N|\!)_\rho \in \langle\!\langle \Pi x{:}B.\, C \rangle\!\rangle_{(\rho,\zeta)}$, or by definition of $\langle\!\langle \Pi x{:}B.\, C \rangle\!\rangle_{(\rho,\zeta)}$ that
$$\forall P \in \langle\!\langle B \rangle\!\rangle_{(\rho,\zeta)}.\ (\!|\lambda x{:}B.\, N|\!)_\rho\ P \in \bigcap_{c \in a(P)} \langle\!\langle B \rangle\!\rangle_{(\rho(x:=P), \zeta(x:=c))} \qquad (*)$$
By induction hypothesis and Lemma 12, $\langle\!\langle B \rangle\!\rangle_{(\rho(x:=P), \zeta(x:=c))}$ is a saturated set whenever $P \in \langle\!\langle B \rangle\!\rangle_{(\rho,\zeta)}$ and $c \in a(P)$. By definition of saturated sets, $(*)$ follows from:
(a) for every $P \in \langle\!\langle B \rangle\!\rangle_{(\rho,\zeta)}$, $(\!|\lambda x{:}B.\, N|\!)_\rho\ P \in \mathsf{SN}((\!|\Pi x{:}B.\, C|\!)_\rho)$;

(b) for every $P \in \langle\!\langle B \rangle\!\rangle_{(\rho,\zeta)}$, $(\!| N |\!)_{\rho(x:=P)} \in \bigcap_{c \in a(P)} \langle\!\langle B \rangle\!\rangle_{(\rho(x:=P),\zeta(x:=c))}$.
The latter is a direct consequence of the induction hypothesis and part 2 of Lemma 6. As for (a), we need to prove by Lemma 3 that $(\!| B |\!)_\rho$ is strongly normalizing and that $(\!| N |\!)_{\rho(x:=P')}$ for every P' such that $P \twoheadrightarrow_{\beta\overline{\eta}} P'$. As $P' \in \langle\!\langle B \rangle\!\rangle_{(\rho,\zeta)}$ for $P \twoheadrightarrow_{\beta\overline{\eta}} P'$, both follow from the induction hypothesis.

It follows that legal terms of the systems of the λ-cube are strongly normalizing w.r.t. β-reduction+η-expansion.

Theorem 3. *For every system of the λ-cube:* $\Gamma \vdash M : A \quad \Rightarrow \quad M \in \mathsf{SN}(\beta\overline{\eta}(\Gamma))$

Proof. Without loss of generality, we can assume $\Gamma = \langle\rangle$, as

$$x_1 : A_1,\ \ldots\ , x_n : A_n \vdash M : B \ \Rightarrow\ \langle\rangle \vdash \lambda x_1 : A_1.\ \ldots\ .\ \lambda x_n : A_n.\ M : \Pi x_1 : A_1.\ \ldots\ .\ \Pi x_n : A_n.\ M$$
$$\lambda x_1 : A_1.\ \ldots\ .\ \lambda x_n : A_n.\ M \in \mathsf{SN}(\beta\overline{\eta}) \ \Rightarrow\ M \in \mathsf{SN}(\beta\overline{\eta})$$

Then for every valuation (ρ, ξ), $(\!| M |\!)_\rho \in \langle\!\langle A \rangle\!\rangle_{(\rho,\zeta)} \in \mathbf{SAT}((\!| A |\!)_\rho)$. In particular, $(\!| M |\!)_\rho \in \mathsf{SN}(\beta\overline{\eta})$. But $(\!| M |\!)_\rho = M$ so we are done.

4 Strong normalization of the algebraic λ-cube

In this section, we introduce the algebraic λ-cube and show that strong normalization is a modular property of the algebraic λ-cube, provided the underlying term-rewriting system is non-duplicating.

4.1 The algebraic λ-cube

The algebraic λ-cube is obtained from the λ-cube by aggregating many-sorted rewriting systems to the type systems. As in [7, 8, 10], we shall only consider first-order term-rewriting systems. Note however that there are other presentations of the algebraic λ-cube based on higher-order rewriting systems, see e.g. [3].

Definition 15. *A* signature Σ *consists of a pair* $(\Lambda, (F_{w,s})_{w \in \mathsf{List}(\Lambda), s \in \Lambda})$ *where Λ is a set of* universes *and* $(F_{w,s})_{w \in \mathsf{List}(\Lambda), s \in \Lambda}$ *is an indexed family of pairwise disjoint sets of* function symbols. *In the sequel, we let* $F = \bigcup_{w \in \mathsf{List}(\Lambda), s \in \Lambda} F_{w,s}$.

Below we let Σ be a fixed signature and assume given a fixed countably infinite set of variables V_τ for every universe τ.

Definition 16. *The set $T_\Sigma(\tau)$ of* τ-terms *is defined by the clauses:*

1. *if $x \in V_\tau$, then $x \in T_\Sigma(\tau)$;*
2. *if $f \in F_{(\tau_1,\ldots,\tau_n),\tau}$ and $t_i \in T_\Sigma(\tau_i)$ for $i = 1, \ldots, n$, then $f(t_1, \ldots, t_n) \in T_\Sigma(\tau)$.*

As usual, we let $\mathsf{var}(t)$ *and* $\mathsf{mvar}(t)$ *denote the set and multiset of variables of t respectively.*

Finally, we define term-rewriting systems.

Definition 17. *A* term-rewriting system $\mathcal{R}$ *is an indexed set* $(R_\tau)_{\tau \in \Lambda}$ *such that* $R_\tau \subseteq T_\Sigma(\tau) \times T_\Sigma(\tau)$ *and* $\mathsf{var}(l) \subseteq \mathsf{var}(r)$ *for every* $(l, r) \in R_\tau$. *We say $\mathcal{R}$ is* non-duplicating *if* $\mathsf{mvar}(l) \subseteq \mathsf{mvar}(r)$ *for every* $(l, r) \in R_\tau$.

Basic concepts, such as the rewriting relation and termination, are defined in the standard way, see e.g. [27].

Throughout this section, we let $\mathcal{R}$ be a fixed term-rewriting system. Every rule set $\mathbf{S}$ yields an algebraic type system $\lambda\mathbf{S} + \mathcal{R}$ as follows.

Definition 18 (Algebraic Type Systems of the algebraic λ-cube).

1. *The set $\mathcal{T}$ of* pseudo-terms *is defined by the abstract syntax*
$$\mathcal{T} = V^* \mid V^\Box \mid\ * \mid \Box \mid \mathcal{T}\mathcal{T} \mid \lambda V : \mathcal{T}.\mathcal{T} \mid \Pi V : \mathcal{T}.\mathcal{T} \mid \Lambda \mid F$$
2. *Algebraic reduction $\to_\mathcal{R}$ is defined by the rules $C[\theta l] \to_\mathcal{R} C[\theta r]$ where (l, r) is a rule, $C[.]$ is a context (in the terminology of rewriting) and θ is a substitution.*
3. *β-reduction, η-reduction... are defined as in Section 2.*
4. *The* derivability *relation $\vdash$ is given by the rules of Figure 1 except for the conversion rule, and the rules of Figure 4.*

In the conversion rule, one could have used η-expansion instead of η-reduction. The important point is not to allow conversion, especially algebraic conversion, as it would lead to an inconsistent system e.g. for the system $\mathcal{R}$ defined by the rules $\mathsf{or}(x, y) \to x$ and $\mathsf{or}(x, y) \to y$, see e.g. [7].

$$\text{(universe)} \quad \langle\rangle \vdash \sigma : * \qquad \text{if } \sigma \in \Lambda$$

$$\text{(function)} \quad \frac{\Gamma \vdash M_1 : \sigma_1 \quad \ldots \quad \Gamma \vdash M_n : \sigma_n}{\Gamma \vdash f(M_1, \ldots, M_n) : \tau} \qquad \text{if } f \in F_{((\sigma_1, \ \ldots \ ,\sigma_n),\tau)}$$

$$\text{(conversion)} \quad \frac{\Gamma \vdash A : B \quad \Gamma \vdash B' : s}{\Gamma \vdash A : B'} \qquad \text{if } B \downarrow_{\beta\eta R} B'$$

Fig. 4. New rules for the algebraic λ-cube

4.2 Proof-irrelevance

One major obstacle with the algebraic λ-cube is subject reduction. As shown in [3], subject reduction can be proved directly, but the proof is extremely complex and lengthy. A simpler alternative, suggested by the author in [7], is to define a *proof-irrelevant* algebraic λ-cube, for which subject reduction is trivial to establish, to show that the algebraic λ-cube is a subsystem of the proof-irrelevant algebraic λ-cube, and to conclude by showing the latter to be strongly normalizing. We begin by defining the proof-irrelevant algebraic λ-cube. As a preliminary, we define pseudo-objects, pseudo-constructors and pseudo-kinds. The definition is almost identical to the one of Section 3.

Definition 19 (Pseudo-objects, pseudo-constructors and pseudo-kinds). *The classes $\mathcal{O}$, $\mathcal{C}$ and $\mathcal{K}$ of* pseudo-objects, pseudo-constructors *and* pseudo-kinds *are given by the abstract syntaxes*

$$\begin{aligned}
\mathcal{O} &= V^* \mid F \mid \lambda V^*{:}\mathcal{C}.\ \mathcal{O} \mid \lambda V^\Box{:}\mathcal{K}.\ \mathcal{O} \mid \mathcal{O}\ \mathcal{O} \mid \mathcal{O}\ \mathcal{C} \\
\mathcal{C} &= V^\Box \mid \Lambda \mid \Pi V{:}\mathcal{C}.\ \mathcal{C} \mid \Pi V{:}\mathcal{K}.\ \mathcal{C} \mid \lambda V^*{:}\mathcal{C}.\ \mathcal{C} \mid \lambda V^\Box{:}\mathcal{K}.\ \mathcal{C} \mid \mathcal{C}\ \mathcal{C} \mid \mathcal{C}\ \mathcal{O} \\
\mathcal{K} &= * \mid \Pi V{:}\mathcal{C}.\ \mathcal{K} \mid \Pi V{:}\mathcal{K}.\ \mathcal{K}
\end{aligned}$$

Intuitively, proof-irrelevant type systems are obtained by identifying all the pseudo-objects in the conversion rule. This motivates the definition below. For technical convenience, we assume the term-rewriting system under consideration to contain a specific constant $\bullet$.

Definition 20.

1. *The* proof-irrelevant skeleton $|.| : \mathcal{T} \to \mathcal{T}$ *is defined inductively as follows:*

$$\begin{aligned}
|M| &= \bullet && \text{if } M \in \mathcal{O} \\
|v| &= v && \text{if } x \in V^\Box \cup \Lambda \cup \mathcal{S} \\
|M\ N| &= |M|\ |N| && \text{if } (M\ N) \notin \mathcal{O} \\
|\lambda x{:}A.\ M| &= \lambda x{:}|A|.\ |M| && \text{if } (\lambda x{:}A.\ M) \notin \mathcal{O} \\
|\Pi x{:}A.\ B| &= \Pi x{:}|A|.\ |B|
\end{aligned}$$

2. *The* proof-irrelevant conversion $\simeq$ *is defined by* $M \simeq N \Leftrightarrow |M| =_{\beta\eta} |N|$.
3. *The relation* $\Gamma \vdash^{pi} M : A$ *is obtained from the rules of the algebraic λ-cube by replacing the conversion rule by*

$$\frac{\Gamma \vdash^{pi} M : A \quad \Gamma \vdash^{pi} B : s}{\Gamma \vdash^{pi} M : B} \quad \text{if } A \simeq B$$

As in [7], one can show the systems of the algebraic λ-cube to be *sound w.r.t. proof-irrelevance*, i.e. for every system of the algebraic λ-cube

$$\Gamma \vdash M : A \quad \Rightarrow \quad \Gamma \vdash^{pi} M : A$$

To prove strong normalization of the algebraic λ-cube, it is thus enough to prove

$$\Gamma \vdash^{pi} M : A \quad \Rightarrow \quad M \in \mathsf{SN}(\beta\overline{\eta}(\Gamma)\mathcal{R})$$

The gain is that proof-irrelevant systems enjoy soundness w.r.t. convertibility and subject reduction. Consequently the model construction of Section 3 only needs to be modified slightly.

4.3 The model construction

The model construction is extended by setting $(\!|\sigma|\!)_\rho = \mathsf{SN}(\sigma)$ for every universe σ. The proof of soundness proceeds as before, except for:

1. the conversion rule: one needs to strengthen Lemma 11 by replacing $=_{\beta\eta}$ by $\simeq$. One proceeds as in [7];
2. the function rule: one needs to use the following result.

Lemma 13. *If $\mathcal{R}$ is non-duplicating and terminating, $f \in F_{((\sigma_1,\ \ldots\ ,\sigma_n),\tau)}$ and $\mathcal{E} \vdash t_i : \sigma_i$ for $1 \leq i \leq n$, then*

$$t_i \in \mathsf{SN}(\sigma_i) \quad \textit{for } 1 \leq i \leq n \quad \Rightarrow \quad f(t_1,\ \ldots,\ t_n) \in \mathsf{SN}(\tau)$$

Proof. (Sketch) Termination is a persistent property of non-duplicating term-rewriting systems [27] thus $\to_{\mathcal{R}}$ is terminating on algebraic pseudo-terms and more generally on pseudo-terms, see [8]. To conclude, proceed exactly as in e.g. [2], using the notions of cap and aliens.

It follows that strong normalization (under β-reduction+η-expansion+algebraic reduction) is a modular property of the systems of the algebraic λ-cube, provided the underlying term-rewriting system is non-duplicating.

Theorem 4. *For every system $\lambda\mathbf{S} + \mathcal{R}$ of the algebraic λ-cube: $\Gamma \vdash M : A \quad \Rightarrow \quad M \in \mathsf{SN}(\beta\overline{\eta}(\Gamma)\mathcal{R})$ provided $\mathcal{R}$ is terminating and non-duplicating.*

Theorem 4 partially solves a conjecture by Di Cosmo and Ghani [10]. Yet Theorem 4 is not fully satisfactory since one would like to drop the condition of the term-rewriting system being non-duplicating. The key is to prove Lemma 13 for an arbitrary terminating term-rewriting system. One possibility, which needs to be further explored, is to use the techniques of [8].

5 Conclusion

This paper brings new light to η-expansion in systems of dependent types. In particular, we have shown:

1. β-reduction+η-expansion is strongly normalizing for the systems of the λ-cube;
2. β-reduction+η-expansion+algebraic reduction is strongly normalizing for the systems of the algebraic λ-cube, provided the underlying term-rewriting system is terminating and non-duplicating.

In the future, it would be interesting to study strong normalization of β-reduction+η-expansion for Pure Type Systems. We therefore conclude with the following conjecture:[4]

Conjecture 1. For every specification $\mathbf{S}$ such that $\lambda_\beta\mathbf{S}$ is strongly normalizing w.r.t. $\to_\beta$, $\lambda\mathbf{S}$ is strongly normalizing w.r.t. $\to_{\beta\overline{\eta}}$.

A similar conjecture for $\to_{\beta\eta}$-reduction has been formulated by Geuvers [14].

Conjecture 2 (Geuvers). For every specification $\mathbf{S}$ such that $\lambda_\beta\mathbf{S}$ is strongly normalizing w.r.t. $\to_\beta$, $\lambda\mathbf{S}$ is strongly normalizing w.r.t. $\to_{\beta\eta}$.

Our conjecture implies his, as

$$\begin{aligned}\lambda\mathbf{S} \text{ is strongly normalizing w.r.t. } \to_{\beta\overline{\eta}} &\Rightarrow \lambda\mathbf{S} \text{ is strongly normalizing w.r.t. } \to_\beta \\ &\Rightarrow \lambda\mathbf{S} \text{ is strongly normalizing w.r.t. } \to_{\beta\eta}\end{aligned}$$

[4] $\lambda_\beta\mathbf{S}$ denotes the usual Pure Type System $\lambda\mathbf{S}$ associated to a specification $\mathbf{S}$.

References

1. T. Altenkirch. *Constructions, inductive types and strong normalisation*. PhD thesis, Laboratory for the Foundations of Computer Science, University of Edinburgh, 1994.
2. F. Barbanera and M. Fernández. Modularity of termination and confluence in combinations of rewrite systems with λ_ω. In A. Lingas, R. Karlsson, and S. Karlsson, editors, *Proceedings of ICALP'93*, volume 700 of *Lecture Notes in Computer Science*, pages 657–668. Springer-Verlag, 1993.
3. F. Barbanera, M. Fernández, and H. Geuvers. Modularity of strong normalisation and confluence in the algebraic λ-cube. *Journal of Functional Programming*, 7(6):613–660, November 1997.
4. H. Barendregt. Introduction to Generalised Type Systems. *Journal of Functional Programming*, 1(2):125–154, April 1991.
5. H. Barendregt. Lambda calculi with types. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, pages 117–309. Oxford Science Publications, 1992. Volume 2.
6. G. Barthe. Existence and uniqueness of normal forms in pure type systems with $\beta\eta$-conversion. Manuscript, 1998.
7. G. Barthe. The relevance of proof-irrelevance. In K.G. Larsen, S. Skyum, and G. Winskel, editors, *Proceedings of ICALP'98*, volume 1443 of *Lecture Notes in Computer Science*, pages 755–768. Springer-Verlag, 1998.
8. G. Barthe and F. van Raamsdonk. Termination of algebraic type systems: the syntactic approach. In M. Hanus and J. Heering, editors, *Proceedings of ALP '97 - HOA '97*, volume 1298 of *Lecture Notes in Computer Science*, pages 174–193. Springer-Verlag, 1997.
9. R. Di Cosmo. A brief history of rewriting with extensionality. Presented at the International Summer School on Type Theory and Term Rewriting, Glasgow, September 1996.
10. R. Di Cosmo and N. Ghani. On modular properties of higher order extensional lambda calculi. In P. Degano, R. Gorrieri, and A. Marchetti-Spaccamela, editors, *Proceedings of ICALP'97*, volume 1256 of *Lecture Notes in Computer Science*, pages 237–247. Springer-Verlag, 1997.
11. R. Di Cosmo and D. Kesner. Simulating expansions without expansions. *Mathematical Structures in Computer Science*, 4(3):315–362, September 1994.
12. D. van Daalen. *The language theory of Automath*. PhD thesis, Technical University of Eindhoven, 1980.
13. G. Dowek, G. Huet, and B. Werner. On the existence of long $\beta\eta$-normal forms in the cube. In H. Geuvers, editor, *Informal Proceedings of TYPES'93*, pages 115–130, 1993.
14. H. Geuvers. *Logics and type systems*. PhD thesis, University of Nijmegen, 1993.
15. H. Geuvers. A short and flexible proof of strong normalisation for the Calculus of Constructions. In P. Dybjer, B. Nordström, and J. Smith, editors, *Proceedings of TYPES'94*, volume 996 of *Lecture Notes in Computer Science*, pages 14–38. Springer-Verlag, 1995.
16. H. Geuvers and B. Werner. On the Church-Rosser property for expressive type systems and its consequence for their metatheoretic study. In *Proceedings of LICS'94*, pages 320–329. IEEE Computer Society Press, 1994.
17. N. Ghani. Eta-expansions in dependent type theory—the calculus of constructions. In P. de Groote and J. Hindley, editors, *Proceedings of TLCA'97*, volume 1210 of *Lecture Notes in Computer Science*, pages 164–180. Springer-Verlag, 1997.
18. F. Joachimski. η-expansion in Gödel's T, Pure Type Systems and calculi with permutative conversions. Manuscript, 1997.
19. J.W. Klop. Term-rewriting systems. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, pages 1–116. Oxford Science Publications, 1992. Volume 2.
20. P.-A. Melliès and B. Werner. A generic proof of strong normalisation for pure type systems. In E. Giménez and C. Paulin-Mohring, editors, *Proceedings of TYPES'96*, volume 1512 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.
21. R. Nederpelt. *Strong normalisation in a typed lambda calculus with lambda structured types*. PhD thesis, Technical University of Eindhoven, 1973.
22. G. Pottinger. Strong normalisation for terms of the theory of constructions. Technical Report TR 11-7, Odissey Research Associates, 1987.
23. D. Prawitz. Ideas and results of proof theory. In J.E. Fenstad, editor, *The 2nd Scandinavian Logical Symposium*, pages 235–307. North-Holland, 1970.
24. A. Salvesen. The Church-Rosser property for $\beta\eta$-reduction. Manuscript, 1991.
25. J. Terlouw. Strong normalization in type systems: a model theoretic approach. *Annals of Pure and Applied Logic*, 73(1):53–78, May 1995.
26. H. Xi. Simulating η-expansions with β-reductions in the second-order polymorphic lambda-calculus. In S. Adian and A. Nerode, editors, *Proceedings of LFCS'97*, volume 1234 of *Lecture Notes in Computer Science*, pages 399–409. Springer-Verlag, 1997.
27. H. Zantema. Termination of term rewriting: Interpretation and type elimination. *Journal of Symbolic Computation*, 17(1):23–50, January 1994.

An Algebraic Characterization of Typability in ML with Subtyping

Marcin Benke*

Institute of Informatics
Warsaw University
ul. Banacha 2
02-097 Warsaw, POLAND

e-mail: benke@mimuw.edu.pl

Abstract. We research the problem of typability in the functional language ML extended with both atomic and polymorphic subtyping. The paper analyses the interaction between polymorphism and subtyping and gives an algebraic characterization of the typability problem.

Introduction

Type reconstruction and type-checking are a form of ensuring program correctness. They prevent run-time errors and often even logical errors in program design.

Subtyping is a form of extending the type discipline which gives programmer more freedom while retaining advantages of type correctness. Since the seminal paper of Mitchell [Mit84] it has attracted a lot of interest. Several ways of extending ML polymorphic type discipline with subtyping have been proposed, but the question of complexity of type reconstruction in such systems remains open.

In this paper we and give an algebraic characterization of typability in such system, similar to one that led to establishing exact complexity of the typability in pure ML. We hope that the characterisation proposed in this paper will allow to achieve similar result for ML with subtyping.

Contributions and organization of the paper

In the first section of the paper we analyse the interaction between polymorphic and atomic subtyping. We introduce a system of subtyping for ML types being a natural restriction of the Mitchell's system [Mit88] to ML types, enriched with the subtyping between type constants. In contrast to to the Mitchell's system, the relation induced by our system turns out to be decidable (in polynomial time, actually). Furthermore, we prove that polymorphic and atomic subtyping can be in certain sense separated.

* This work has been partially supported by Polish KBN grant 8 T11C 035 14

W. Thomas (Ed.): FOSSACS'99, LNCS 1578, pp. 104-119, 1999.

In Section 2 we introduce a type system for ML with subtyping. In contrast to the previous work in this area, we try to keep this extension as simple as possible and thus do not propose to extend the syntax of types, but merely to add the subtyping rule. The resulting system turns out to behave very differently to its ancestor and enjoys some interesting (even if not very encouraging from the practical point of view) properties.

The paper [KTU90] provides an algebraic characterization of typability in ML by the "acyclic semiunification problem" (ASUP) — a variant of the unification problem. In section 3 we propose a generalization of ASUP allowing to accommodate subtyping. In the final section we show that such extension is sufficient to provide an algebraic characterization of typability in our system.

1 Preliminaries

1.1 Terms

We concentrate on a subset of ML terms essential for type analysis:

$$M ::= c \mid x \mid \lambda x.M \mid M_1 M_2 \mid \textbf{let } x = M_1 \textbf{ in } M_2$$

(x stands for variables and c for constants)

1.2 Types and type schemes

Given a set of type variables $(\alpha, \beta, \gamma, \ldots)$ and a (finite) set of type constants (like *char,int,real,...*), we define the set of (monomorphic) types

$$\tau ::= \kappa \mid \alpha \mid \tau \to \tau$$

where κ stands for type constants.

Further, we define the set of (polymorphic) type schemes

$$\sigma ::= \forall \alpha_1 \ldots \alpha_n.\tau$$

In the sequel we shall use the abbreviation $\forall \vec{\alpha}.\tau$, and a notational convention that σ (possibly with indices) will stand for type schemes and τ, ρ (possibly with indices) for (monomorphic) types.

If $\alpha \notin FV(\sigma)$ then $\forall \alpha.\sigma$ is called an empty binding. A type scheme containing only empty bindings is called singular.

1.3 Subtype partial orders

We assume there is some predefined partial ordering $\leq_\kappa$ on the type constants.

We introduce a system of subtyping for ML types being a restriction of the Mitchell's system [Mit88] to ML types, enriched with the subtyping between constants. The system derives formulas of the form $\sigma \leq \tau$, where σ and τ are type schemes.

Axioms:

(refl) $\sigma \leq \sigma$

(inst) $\forall \vec{\alpha}.\, \sigma \leq \forall \vec{\beta}.\, \sigma[\vec{\rho}/\vec{\alpha}]$, ρ_i are types; $\beta_i \notin FV(\forall \vec{\alpha}.\, \sigma)$

Rules:

$$\text{(const)} \quad \frac{\kappa_1 \leq_\kappa \kappa_2}{\kappa_1 \leq \kappa_2}$$

$$(\rightarrow) \quad \frac{\rho' \leq \rho \quad \tau \leq \tau'}{\rho \rightarrow \tau \leq \rho' \rightarrow \tau'} \qquad (\forall) \quad \frac{\sigma \leq \sigma'}{\forall \alpha.\, \sigma \leq \forall \alpha.\, \sigma'}$$

$$\text{(trans)} \quad \frac{\sigma \leq \sigma_1 \quad \sigma_1 \leq \sigma'}{\sigma \leq \sigma'}$$

We write $\vdash \sigma \leq \sigma'$ to indicate that $\sigma \leq \sigma'$ is derivable in the above system. We shall also write $\vdash_\sigma$ for derivability without using the axiom **(inst)** (and use $\vdash_\sigma \sigma_1 \leq \sigma_2$ as a a shorthand for $\sigma_1 \leq_\sigma \sigma_2$) and $\vdash_\tau$ for derivability without mentioning type schemes at all.

We shall use the symbol $\preceq$ to denote the relation generated by the **(inst)** axiom.

It is worthwhile to observe that this is not a conservative restriction of Mitchell's system, i.e. if we allow substituting polymorphic types in **(inst)**, we can infer more inequalities between ML types. A simple example here is the inequality

$$\forall \alpha.((\alpha \rightarrow \beta) \rightarrow \alpha) \leq (\alpha \rightarrow \beta) \rightarrow \gamma$$

which is derivable in the Mitchell's system but (as follows from Theorem 1.7) not in ours. On the other hand an important consequence of allowing only monomorphic instance is the following

Proposition 1.1. *The relation $\leq$ is decidable.*

Proof. This can be deduced from the results contained in [OL96]. In fact one can even prove that it is decidable in polynomial time.

Lemma 1.2. *If $\vdash \sigma \leq \sigma'$ and σ is singular then*

1. *$FV(\sigma) = FV(\sigma')$*
2. *the derivation contains only singular type schemes (in particular σ' is singular).*

Lemma 1.3. *If $\vdash \tau \leq \tau'$ (with τ, τ' monomorphic) then there is a derivation of this inequality which contains only monomorphic types.*

Lemma 1.4. *The relation $\preceq$ is reflexive and transitive.*

Lemma 1.5. *If $\sigma \leq_\sigma \sigma_1 \preceq \sigma'$ then there exists σ_2 such that*

$$\sigma \preceq \sigma_2 \leq_\sigma \sigma'.$$

Lemma 1.6. *If $\sigma_0 \preceq \sigma_1$ then $\forall\gamma.\,\sigma_0 \preceq \forall\gamma.\,\sigma_1$.*

Proof. By the definition of $\preceq$, we have $\sigma_0 \equiv \forall\vec{\alpha}.\,\sigma$ and $\sigma_1 \equiv \sigma[\vec{\rho}/\vec{\alpha}]$ for some σ, ρ. We have to consider two cases:

1. $\gamma \in FV(\forall\vec{\alpha}.\,\sigma)$. Then we have

$$\forall\gamma.\forall\vec{\alpha}.\,\sigma \preceq \forall\vec{\alpha}.\,\forall\gamma.\,\sigma[\gamma/\gamma, \vec{\alpha}/\vec{\alpha}] \equiv \forall\vec{\alpha}.\,\forall\gamma.\,\sigma \preceq \forall\gamma.\sigma[\vec{\rho}/\vec{\alpha}]$$

2. $\gamma \notin FV(\forall\vec{\alpha}.\,\sigma)$. Then we have

$$\forall\gamma.\forall\vec{\alpha}.\,\sigma \preceq \forall\vec{\alpha}.\,\sigma[\gamma/\gamma] \equiv \forall\vec{\alpha}.\,\sigma \preceq \forall\gamma.\,\sigma[\vec{\rho}/\vec{\alpha}]$$

In both cases the thesis follows from the transitivity of $\preceq$.

Theorem 1.7 (Normalization for $\leq$). *If $\vdash \sigma \leq \sigma'$ then there exists σ_1 such that*

$$\sigma \preceq \sigma_1 \leq_\sigma \sigma'$$

Proof. Again we proceed by induction on the derivation. The basic cases i.e. axioms, and (const) are trivial. The rule (trans) can be handled by Lemma 1.5.

If the last rule in the derivation was $(\rightarrow)$ then all components must be monomorphic, and by Lemma 1.3 there is a derivation of the inequality in $\vdash_\sigma$.

Having said that, we only have to consider the case when the last rule was $(\forall)$:

$$\frac{\sigma \leq \sigma'}{\forall\alpha.\,\sigma \leq \forall\alpha.\,\sigma'}$$

By the induction assumption, there exists σ_1 such that

$$\sigma \preceq \sigma_1 \leq_\sigma \sigma'.$$

But then, by Lemma 1.6 we have that

$$\forall\vec{\alpha}.\,\sigma \preceq \forall\vec{\alpha}.\,\sigma_1$$

On the other hand, obviously if $\sigma_1 \leq_\sigma \sigma'$ then $\forall\vec{\alpha}.\,\sigma_1 \leq_\sigma \forall\vec{\alpha}.\,\sigma'$.

Proposition 1.8. *In the $\leq$ ordering, every set of type schemes has a lower bound.*

Proof. It is easy to see that for every type scheme σ

$$\forall\alpha.\alpha \leq \sigma$$

$$\text{(CON)} \qquad \vdash c : \kappa(c) \quad \text{for } c \in K$$

$$\text{(VAR)} \qquad E(x : \sigma) \vdash x : \sigma$$

$$\text{(ABS)} \qquad \frac{E(x : \tau) \vdash M : \rho}{E \vdash \lambda x.M : \tau \to \rho}$$

$$\text{(APP)} \qquad \frac{E \vdash M : \tau \to \rho \quad E \vdash N : \tau}{E \vdash MN : \rho}$$

$$\text{(GEN)} \quad \frac{E \vdash M : \sigma}{E \vdash M : \forall \alpha.\sigma} \qquad \alpha \notin FV(E)$$

$$\text{(INST)} \qquad \frac{E \vdash M : \forall \alpha.\sigma}{E \vdash M : \sigma[\rho/\alpha]}$$

$$\text{(LET)} \quad \frac{E \vdash M : \sigma_0 \quad E(x : \sigma_0) \vdash N : \sigma}{E \vdash \textbf{let } x = M \textbf{ in } N : \sigma}$$

Fig. 1. A reference ML type system, $\vdash_{ML}$

2 Type systems

2.1 The traditional type system for pure ML

We assume that we have a fixed set of constants Q, and for each $c \in Q$ its type $\kappa(c)$, built only with type constants and arrows, is known.

The system depicted in Figure 1 will serve as a reference type system for ML [DM82,CDDK86,KTU90]. We shall use the symbol $\vdash_{\mathrm{ML}}$ to denote derivability in this system.

The simplest way to extend this system with subtyping is by adding the subsumption rule

$$\text{(SUB)} \quad \frac{E \vdash M : \tau \quad \tau \leq \rho}{E \vdash M : \rho}$$

In the sequel by $\vdash_{\mathrm{ML}_\leq}$ we shall understand the system $\vdash_{\mathrm{ML}}$ with the subsumption rule.

We shall say that a derivation is *normal* if subsumption rule is applied only to variables and constants.

Lemma 2.1. *If $E \vdash_{\mathrm{ML}_\leq} M : \tau$ then there is a normal derivation ending with this judgement.*

2.2 An alternative type system for ML

Kfoury et. al, in [KTU89,KTU94] suggest an alternative (equivalent) type inference system for ML, which is better suited for complexity studies:[1]

$$(\text{CON}) \qquad \vdash c : \kappa(c) \quad \text{for } c \in K$$

$$(\text{VAR}) \qquad (x : \sigma) \vdash x : \tau \qquad \text{for } \sigma \preceq \tau$$

$$(ABS) \qquad \frac{E(x : \tau) \vdash M : \rho}{E \vdash \lambda x.M : \tau \to \rho}$$

$$(APP) \qquad \frac{E \vdash M : \tau \to \rho \quad E \vdash N : \tau}{E \vdash MN : \rho}$$

$$(LET) \quad \frac{E \vdash M : \rho \quad E(x : \forall \vec{\alpha}.\rho) \vdash N : \tau}{E \vdash \textbf{let } x = M \textbf{ in } N : \tau} \quad \alpha$$

Fig. 2. An alternative type system for ML, $\vdash_{ML*}$

Proposition 2.2. *For every term M, it is typable in $\vdash_{\text{ML}^*}$ iff it is typable in $\vdash_{\text{ML}}$.*

For proof, cf. [CDDK86,KTU90].

Subtyping can be added here by replacing the instance relation in the axiom with the subtyping relation defined in the section 1.3...

$$E(x : \sigma) \vdash x : \tau \qquad \text{if } \vdash \sigma \leq \tau$$

... and modifying an axiom for constants:

$$(\text{CON}) \qquad \vdash c : \tau \qquad \text{if } \vdash \kappa(c) \leq \tau$$

We shall denote derivability in the resulting system by the symbol $\vdash_{\text{ML}^*_\leq}$.

Theorem 2.3. *For every environment E, term M and monomorphic type τ, if $\vec{\alpha} \subseteq FV(\tau) - FV(E)$ then*

$$E \vdash_{\text{ML}_\leq} M : \forall \vec{\alpha}.\tau \text{ iff } E \vdash_{\text{ML}^*_\leq} M : \tau$$

In other words for every term M, it is typable in $\vdash_{\text{ML}^*_\leq}$ iff it is typable in $\vdash_{\text{ML}_\leq}$.

[1] This system is called $\vdash^*$ in [KTU90]

Proof. The right-to-left implication becomes obvious when we observe that $\vdash_{\mathrm{ML}^\star_\leq}$ can be viewed as a subset of $\vdash_{\mathrm{ML}_\leq}$, and that under given assumption generalization over $\vec{\alpha}$ is allowed in $\vdash_{\mathrm{ML}_\leq}$.

The proof of the left-to-right implication may proceed by induction on derivation; the only difference from the Proposition 2.2 lies in the rule (SUB):

$$\text{(SUB)} \quad \frac{E \vdash M : \forall \vec{\alpha}.\, \tau \quad \forall \vec{\alpha}.\, \tau \leq \forall \vec{\beta}.\, \tau'}{E \vdash M : \forall \vec{\beta}.\, \tau'}$$

Because of the normalization theorem for $\leq$, it is sufficient to consider the cases when $\forall \vec{\alpha}.\, \tau \preceq \forall \vec{\beta}.\, \tau'$ and when $\forall \vec{\alpha}.\, \tau \leq_\sigma \forall \vec{\beta}.\, \tau'$.

In the first case it follows that $\tau' \equiv \tau[\vec{\rho}/\vec{\alpha}]$. By the induction assumption we have that

$$E \vdash_{\mathrm{ML}^\star_\leq} M : \tau$$

and want to conclude that

$$E \vdash_{\mathrm{ML}^\star_\leq} M : \tau[\vec{\rho}/\vec{\alpha}].$$

It is easy to see that

$$E[\vec{\rho}/\vec{\alpha}] \vdash_{\mathrm{ML}^\star_\leq} M : \tau[\vec{\rho}/\vec{\alpha}]$$

but since α's cannot be possibly free in E, we have that $E[\vec{\rho}/\vec{\alpha}] = E$.
Now consider the case when

$$\vdash_\sigma \forall \vec{\alpha}.\, \tau \leq \forall \vec{\beta}.\, \tau'$$

It is easy to prove that in this case $\vec{\alpha} = \vec{\beta}$ and $\vdash_\sigma \tau \leq \tau'$ with τ, τ' monomorphic. A routine check that in this case if $E \vdash_{\mathrm{ML}^\star_\leq} M : \tau$ then $E \vdash_{\mathrm{ML}^\star_\leq} M : \tau'$ is left to the reader.

Lemma 2.4. *Let M be an arbitrary term, x a free variable, occurring k times ($k \geq 1$) in M, and let $N = \lambda x_1 \ldots x_k.M'$, where M' is a term obtained from M by replacing subsequent occurrences of x with $x_1, \ldots, x_k$ respectively. Then M is typable iff N is, or, more precisely*

1. *If $E(x : \sigma) \vdash M : \tau$ then $E \vdash N : \rho_1 \to \cdots \to \rho_k \to \tau$ for some $\rho_1, \ldots, \rho_k$ such that $\sigma \leq \rho_i$ for $i = 1, \ldots, k$.*
2. *If $E \vdash N : \rho_1 \to \cdots \to \rho_k \to \tau$, then there is σ such that $E(x : \sigma) \vdash M : \tau$.*

2.3 Example

The example depicted in Fig. 3 illustrates an important difference between ML and $\mathrm{ML}_\leq$:

Assume we have two type constants i, r (one can think of them as representing for example *int* and *real*), with $i \leq r$, an atomic constant pi of type r and a functional constant *round* of type $r \to i$. Now consider the following term:

$$\textbf{let } t = (\lambda f.\, \lambda x.\, f\ (f\ x)) \textbf{ in } t\ round\ pi$$

Assumptions:
r and i are atomic type constants with $i \leq r$.

pi is an object constant of type r

$round$ is an object constant of type $r \to i$

$E_{f,x} = \{f : r \to i, x : r\}$

$E_t = \{t : (r \to i) \to r \to i\}$

Derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{E_{f,x} \vdash f : r \to i \quad \dfrac{E_{f,x} \vdash f : r \to r \quad E_{f,x} \vdash x : r}{E_{f,x} \vdash fx : r}}{E_{f,x} \vdash f(fx) : i}}{f : r \to i \vdash \lambda x.f(fx) : r \to i}}{\vdash \lambda f.\lambda x.f(fx) : (r \to i) \to r \to i} \quad \dfrac{\dfrac{E_t \vdash t : (r \to i) \to r \to i \quad \vdash round : r \to i}{E_t \vdash t\ round : r \to i} \quad E_t \vdash pi : r}{E_t \vdash t\ round\ pi : i}}{\vdash \lambda f.\lambda x.f(fx) : (r \to i) \to r \to i}}{\vdash \textbf{ let } t = \lambda f.\lambda x.f(fx) \textbf{ in } \text{t round pi : i}}$$

Fig. 3. A type derivation in $ML_{\leq}$

This term is not typable in ML but is typable in $ML_{\leq}$. In fact all its normal typings mention only monomorphic types. One of this typings is presented in Fig. 3 Here, the context in which t is used, has 'forced' inferring a monomorphic type for it, even though its definition allows to infer an universal type, e.g. $\forall\alpha.\alpha \to \alpha$.

This example illustrates consequences of the fact that $ML_{\leq}$ has no principal types property, and shows that this type system is in a way 'not compositional', whereas for ML, the following holds:

Proposition 2.5. *For any terms* N_1, N_2

$$E \vdash_{\mathrm{ML}} \ \textbf{\textit{let}}\ x = N_1\ \textbf{\textit{in}}\ N_2 : \tau$$

iff

$$E(x:\sigma) \vdash_{\mathrm{ML}} N_2 : \tau$$

where σ *is a principal type for* N_1.

3 Subtyping and Semi-Unification

3.1 Semi-unification

The Semi-Unification Problem (SUP) can be formulated as follows: An instance Γ of SUP is a set of pairs of (monomorphic) types. A substitution S is a *solution* of $\Gamma = \{(t_1, u_1), \ldots, (t_n, u_n)\}$ iff there are substitutions $R_1, \ldots, R_n$ such that

$$R_1(S(t_1)) = S(u_1), \ldots, R_n(S(t_n)) = S(u_n)$$

The problem is to decide, whether given instance has a solution.

A variation of this problem including subtyping can be formulated by redefining solution of Γ as follows: S is a *sub-solution* of Γ iff there are substitutions $R_1, \ldots, R_n$ such that

$$R_1(S(t_1)) \leq S(u_1), \ldots, R_n(S(t_n)) \leq S(u_n)$$

The Semi Sub-Unification Problem (SSUP) is to decide whether given instance has a sub-solution.

Proposition 3.1 ([KTU93]). *SUP is undecidable.*

Corollary 3.2. *SSUP is undecidable.*

3.2 Acyclic semi-unification

An instance Γ of semi-unification is *acyclic* if for some $n \geq 1$, there are integers $r_1, \ldots, r_n$ and $n+1$ disjoint sets of variables, $V_0, \ldots, V_n$, such that the pairs of Γ can be placed in n columns (possibly of different height; column i contains r_i pairs):

$$\begin{array}{cccc} (t^{1,1}, u^{1,1}) & (t^{2,1}, u^{2,1}) & \cdots & (t^{n,1}, u^{n,1}) \\ (t^{1,2}, u^{1,2}) & (t^{2,2}, u^{2,2}) & \cdots & (t^{n,2}, u^{n,2}) \\ \vdots & \vdots & \ddots & \vdots \\ (t^{1,r_1}, u^{1,r_1}) & (t^{2,r_2}, u^{2,r_2}) & \cdots & (t^{n,r_n}, u^{n,r_n}) \end{array}$$

where:

$$\begin{aligned} V_0 &= \mathrm{FV}(t^{1,1}) \cup \cdots \cup \mathrm{FV}(t^{1,r_1}) \\ V_i &= \mathrm{FV}(u^{i,1}) \cup \cdots \cup \mathrm{FV}(u^{i,r_i}) \cup \\ &\quad \cup \mathrm{FV}(t^{i+1,1}) \cup \cdots \cup \mathrm{FV}(t^{i+1,r_{i+1}}) \text{ for } 1 \leq i < n \\ V_n &= \mathrm{FV}(u^{n,1}) \cup \cdots \cup \mathrm{FV}(u^{n,r_n}) \end{aligned}$$

The set of terms with variables from V_i is also called *zone* i.

The Acyclic Semi-Unification Problem (ASUP) is the problem of deciding whether given acyclic instance has a solution.

Here again, subtyping can be introduced to yield an Acyclic Semi-Sub-Unification problem: whether a sub-solution of given acyclic instance exists.

Proposition 3.3 ([KTU90]). *ASUP is DEXPTIME-complete.*

4 The equivalence between $\mathbf{ML}_{\leq}$ and ASSUP

This section is devoted to the proof of the following

Theorem 4.1. *ASSUP is* log*-space equivalent to* $\mathbf{ML}_{\leq}$ *typability.*

The reduction from $\mathrm{ML}_{\leq}$ to ASSUP can be inferred from the original reduction from ML to ASUP given in [KTU90]. The differences are mostly of technical nature, therefore we omit it here[2] and focus on the reduction from ASSUP to typability.

4.1 Constraining terms

For every type variable α_i we introduce object variables w_i, v_i. We assume that for every type constant $\kappa \in Q$ we there is a constant c_κ of this type, and a constant $c_{(\kappa \to \kappa)}$ of type $\kappa \to \kappa$. Now, for every monomorphic type τ, we define a term M_τ and a context $C_\tau[\,]$ with one hole simultaneously by induction on τ (bear in mind that $K = \lambda x.\lambda y.x$):

1. $M_\kappa = c_\kappa$
   $C_\kappa[\,] = c_{(\kappa \to \kappa)}[\,]$

[2] The readers keen on details are referred to [Ben96].

2. $M_{\alpha_i} = v_i w_i$
   $C_{\alpha_i}[\,] = v_i[\,]$
3. $M_{\tau_1 \to \tau_2} = \lambda x.K M_{\tau_2} C_{\tau_1}[x]$
   $C_{\tau_1 \to \tau_2}[\,] = C_{\tau_2}[[\,] M_{\tau_1}]$

Intuitively, M_τ can be used wherever enforcing τ as a lower bound is needed. Dually, the context $C_\tau[\,]$ imposes τ as an upper bound for types of the term placed inside it. These intuitions are formalised in the following

Lemma 4.2. *Let $\tau, \rho_1, \ldots, \rho_\ell, \rho_1^1, \ldots, \rho_\ell^1, \rho_1^2, \ldots, \rho_\ell^2$ be arbitrary types such that*

$$\mathrm{FV}(\tau) \subseteq \{\alpha_1, \ldots, \alpha_\ell\}.$$

$$\rho_i^1 \leq \rho_i^2 \text{ for } 1 \leq i \leq \ell$$

Furthermore, let S be a substitution such that $\rho_i^1 \leq S(\alpha_i) \leq \rho_i^2$ for $1 \leq i \leq \ell$. Then for any term N and environment E such that

$$E \supseteq \{v_i : \rho_i^1 \to \rho_i^2, w_i : \rho_i \mid 1 \leq i \leq \ell\}$$

we have for every type τ'':

1. *If*
   $$E \vdash M_\tau : \tau''$$
   then $S(\tau) \leq \tau''$
2. *If*
   $$E \vdash C_\tau[N] : \tau''$$
   then
   $$E \vdash N : S(\tau)$$

Lemma 4.3. *Let $\tau, \rho_1, \ldots, \rho_\ell$ be arbitrary types such that*

$$\mathrm{FV}(\tau) \subseteq \{\alpha_1, \ldots, \alpha_\ell\}.$$

Furthermore, let S be a substitution such that $S(\alpha_i) = \rho_i$ for $1 \leq i \leq \ell$. Then for any term N and environment E such that

$$E \supseteq \{v_i : \rho_i \to \rho_i, w_i : \rho_i \mid 1 \leq i \leq \ell\}$$

we have

1. *$E \vdash M_\tau : S(\tau)$*
2. *If $E \vdash N : S(\tau)$, then there exists τ'' such that $E \vdash C_\tau[N] : \tau''$*

4.2 The encoding

Consider an instance Γ of ASSUP. Without loss of generality we may assume that all the columns of Γ have an equal number of inequalities r.

Let type variables in zone i, for $i = 0, 1, \ldots, n$, be:

$$\alpha_{i,1},\ \alpha_{i,2},\ \ldots\ ,\ \alpha_{i,\ell_i}$$

for some $\ell_i \geq 1$, corresponding to which we introduce object variables:

$$v_{i,1}, w_{i,1},\ v_{i,2}, w_{i,2},\ \ldots,\ v_{i,\ell_i}, w_{i,\ell_i}$$

The notation M_τ and $C_\tau[\,]$ introduced earlier relative to singly subscripted variables, α_i and v_i, w_i, is now extended to doubly subscripted variables, $\alpha_{i,j}$ and $v_{i,j}, w_{i,j}$. We can assume that all the zones have an equal number ℓ of variables, i.e.,

$$\ell \ = \ \ell_0 \ = \ \ell_1 \ = \ \cdots \ = \ \ell_n$$

With these assumptions about Γ, let us introduce some building blocks which shall be used in the construction of the term M_Γ:

In the sequel by $C_i^\ell[\,]$ we shall mean the context

$$\lambda w_{i,1}. \ldots \lambda w_{i,\ell}.(\lambda v_{i,1}. \ldots \lambda v_{i,\ell}.[\,])\ \underbrace{I \ldots I}_{\ell \text{ times}}$$

where $I = \lambda x.x$.

Define

$$N_1' = \lambda z.z M_{t^{1,1}} \ldots M_{t^{1,r}}$$
$$N_1 = C_0^\ell[N_1']$$

Further for $1 \leq j \leq r$, define

$$P_{1,j} = \lambda p_1 \ldots \lambda p_l.x_i\, p_1 \ldots p_l(\lambda y_1 \ldots \lambda y_r.C_{u^{1,j}}[y_j])$$

and for $2 \leq i \leq n$ and $1 \leq j \leq r$

$$P_{i,j} = \lambda p_1 \ldots \lambda p_l.x_i\, p_1 \ldots p_l(\lambda z_1 \ldots \lambda z_r \lambda y_1 \ldots \lambda y_r.C_{u^{i,j}}[y_j])$$

and

$$N_i' = \lambda z.z P_{i-1,1} \ldots P_{i-1,r} M_{t^{i,1}} \ldots M_{t^{i,r}} \tag{1}$$
$$N_i = C_i^\ell[N_i'] \tag{2}$$

Finally we define the term M_Γ as follows:

$$\begin{aligned} M_\Gamma \equiv\ & \textbf{let}\ \ x_1 \ = \ N_1 \quad \textbf{in} \\ & \textbf{let}\ \ x_2 \ = \ N_2 \quad \textbf{in} \\ & \qquad \vdots \\ & \textbf{let}\ \ x_{n-1} = N_{n-1}\ \textbf{in} \\ & \textbf{let}\ \ x_n \ = \ N_n \quad \textbf{in}\ \ N_{n+1} \end{aligned}$$

4.3 Soundness of the encoding

Theorem 4.4. *If Γ has a sub-solution then M_Γ is typable.*

Proof. Suppose Γ has a sub-solution S:

$$S = [\alpha_{i,j} := \rho_{i,j} \mid i = 0, \ldots, n, \text{ and } j = 1, \ldots, \ell\,]$$

There are therefore substitutions $R_{i,j}$ such that: $R_{i,j}(S(t^{i,j})) \leq S(u^{i,j})$, for every $i = 1, \ldots, n$ and $j = 1, \ldots, r$. We shall show that M_Γ is typable. For $i = 1, \ldots, n+1$, define the environment E_i:

$$E_i = \{v_{i-1,j} : \rho_{i-1,j} \to \rho_{i-1,j}, w_{i-1,j} : \rho_{i-1,j} \mid 1 \leq j \leq \ell\}$$

For $i = 1, \ldots, n$ and $j = 1, \ldots, r$, by Lemma 4.3:

$$E_i \vdash M_{t^{i,j}} : S(t^{i,j})$$

and for every $i = 2, \ldots, n+1$ and $j = 1, \ldots, r$ there exists a (monomorphic) type $\psi_{i,j}$, such that

$$E_i(y_j : S(u^{i-1,j})) \vdash C_{u^{i-1,j}}[y_j] : \psi_{i,j}$$

We shall prove that $\vdash N_1 : \xi_1$, where:

$$\tau_1 = \rho_{0,1} \to \cdots \to \rho_{0,\ell} \to (S(t^{1,1}) \to \cdots \to S(t^{1,r}) \to \beta_1) \to \beta_1$$

where β_1 is a type variable.

Let

$$\chi_1 = S(t^{1,1}) \to \cdots \to S(t^{1,r}) \to \beta_1$$

The desired property of N_1 is easily seen from the following derivation:

$$\dfrac{\dfrac{\dfrac{E_1(z : \chi_1) \vdash z : \chi_1 \quad E_1(z : \chi_1) \vdash M_{t^{1,1}} : S(t^{1,1})}{\vdots}}{E_1(z : \chi_1) \vdash z M_{t^{1,1}} \ldots M_{t^{1,r}} : \beta_1}}{E_1 \vdash N_1' : \chi_1 \to \beta_1}$$

$$\dfrac{\vdots}{\vdash C_1^\ell[N_1'] : \xi_1}$$

By an argument similar to the one about N_1, one may prove that

$$\begin{aligned} x_1 : \xi_1 \vdash N_2 : \rho_{1,1} \to \cdots \to \rho_{1,\ell} \to \\ (\theta_{1,1} \to \cdots \to \theta_{1,r} \to \\ S(t^{2,1}) \to \cdots \to S(t^{2,r}) \to \beta_2) \to \beta_2 \end{aligned}$$

we shall call this type ξ_2. Similarly, for $i = 3, \ldots, n+1$ and $j = 1, \ldots, r$, using the fact that $R_{i-1,j}(S(t^{i-1,j})) \leq S(u^{i-1,j})$, it is not difficult to check that:

$$\{x_{i-1} : \forall \bar{\varphi}.\, \xi_{i-1}\} \vdash N_i : \xi_i$$

where

$$\xi_i = \rho_{i-1,1} \to \cdots \to \rho_{i-1,\ell} \to (\theta_{i-1,1} \to \cdots \to \theta_{i-1,r} \to S(t^{i,1}) \to \cdots \to S(t^{i,r}) \to \beta_i) \to \beta_i$$

$$\theta_{i,j} = R_{i,j}(\rho_{i,1}) \to \cdots \to R_{i,j}(\rho_{i,\ell}) \to \psi_{i,j}$$

for some type $\psi_{i,j}$.

4.4 Completeness of the encoding

Theorem 4.5. *If M_Γ is typable then Γ has a sub-solution, i.e. there exist substitutions $S, R_{1,1}, \ldots, R_{n,r}$ such that*

$$R_{i,j}(S(t_{i,j})) \leq S(u_{i,j}) \text{ for } 1 \leq i \leq n, \quad 1 \leq j \leq r$$

Proof. Suppose that M_Γ is typable. This means that, for $i = 1, \ldots, n+1$, N_i is typable in an environment E_i of the form:

$$E_i \;=\; \{x_1 : \sigma_1, \ldots, x_{i-1} : \sigma_{i-1}\}$$

where $\sigma_1, \ldots \sigma_n$ are type schemes Although the only free variable in N_i is x_{i-1}, E_i must include a type assumption for every variable whose binding includes N_i in its scope. Note that $\emptyset = E_1 \subset \cdots \subset E_{n+1}$.

Let V_i denote the set of variables in zone i of Γ. Note that

$$FV(N_i) = \begin{cases} V_0 & \text{if } i = 1 \\ V_{i-1} \cup \{x_{i-1}\} & \text{otherwise} \end{cases} \tag{3}$$

If $N_i = C_i^\ell[N_i']$ is typable in E_i, then there exists an environment E_i^v and types $\xi_i, \rho_{i,1}^1, \ldots, \rho_{i,\ell}^1, \rho_{i,1}^2, \ldots, \rho_{i,\ell}^2$ such that

$$\rho_{i,j}^1 \leq \rho_{i,j}^2 \tag{4}$$

$$E_i^v(v_{i,j}) = \rho_{i,j}^1 \to \rho_{i,j}^2 \tag{5}$$

$$E_i^v \vdash N_i : \xi_i \tag{6}$$

Take any S such that $\rho_{i,j}^1 \leq S(\alpha_{i,j}) \leq \rho_{i,j}^1$ for $1 \leq i \leq n$, $1 \leq j \leq \ell$. The existence of such S follows from acyclicity of Γ and (3). Note that S and E_i^v satisfy assumptions of Lemma 4.2.

First let us focus on the term N_1. The type ξ_1 must be of the form

$$\xi_1 = \rho_{0,1} \to \cdots \to \rho_{0,\ell} \to (\tau_{1,1} \to \cdots \to \tau_{1,r} \to \varphi_1) \to \psi_1$$

where

$$S(t^{1,j}) \leq \tau_{1,j} \text{ for } 1 \leq j \leq r \tag{7}$$

$$\varphi_1 \leq \psi_1 \tag{8}$$

Now consider the term $P_{1,j}$. There exist an environment $E^y_{1,j} \supseteq E^v_1$ and a type $\psi_{1,j}$ such that

$$E^y_{1,j} \vdash C_{u^{i,j}} : \psi_{1,j}$$
$$E^y_{1,j} \vdash y : S(u_{1,j})$$

Since $P_{1,j}$ occurs in a context **let** $x_1 = N_1$ **in** $\ldots$, the occurrence of x_1 in it must be assigned a type ξ^j_1 that is an instance of $\forall\bar{\varphi}.\,\xi_1$. Therefore by the Lemma 1.7, there exists a substitution $R_{1,j}$ such that

$$R_{1,j}(\xi_1) \leq \xi^j_1. \tag{9}$$

From this, it follows that ξ^j_1 must be of the form

$$\xi^j_1 = \rho^j_{0,1} \to \cdots \to \rho^j_{0,\ell} \to (\tau^j_{1,1} \to \cdots \to \tau^j_{1,r} \to \varphi^j_1) \to \psi^j_1$$

since x_1 occurs in $P_{1,j}$ in the context $x_1 p_1 \ldots p_\ell(\lambda y_1 \ldots \lambda y_r . C_{u^1,j}[y_j]$, by Lemma 4.2 we have

$$\tau^j_{1,j} \leq S(u^{1,j})$$

Since $\tau_{1,j}$ occurs positively in ξ_1, we have $R_{1,j}(S(t^{1,j}) \leq R_{1,j}(\tau_{1,j})$. In turn $R_{1,j}(\tau_{1,j}) \leq \tau^j_{1,j}$ by 9. From this inequalities, we conclude that

$$R_{1,j}(S(t^{1,j})) \leq S(u^{1,j}).$$

By the same token, for $2 \leq i \leq n+1$,[3] the type ξ_i must be of the form

$$\begin{aligned}\xi_i = {} & \rho_{i-1,1} \to \cdots \to \rho_{i-1,\ell} \to \\ & (\theta_{i-1,1} \to \cdots \to \theta_{i-1,r} \to \tau_{i,1} \to \cdots \to \tau_{i,r} \to \varphi_i) \to \psi_i\end{aligned}$$

and for all j the occurrence of x_i in $P_{i,j}$ must be assigned a type of the form

$$\begin{aligned}\xi^j_i = {} & \rho^j_{i-1,1} \to \cdots \to \rho^j_{i-1,\ell} \to \\ & (\theta^j_{i-1,1} \to \cdots \to \theta^j_{i-1,r} \to \tau^j_{i,1} \to \cdots \to \tau^j_{i,r} \to \varphi^j_i) \to \psi^j_i\end{aligned}$$

and there must be a substitution $R_{i,j}$ such that

$$R_{i,j}(\xi_i) \leq \xi^j_i. \tag{10}$$

From this and from the construction of $P_{i,j}$ we can again conclude that in fact $R_{i,j}(S(t^{i,j})) \leq R_{i,j}(\tau_{i,j}) \leq \tau^j_{i,j}$ and hence

$$R_{i,j}(S(t^{i,j})) \leq S(u^{i,j})$$

for all i and j.

[3] Note that there is no x_{n+1}, but there is N_{n+1}.

References

[Ben93] Marcin Benke. Efficient type reconstruction in the presence of inheritance (extended abstract). In *Proc. Int. Symp. MFCS 1993*. Springer Verlag, 1993.

[Ben96] Marcin Benke. An algebraic characterization of typability in ML with subtyping. Technical Report TR96-14(235), Institute of Informatics, Warsaw University, December 1996. Available from `http://zls.mimuw.edu.pl/~ben/Papers/`.

[Ben98] Marcin Benke. *Complexity of type reconstruction in programming languages with subtyping*. PhD thesis, Warsaw University, 1998.

[CDDK86] D. Clément, J. Despeyroux, T. Despeyroux, and G. Kahn. A simple applicative language: Mini-ML. In *Proc. ACM Conference on Lisp and Functional Programming*, pages 13–27, 1986.

[DM82] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Conf. Rec. ACM Symp. Principles of Programming Languages*, pages 207–211, 1982.

[FM89] You-Chin Fuh and Prateek Mishra. Polymorphic subtype inference: closing the theory-practice gap. In *Proc. Theory and Practice of Software Development*, pages 167–184, March 1989.

[FM90] You-Chin Fuh and Prateek Mishra. Type inference with subtypes. *Theoretical Computer Science*, 73:155–176, 1990.

[HM95] My Hoang and John C. Mitchell. Lower bounds on type inference with subtypes. In *Conf. Rec. ACM Symp. Principles of Programming Languages*, 1995.

[Jim96] Trevor Jim. What are principal typings and what are they good for? In *Conf. Rec. ACM Symp. Principles of Programming Languages*, pages 42–53, 1996.

[KTU89] Assaf J. Kfoury, Jerzy Tiuryn, and Paweł Urzyczyn. An analysis of ML typability. Technical Report 89-009, Boston University, 1989.

[KTU90] Assaf J. Kfoury, Jerzy Tiuryn, and Paweł. Urzyczyn. ML typability is DEXPTIME-complete. In *Proc. 15th Colloq. on Trees in Algebra and Programming*, pages 206–220. Springer LNCS 431, 1990.

[KTU93] Assaf J. Kfoury, Jerzy Tiuryn, and Paweł Urzyczyn. The undecidability of the semi-unification problem. *Information and Computation*, 102(1):83–101, January 1993.

[KTU94] Assaf J. Kfoury, Jerzy Tiuryn, and Paweł Urzyczyn. An analysis of ML typability. *Journal of the ACM*, 41(2):368–398, March 1994.

[Mit84] John C. Mitchell. Coercion and type inference. In *Conf. Rec. ACM Symp. Principles of Programming Languages*, pages 175–185, 1984.

[Mit88] John C. Mitchell. Polymorphic type inference and containment. *Information and Computation*, 76(2/3):211–249, 1988. Reprinted in *Logical Foundations of Functional Programming*, ed. G. Huet, Addison-Wesley (1990) 153–194.

[OL96] Martin Odersky and Konstantin Läufer. Putting type annotations to work. In *Conf. Rec. ACM Symp. Principles of Programming Languages*, pages 54–67, 1996.

[Smi94] Geoffrey S. Smith. Principal type schemes for functional programs with overloading and subtyping. *Science of Computer Programming*, 23:197–226, 1994.

[TU96] Jerzy Tiuryn and Paweł Urzyczyn. The subtyping problem for second-order types is undecidable. In *Proc. 11th IEEE Symposium on Logic in Computer Science*, 1996.

Static Analysis of Processes for No Read-Up and No Write-Down

Chiara Bodei, Pierpaolo Degano,[1]
Flemming Nielson, Hanne Riis Nielson[2]

[1] Dipartimento di Informatica, Università di Pisa
Corso Italia 40, I-56100 Pisa, Italy
{chiara,degano}@di.unipi.it
[2] Computer Science Department, Aarhus University
Ny Munkegade, DK-8000 Aarhus C, Denmark
{fn,hrn}@daimi.au.dk

Abstract. We study a variant of the *no read-up/no write-down* security property of Bell and LaPadula for processes in the π-calculus. Once processes are given levels of security clearance, we statically check that a process at a high level never sends names to processes at a lower level. The static check is based on a Control Flow Analysis for the π-calculus that establishes a super-set of the set of names to which a given name may be bound and of the set of names that may be sent and received along a given channel, taking into account its directionality. The static check is shown to imply the natural dynamic condition.

1 Introduction

System security is receiving more and more attention but obtaining precise answers is often undecidable [10]. However, static analysis provides a repertoire of automatic and decidable methods for analysing properties of programs, and these can often be used as the basis for establishing security properties. We use here Control Flow Analysis that is a static technique for predicting safe and computable approximations to the set of values that the objects of a program may assume during its execution. To circumvent the undecidability issues the analysis "errs on the safe side" by never omitting values that arise, but perhaps including values that never arise in the semantics. The approach is related to Data Flow Analysis and Abstract Interpretation and naturally leads to a general treatment of semantic correctness and the existence of best solutions. A more widely used alternative for calculi of computation is to use Type Systems; they also allow a clear statement of semantic correctness (sometimes called type soundness) and allow to study whether or not best solutions exist (in the form of principal types). The interplay between Type Systems and Control Flow Analysis is not yet fully understood, but simple Type Systems and simple Control Flow Analyses seem to be equally expressive; however, a main difference is that the Control Flow Analysis guarantees that best solutions always exist whereas many Type Systems do not admit principal types (and occasionally the issue is left open when presenting the type system).

W. Thomas (Ed.): FOSSACS'99, LNCS 1578, pp. 120-134, 1999.

Here we elaborate on our proposal made in [5], that presents a Control Flow Analysis for the π-calculus, which is a model of concurrent communicating processes based on naming, and where we applied it to statically check that a process has *no leaks*, i.e. that a process confines a set of values, devised to be secret, within itself. Our new analysis is more accurate than the one in [5], because a more careful check is made on the input and the output prefixes, so as to identify unreachable code. The result of our Control Flow Analysis establishes a super-set of the set of names to which a given name may be bound and of the sets of names that may be sent and received along a given channel, when used by a process with clearance l. These super-sets give rise to *solutions* of the form (ρ, σ) and we formulate the Control Flow Analysis as a specification of the correctness of a candidate solution. This takes the form of judgements of the form $(\rho, \sigma) \models^{l}_{me} P$ (where *me* will be explained later and l is the level of security clearance), and a set of clauses that operate on them. We show that best solutions always exist and we establish the semantic correctness of solutions in the form of a subject-reduction result.

We apply our analysis for statically checking a dynamic version of the *no read-up/no write-down* property of Bell and LaPadula [4, 11, 12]: a process classified at a high level cannot write any value to a process of low level, while communications in any other direction is permitted. This requirement is part of a security model, based on a multi-level access control, see [4, 10]. We first define a static check on solutions (ρ, σ), called *discreetness*, for when a process respects the classification hierarchy. Then we show that a discreet process enjoys the dynamic version of the no read-up/no write-down property.

Overview. Section 2 gives the syntax and the operational semantics of our version of the π-calculus with clearance levels. Our Control Flow Analysis is in Section 3, together with the semantic correctness of solutions. The no read-up/no write-down property is then studied in Section 4. Some proofs are omitted or only sketched because of lack of space.

2 The π-calculus

Syntax. In this section we briefly recall the π-calculus [21], a model of concurrent communicating processes based on the notion of *naming*. The formulation of our analysis requires a minor extension to the standard syntax of the π-calculus, namely assigning "channels" to the binding occurrences of names within restrictions and "binders" to the binding occurrences of names within input prefixes; as will emerge later, this is because of the α-conversion allowed by the structural congruence, and the syntactic extension will allow to compute a super-set of the actual links that a name can denote. Also, we need a further extension to assign a security level to π-calculus processes.

More precisely, we introduce a finite set $\mathcal{L} = \{\#\} \cup \{0, \cdots, k\}$ of level labels, with metavariable l, consisting both of natural numbers and of the distinguished label #, intended as the label of the environment, which is intuitively assumed

to have "no level". The set $\mathcal{L}$ is ordered (see Fig. 1) with the usual $\leq$ relation on natural numbers and assuming that $\forall l \in \mathcal{L} \setminus \{\#\}, l \not\leq \#$ and $\# \not\leq l$.

Definition 1. *Let $\mathcal{N}$ be a infinite set of* names *ranged over by $a, b, \cdots, x, y$ and let τ be a distinguished element such that $\mathcal{N} \cap \{\tau\} = \emptyset$. Also let $\mathcal{B}$ be a non-empty set of* binders *ranged over by $\beta, \beta', \cdots$; and let $\mathcal{C}$ be a non-empty set of* channels *ranged over by $\chi, \chi', \cdots$; moreover let $\mathcal{B} \cup \mathcal{C}$ be the set of* markers. *Then* processes*, denoted by $P, P_1, P_2, Q, R, \cdots \in \mathcal{P}$ are built from names according to the syntax*

$$P ::= \mathbf{0} \mid \mu.P \mid P + P \mid P|P \mid (\nu x^{\chi})P \mid [x = y]P \mid\ !P \mid \langle P \rangle^{l}$$

where μ may either be $x(y^{\beta})$ for input*, or $\overline{x}y$ for* output *or τ for* silent *moves, and where $l \in \mathcal{L} \setminus \{\#\}$. Hereafter, the trailing $\mathbf{0}$ will be omitted (i.e. we write π instead of $\pi.\mathbf{0}$).*

In this paper we consider the *early* operational semantics defined in SOS style. The intuition of process constructors is the standard one. Indeed, $\langle P \rangle^{l}$ behaves just as P but expresses that P has level l, where $l \in \mathcal{L} \setminus \{\#\}$. The labels of transitions are τ for silent actions, xy for free input, $\overline{x}y$ for free output, and $\overline{x}(y)$ for bound output. We will use μ as a metavariable for the labels of transitions. We recall the notion of free names $fn(\mu)$, bound names $bn(\mu)$, and names $n(\mu) = fn(\mu) \cup bn(\mu)$ of a label μ. Also two partial functions, *sbj* and *obj*, are defined that give, respectively, the subject x and the object y of input and output actions, i.e. the channel x on which y is transmitted.

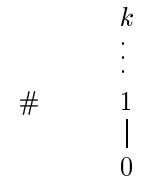


Fig. 1. Levels of processes ($i \leq i + 1$) and of the environment.

Kind	μ	$fn(\mu)$	$bn(\mu)$	$sbj(\mu)$	$obj(\mu)$
Silent move	τ	$\emptyset$	$\emptyset$		
Free input and output	$xy, \overline{x}y$	$\{x, y\}$	$\emptyset$	x	y
Bound output	$\overline{x}(y)$	$\{x\}$	$\{y\}$	x	y

Functions fn, bn and n are extended in the obvious way to processes.

Congruence. Below we shall need the *structural congruence* $\equiv$ on processes, defined as in [22], apart from the last rule, where restrictions can be exchanged only if the restricted names are different, because otherwise $(\nu x^{\chi})P \equiv (\nu x^{\chi'})P$. Then $\equiv$ is the least congruence satisfying:

- if P and Q are α-equivalent ($P =_{\alpha} Q$) then $P \equiv Q$;

$$Act: \vdash^l \mu.P \xrightarrow[\epsilon,\epsilon]{\mu} P, \quad \mu \neq x(y^\beta) \qquad Ein: \vdash^l x(y^\beta).P \xrightarrow[\lambda,\epsilon]{x\,w} P\{w/y\}$$

$$Par: \frac{\vdash^l P_1 \xrightarrow[\lambda,L]{\mu} Q_1}{\vdash^l P_1|P_2 \xrightarrow[\lambda,L]{\mu} Q_1|P_2}, bn(\mu) \cap fn(P_2) = \emptyset \qquad Sum: \frac{\vdash^l P_1 \xrightarrow[\lambda,L]{\mu} Q_1}{\vdash^l P_1 + P_2 \xrightarrow[\lambda,L]{\mu} Q_1}$$

$$Res: \frac{\vdash^l P \xrightarrow[\lambda,L]{\mu} Q}{\vdash^l (\nu x^\chi)P \xrightarrow[\lambda,L]{\mu} (\nu x^\chi)Q}, x \notin n(\mu) \qquad Open: \frac{\vdash^l P \xrightarrow[\epsilon,L]{\overline{x}y} Q}{\vdash^l (\nu y^\chi)P \xrightarrow[\chi,L]{\overline{x}(y)} Q}, y \neq x$$

$$Close: \frac{\vdash^l P_1 \xrightarrow[\chi,L_1]{\overline{x}(y)} Q_1, \vdash^l P_2 \xrightarrow[\chi,L_2]{x\,y} Q_2}{\vdash^l P_1|P_2 \xrightarrow[\epsilon,\epsilon]{\tau} (\nu y^\chi)(Q_1|Q_2)}, y \notin fn(P_2) \qquad Com: \frac{\vdash^l P_1 \xrightarrow[\epsilon,L_1]{\overline{x}y} Q_1, \vdash^l P_2 \xrightarrow[\epsilon,L_2]{x\,y} Q_2}{\vdash^l P_1|P_2 \xrightarrow[\epsilon,\epsilon]{\tau} Q_1|Q_2}$$

$$Match: \frac{\vdash^l P \xrightarrow[\lambda,L]{\mu} Q}{\vdash^l [x = x]P \xrightarrow[\lambda,L]{\mu} Q} \qquad Var: \frac{P' \equiv P, \vdash^l P \xrightarrow[\lambda,L]{\mu} Q \equiv Q'}{\vdash^l P' \xrightarrow[\lambda,L]{\mu} Q'}$$

$$Lev: \frac{\vdash^l P \xrightarrow[\lambda,L]{\mu} Q}{\vdash^{l'} \langle P \rangle^l \xrightarrow[\lambda,L\,l]{\mu} \langle Q \rangle^l}$$

Table 1. Early transition system for the π-calculus with security levels.

- $(\mathcal{P}/_\equiv, +, \mathbf{0})$ and $(\mathcal{P}/_\equiv, |, \mathbf{0})$ are commutative monoids;
- $!P \equiv P|!P$.
- $(\nu x^\chi)(\nu y^{\chi'})P \equiv (\nu y^{\chi'})(\nu x^\chi)P$ if $x \neq y$, $(\nu x^\chi)(P_1|P_2) \equiv (\nu x^\chi)P_1|P_2$ if $x \notin fn(P_2)$, and $(\nu x^\chi)P \equiv P$ if $x \notin fn(P)$;

Semantics. Table 1 shows the annotated *early* transition system of the π-calculus. The transitions have the form $\vdash^l P \xrightarrow[\lambda,L]{\mu} Q$, with $\lambda \in \mathcal{C} \cup \{\epsilon\}$, $l \in \mathcal{L}$, $L \in \mathcal{L}^*$. The string of level labels L records the clearances passed through while deducing the transition, while the index l on $\vdash$ represents the current level. Note that the sequence of security levels of the sender and of the receiver are discarded when a communication is derived, leading to a transition of the form $\vdash^l P \xrightarrow[\epsilon,\epsilon]{\tau} Q$ (see the rules *Com* and *Close* in Tab. 1). As far as the label $\lambda \in \mathcal{C} \cup \{\epsilon\}$ is concerned, we have that ϵ is used in all cases, apart from extrusions or when input transitions have to be used as a premise of a *Close* rule. In that case the label is χ and records the actual channel to be associated with the object of the input. Rule *Match* takes care of matching; we have formulated it as a transition rather than

as a structural law in order to simplify the technical development (compare [5]). Rule Var ensures that all the rules and axioms can also be used upon all its variants.

3 Control Flow Analysis

The result of our analysis for a process P (with respect to an additional marker environment me for associating names with markers and a label l recording a clearance) is a pair (ρ, σ): the abstract environment ρ gives information about which channels names can be bound to, while the abstract communication environment $\sigma = \langle \sigma_{in}, \sigma_{out} \rangle$ gives information about the channels sent and received by the sub-processes of P with clearance l. Besides the usage of security levels, our present solutions refine those in [5]. In fact, the σ component controls the values that pass along a channel, more accurately than there. We now make the above more precise.

3.1 Validation

To *validate* the correctness of a proposed solution (ρ, σ) we state a set of clauses operating upon judgments of the form:

$$(\rho, \sigma) \ \models^{l}_{me} P$$

The purpose of me, l, ρ, σ is clarified by:

- $me : \mathcal{N} \to (\mathcal{B} \cup \mathcal{C})$ is the *marker environment* that associates names (in particular the free names of a process) with the appropriate channel or binder where the name was introduced; so $me(x)$ will be the marker (in $\mathcal{B}$ or $\mathcal{C}$) where the current name x is bound.
- $l \in \mathcal{L}$ keeps track of the current security level that the process under validation has.
- $\rho : \mathcal{B} \to \wp(\mathcal{C})$ is the *abstract environment* that associates binders with the set of channels that they can be bound to; more precisely, $\rho(\beta)$ must include the set of channels that β could evaluate to.
  By setting $\forall \chi : \rho(\chi) = \{\chi\}$ we shall allow to regard the abstract environment as a function $\rho : (\mathcal{B} \cup \mathcal{C}) \to \wp(\mathcal{C})$.
- $\sigma_{in}, \sigma_{out} : \mathcal{L} \to (\mathcal{C} \to \wp(\mathcal{C}))$ constitute the *abstract communication environment*. They give the set of the channels that can be bound to the possible objects of an input and an output action[1] respectively, performed by the sub-processes labelled by l, on a given channel χ.

[1] The relation between the abstract communication environment σ and the abstract channel environment κ in [5] is $\forall \chi \in \mathcal{C} : \kappa(\chi) = \bigcup_{l \in \mathcal{L}} (\sigma_{in}(l)(\chi) \cup \sigma_{out}(l)(\chi))$ in case of least solutions.

$$
\begin{array}{lll}
(\rho,\sigma) \models^l_{me} \mathbf{0} & \text{iff } \mathit{true} \\
(\rho,\sigma) \models^l_{me} \tau.P & \text{iff } (\rho,\sigma) \models^l_{me} P \\
(\rho,\sigma) \models^l_{me} \overline{x}y.P & \text{iff } (\rho(me(y)) \neq \emptyset \wedge \rho(me(x)) \neq \emptyset) \Rightarrow (\rho,\sigma) \models^l_{me} P \wedge \\
 & \quad \forall \chi \in \rho(me(x)) : \rho(me(y)) \subseteq \sigma_{out}(l)(\chi) \\
(\rho,\sigma) \models^l_{me} x(y^\beta).P & \text{iff } \bigcup_{l' \in \mathcal{L}, \chi \in \rho(me(x))} \sigma_{out}(l')(\chi) \neq \emptyset \Rightarrow (\rho,\sigma) \models^l_{me[y \mapsto \beta]} P \wedge \\
 & \quad \forall \chi \in \rho(me(x)) : \bigcup_{l' \in \mathcal{L}} \sigma_{out}(l')(\chi) \subseteq \sigma_{in}(l)(\chi) \wedge \\
 & \quad \forall \chi \in \rho(me(x)) : \sigma_{in}(l)(\chi) \subseteq \rho(\beta) \\
(\rho,\sigma) \models^l_{me} P_1 + P_2 & \text{iff } (\rho,\sigma) \models^l_{me} P_1 \wedge (\rho,\sigma) \models^l_{me} P_2 \\
(\rho,\sigma) \models^l_{me} P_1 | P_2 & \text{iff } (\rho,\sigma) \models^l_{me} P_1 \wedge (\rho,\sigma) \models^l_{me} P_2 \\
(\rho,\sigma) \models^l_{me} (\nu x^\chi)P & \text{iff } (\rho,\sigma) \models^l_{me[x \mapsto \chi]} P \\
(\rho,\sigma) \models^l_{me} [x=y]P & \text{iff } (\rho(me(x)) \cap \rho(me(y)) \neq \emptyset \vee me(x) = me(y)) \\
 & \quad \Rightarrow (\rho,\sigma) \models^l_{me} P \\
(\rho,\sigma) \models^l_{me} !P & \text{iff } (\rho,\sigma) \models^l_{me} P \\
(\rho,\sigma) \models^l_{me} \langle P \rangle^{l'} & \text{iff } (\rho,\sigma) \models^{l'}_{me} P \wedge \sigma_{in}(l') \sqsubseteq \sigma_{in}(l) \wedge \sigma_{out}(l') \sqsubseteq \sigma_{out}(l)
\end{array}
$$

Table 2. Control flow analysis for the π-calculus.

Note that we use a marker environment because the identity of names is not preserved under α-conversions (see rules Ein and Var). In particular, it would not suffice to "α-rename the program apart" because as in [5] this property is not preserved under reduction.

The analysis is in Tab. 2. As we are analysing a process P from scratch, we assume that the initial clearance label is #. All the rules for validating a compound process require that the components are validated, apart from the rules for output, input and matching. The rules for output and input require a preliminary check to decide whether the continuation P needs to be validated and this makes the analysis more accurate than the one in [5]. In case of output, one has to make sure that the (set of channels associated with the) object is bound to some channels and similarly for the subject. In the case of input we control that the (set of channels associated with the) subject has some channels to read; actually, we ensure that some value can be sent along the subject. The last conjunct of the rule for output takes care of the clearance l of the process under analysis. The channels that can be bound to the object of an output action along channel χ must be included in $\sigma_{out}(l)(\chi)$. Analogously for the case of input, where we ensure that $\sigma_{in}(l)(\chi)$ and $\rho(\beta)$ contain all the outputs on $\chi \in \rho(me(x))$, regardless of the clearance level l of the sending process. The condition for matching says that P needs to be validated if there is at least one

channel to which both x and y can evaluate; note that both can evaluate to $\emptyset$ and thus we need to check whether they actually denote the same channel by allowing $me(x) = me(y)$. The rule for the process $\langle P \rangle^{l'}$ simply says that the channels that can be read and written by it must be included in those read or written by its surrounding process, labelled l. It makes use of the following definition.

Definition 2. *The set of proposed solutions can be partially ordered by setting* $(\rho, \sigma) \sqsubseteq (\rho', \sigma')$ *iff* $\forall \beta \in \mathcal{B} : \rho(\beta) \subseteq \rho'(\beta)$, $\forall \chi \in \mathcal{C}, \forall l \in \mathcal{L} : \sigma_{in}(l)(\chi) \subseteq \sigma'_{in}(l)(\chi)$ *and* $\forall \chi \in \mathcal{C}, \forall l \in \mathcal{L} : \sigma_{out}(l)(\chi) \subseteq \sigma'_{out}(l)(\chi)$.

It is immediate that this suffices for making the set of proposed solutions into a complete lattice; using standard notation we write $(\rho, \sigma) \sqcup (\rho', \sigma')$ for the binary least upper bound (defined pointwise), $\sqcap \mathcal{I}$ for the greatest lower bound of a set $\mathcal{I}$ of proposed solutions (also defined pointwise), and $(\bot, \bot)$ for the least element (where $\bot$ maps everything[2] to $\emptyset$).

Example 1. Consider the following process

$$S = !(R \mid Q \mid P) =$$

$$!(\langle \overline{a}b.\overline{a}b.\overline{b}c \rangle^{l_R} \mid \langle a(x^{\beta_x}).\overline{x}x \rangle^{l_Q} \mid \langle a(y^{\beta_y}).y(z^{\beta_z}).([y = z]\overline{y}a \ + \ y(w^{\beta_w}) \rangle^{l_P}),$$

where the marker environment me is such that $me(fv) = \chi_{fv}$ for all the free names $fv \in \{a, b, c\}$. The pair (ρ, σ) is defined as follows, where the bound names are $bv \in \{x, y, z, w\}$ and the level labels are $l \in \{\#, l_R, l_Q, l_P\}$:

$$\rho(\beta_{bv}) = \begin{cases} \{\chi_b\} & \text{if } bv = x, y \\ \{\chi_a, \chi_b, \chi_c\} & \text{if } bv = z, w \end{cases} \qquad \text{(Recall that } \rho(\chi) = \{\chi\})$$

$$\sigma_{in}(l)(\chi_a) = \begin{cases} \{\chi_b\} & \text{if } l = \# \\ \emptyset & \text{if } l = l_R \\ \{\chi_b\} & \text{if } l = l_Q, l_P \end{cases} \qquad \sigma_{out}(l)(\chi_a) = \begin{cases} \{\chi_b\} & \text{if } l = \#, l_R \\ \emptyset & \text{if } l = l_Q, l_P \end{cases}$$

$$\sigma_{in}(l)(\chi_b) = \begin{cases} \{\chi_a, \chi_b, \chi_c\} & \text{if } l = \# \\ \emptyset & \text{if } l = l_R, l_Q \\ \{\chi_a, \chi_b, \chi_c\} & \text{if } l = l_P \end{cases} \qquad \sigma_{out}(l)(\chi_b) = \begin{cases} \{\chi_a, \chi_b, \chi_c\} & \text{if } l = \# \\ \{\chi_c\} & \text{if } l = l_R \\ \{\chi_b\} & \text{if } l = l_Q \\ \{\chi_a\} & \text{if } l = l_P \end{cases}$$

$$\sigma_{in}(l)(\chi_c) = \emptyset, \quad \text{if } l = \#, l_R, l_Q, l_P \qquad \sigma_{out}(l)(\chi_c) = \emptyset, \quad \text{if } l = \#, l_R, l_Q, l_P$$

A simple check shows that $(\rho, \sigma) \models^{\#}_{me} S$.

3.2 Existence of solution

So far we have only considered a procedure for validating whether or not a proposed solution (ρ, σ) is in fact acceptable. We now show that there always exists a least choice of (ρ, σ) that is acceptable in the manner of Tab. 2.

[2] However, note that $\bot_\rho : \mathcal{B} \to \wp(\mathcal{C})$ viewed as $\bot_\rho : (\mathcal{B} \cup \mathcal{C}) \to \wp(\mathcal{C})$ has $\bot_\rho(\beta) = \emptyset$ for $\beta \in \mathcal{B}$ but $\bot_\rho(\chi) = \{\chi\}$ for $\chi \in \mathcal{C}$ (rather than $\bot_\rho(\chi) = \emptyset$).

Definition 3. *A set $\mathcal{I}$ of proposed solutions is a* Moore family *if and only if it contains $\sqcap\mathcal{J}$ for all $\mathcal{J} \subseteq \mathcal{I}$ (in particular $\mathcal{J} = \emptyset$ and $\mathcal{J} = \mathcal{I}$).*

This is sometimes called the model intersection property and is fundamental for many approaches to program analysis [7]. When $\mathcal{I}$ is a Moore family it contains a greatest element ($\sqcap\emptyset$) as well as a least element ($\sqcap\mathcal{I}$). The following theorem then guarantees that there always is a least solution to the specification in Tab. 2.

Theorem 1. *The set $\{(\rho,\sigma) \mid (\rho,\sigma) \models^l_{me} P\}$ is a Moore family for all me, l, P.*

Proof. By induction on P.

There is also a constructive procedure for obtaining the least solution; it has a low polynomial complexity. Essentially, establishing $(\rho,\sigma) \models^l_{me} P$ amounts to checking a number of individual constraints. In the full paper we define a function $\mathcal{G}_C[\![P]\!]_{me}$ for explicitly extracting these constraints, proceeding by induction on the structure of processes. This is not entirely straightforward because of the conditional analysis of the continuation process in the case of output, input and matching. The resulting constraints can be solved in low polynomial time.

3.3 Correctness

We state now some auxiliary results that will allow us to establish semantic correctness of our analysis. They are all independent of the semantics and only rely on Tab. 2; their proofs are all by induction.

Lemma 1. *Assume that $\forall x \in fn(P) : me_1(x) = me_2(x)$; then $(\rho,\sigma) \models^l_{me_1} P$ if and only if $(\rho,\sigma) \models^l_{me_2} P$.*

Lemma 2. *Assume that $me(y) = me(z)$; then $(\rho,\sigma) \models^l_{me} P$ if and only if $(\rho,\sigma) \models^l_{me} P\{z/y\}$.*

Corollary 1. *Assume that $z \notin fn(P)$ and $\eta \in \mathcal{B} \cup \mathcal{C}$; then $(\rho,\sigma) \models^l_{me[y\mapsto\eta]} P$ if and only if $(\rho,\sigma) \models^l_{me[z\mapsto\eta]} P\{z/y\}$.*

Lemma 3. *Assume that $P \equiv Q$; then $(\rho,\sigma) \models^l_{me} P$ iff $(\rho,\sigma) \models^l_{me} Q$.*

Lemma 4. *Assume that $(\rho,\sigma) \models^l_{me} P$ and $me(w) \in \rho(me(z)) \subseteq \mathcal{C}$; then $(\rho,\sigma) \models^l_{me} P\{w/z\}$.*

Subject reduction. To establish the semantic correctness of our analysis we rely on the definition of the early semantics in Tab. 1 as well as on the analysis in Tab. 2. The subject reduction result below applies to *all* the solutions of the analysis, and hence in particular to the least. The operational semantics only rewrites processes at "top level" where it is natural to demand that all free names are bound to channels (rather than binders); this is formalised by the condition $me[fn(-)] \subseteq \mathcal{C}$ that occurs several times. Note that item (*3b*) corresponds to "bound" input, mainly intended to be used to match a corresponding bound output in the rule *Close* of the semantics; therefore the name y read along link x must be fresh in P, i.e. $y \notin fn(P)$.

Theorem 2. *If* $me[fn(P)] \subseteq \mathcal{C}$, $(\rho,\sigma) \models^l_{me} P$ *and* $\vdash^l P \xrightarrow[\lambda,L]{\mu} Q$ *we have:*

(1) *If* $\mu = \tau$ *then*
$\lambda = \epsilon$, $(\rho,\sigma) \models^l_{me} Q$, *and* $me[fn(Q)] \subseteq \mathcal{C}$

(2a) *If* $\mu = \overline{x}y$ *then*
$\lambda = \epsilon$, $(\rho,\sigma) \models^l_{me} Q$, $me[fn(Q)] \subseteq \mathcal{C}$ *and* $\forall l' \in Ll.\ me(y) \in \sigma_{out}(l')(me(x))$

(2b) *If* $\mu = \overline{x}(y)$ *then*
$\lambda = \chi$ *for some* $\chi \in \mathcal{C}$, $(\rho,\sigma) \models^l_{me[y\mapsto\chi]} Q$, $(me[y \mapsto \chi])[fn(Q)] \subseteq \mathcal{C}$, *and*
$\forall l' \in Ll.\ \chi \in \sigma_{out}(l')(me(x))$

(3a) *If* $\mu = xy$, $\lambda = \epsilon$, $me(y) \in \mathcal{C}$ *and*
$me(y) \in \bigcup_{l'\in\mathcal{L}}(\sigma_{in}(l')(me(x)) \cup \sigma_{out}(l')(me(x)))$ *then*
$(\rho,\sigma) \models^l_{me} Q$, $me[fn(Q)] \subseteq \mathcal{C}$, *and* $\forall l' \in Ll.\ me(y) \in \sigma_{in}(l')(me(x))$

(3b) *If* $\mu = xy$, $\lambda = \chi$, $\chi \in \bigcup_{l'\in\mathcal{L}}(\sigma_{in}(l')(me(x)) \cup \sigma_{out}(l')(me(x)))$ *and* $y \notin fn(P)$ *then*
$(\rho,\sigma) \models^l_{me[y\mapsto\chi]} Q$, $(me[y \mapsto \chi])[fn(Q)] \subseteq \mathcal{C}$, *and*
$\forall l' \in Ll.\ \chi \in \sigma_{in}(l')(me(x))$

Proof. A lengthy proof by induction on the construction of

$$\vdash^l P \xrightarrow[\lambda,L]{\mu} Q$$

and with subcases depending on whether case *(1)*, *(2a)*, *(2b)*, *(3a)* or *(3b)* applies. The proof makes use of Lemmata 1, 2, 3 and 4.

4 Multi-level Security

System security is typically based on putting objects and subjects into security classes and preventing information from flowing from higher levels to lower ones. Besides the no-leaks property studied in [5], here we offer another evidence that Control Flow Analysis helps in statically detecting useful information on security.

The literature reports a security property called no read-up/no write-down [11, 12]. The security requirement is that a process classified at a high level cannot write any value to a process of low level, while the converse is allowed. These requirements are part of a security model, based on a multi-level access control, see [4, 10]. The no read-up/no write-down property is commonly studied for a set of processes put in parallel (see, e.g. [29]). We follow this view and consider in the following only processes of the form $\langle P_0\rangle^{l_0} | \langle P_1\rangle^{l_1} | \ldots | \langle P_n\rangle^{l_n}$, where each process P_i has no labelling construct inside.

A dynamic notion. Now we are ready to introduce the dynamic version of the no read-up/no write-down property.

We assume that the environment is always willing to listen to P, i.e. its sub-processes at any level can perform free outputs to the environment. On the

contrary, some parts of P are reachable only if the environment supplies some information to a sub-process R with a particular clearance. To formalize the intentions of the environment, we use a function

$$\varsigma : \mathcal{L} \to (\mathcal{C} \to \wp(\mathcal{C}))$$

that associates a label l and a channel χ with the set of channels that the environment considers secure to communicate to R.

Definition 4. *Given P, me_P, ς, a* granted step *is $(P, me_P) \xrightarrow[\lambda, L]{\mu} (Q, me_Q)$, and is defined whenever*

1. $\vdash^{\#} P \xrightarrow[\lambda, L]{\mu} Q$, *and*
2. *if $\mu = xy$, then* $\begin{cases} (a)\ me_P(y) \in \varsigma(\#)(me_P(x)) & \text{if } \lambda = \epsilon \\ (b)\ \lambda \in \varsigma(\#)(me_P(x)) \text{ and } y \notin fn(P) & \text{if } \lambda = \chi \end{cases}$

where $me_Q = \begin{cases} me_P & \text{if } \lambda = \epsilon \\ me_P[obj(\mu) \mapsto \chi] & \text{if } \lambda = \chi \end{cases}$

A granted computation $(P, me_P) \Longrightarrow^{*} (Q, me_Q)$ *is made of granted steps.*

The definition of our version of the no read-up/no write-down property follows. Essentially, it requires that in all the communications performed by a process, the sender R_o has a clearance level lower than the clearance level of the receiver R_i.

Definition 5. *A process P is* no read-up/no write-down (nru/nwd *for short) with respect to ς, me_P if and only if the following holds:*

whenever $(P, me_P) \Longrightarrow^{*} (P', me_{P'}) \xrightarrow[\epsilon, L]{\tau} (Q, me_Q)$ where the last granted step is a communication (between R_o and R_i) that has been deduced with either

(a) the rule Com, using the premises $\vdash^{l} R_o \xrightarrow[\epsilon, L_o]{\overline{x}y} R'_o$ and $\vdash^{l} R_i \xrightarrow[\epsilon, L_i]{xy} R'_i$, or

(b) the rule Close, using the premises $\vdash^{l} R_o \xrightarrow[\chi, L_o]{\overline{x}(y)} R'_o$ and $\vdash^{l} R_i \xrightarrow[\chi, L_i]{xy} R'_i$,

then no element of L_o is strictly greater than any element of L_i.

A static notion. We define now a static property that guarantees that a process is nru/nwd. Besides finding a solution (ρ, σ) for a process P, we require that the channels read along χ should include those that the environment is willing to supply, expressed by ς. The last condition below requires that the same channel cannot be used for sending an object from a process with high level l'' to a process with low level l'.

Definition 6. *Let P, me be such that $me[fn(P)] \subseteq \mathcal{C}$. Then P is* discreet *(w.r.t. ς, me) if and only if there exists (ρ, σ) such that*

1. $(\rho, \sigma) \models^{\#}_{me} P$
2. $\forall l \in \mathcal{L}, \chi \in \mathcal{C} : \sigma_{in}(l)(\chi) \supseteq \varsigma(l)(\chi)$
3. $\forall l', l'' \in \mathcal{L},\ l' < l''$ *and* $\forall \chi \in \mathcal{C} : \sigma_{out}(l'')(\chi) \cap \sigma_{in}(l')(\chi) = \emptyset$.

The property of discreetness can be checked in low polynomial time by building on the techniques mentioned in Section 3.2. Below, we show that the property of being discreet is preserved under granted steps.

Lemma 5 (Subject reduction for discreetness).

If P is discreet with respect to ς, me_P, and $(P, me_P) \stackrel{\mu}{\Longrightarrow}_{\lambda, L} (Q, me_Q)$, then Q is discreet with respect to ς, me_Q.

Proof. Theorem 2 suffices for proving that $me_Q[fn(Q)] \subseteq \mathcal{C}$ and $(\rho, \sigma) \models^{\#}_{me} Q$. The proof of the second and third items is immediate, because the solution does not change. The only delicate point for the application of Theorem 2 is when the granted step is an input. Consider first, the case in which $\lambda = \epsilon$. It suffices to make sure that $me(y) \in \mathcal{C}$ and that $me(y) \in \sigma_{in}(\#)(me(x))$. Condition 2 of Def. 6 guarantees that $\sigma_{in}(\#)(me(x)) \supseteq \varsigma(\#)(me(x))$. In turn $me(y) \in \varsigma(\#)(me(x)) \subseteq \mathcal{C}$ because the step is granted. The case when $\lambda = \chi$ is just the same, while in the other cases the proof is trivial.

The following lemma further illustrates the links between the transitions of processes and the results of our static analysis. It will be used in proving that discreetness is sufficient to guarantee that P enjoys the nru/nwd property.

Lemma 6. *If $\vdash^{l} P \stackrel{\mu}{\rightarrow}_{\lambda, L} Q$ has been deduced with premise $\vdash^{l'} R \stackrel{\mu}{\rightarrow}_{\lambda', L'} R'$, and $(\rho, \sigma) \models^{l}_{me} P$, then there exists me' such that $(\rho, \sigma) \models^{l'}_{me'} R$.*

Proof. The proof is by induction on the derivation of the transition.

Theorem 3.
If P is discreet (w.r.t. ς, me), then P is nru/nwd (w.r.t. ς, me).

Proof. By Lemma 5 it is enough to check that, if $(P, me_P) \stackrel{\tau}{\Longrightarrow}_{\epsilon, L} (Q, me_Q)$, then Q is nru/nwd, with τ being a communication between R_o and R_i, defined as in Def. 5. Assume, per absurdum, that an element $l_o \in L_o$ is strictly greater than an element element $l_i \in L_i$. By Lemma 6, $(\rho, \sigma) \models^{l}_{me} R_o$ and $(\rho, \sigma) \models^{l}_{me} R_i$. Consider first the case (a) of Def. 5. The analysis and Theorem 2 tell us that $\forall l' \in L_o.\ me(y) \in \sigma_{out}(l')(me(x))$ and $\forall l' \in L_i.\ me(y) \in \sigma_{in}(l')(me(x))$. But this contradicts item 3 of Def. 6, because, in particular, $me(y) \in \sigma_{out}(l_o)(me(x))$ as well as $me(y) \in \sigma_{in}(l_i)(me(x))$.
As for case (b) of Def. 5, just replace χ for $me(y)$ and proceed as in case (a).

Example 2. Consider again the process S validated in Example 1:

$$!(\langle \overline{a}b.\overline{a}b.\overline{b}c\rangle^{l_R} \mid \langle a(x^{\beta_x}).\overline{x}x\rangle^{l_Q} \mid \langle a(y^{\beta_y}).y(z^{\beta_z}).([y=z]\overline{y}a \ + \ y(w^{\beta_w}))\rangle^{l_P}),$$

and suppose that $l_R < l_Q < l_P$. Then it is easy to prove that the process is discreet. In particular the following five conditions hold.

- $\sigma_{out}(l_Q)(\chi_{fv}) \cap \sigma_{in}(l_R)(\chi_{fv}) = \sigma_{out}(l_Q)(\chi_{fv}) \cap \emptyset = \emptyset$;
- $\sigma_{out}(l_P)(\chi_{fv}) \cap \sigma_{in}(l_R)(\chi_{fv}) = \sigma_{out}(l_P)(\chi_{fv}) \cap \emptyset = \emptyset$;
- $\sigma_{out}(l_P)(\chi_a) \cap \sigma_{in}(l_Q)(\chi_a) = \emptyset \cap \{\chi_b\} = \emptyset$.
- $\sigma_{out}(l_P)(\chi_b) \cap \sigma_{in}(l_Q)(\chi_b) = \{\chi_a\} \cap \emptyset = \emptyset$.
- $\sigma_{out}(l_P)(\chi_c) \cap \sigma_{in}(l_Q)(\chi_c) = \emptyset \cap \emptyset = \emptyset$.

Note that the clearance levels of processes introduced here are orthogonal to the security levels of channels as defined in [5]. There channels are partitioned into secret and public and a static check is made that secret channels never pass along a public one. Therefore channels have always the same level. On the contrary, here it is possible that a channel a can be sent along b by one process but *not* by another. Discreetness cannot be checked with the analysis of [5], because in that analysis a can be either sent on b always or never. The combination of the two analyses may permit a static check of even more demanding properties.

5 Conclusions

There is a vast literature on the topics of our paper. Here we only mention very briefly some papers related to security issues.

The first studies in system security reach back to the 1970's and were mainly carried out in the area of operating systems; see the detailed survey by Landwehr [17] and Denning's book [10] reporting on the static detection of secure flow violation while analysing the code.

Recently, security classes have been formalized as types and the control of flow is based on type checking. Heintze and Riecke [15] study a non-interference property on the SLam Calculus (Secure λ-calculus). Volpano, Smith and Irvine develop a type system to ensure secure information flow in a sequential imperative language in [30], later extended by the first two authors in a concurrent, shared memory based setting [29]. Abadi studies in [1] the secrecy of channels and of encrypted messages, using the spi-calculus, an extension of the π-calculus devised for writing secure protocols. Venet [27, 28] uses Abstract Interpretation techniques to analyse processes in a fragment of the π-calculus, with particular attention to the usage of channels.

Other papers interesting for this area are [24, 26, 13, 8, 3, 9, 25, 16]. Particularly relevant are Hennessy and Riely's papers [25, 16] who give a type system for Dπ, a variant of the π-calculus with explicit sites that harbour mobile processes. Cardelli and Gordon [6] propose a type system for the Mobile Ambient calculus

ensuring that a well-typed mobile computation cannot cause certain kinds of run-time faults, even when capabilities can be exchanged between processes.

The idea of static analysis for security has been followed also in the Java world, for example in the Java Bytecode Verifier [18], and in the techniques of proof-carrying code [23]. Also Abadi faces in [2] the problem of implementing secure systems and proposes to use full abstraction to check that the compile code enjoys the same security properties as the source program.

A different approach consists in dynamically checking properties. This point of view has been adopted by a certain number of information flow models [14, 19, 20, 11, 12] (to cite only a few), mainly concerned with checking (variants of) the security property we studied here. All these papers, consider the external observable behaviour only as the object of the analysis.

Here, we presented a Control Flow Analysis for the π-calculus that statically predicts how names will be bound to actual channels at run-time. The only extensions made to the syntax of processes are that a channel χ is explicitly assigned to a restricted name, and that an input action has the form $x(y^{\beta})$, making explicit the rôle of the placeholder y; this change was motivated by the inclusion of α-conversion in the semantics. Our intention was to apply our analysis for detecting violations of a security property that needs security levels, so processes may carry labels expressing their clearance. The result of our analysis for a process P is a solution (ρ, σ). The abstract environment ρ gives information about which channels a binder β may be bound to, by means of communication. The abstract communication environment σ gives information about the channels sent and received by a process with clearance l. All the solution components approximate the actual solution, because they may give a super-set of the corresponding actual values.

We defined judgements of the form $(\rho, \sigma) \models^{l}_{me} P$ and a set of clauses that operate on them so as to validate the correctness of the solution. The additional marker environment me binds the free names of P to actual channels. The label l records the security level of P. We proved that a best solution always exists. In the full paper we shall give a constructive procedure for generating solutions that essentially generates a set of constraints corresponding to the checks necessary to validate solutions. These constraints can be solved in low polynomial time.

We used our analysis to establish the no read-up/no write-down security property of Bell and LaPadula. This property requires that a process with high clearance level never sends channels to processes with a low clearance. We defined a static check on solutions and proved that it implies the no read-up/no write-down property. Also, the check that a process P is discreet with respect to given ς,me has a polynomial time complexity. A web-based system that validates the solutions and checks discreetness can be found at the URL: `http://www.daimi.au.dk/~rrh/discreet.html`.

We have not considered here the more general notion of the no read-up/no write-down property, that assigns levels of confidentiality also to the exchanged data (i.e. the objects of input and output actions). Processes with low level clearance are then not allowed to access (i.e. they can neither send nor receive)

highly classified data. The reason is that the dynamic version of this property is surprisingly more intricate than its static version. The latter, that entails the former, only requires an additional check on the second component of a solution (i.e., $\forall \chi \in \sigma_{in}(l)(\chi')$ the confidentiality level of χ, possibly read along channel χ', should be smaller than the security level of the process under check, namely l; similarly for σ_{out}).

Other properties that deserve further investigation are connected with the so-called "indirect flow" of information, i.e. on the possibility of a low level process to detect the value of some confidential datum by "observing" the behaviour of higher level processes.

Acknowledgments. We wish to thank Martín Abadi, Roberto Gorrieri and Dennis Volpano for interesting discussions, and René Rydhof Hansen for writing the web-based system that checks discreetness.

The first two authors have been partially supported by the CNR Progetto Strategico *Modelli e Metodi per la Matematica e l'Ingegneria* and by the MURST Progetto *Tecniche Formali per la Specifica, l'Analisi, la Verifica, la Sintesi e la Trasformazione di Sistemi Software*. The last two authors have been partially supported by the DART project (funded by The Danish Research Councils).

References

1. M. Abadi. Secrecy by typing in security protocols. In *Proceedings of Theoretical Aspects of Computer Software, Third International Symposioum, LNCS 1281*, pages 611–638. Springer-Verlag, 1997.
2. M. Abadi. Protection in programming-language translations. In *Proceedings of ICALP'98, LNCS 1443*, pages 868–883. Springer-Verlag, 1998.
3. R.M. Amadio. An asynchronous model of locality, failure, and process mobility. In *Proceedings of COORDINATION'97, LNCS 1282*, pages 374–391. Springer-Verlag, 1997.
4. D.E. Bell and L.J. LaPadula. Secure computer systems: Unified exposition and Multics interpretation. Technical Report ESD-TR-75-306, Mitre C., 1976.
5. C. Bodei, P. Degano, F. Nielson, and H. Riis Nielson. Control flow analysis for the π-calculus. In *Proceedings of CONCUR'98, LNCS 1466*, pages 84–98. Springer-Verlag, 1998.
6. L. Cardelli and A.D. Gordon. Types for mobile ambients. In *Proceedings of POPL'99*, page (to appear). ACM Press, 1999.
7. P. Cousot and R. Cousot. Systematic Design of Program Analysis Frameworks. In *Proceedings of POPL '79*, pages 269–282. ACM Press, 1979.
8. M. Dam. Proving trust in systems of second-order processes: Preliminary results. Technical Report LOMAPS-SICS 19, SICS, Sweden, 1997.
9. R. De Nicola, G. Ferrari, and R. Pugliese. Coordinating mobile agents via blackboards and access rights. In *Proceedings of COORDINATION'97, LNCS 1282*, pages 220–237. Springer-Verlag, 1997.
10. D.E Denning. *Cryptography and Data Security*. Addison-Wesley, Reading Mass., 1982.
11. R. Focardi. Comparing two information flow security properties. In *Proceedings of* 9^{th} *IEEE Computer Science Security Foundation W/S*, pages 116–122, 1996.

12. R. Focardi and R. Gorrieri. The compositional security checker: A tool for the verification of information flow security properties. *IEEE Transactions on Software Engineering*. To appear.
13. C. Fournet, C. Laneve, L. Maranget, and D. Remy. Implicit typing à la ML for the join calculus. In *Proceedings of CONCUR'97, LNCS 1243*, pages 196–212. Springer-Verlag, 1997.
14. J.A. Goguen and J. Meseguer. Security policy and security models. In *Proceedings of the 1982 IEEE Symposioum on Research on Security and Privacy*, pages 11–20. IEEE Press, 1982.
15. N. Heintze and J.G Riecke. The SLam calculus: Programming with secrecy and integrity. In *Proceedings of POPL'98*, pages 365–377. ACM Press, 1998.
16. M. Hennessy and J. Riely. Resource access control in systems of mobile agents. Technical Report 2/98, University of Sussex, 1998.
17. C. E. Landwehr. Formal models for computer security. *ACM Computing Surveys*, 13(3):247–278, 1981.
18. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.
19. D. McCullough. Specifications for multi-level security and hook-up property. In *Proceedings of the 1987 IEEE Symposioum on Research on Security and Privacy*. IEEE Press, 1987.
20. D. McCullough. Noninterference and the composability of security properties. In *Proceedings of the 1988 IEEE Symposium on Research on Security and Privacy*, pages 177–186. IEEE Press, 1988.
21. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes (I and II). *Information and Computation*, 100(1):1–77, 1992.
22. R. Milner, J. Parrow, and D. Walker. Modal logics for mobile processes. *TCS*, 114:149–171, 1993.
23. G. Necula. Proof-carrying code. In *Proceedings of POPL'97*, pages 106–119. ACM Press, 1997.
24. B.C. Pierce and D. Sangiorgi. Typing and sub-typing for mobile processes. *Mathematical Structures in Computer Science*, 6(5):409–454, 1996.
25. J. Riely and M. Hennessy. A typed language for distributed mobile processes. In *Proceedings of POPL'98*, pages 378–390. ACM Press, 1998.
26. P. Sewell. Global/local subtyping and capability inference for a distributed π-calculus. In *Proceedings of ICALP'98, LNCS 1443*, pages 695–706. Springer-Verlag, 1998.
27. A. Venet. Abstract interpretation of the π-calculus. In *Analysis and Verification of Multiple-Agent Languages, LNCS 1192*, pages 51–75. Springer-Verlag, 1997.
28. A. Venet. Automatic determination of communication topologies in mobile systems. In *Static Analysis Symposium, LNCS 1503*, pages 152–167. Springer-Verlag, 1998.
29. D. Volpano and G. Smith. Secure information flow in a multi-threaded imperative language. In *Proceedings of POPL'98*, pages 355–364. ACM Press, 1998.
30. D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4:4–21, 1996.

A WP-calculus for OO

F.S de Boer[1]

Utrecht University, The Netherlands
Email: frankb@cs.uu.nl

Abstract. A sound and complete Hoare-style proof system is presented for a sequential object-oriented language, called SPOOL. The proof system is based on a weakest precondition calculus for aliasing and object-creation.

1 Introduction

This paper introduces a Hoare-style proof system for an object-oriented language, called SPOOL. SPOOL is a sequential version of the parallel object-oriented language POOL [2].

The main aspect of SPOOL that is dealt with is the problem of how to reason about *pointer structures*. In SPOOL, objects can be created at arbitrary points in a program, references to them can be stored in variables and passed around as parameters in messages. This implies that complicated and dynamically evolving structures of references between objects can occur. We want to reason about these structures on an abstraction level that is *at least as high as that of the programming language*. In more detail, this means the following: The only operations on "pointers" (references to objects) are testing for equality and dereferencing (looking at the value of an instance variable of the referenced object). Furthermore, in a given state of the system, it is only possible to mention the objects that exist in that state. Objects that do not (yet) exist never play a role.

Strictly speaking, direct dereferencing is not even allowed in the programming language, because each object only has access to its own instance variables. However, for the time being we allow it in the assertion language. Otherwise, even more advanced techniques would be necessary to reason about the correctness of a program.

The above restrictions have quite severe consequences for the proof system. The limited set of operations on pointers implies that first-order logic is too weak to express some interesting properties of pointer structures (for example, the property, as considered in [9], that it is possible to go from one object to the other by following a finite number of x-links). Therefore we have to extend our assertion language to make it more expressive. In this paper we do so by allowing the assertion language to reason about finite sequences of objects.

The proof system itself is based on a *weakest precondition* calculus for aliasing and object-creation. This means that in the proof system aliasing and object-creation are modelled by substitutions which, when applied to a given postcondition, yield the corresponding weakest precondition.

W. Thomas (Ed.): FOSSACS'99, LNCS 1578, pp. 135-149, 1999.

Plan of the paper In the following section we introduce the programming language SPOOL. In section 3 the assertion language for describing object structures is introduced. The proof system is discussed in section 4. In the final section related work is discussed and some general conclusions are drawn.

2 The language SPOOL

The most important concept of SPOOL is the concept of an *object*. This is an entity containing data and procedures (*methods*) acting on these data. The data are stored in *variables*, which come in two kinds: *instance variables*, whose lifetime is the same as that of the object they belong to, and *temporary variables*, which are local to a method and last as long as the method is active. Variables can contain references to other objects in the system (or even the object under consideration itself). The object a variable refers to (its *value*) can be changed by an *assignment*. The value of a variable can also be nil, which means that it refers to no object at all.

The variables of an object cannot be accessed directly by other objects. The only way for objects to interact is by sending *messages* to each other. If an object sends a message, it specifies the receiver, a method name, and possibly some parameter objects. Then control is transferred from the sender object to the receiver. This receiver then executes the specified method, using the parameters in the message. Note that this method can, of course, access the instance variables of the receiver. The method returns a result, an object, which is sent back to the sender. Then control is transferred back to the sender which resumes its activities, possibly using this result object.

The sender of a message is *blocked* until the result comes back, that is, it cannot answer any message while it still has an outstanding message of its own. Therefore, when an object sends a message to itself (directly or indirectly) this will lead to abnormal termination of the program.

Objects are grouped into *classes*. Objects in one class (the *instances* of the class) share the same methods, so in a certain sense they share the same behaviour. New instances of a given class can be created at any time. There are two standard classes, Int and Bool, of integers and booleans, respectively. They differ from the other classes in that their instances already exist at the beginning of the execution of the program and no new ones can be created. Moreover, some standard operations on these classes are defined.

A program essentially consists of a number of class definitions, together with a statement to be executed by an instance of a specific class. Usually, but not necessarily, this instance is the only non-standard object that exists at the beginning of the program: the others still have to be created.

In order to describe the language SPOOL, which is strongly typed, we use *typed* versions of all variables, expressions, etc. These types however are implicitly assumed in the language description below.

We assume the following sets to be given: A set C of *class names*, with typical element c (this means that metavariables like $c, c', c_1, \ldots$ range over elements of

the set C). We assume that $\mathsf{Int}, \mathsf{Bool} \notin C$ and define the set $C^+ = C \cup \{\mathsf{Int}, \mathsf{Bool}\}$, with typical element d. For each $c \in C$, $d \in C^+$ we assume a set $IVar_d^c$, with typical element x, of *instance variables* in class c which are of type d. For each $d \in C$ we assume a set $TVar_d$ of *temporary variables* of type d, with typical element u. Finally, for each $c \in C$ and $d_0, \ldots, d_n \in C^+$ $(n \geq 0)$ we assume a set $MName^c_{d_0,\ldots,d_n}$ of *method names* of class c with result type d_0 and parameter types $d_1, \ldots, d_n$. The set $MName^c_{d_0,\ldots,d_n}$ will have m as a typical element.

Now we can specify the syntax of our (strongly typed) language (we omit the typing information).

Definition 1. For any $c \in C$ and $d \in C^+$ the set Exp_d^c of *expressions* of type d in class c, with typical element e, is defined as usual. We give the following base cases.

$$e ::= x \mid u \mid \mathsf{nil} \mid \mathsf{self} \cdots$$

The set $SExp_d^c$ of expressions with possible *side effect* of type d in class c, with typical element s, is defined as follows:

$$s ::= e \mid \mathsf{new} \mid e_0!m(e_1, \ldots, e_n)$$

The first kind of side effect expression is a normal expression, which has no actual side effect, of course. The second kind is the creation of a new object. This new object will also be the value of the side effect expression. The third kind of side effect expression specifies that a message is to be sent to the object that results from e_0, with method name m and with arguments (the objects resulting from) $e_1, \ldots, e_n$.

The set $Stat^c$ of *statements* in class c, with typical element S, are constructed from assignments by means of the standard sequential operations of sequential composition, (deterministic) choice and iteration.

Definition 2. The set $MethDef^c$ of *method definitions* in class c, with typical element μ, is defined by:

$$\mu ::= (u_1, \ldots, u_n : S \uparrow e)$$

Here we require that the u_i are all different and that none of them occurs at the left hand side of an assignment in $S \in Stat^c$ (and that $n \geq 0$).

When an object is sent a message, the method named in the message is invoked as follows: The variables $u_1, \ldots, u_n$ (the parameters of the method) are given the values specified in the message, all other temporary variables (i.e. the *local* variables of the method, are initialized to nil, and then the statement S is executed. After that the expression e is evaluated and its value, the result of the method, is sent back to the sender of the message, where it will be the value of the send-expression that sent the message.

Definition 3. The set $ClassDef^c$ of definitions of class c, with typical element D, is defined by:

$$D ::= c : \langle m_1 = \mu_1, \ldots, m_n = \mu_n \rangle$$

where we require that all the method names are different (and $n \geq 0$).

Definition 4. Finally, the set $Prog^c$ of *programs* in class c, with typical element ρ, is defined by:

$$\rho ::= \langle U | c : S \rangle$$

where U denotes a finite set of class definitions and $S \in Stat^c$. The interpretation of such a program is that the statement S is executed by some object of class c (the root object) in the context of the declarations contained in the unit U. In many cases (including the following example) we shall assume that at the beginning of the execution this root object is the only existing non-standard object.

Example 1. The following program generates prime numbers using the sieve method of Eratosthenes.

```
⟨Sieve : ⟨input ⇐ (q) : if next = nil
                        then next := new;
                             p := q
                        else if q mod p ≠ 0
                             then next ! input(q)
                             fi
                        fi
                        ↑ self ⟩,
Driver : ⟨ ⟩
|
Driver : i := 2;
         first := new;
         while i < bound
         do first ! input(i);
            i := i + 1
         od
⟩
```

Figure 1 represents the system in a certain stage of the execution of the program.

3 The assertion language

In this section a formalism is introduced for expressing certain properties of a complete system, or configuration, of objects. Such a system consists for each class of a set of *existing* objects in that class (i.e. the objects in that class which have been created sofar) together with their internal states (i.e. an assignment of values to their own instance variables), and the currently active object together with an assignment of values to its temporary variables.

One element of this assertion language will be the introduction of *logical variables*. These variables may not occur in the program, but only in the assertion

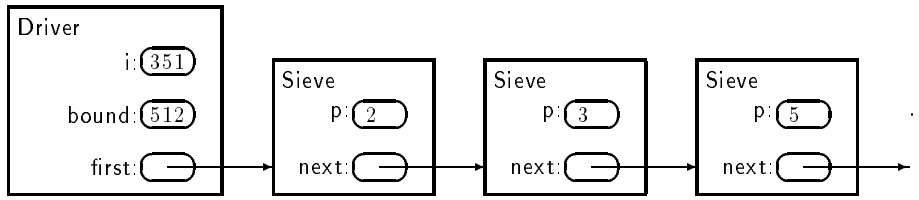


Fig. 1. Objects in the sieve program in a certain stage of the execution

language. Therefore we are always sure that the value of a logical variable can never be changed by a statement. Apart from a certain degree of cleanliness, this has the additional advantage that we can use logical variables to express the constancy of certain expressions (for example in the proof rule for message passing). Logical variables also serve as bound variables for quantifiers.

The set of expressions in the assertion language is larger than the set of programming language expressions not only because it contains logical variables, but also because by means of a dereferencing operator it is allowed to refer to instance variables of other objects. Furthermore we include conditional expressions in the assertion language. These conditional expressions will be used for the analysis of the phenomenon of aliasing which arises because of the presence of a dereferencing operator.

In two respects our assertion language differs from the usual first-order predicate logic: Firstly, the range of quantifiers is limited to the *existing* objects in the current state of the system. For the classes different from Int and Bool this restriction means that we cannot talk about objects that have not yet been created, even if they could be created in the future. This is done in order to satisfy the requirements on the proof system stated in the introduction. Because of this the range of the quantifiers can be different for different states. More in particular, a statement can change the truth of an assertion even if none of the program variables accessed by the statement occurs in the assertion, simply by creating an object and thereby changing the range of a quantifier. (The idea of restricting the range of quantifiers was inspired by [11].)

Secondly, in order to strengthen the expressiveness of the logic, it is augmented with quantification over finite sequences of objects. It is quite clear that this is necessary, because simple first-order logic is not able to express certain interesting properties.

Definition 5. For each $d \in C^+$ we introduce the symbol d^* for the type of all finite sequences with elements from d, we let C^* stand for the set $\{d^* | d \in C^+\}$, and we use $C^\dagger$, with typical element a, for the union $C^+ \cup C^*$. We assume that for every a in $C^\dagger$ we have a set $LVar_a$ of logical variables of type a, with typical element z.

Definition 6. We give the following typical elements of the set $LExp_a^c$ of logical expressions of type a in class c (we omit the typing information):

$$l ::= e \mid z \mid l.x \mid \mathsf{if}\ l_0\ \mathsf{then}\ l_1\ \mathsf{else}\ l_2\ \mathsf{fi}$$

An expression e is evaluated in the internal state of the currently active object which is denoted by self. The difference with the set Exp_d^c of expressions in the programming language is that in logical expressions we can use logical variables, refer to the instance variables of other objects (the expression $l.x$ refers to the local value of the instance variable x of the object denoted by l), and write conditional expressions. Furthermore, we extended the domain of discourse by means of logical variables ranging over sequences. In order to reason about sequences we assume the presence of notations to express, for example, the length of a sequence (denoted by $|l|$) and the selection of an element of a sequence (denoted by $l(n)$, where n is an integer expression).

Definition 7. The set Ass^c of assertions in class c, with typical elements P and Q, is defined by:

$$P ::= l \mid P \wedge Q \mid \neg P \mid \exists z P$$

Here l denotes a boolean expression (i.e. $l \in Exp^c_{\mathsf{Bool}}$).

As already explained above, a formula $\exists z P$, with z of some type $c \in C$ states that P holds for some *existing* object in class c. A formula $\exists z P$, with z of a sequence type c^*, states the existence of a sequence of existing objects in class c.

Example 2. The formula $\exists z$ true, where z is of some type $c \in C$, thus states the existence of an object in class c. As such this formula is false in case no such objects exist. As another example, the following formula states the existence of a sequence of objects in class Sieve (of the example program in the previous section) such that the value of p of the nth element in this sequence is the nth prime number and next refers to the next element, i.e. the $n+1$th element, in the sequence.

$$\exists z \forall n \left(\begin{array}{c} (0 < n \wedge n \leq |z| \rightarrow z(n).\mathsf{p} = prime(n)) \\ \wedge \\ (0 < n \wedge n < |z| \rightarrow z(n).\mathsf{next} = z(n+1)) \end{array} \right)$$

Here n denotes a logical variable ranging over integers and z ranges over sequences of objects in class Sieve. The predicate $prime(n)$ holds if n is a prime.

Definition 8. A *correctness formula* in class c is a Hoare triple of the form $\{P\}\rho\{Q\}$, where $P, Q \in Ass^c$ and $\rho \in Prog^c$.

A Hoare-triple $\{P\}\rho\{Q\}$ expresses a *partial correctness* property of the program ρ: It holds if every *successfully terminating* execution of the program ρ in a system of objects which satisfies the precondition P results in a final configuration which satisfies the postcondition Q.

4 The proof system

In this section we present a Hoare-style proof system which provides a view of programs in SPOOL as *predicate-transformers*.

Simple assignments We shall call a statement a *simple assignment* if it is of the form $x := e$ or $u := e$ (that is, it uses the first form of a side effect expression: the one without a side effect). For the axiomatization of simple assignments to temporary variables the standard assignment axiom suffices because objects are only allowed to refer to the instance variables of other objects and therefore *aliasing*, i.e. the situation that different expressions refer to the same variable, does not arise in case of temporary variables.

In the case that the target variable of an assignment statement is an instance variable, we use the following axiom:

$$\Big\{P[e/x]\Big\}\ \langle U|c : x := e\rangle\ \Big\{P\Big\}$$

The substitution operation $[e/x]$ has to account for possible aliases of the variables x, namely, expressions of the form $l.x$: It is possible that, after substitution, l refers to the currently active object (i.e. the object denoted by self), so that $l.x$ is the same variable as x and should be substituted by e. It is also possible that, after substitution, l does not refer to the currently executing object, and in this case no substitution should take place. Since we cannot decide between these possibilities by the form of the expression only, a conditional expression is constructed which decides "dynamically".

Definition9. We have the following main cases of the substitution operation $[e/x]$:

$$\begin{aligned} l\,.\,x\ [e/x] &= \mathsf{if}\ (l[e/x]) = \mathsf{self}\ \mathsf{then}\ e\ \mathsf{else}\ (l[e/x])\,.\,x\ \mathsf{fi} \\ l\,.\,x'\,[e/x] &= (l[e/x])\,.\,x' \qquad \text{if } x' \neq x \end{aligned}$$

The definition is extended to assertions other than logical expressions in the standard way.

Object creation Next we consider the creation of objects. We will introduce two different axiomatizations of object-creation which are based on the logical formulation of the *weakest precondition* and the *strongest postcondition*, respectively. First we consider a weakest precondition axiomatization.

For an assignment of the form $u :=$ new we have a axiom similar to the previous two:

$$\Big\{P[\mathsf{new}/u]\Big\}\ \langle U|c : u := \mathsf{new}\rangle\ \Big\{P\Big\}$$

We have to define the substitution $[\mathsf{new}/u]$. As with the notions of substitution used in the axioms for simple assignments, we want the expression after substitution to have the same meaning in a state before the assignment as the unsubstituted expression has in the state after the assignment. However, in the case of a new-assignment, there are expressions for which this is not possible,

because they refer to the new object (in the new state) and there is no expression that could refer to that object in the old state, because it does not exist yet. Therefore the result of the substitution must be left undefined in some cases.

However we *are* able to carry out the substitution in case of assertions, assuming, without loss of expressiveness, that in the assertion language the operations on sequences are limited to $|l|$, i.e. the length of the sequence l, and $l(n)$, i.e. the operation which yields the nth element of l. The idea behind this is that in an assertion the variable u referring to the new object can essentially occur only in a context where either one of its instance variables is referenced, or it is compared for equality with another expression. In both of these cases we can predict the outcome without having to refer to the new object.

Definition 10. Here are the main cases of the formal definition of the substitution $[\mathsf{new}/u]$ for logical expressions. As already explained above the result of the substitution $[\mathsf{new}/u]$ is undefined for the expression u. Since the (instance) variables of a newly created object are initialized to nil we have

$$u.x[\mathsf{new}/u] \;=\; \mathsf{nil}$$

If neither l_1 nor l_2 is u or a conditional expression they cannot refer to the newly created object and we have

$$\Big(l_1 = l_2\Big)[\mathsf{new}/u] = \Big(l_1[\mathsf{new}/u]\Big) = \Big(l_2[\mathsf{new}/u]\Big)$$

If either l_1 is u and l_2 is neither u nor a conditional expression (or vice versa) we have that after the substitution operation l_1 and l_2 cannot denote the same object (because one of them refers to the newly created object while the other one refers to an already existing object):

$$\Big(l_1 = l_2\Big)[\mathsf{new}/u] = \mathsf{false}$$

On the other hand if both the expressions l_1 and l_2 equal u we obviously have

$$\Big(l_1 = l_2\Big)[\mathsf{new}/u] = \mathsf{true}$$

We have that $l[\mathsf{new}/u]$ is defined for boolean expressions l.

Definition 11. We extend the substitution operation $[\mathsf{new}/u]$ to assertions other than logical expressions as follows (we assume that the type of u is $d \in C$):

$$\begin{array}{lll}
(P \to Q) & [\mathsf{new}/u] & = (P[\mathsf{new}/u]) \to (Q[\mathsf{new}/u]) \\
(\neg P) & [\mathsf{new}/u] & = \neg(P[\mathsf{new}/u]) \\
(\exists z\ P) & [\mathsf{new}/u] & = (\exists z(P[\mathsf{new}/u])) \vee (P[u/z][\mathsf{new}/u]) \\
(\exists z\ P) & [\mathsf{new}/u] & = \exists z\ \exists z'\ (|z| = |z'| \wedge (P[z', u/z][\mathsf{new}/u])) \\
(\exists z\ P) & [\mathsf{new}/u] & = (\exists z(P[\mathsf{new}/u]))
\end{array}$$

In the third and fourth clause the (bound) variable z is assumed to be of type d and d^*, respectively. The type of the variable z in the last clause is of a type

different from d and d^*. The (bound) variable z' in the fourth clause is assumed to be of type boolean (this variable is also assumed not to occur in P).

The idea of the application of $[\mathsf{new}/u]$ to $(\exists z\ P)$ (in case z is of the same type as u) is that the first disjunct $(\exists z(P[\mathsf{new}/u]))$ represents the case that the object for which P holds is an 'old' object (i.e. which exists already before the creation of the new object) whereas the second disjunct $P[u/z][\mathsf{new}/u]$ represents the case that the new object itself satisfies P.

The idea of the fourth clause is that z and z' together code a sequence of objects in the state after the new-statement. At the places where z' yields true the value of the coded sequence is the newly created object. Where z' yields false the value of the coded sequence is the same as the value of z. This encoding is described by the substitution operation $[z', u/z]$ the main characteristic cases of which are:

$$\begin{array}{ll} z[z', u/z] & \text{is undefined} \\ \Big(z(l)\Big)[z', u/z] = \mathsf{if}\ z'(l')\ \mathsf{then}\ u\ \mathsf{else}\ z(l')\ \mathsf{fi}, & \text{where } l' = l[z', u/z] \end{array}$$

This substitution operation $[z', u/z]$ is defined for boolean expressions.

Example 3. Let z be a logical variable of the same type as u. We have

$$\begin{array}{l} \Big(\exists z(u = z)\Big)[\mathsf{new}/u] \qquad\qquad\qquad\qquad \equiv \\ \Big(\exists z(u = z)[\mathsf{new}/u]\Big) \vee (u = u)[\mathsf{new}/u] \equiv \\ \exists z\ \mathsf{false} \vee \mathsf{true} \end{array}$$

where the last assertion obviously reduces to true, which indeed is the weakest precondition of $\exists z(u = z)$ with respect to $u := \mathsf{new}$.

Note that we cannot apply the substitution operation $[\mathsf{new}/u]$ directly to assertions involving more high-level operations on sequences. For example, an assertion like $l_1 \leq l_2$, which expresses that the sequence l_1 is a prefix of l_2, we have first to reformulate into a logically equivalent one which uses only the sequence operations $|l|$ and $l(n)$. Thus, $l_1 \leq l_2$ should be first translated into

$$\forall n(0 < n \wedge n \leq |l_1| \rightarrow l_1(n) = l_2(n))$$

If our assignment is of the form $x := \mathsf{new}$ we have the following axiom:

$$\Big\{P[\mathsf{new}/x]\Big\}\ \langle U|c : x := \mathsf{new}\rangle\ \Big\{P^c\Big\}$$

The substitution operation $[\mathsf{new}/x]$ is defined by: $P[u/x]\ [\mathsf{new}/u]$, where u is a temporary variable that does not occur in P. (It is easy to see that this definition does not depend on the actual u used.)

Thus we see that we are able to compute the weakest precondition of a new-statement despite the fact that we cannot refer to the newly created object in the state prior to its creation. Alternatively, we have the following strongest postcondition axiomatization of object-creation. Let u be a temporary variable

of type c, and the logical variables z and z' be of type c^* and c, respectively. Moreover, let V be a finite set of instance variables in class c. For an assignment of the form $u := \mathsf{new}$ we have the following axiom.

$$\Big\{P\Big\}\ \langle U|c : u := \mathsf{new}\rangle\ \Big\{\exists z\Big(P' \downarrow z \wedge Q(V,z)\Big)\Big\}$$

where $P' = \exists z'(P[z'/u])$ and $Q(V,z)$ denotes the following assertion

$$u \notin z \wedge \forall z'(z' \in z \vee z' = u) \wedge \bigwedge_{x \in V} u.x = \mathsf{nil}$$

The operation $\downarrow z$ applied to an assertion R restricts all quantifications in R to z. It is described in more detail below. Let us first explain the role of the logical variables z and z' (which are assumed not to occur in P). The logical variable z in the postcondition is intended to the store all the objects in class c which exist in the state prior to the creation of the new object. The logical variable z' is intended to represent the old value of u. Given that z' denotes the old value of u, that P holds for the old value of u then can be expressed in the postcondition simply by $P[z'/u]$. However the quantification $\exists z'(P[z'/u])$ in the postcondition will also include the newly created object. In general we thus have to take into account the changing scope of the quantifiers. For example, consider $P = \forall z''.\mathsf{false}$ (with z'' of type c). Obviously P, which states that there do not exist objects in class c, does not hold anymore after the creation of a new object in class c. Our solution to this problem is to restrict the scope of all quantifications involving objects in class c to the old objects in class c, which are given by z. This restriction operation is denoted by $R \downarrow z$. Its main characteristic defining clauses are the following two:

$$(\exists z'' R) \downarrow z = \exists z''(z'' \in z \wedge R \downarrow z)$$
$$(\exists z'' R) \downarrow z = \exists z''(z'' \subseteq z \wedge R \downarrow z)$$

where in the first clause z'' is of type c while in the second clause z'' is of type c^* (for convenience we assume the presence of the relation 'is an element of the sequence', denoted by $\in$, and the containment-relation $\subseteq$, which holds whenever all the elements of its first argument occur in its second argument). Finally, the assertion $Q(V,z)$ in the postcondition of axiom above expresses that u denotes the newly created object and specifies the initial values of the variables in V (of the newly created object).

For new-statements involving instance variables we have a similar axiom characterizing its strongest postcondition semantics.

It is of interest to observe here that the strongest postcondition axiomatization does not require a restricted repetoire of primitive sequence operations.

Method calls Next we present proof rules for verifying the third kind of assignments: the ones where a message is sent and the result stored in the variable on the left hand side. We present here a rule for non-recursive methods (recursion is handled by a straightforward adaptation of the classical recursion rule, see for example [3]).

For the statement $x := e_0!m(e_1, \ldots, e_n)$, we have the following proof rule (for the statement $u := e_0!m(e_1, \ldots, e_n)$ we have a similar rule):

$$\frac{\left\{P \wedge \bigwedge_{i=1}^{k} v_i = \mathsf{nil} \wedge \mathsf{self} \notin \delta_{c'}\right\}\langle U|c' : S\rangle\left\{Q[e/r]\right\}, \quad Q'[\bar{f}/\bar{z}] \rightarrow R[r/x]}{\left\{P'[\bar{f}/\bar{z}]\right\}\langle U|c : x := e_0!m(e_1, \ldots, e_n)\rangle\left\{R\right\}}$$

where $S \in Stat^{c'}$ and $e \in Exp_{d_0}^{c'}$ are the statement and expression occurring in the definition of the method m in the unit U, $u_1, \ldots, u_n$ are its formal parameters, $v_1, \ldots, v_k$ is a row of temporary variables that are *not* formal parameters ($k \geq 0$), r is a logical variable of type of the result of the method m (it is assumed that r does not occur in R), $\bar{f}$ is an arbitrary row of expressions (*not* logical expressions) in class c, and $\bar{z}$ is a row of logical variables, mutually different and different from r, such that the type of each z_i is the same as the type of the corresponding f_i. Furthermore, we assume given for each $d \in C$, a logical variable δ_d of type d^*. These variables will store the objects in class d that are blocked (as will be explained below). Finally, P' and Q' denote the result of applying to P and Q a *simultaneous* substitution having the "components" $[e_0/\mathsf{self}], [\delta_c \cdot \mathsf{self}/\delta_c], [e_1/u_1], \ldots, [e_n/u_n]$ (a formal definition will follow). We require that no temporary variables other than the formal parameters $u_1, \ldots, u_n$ occur in P or Q.

Before explaining the above rule let us first summerize the execution of a method call: First, control is transferred from the sender of the message to the receiver (*context switching*). The formal parameters of the receiver are initialized with the values of the expressions that form the actual parameters of the message and the other temporary variables are initialized to nil. Then the body S of the method is executed. After that the result expression e is evaluated, control is returned to the sender, the temporary variables are restored, and the result object is assigned to the variable x.

The first thing, the context switching, is represented by the substitutions $[e_0/\mathsf{self}]$, $[\delta_c \cdot \mathsf{self}/\delta_c]$ (the append operation is denoted by $\cdot$), and $[\bar{e}/\bar{u}]$ (where $\bar{e} = e_1, \ldots, e_n$ and $\bar{u} = u_1, \ldots, u_n$).

The transfer of control itself corresponds with a 'virtual' statement $\mathsf{self} := e_0$. Thus we see that if $P[e_0/\mathsf{self}]$ holds from the viewpoint of the sender then P holds from the viewpoint of the receiver after the transfer of control (i.e. after $\mathsf{self} := e_0$). Or, in other words, an assertion P as seen from the receiver's viewpoint is equivalent to $P[e_0/\mathsf{self}]$ from the viewpoint of the sender.

Definition 12. We have the following main cases of the substitution operation $[e/\mathsf{self}]$: $x[e/\mathsf{self}] = e\,.\,x$ and $\mathsf{self}[e/\mathsf{self}] = e$.

Note that this substitution changes the class of the assertion: $P[e_0/\mathsf{self}] \in Ass^c$ whereas $P \in Ass^{c'}$.

The (standard) substitution $[\delta_c \cdot \mathsf{self}/\delta_c]$ models the other aspect of the context switch, namely that the sender of the message is blocked when the receiver

is active. This aspect of the control switch thus corresponds with a virtual statement $\delta_c := \delta_c \cdot \mathsf{self}$. Moreover, the implicit check that the receiver itself is not blocked is expressed by the additional information $\mathsf{self} \notin \delta_{c'}$ in the precondition of the receiver (below is given an example of how this information can be used).

Now the passing of the parameters is simply represented by the *simultaneous* substitution $[\bar{e}/\bar{u}]$. (Note that we really need *simultaneous* substitution here, because u_i might occur in an e_j with $j < i$, but it should not be substituted again.) In reasoning about the body of the method we may also use the information that temporary variables that are not parameters are initialized to nil.

The second thing to note is the way the result is passed back. Here the logical variable r plays an important role. This is best understood by imagining after the body S of the method the statement $r := e$ (which is syntactically illegal, however, because r is a *logical* variable). In the sending object one could imagine the (equally illegal) statement $x := r$. Now if the body S terminates in a state where $Q[e/r]$ holds (a premiss of the rule) then after this "virtual" statement $r := e$ we would have a situation in which Q holds. Otherwise stated, the assertion Q describes the situation after executing the method body, in which the result is represented by the logical variable r, everything seen from the viewpoint of the receiver. Now if we context-switch this Q to the sender's side, and if it implies $R[r/x]$, then we know that after assigning the result to the variable x (our second imaginary assignment $x := r$), the assertion R will hold.

Now we come to the role of $\bar{f}$ and $\bar{z}$. We know that during the evaluation of the method the sending object becomes blocked, that is, it cannot answer any incoming messages. Therefore its instance variables will not change in the meantime. The temporary variables will be restored after the method is executed, so these will also be unchanged and finally the symbol self will retain its meaning over the call. All the expressions in class c (and in particular the f_i) are built from these expressions plus some inherently constant expressions and therefore their value will not change during the call. However, the method can change the variables of other objects and new objects can be created, so that the properties of these unchanged expressions *can* change. In order to be able to make use of the fact that the expressions $\bar{f}$ are constant during the call, the rule offers the possibility to replace them temporarily by the logical variables $\bar{z}$, which are automatically constant. So, in reasoning from the receiver's viewpoint (in the rule this applies to the assertions P and Q) the value of the expression f_i is represented by z_i, and in context switching f_i comes in again by the substitution $[\bar{f}/\bar{z}]$. Note that the constancy of $\bar{f}$ is guaranteed up to the point where the result of the method is assigned to x, and that x may occur in f_i, so that it is possible to make use of the fact that x remains unchanged right up to the assignment of the result.

Example 4. Let us illustrate the use of the above rule by a small example. Consider the unit $U = c : \langle m \Leftarrow (u_0) : x_1 := u_0 \uparrow x_2 \rangle$ and the program $\rho = \langle U | c : x_1 := u_1 ! m(x_2) \rangle$. We want to show

$$\Big\{ u_1 \,.\, x_1 = x_1 \wedge \neg u_1 = \mathsf{self} \Big\} \rho \Big\{ u_1 \,.\, x_1 = x_2 \wedge x_1 = u_1 \,.\, x_2 \Big\}.$$

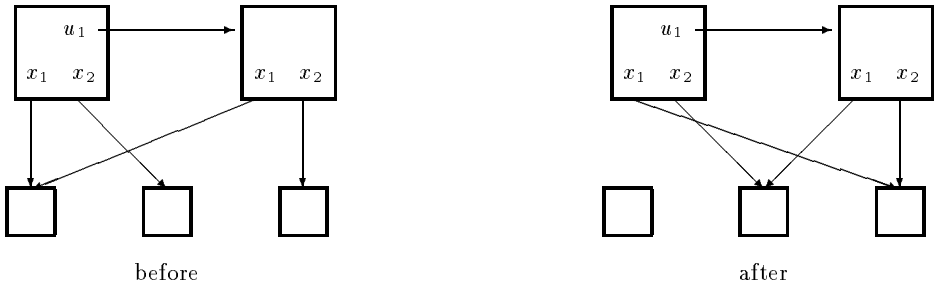


Fig. 2. The situation before and after sending the message (example 4)

So let us apply the rule (MI) with the following choices:

$$\begin{aligned} P &\equiv x_1 = z_1 \wedge \neg \mathsf{self} = z_2 \\ Q &\equiv x_1 = u_0 \wedge r = x_2 \wedge \neg \mathsf{self} = z_2 \\ R &\equiv u_1 \,.\, x_1 = x_2 \wedge x_1 = u_1 \,.\, x_2 \\ k &= 0 \qquad \text{(we shall use no } v_i\text{)} \\ f_1 &\equiv x_1 \qquad \text{(represented by } z_1 \text{ in } P \text{ and } Q\text{)} \\ f_2 &\equiv \mathsf{self} \qquad \text{(represented by } z_2 \text{ in } P \text{ and } Q\text{)} \end{aligned}$$

First notice that $P[u_1, x_2/\mathsf{self}, u_0][x_1, \mathsf{self}/z_1, z_2] \equiv u_1 \,.\, x_1 = x_1 \wedge \neg u_1 = \mathsf{self}$ so that the result of the rule is precisely what we want.

For the first premiss we have to prove

$$\Big\{x_1 = z_1 \wedge \neg \mathsf{self} = z_2\Big\}\langle U | c : x_1 := u_0\rangle\Big\{x_1 = u_0 \wedge x_2 = x_2 \wedge \neg \mathsf{self} = z_2\Big\}.$$

This is easily done with the appropriate assignment axiom and the rule of consequence.

With respect to the second premiss, we have

$$\begin{aligned} Q[u_1, x_2/\mathsf{self}, u_0][x_1, \mathsf{self}/z_1, z_2] &\equiv u_1 \,.\, x_1 = x_2 \wedge r = u_1 \,.\, x_2 \wedge \neg u_1 = \mathsf{self} \\ R[r/x_1] &\equiv \mathsf{if}\ u_1 = \mathsf{self}\ \mathsf{then}\ r\ \mathsf{else}\ u_1 \,.\, x_1\ \mathsf{fi} = x_2 \wedge r = u_1 \,.\, x_2 \end{aligned}$$

It is quite clear that the first implies the second, and we can use this implication as an axiom.

In the above example we did not need to use the information represented by the logical variables δ_c. The following example illustrates the use of these variables in reasoning about deadlock.

Example 5. Consider the program $\rho = \langle U \mid c : x := \mathsf{self}!m\rangle$, where m is defined in U without parameters. Since this program obviously deadlocks (in general we will have to deal with longer cycles in the calling chain) we have the validity of $\Big\{\mathsf{true}\Big\}\rho\Big\{\mathsf{false}\Big\}$. This can be proved simply by observing that true is equivalent to $\mathsf{self} \in \delta_c \cdot \mathsf{self}$ and that the latter assertion can be obtained by applying the substitution $[\delta_c \cdot \mathsf{self}/\delta_c]$ to the assertion $\mathsf{self} \in \delta_c$. But this latter assertion, which

by the above we can use as the part P of the precondition of the receiver in the rule (MI), obviously contradicts the additional assumption that $\mathsf{self} \notin \delta_c$. Thus the entire precondition of the receiver reduces to false from which we can derive false as the postcondition of the body of the method m. From which in turn we can derive easily by rule (MI) the correctness formula above.

5 Conclusions

In this paper we have given a proof system for a sequential object-oriented programming language, called SPOOL, that fulfills the requirements we have listed in the introduction.

In [6] detailed proofs are given of both *soundness* (i.e. every derivable correctness formula is valid) and *(relative) completeness* (every valid correctness formula is derivable, assuming, as additional axioms, all the valid assertions). These proofs are considerable elaborations of the corresponding proofs of the soundness and completeness of a simple sequential programming language with recursive procedures (as described in, for example, [3] and [5]).

Related work To the best of our knowledge the proof system presented is the first sound and complete proof system for a sequential object-oriented language. In [1] and [10] different Hoare-style proof systems for sequential object-oriented languages are given which are based on the *global store* model as it has been developed for the semantics of Algol-like languages. This model however introduces a difference between the abstraction level of the assertion language and that of the programming language itself. Moerover, as observed in [1], the global store model gives rise to incompleteness.

Future research The proof rule for message passing, incorporating the passing of parameters and result, context switching, and the constancy of the variables of the sending object, is rather complex. It seems to work fine for our proof system, but its properties have not yet been studied extensively enough. It would be interesting to see whether the several things that are handled in one rule could be dealt with by a number of different, simpler rules.

We have considered in this paper only partial correctness. But we are currently working on extensions which allow one to prove absence of deadlock and termination.

In the present proof system the protection properties of objects are not reflected very well. While in the programming language it is not possible for one object to access the internal details (variables) of another one, in the assertion language this *is* allowed. In order to improve this it might be necessary to develop a system in which an object presents some abstract view of its behaviour to the outside world. Such an abstract view of an object we expect to consist of a specification of the *interface* of an object as it is used in [4, 6, 7, 8] for reasoning about systems composed of objects which execute in parallel.

Related to the above is the problem of a formal justification of the appropriateness of the abstraction level of a formalism for describing properties of

dynamically evolving object structures. We expect that such a formal justification involves a fully abstract semantics of the notion of on object. A related question, as already described above, is to what extent the problems with the incompleteness of the global store model are due to the particular choice of the abstraction level.

In any case, we expect that our approach provides an appropriate basis for specifying such high-level object-oriented programming mechanisms like subtyping, abstract types and inheritance.

Acknowledgement

This paper reports on joint work with P. America.

References

1. M. Abadi and K.R. M. Leino: A logic of object-oriented programs. Proceedings of the 7th International Joint Conference CAAP/FASE, vol. 1214 of Lecture Notes in Computer Science, April 1997.
2. P. America: Definition of the programming language POOL-T. ESPRIT project 415A, Doc. No. 0091, Philips Research Laboratories, Eindhoven, the Netherlands, September 1985.
3. K.R. Apt: Ten years of Hoare logic: a survey — part I. ACM Transactions on Programming Languages and Systems, Vol. 3, No. 4, October 1981, pp. 431–483.
4. P. America and F.S. de Boer: Reasoning about dynamically evolving process structures. Formal Aspects of Computing. Vol. 6, No. 3, 1994.
5. J.W. de Bakker: Mathematical Theory of Program Correctness. Prentice-Hall International, Englewood Cliffs, New Jersey, 1980.
6. F.S. de Boer: Reasoning about dynamically evolving process structures (A proof theory of the parallel object-oriented language POOL). PhD. Thesis. Free University, Amsterdam, 1991.
7. F.S. de Boer: A proof system for the parallel object-oriented laguage POOL. Proceedings of the seventeenth International Colloquium on Automata, Languages and Programming (ICALP), Lecture Notes in Computer Science, Vol. 443, Warwick, England, 1990.
8. F.S. de Boer: A compositional proof system for dynamic process creation. Proceedings of the sixth annual IEEE symposium on Logics in Computer Science (LICS), IEEE Computer Society Press, Amsterdam, The Netherlands, 1991.
9. J.M. Morris: Assignment and linked data structures. Manfred Broy, Gunther Schmidt (eds.): Theoretical Foundations of Programming Methodology. Reidel, 1982, pp. 35–41.
10. A. Poetzsch and P. Mueller: Logical foundations for typed object-oriented languages. Proceedings of the IFIP Working Conference on Programming Concepts and Methods (PROCOMET98).
11. D. S. Scott: Identity and existence in intuitionistic logic. M.P. Fourman, C.J. Mulvey, D.S. Scott (eds.): Applications of Sheaves. Proceedings, Durham 1977, Springer-Verlag, 1979, pp. 660–696 (Lecture Notes in Mathematics 753).

The Recognizability Problem for Tree Automata with Comparisons between Brothers

Bruno Bogaert, Franck Seynhaeve, and Sophie Tison

LIFL, Bât M3, Université Lille 1, F59655 Villeneuve d'Ascq cedex, France
email: bogaert,seynhaev,tison@lifl.fr

Abstract. Several extensions of tree automata have been defined, in order to take in account non-linearity in terms. Roughly, these automata allow equality or disequality constraints between subterms. They have been used to get decision results, e.g. in term rewriting. One natural question arises when we consider a language recognized by such an automaton: is this language recognizable, i.e. are the constraints necessary? Here we study this problem in the class $REC_{\neq}$ corresponding to comparisons between brothers and we prove its decidability. It gives e.g. a decision procedure for testing whether the image by a quasi-alphabetic homomorphism of a recognizable tree language is recognizable.

1 Introduction

Even if many concepts in tree languages can be viewed as extensions of the word case, some new difficulties and phenomena arise when we consider trees, in particular "non-linearity" (a term is non linear if it contains two occurrences of the same variable). For example, the family of recognizable sets is not closed under non-linear homomorphism. Actually tree automata can't deal with non linear terms: e.g. the set of terms containing an occurrence of $f(x,x)$ is not recognizable. As non linear terms occur very often, e.g. in logic or equational programming, several extensions of tree automata have been defined, in order to take in account non-linearity in terms.

The first one is the class of automata with equality tests (*Rateg* automata) [13]; unfortunately, the emptiness property is undecidable for this class. Several "decidable" classes have then been defined, dealing with restrictions to the tests in order to keep good decidability and closure properties.

First, Bogaert and Tison [3] introduced $REC_{\neq}$ automata (tree automata with comparisons between brothers) and denoted $REC_{\neq}$ the set of languages recognized by these automata. The rules use tests in order to impose either equalities, or differences between brother terms: rules like $f(q,q)[x_1 = x_2] \to q_1$ or $f(q,q)[x_1 \neq x_2] \to q_2$ are allowed. The emptiness problem in $REC_{\neq}$ has been proved decidable in [3] and the class has good closure properties.

One more general class with good decidability properties has then been introduced (Caron et al. [5,4,6]): the class of reduction automata, which roughly allow arbitrary disequality constraints but only finitely many equality constraints on

W. Thomas (Ed.): FOSSACS'99, LNCS 1578, pp. 150-164, 1999.

each run of the automaton. By using these classes interesting decision results have been got; for example, the encompassment theory [1] can be shown decidable by using reduction automata and decidability of ground reducibility is a direct consequence of this result ([7]).

One natural question arises when we consider a language recognized by an automaton with tests: is this language recognizable, and in this case can we compute the corresponding "classic" automaton? In other words, can we decide whether "constraints are really necessary to define the language"? Getting rid of constraints allows e.g. to use classical algorithms for recognizable sets. For the class of reduction automata, this problem contains strictly the decidability of recognizability for the set of normal forms of a rewrite system, problem solved but whose proofs are very technical [12,14].

Here we give a positive answer to this problem for $REC_{\neq}$ languages: we can decide whether such a language is recognizable (and compute a classic automaton when it exists). This partial result has some interesting corollaries; it gives e.g. a decision procedure for testing whether the image by a quasi-alphabetic homomorphism of a recognizable tree language is recognizable. (This result can be connected with the cross-section theorem; the cross-section theorem is false in general for trees; it is true when the morphism is linear [1], or when the morphism is quasi-alphabetic and the image is recognizable. It is conjectured true when the image is recognizable [8]). The result can also be used to decide properties of term rewrite systems. When a rewrite system R has "good" properties (same occurrences of a variable are "brothers": it includes the case of shallow systems [11]), it gives a procedure to test recognizability of the set of normal forms of R which is much easier than the general one and it allows testing whether the set of direct descendants $R(L)$ is recognizable for a recognizable language L: testing these properties can be useful e.g for computing normalizing terms, for computing reachable terms... ([15],[10]).

The spirit of the proof is natural: we define a kind of "minimization" very similar to the classical one (Myhill-Nerode theorem for tree languages [9,6]). The difficulty is to extend the notion of context by adding equality or disequality constraints. Then the point is that in the "minimized" automaton, it should appear "clearly" whether the constraints are necessary or not: e.g., when we get two rules $f(q,q)[x_1 = x_2] \to q_1$ and $f(q,q)[x_1 \neq x_2] \to q_2$, with q_1 and q_2 non equivalent, it should mean that we need the constraints and so that the language is not recognizable. Actually, the proof is a little more intricate and finite languages can disturb the "natural" minimization. E.g. the "minimized" automaton associated with the recognizable language $h^*(\{f(a,a), f(b,b)\})$ is $a \to q, b \to q, f(q,q)[x_1 = x_2] \to q_f, h(q_f) \to q_f$ and then uses constraints. So, a first step of the proof is devoted to eliminate these degenerate cases.

After basic definitions given in Section 2, $REC_{\neq}$ automata are introduced in Section 3. The Section 4 is devoted to the proof.

[1] The encompassment theory is the set of first order formula with predicates $red_t(x)$, t term. In the theory $red_t(x)$ holds if and only if x is a ground term encompassing t i.e. an instance of t is a subterm of x.

2 Preliminaries

The set of nonnegative integers is denoted $\mathbb{N}$ and $\mathbb{N}^*$ denotes the set of finite–length strings over $\mathbb{N}$. For $n \in \mathbb{N}$, $[n]$ denotes the set $\{1, \ldots, n\}$, so that $[0]$ is another name for the empty set $\emptyset$.

An alphabet Σ is *ranked* if $\Sigma = \bigcup_p \Sigma_p$ where $\Sigma_p \neq \emptyset$ only for a finite number of p's and the non empty Σ_p are finite and pairwise disjoint. Elements of Σ_p are said to be of *arity* p. Elements of arity 0 are called *constants*. We suppose that Σ contains at least one constant.

Let $\mathcal{X}$ be a set of variables. A *term* over $\Sigma \cup \mathcal{X}$ is a partial function $t : \mathbb{N}^* \to \Sigma \cup \mathcal{X}$ with domain $\mathcal{P}os(t)$ satisfying the following properties:

- $\mathcal{P}os(t)$ is nonempty and prefix-closed;
- If $t(\alpha) \in \Sigma_n$, then $\{i \in \mathbb{N} \mid \alpha i \in \mathcal{P}os(t)\} = \{1, 2, \ldots, n\}$;
- If $t(\alpha) \in \mathcal{X}$, then $\{i \in \mathbb{N} \mid \alpha i \in \mathcal{P}os(t)\} = \emptyset$.

The set of all terms (or *trees*) is denoted by $T_\Sigma(\mathcal{X})$. If $\mathcal{X} = \emptyset$ then $T_\Sigma(\mathcal{X})$ is denoted by T_Σ. Each element of $\mathcal{P}os(t)$ is called a *position*.
Let $t \in T_\Sigma(\mathcal{X})$ and $p \in \mathcal{P}os(t)$. We denote by $t|_p$ the subterm of t rooted at position p and by $t(p)$ the label of t at position p. $\forall i \in [n]$ such that $pi \in \mathcal{P}os(t)$, $t|_{pi}$ is said to be a *son* of the label $t(p)$.
Let $\mathcal{X}_n$ be a set of n variables. A term $C \in T_\Sigma(\mathcal{X}_n)$ where each variable occurs at most once in C is called a context. The term $C[t_1, \ldots, t_n]$ for $t_1, \ldots, t_n \in T_\Sigma$ denotes the term in T_Σ obtained from C by replacing for each $i \in [n]$ x_i by t_i. We denote by $\mathcal{C}^n(\Sigma)$ the set of contexts over n variables $\{x_1, \ldots, x_n\}$ and $\mathcal{C}(\Sigma)$ the set of contexts containing a single variable.

3 Tree Automata with Comparisons between Brothers

Automata with comparisons between brothers ($REC_{\neq}$ automata) have been introduced by Bogaert and Tison [3]. They impose either equalities, or differences between brother terms. These equalities and differences are expressed by constraint expressions. Here we will restrict to define normalized-complete $REC_{\neq}$ automata (each $REC_{\neq}$ automaton is equivalent to a automaton called normalized-complete $REC_{\neq}$ automaton [3]).
Rules of normalized-complete $REC_{\neq}$ automata impose, for each pair (pi, pj) of positions of a term t where p is a position and $i \neq j \in \mathbb{N}$, that $t|_{pi} = t|_{pj}$ or $t|_{pi} \neq t|_{pj}$. These comparisons are expressed by full constraint expressions.
First, we define the notion of full constraint expressions. Then we give the definition of normalized-complete $REC_{\neq}$ automata.

Definition 1. A full constraint expression c over n variables $(x_i)_{i \in [n]}$, $n \in \mathbb{N}$, (in the following x_i will always denote the i^{th} son of a node) is a conjunction of equalities $x_i = x_j$ and of disequalities $x_i \neq x_j$ such that there exists a partition $(E_i)_{i \in [m]}$ of $[n]$, $m \leq n$ satisfying:

$$c = \bigwedge_{k \in [m]} \bigwedge_{l, l' \in E_k} x_l = x_{l'} \wedge \bigwedge_{k, k' \in [m], k \neq k'} \bigwedge_{l \in E_k, l' \in E_{k'}} x_l \neq x_{l'} \qquad (1)$$

We denote $c = (E_i)_{i \in [m]}$ in order to simplify the notation, $card(c) = m$ the cardinality of c and CE'_n the set of full constraint expressions over n variables.

For example, $CE'_3 = \{(\{1,2,3\}), (\{1,2\},\{3\}), (\{1,3\},\{2\}), (\{2,3\},\{1\}), (\{1\},\{2\},\{3\})\}$.
In the case $n = 0$, the full constraint expression over no variable is denoted by $\top$ (null constraint).

Definition 2. A tuple of terms $(t_i)_{i\in[n]}$ satisfies a full constraint expression c iff the evaluation of c for the valuation $(\forall i \leq n, x_i = t_i)$ is $true$, when "$=$" is interpreted as equality of terms, "$\neq$" as its negation, "$\top$" as $true$, $\wedge$ as the usual boolean function and. For example, the tuple of constants (a, b, a) satisfies the full constraint expression $x_1 \neq x_2 \wedge x_1 = x_3 \wedge x_2 \neq x_3$.

Let us remark that if c and c' are full constraint expressions over n variables then $c \wedge c'$ is unsatisfiable if $c \neq c'$.

Definition 3. Let c be a full constraint of CE'_n and $(q_i)_{i\in[n]}$ be a n-tuple of states. We say that $(q_i)_{i\in[n]}$ satisfies the equality constraints of c if for $\forall k, l \in [n]$, $(c \Rightarrow (x_k = x_l)) \Rightarrow (q_k = q_l)$.

Let us now define normalized-complete $REC_{\neq}$ automata.

Definition 4. A normalized-complete automaton $\mathcal{A}$ with comparisons between brothers (normalized-complete $REC_{\neq}$ automaton) is a tuple $(\Sigma, Q, F, \mathcal{R})$ where Σ is a ranked alphabet, Q a finite set of states, $F \subseteq Q$ a set of final states and $\mathcal{R} \subseteq \bigcup_i \Sigma_i \times CE'_i \times Q^{i+1}$ a set of rules (a rule $(f, c, q_1, \ldots, q_n, q)$ will be denoted $f(q_1, \ldots, q_n)[c] \to q$) with:

- $\mathcal{A}$ deterministic i.e. for all rules $f(q_1, \ldots, q_n)[c] \to q$ and $f(q_1, \ldots, q_n)[c] \to q'$, $q = q'$;
- For each letter $f \in \Sigma_n$, each n-tuple $(q_i)_{i\in[n]} \in Q$, each constraint c of CE'_n such that $(q_i)_{i\in[n]}$ satisfies the equality contraints of c, there exists at least one rule $f(q_1, \ldots, q_n)[c] \to q$;
- And for each letter $f \in \Sigma_n$, each n-tuple $(q_i)_{i\in[n]} \in Q$, each constraint c of CE'_n such that $(q_i)_{i\in[n]}$ doesn't satisfy the equality contraints of c, there exists no rule $f(q_1, \ldots, q_n)[c] \to q \in \mathcal{R}$.

Let $f \in \Sigma_n$ and $(t_i)_{i\in[n]}$ be terms of T_Σ. The relation $\xrightarrow{*}_{\mathcal{A}}$ is defined as follows:

$$f(t_1, \ldots t_n) \xrightarrow{*}_{\mathcal{A}} q \text{ if and only if}$$
$$\left(\begin{array}{r} \exists f(q_1, \ldots, q_n)[c] \to q \in \mathcal{R} \text{ such that } \forall i \in [n], t_i \xrightarrow{*}_{\mathcal{A}} q_i \\ \text{and } (t_i)_{i\in[n]} \text{ satisfies the constraint } c \end{array}\right)$$

Let q be a state of Q. We denote by $\mathcal{L}_{\mathcal{A}}(q)$ the set of terms t such that $t \xrightarrow{*}_{\mathcal{A}} q$. A tree $t \in T_\Sigma$ is accepted by $\mathcal{A}$ if there exists a final state q such that $t \in \mathcal{L}_{\mathcal{A}}(q)$. The language $\mathcal{L}(\mathcal{A})$ recognized by $\mathcal{A}$ is the set of accepted terms. We denote by $REC_{\neq}$ the set of tree languages recognized by the class of $REC_{\neq}$ automata.

Example 5. Let $\mathcal{A} = (\{a, h, f\}, \{q, q_f, q_p\}, \{q_f\}, \mathcal{R})$ with $\mathcal{R}$:

$$\begin{array}{ccccc} a \to q & & h(q) \to q & & h(q_f) \to q_p \\ h(q_p) \to q_p & & f(q,q,q)[c] \to q_f & & f(q,q,q)[c'] \to q_p \ \forall c' \in CE'_3 \setminus \{c\} \\ f(q_1,q_2,q_3)[c'] \to q_p & \forall (q_1,q_2,q_3) \in Q^3 \setminus \{(q,q,q)\}, \forall c' \in CE'_3 & & & \end{array}$$

where c is the full constraint expression $x_1 = x_2 \wedge x_3 \neq x_1 \wedge x_3 \neq x_2$. Then $\mathcal{A}$ recognizes the language $\{f(h^n(a), h^n(a), h^m(a)) \mid m, n \in \mathbb{N}, m \neq n\}$.

4 Recognizability Problem

We consider the recognizability problem in the class $REC_{\neq}$:

Input: A ranked alphabet Σ and a language $\mathcal{F} \in REC_{\neq}$.

Question: Is $\mathcal{F}$ recognizable?

We will prove that the recognizability problem is decidable; furthermore, when the input language is recognizable, our algorithm computes a corresponding tree automaton.

The idea of the algorithm is the following: we define a kind of minimization, close to the classic one (Myhill-Nerode theorem for tree languages [9,6]) but dealing with constraints: roughly, two states will be equivalent, when they have the same behaviour for the same context with constraints. This needs defining constrained terms which are terms labeled with equality and disequality constraints. Then, the point is that, when the reduction works well, it should be the case that non necessary constraints are dropped. For example, let us suppose that we have two rules $f(q,q)[x_1 = x_2] \to q_1$ and $f(q,q)[x_1 \neq x_2] \to q_2$; when q_1 and q_2 are equivalent, it means that the constraints are not necessary.

However, the reasoning fails when the language associated with a state is finite: $a \to q, b \to q, f(q,q)[x_1 = x_2] \to q_f$ use constraints to define the finite (thus recognizable) language $\{f(a,a), f(b,b)\}$. So in a first step, we eliminate states q s.t. $\mathcal{L}_{\mathcal{A}}(q)$ is finite (section 4.1). Then we extend the notion of context to take in account equality and disequality constraints (section 4.2) and then, we define and compute "the" reduced automaton (section 4.3). Finally, we prove that the language is recognizable iff the reduced automaton is not "constraint-sensitive" (section 4.4), i.e. two rules whose left-hand-side differ only by constraints have the same right-hand-side. We deduce decidability of the recognizability problem in the class $REC_{\neq}$ and obtain an effective construction of the corresponding automaton, when the language is recognizable.

4.1 How to reduce to the "infinite" case

Let $\mathcal{F} \in REC_{\neq}$ and $\mathcal{A} = (\Sigma, Q, F, \mathcal{R})$ be a normalized-complete $REC_{\neq}$ automaton recognizing $\mathcal{F}$. Let us suppose that there exists at least a state q of $\mathcal{A}$ such that $\mathcal{L}_{\mathcal{A}}(q)$ is finite. Let us denote:

$$\mathcal{F}_1 = \bigcup_{q \in F, \mathcal{L}_{\mathcal{A}}(q) \text{ finite}} \mathcal{L}_{\mathcal{A}}(q) \quad \text{and} \quad \mathcal{F}_2 = \bigcup_{q \in F, \mathcal{L}_{\mathcal{A}}(q) \text{ infinite}} \mathcal{L}_{\mathcal{A}}(q).$$

Since $\mathcal{L}(\mathcal{A}) = \mathcal{F}_1 \cup \mathcal{F}_2$ and $\mathcal{F}_1$ is finite, $\mathcal{L}(\mathcal{A})$ is recognizable iff $\mathcal{F}_2$ is recognizable. The language $\mathcal{F}_2$ is recognized by the $REC_{\neq}$ automaton $\mathcal{B} = (\Sigma, Q, F', \mathcal{R})$ where $F' = \{q | q \in F, \mathcal{L}_{\mathcal{A}}(q) \text{ infinite}\}$. We construct a new alphabet Γ by encoding, for each state q such that $\mathcal{L}_{\mathcal{B}}(q)$ is finite, the terms of $\mathcal{L}_{\mathcal{B}}(q)$ in the symbols of Γ. We define a $REC_{\neq}$ automaton $\mathcal{B}'$ on Γ and a linear morphism φ from $T_{\Gamma}(\mathcal{X})$ onto $T_{\Sigma}(\mathcal{X})$ such that for each state q of $\mathcal{B}'$, $\mathcal{L}_{\mathcal{B}'}(q)$ is infinite and such that $\varphi(\mathcal{L}(\mathcal{B}')) = \mathcal{L}(\mathcal{B})$ and $\varphi^{-1}(\mathcal{L}(\mathcal{B})) = \mathcal{L}(\mathcal{B}')$. We deduce that ($\mathcal{L}(\mathcal{B})$ is recognizable) $\Leftrightarrow$ ($\mathcal{L}(\mathcal{B}')$ is recognizable) since φ is linear (the entire proof can be found in [2]). We deduce that the general case can be reduced to the infinite case since for each state q of the automaton $\mathcal{B}'$, $\mathcal{L}_{\mathcal{B}'}(q)$ is infinite.

Before studying the "infinite" case, let us give an example of construction of $\mathcal{B}'$ and φ. Let $\Sigma = \{a/0, f/2\}$ and $\mathcal{B} = (\Sigma, Q, F', \mathcal{R})$ where $Q = \{q, q_p, q_f\}$, $F' = \{q_f\}$ and $\mathcal{R}$ is composed of the following rules:

$$\begin{array}{rr} a \to q & f(q,q)[x_1 = x_2] \to q_f \\ f(q_f, q_f)[x_1 = x_2] \to q_f & f(q_p, q_p)[x_1 = x_2] \to q_p \\ f(q_1, q_2)[x_1 \neq x_2] \to q_p \ \forall (q_1, q_2) \in (Q \times Q) & \end{array}$$

$\mathcal{B}$ is a normalized-complete $REC_{\neq}$ automaton. Obviously $\mathcal{L}_{\mathcal{B}}(q) = \{a\}$ and, $\mathcal{L}_{\mathcal{B}}(q_p)$ and $\mathcal{L}_{\mathcal{B}}(q_f)$ are infinite. Then we consider $\Box$ a symbol not in Σ and we define the alphabet $\Gamma = \{f_{(\Box,\Box)}, f_{(\Box,a)}, f_{(a,\Box)}, f_{(a,a)}\}$.
Then $\mathcal{B}' = (\Gamma, Q', F', \mathcal{R}')$ is the $REC_{\neq}$ automaton where $Q' = \{q_p, q_f\}$ and $\mathcal{R}'$ is composed of the following rules:

$$\begin{array}{ll} f_{(a,a)} \to q_f & f_{(\Box,\Box)}(q_1, q_2)[x_1 \neq x_2] \to q_p \ \forall (q_1, q_2) \in (Q' \times Q') \\ f_{(\Box,a)}(q_1) \to q_p \ \forall q_1 \in Q' & f_{(\Box,\Box)}(q_f, q_f)[x_1 = x_2] \to q_f \\ f_{(a,\Box)}(q_2) \to q_p \ \forall q_2 \in Q' & f_{(\Box,\Box)}(q_p, q_p)[x_1 = x_2] \to q_p \end{array}$$

And $\varphi : T_{\Gamma}(\mathcal{X}) \to T_{\Sigma}(\mathcal{X})$ is the linear morphism defined as follows:

$$\begin{array}{rr} \varphi(f_{(a,a)}) = f(a,a) & \varphi(f_{(\Box,a)})(x_1) = f(x_1, a) \\ \varphi(f_{(\Box,\Box)})(x_1, x_2) = f(x_1, x_2) & \varphi(f_{(a,\Box)})(x_1) = f(a, x_1) \end{array}$$

So we can suppose in the rest of the proof that for each state q of the normalized-complete automaton $\mathcal{A} = (\Sigma, Q, F, \mathcal{R})$ recognizing $\mathcal{F}$, $\mathcal{L}_{\mathcal{A}}(q)$ is infinite.

4.2 Constrained Terms

In the class of recognizable tree languages, an equivalence relation using contexts is used in order to minimize the automata (Myhill-Nerode theorem for tree languages [9,6]). We define a similar notion in the class of $REC_{\neq}$ automata. As the rules of $REC_{\neq}$ automata contain comparisons between brother terms, we introduce the notion of terms imposing equalities and disequalities between brother terms, these comparisons being expressed by full constraint expressions. Such terms are called constrained terms. The label of a constrained term at a position p is the combination of a symbol and of a full constraint expression c such that the equality constraints of c are satisfied by the sons of the label and such that there is no disequality constraint between equal ground sons of the label. Leaves of a constrained term may also be states or occurences of an unique variable.
More formally, let x be a variable and Σ' be the ranked alphabet defined by $\forall n \in \mathbb{N}$, $\Sigma'_n = \{f_c \mid f \in \Sigma_n, c \in CE'_n\}$. A *constrained term* C over $\Sigma \cup Q$ is a term of $T_{\Sigma'}(Q \cup \{x\})$ where the states of Q are constants and $\forall p$ non leaf position of C, $\exists n > 0$, such that $C(p) = f_c \in \Sigma'_n$ with:

- The n-tuple $(C|_{pi})_{i \in [n]}$ satisfies the equality constraints of c;
- c contains no disequality constraint between equal ground sons i.e. $\forall i, j \in [n], (C|_{pi} \in T_{\Sigma'}$ and $C|_{pi} = C|_{pj}) \Rightarrow (c \Rightarrow (x_i = x_j))$.

Example 6. Let $g, f \in \Sigma_2$ and $q_1, q_2 \in Q$. Then $f_c(g_{c'}(q_1, x), g_{c'}(q_2, x))$ with $c = [x_1 = x_2]$ and $c' = [x_1 \neq x_2]$ is not a constrained term since $g_{c'}(q_1, x) \neq g_{c'}(q_2, x)$. But $f_c(g_{c'}(q_1, x), g_{c'}(q_1, x))$ with $c = [x_1 = x_2]$ and $c' = [x_1 \neq x_2]$ is a constrained term.

Constrained terms are terms hence we use the usual notion of $height$ of a term on constrained terms with $height(c) = 0$ if $c \in Q \cup \{x\}$ and $height(c) = 1$ if $c \in \Sigma'_0$. Let C be a constrained term and $q \in Q$, we denote by $C[q]$ the constrained term obtained from C by replacing each occurence of x by q.

Run on constrained terms We extend the notion of run on terms to run on constrained terms. Let C be a constrained term and q, q' be states. We denote $C[q] \xrightarrow{*}_{\mathcal{A}} q'$ iff

- Either $C = q'$ or $(C = x$ and $q = q')$;
- Or $C = f_c(C_1, \ldots, C_n)$ with $f_c \in \Sigma'_n$, $(C_i)_{i \in [n]}$ constrained terms such that $\forall i \in [n]\, C_i[q] \xrightarrow{*}_{\mathcal{A}} q_i$ and $f(q_1, \ldots, q_n)[c] \to q' \in \mathcal{R}$.

Let us now extend the notion of run to run between constrained terms. Let C, C' be constrained terms and q be a state. We denote $C[q] \xrightarrow{*}_{\mathcal{A}} C'$ iff there exists a set P of positions of C such that:

- $\forall p \in P$, $C'|_p \in Q$ and $C[q]|_p \xrightarrow{*}_{\mathcal{A}} C'|_p$;
- $\forall p \in \mathcal{P}os(C)$ not prefixed by a position of P, $C'(p) = C[q](p)$.

4.3 Minimization

Definition 7. let $\equiv_{\mathcal{A}}$ be the relation on Q defined by for all $q, q' \in Q$, $q \equiv_{\mathcal{A}} q'$ if for each constrained term C, $(C[q] \xrightarrow{*}_{\mathcal{A}} q_1 \in F \Leftrightarrow C[q'] \xrightarrow{*}_{\mathcal{A}} q_2 \in F)$.

The relation $\equiv_{\mathcal{A}}$ is obviously an equivalence relation. In the following, we associate with the automaton $\mathcal{A}$ a normalized-complete $REC_{\neq}$ $\mathcal{A}_m$ said "minimized" whose states are the equivalence classes of the relation $\equiv_{\mathcal{A}}$ and such that $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}_m)$.
First we prove that the equivalence classes of the relation $\equiv_{\mathcal{A}}$ are computable. Then we define the automaton $\mathcal{A}_m$.

Equivalence Classes Algorithm EQUIV
input: Normalized-complete $REC_{\neq}$ automaton $\mathcal{A} := (\Sigma, Q, F, \mathcal{R})$
begin
  Set P to $\{F, Q \setminus F\}$ /* P is the initial equivalence relation*/
  repeat
    $P' := P$
    /* Refine equivalence P' in P */
    qPq' if $qP'q'$ and $\forall C$ constrained term of height 1,
      $C[q] \xrightarrow{*}_{\mathcal{A}} q_1$ and $C[q'] \xrightarrow{*}_{\mathcal{A}} q_2$ with $q_1 P' q_2$
  until $P' = P$
output: P set of equivalence classes of $\equiv_{\mathcal{A}}$
end

We denote by $\tilde{q}$ the equivalence class of a state q w.r.t. P, the set computed by the algorithm EQUIV. Let us prove that the algorithm EQUIV is correct i.e. that P is the set of equivalence classes of $\equiv_{\mathcal{A}}$ (Lemma 10). First we consider two rules whose left hand sides differ only by replacing all occurences of one state bounded by equalities imposed by the constraint by a state of the same equivalence class w.r.t. P. Then we prove that the right hand side of the two

rules belong to the same equivalence class w.r.t. P (Lemma 8). We deduce that the equivalence classes w.r.t. P are compatible with the rules of the automaton $\mathcal{A}$ (Corollary 9).

Lemma 8. *Let $f(q_1,\ldots,q_n)[c] \to q \in \mathcal{R}$ and $f(q'_1,\ldots,q'_n)[c] \to q' \in \mathcal{R}$ such that there exists $j \in [n]$ such that $q_j \in \tilde{q'_j}$, $\forall i \in [n], ((c \Rightarrow x_i = x_j) \Rightarrow q'_i = q'_j)$ and $((c \Rightarrow x_i \neq x_j) \Rightarrow q'_i = q_i)$. Then $q \in \tilde{q'}$.*

Proof. First the rule $f(q'_1,\ldots,q'_n)[c] \to q'$ is well defined since we can prove that $(q')_{i\in[n]}$ satisfies the equality constraints of c. Let us now consider the constrained term C defined by $head(C) = f_c$ and for each $i \in [n]$ if $c \Rightarrow x_i = x_j$ then $C(i) = x$ else $C(i) = q_i$.
Obviously $\forall i \in [n], C[q_j]|_i = f(q_1,\ldots,q_n)|_i$ then $C[q_j] \to_{\mathcal{A}} q$. Let us now prove that $\forall i \in [n], C[q'_j]|_i = f(q'_1,\ldots,q'_n)|_i$. Let $i \in [n]$. If $c \Rightarrow x_i = x_j$ then $q'_i = q'_j$. Then $C[q'_j]|_i = q'_j = q'_i = f(q'_1,\ldots,q'_n)|_i$. If $c \Rightarrow x_i \neq x_j$ then $C[q'_j]|_i = q_i = q'_i = f(q'_1,\ldots,q'_n)|_i$. Hence $\forall i \in [n], C[q'_j]|_i = f(q'_1,\ldots,q'_n)|_i$ then $C[q'_j] \to_{\mathcal{A}} q'$.
Moreover C is a constrained term of height 1 hence according to the EQUIV algorithm, we have $q \in \tilde{q'}$ since $q_j \in \tilde{q'_j}$ which ends the proof of Lemma 8.

Corollary 9. *Let $f(q_1,\ldots,q_n)[c] \to q \in \mathcal{R}$ and $f(q'_1,\ldots,q'_n)[c] \to q' \in \mathcal{R}$ such that $\forall j \in [n]\ q_j \in \tilde{q'_j}$. Then $q \in \tilde{q'}$.*

Let us now prove that the algorithm EQUIV is correct.

Lemma 10. *P is the set of equivalence classes of $\equiv_{\mathcal{A}}$ i.e.:*

$$\forall q, q' \in Q\ (q \equiv_{\mathcal{A}} q') \Leftrightarrow (q \in \tilde{q'}.)$$

Proof. First, we can prove that $\forall q, q' \in Q\ (q \notin \tilde{q'}) \Rightarrow (q \not\equiv_{\mathcal{A}} q')$ by induction on the step of the algorithm EQUIV where $q \notin \tilde{q'}$ appears. We deduce that $\forall q, q' \in Q\ (q \equiv_{\mathcal{A}} q') \Rightarrow (q \in \tilde{q'})$.
In order to prove the implication $\Leftarrow$, we first prove that:

$$\forall q, q' \in Q,\ q \in \tilde{q'} \Rightarrow \left(\forall C \text{ constrained term } \begin{cases} C[q] \xrightarrow{*}_{\mathcal{A}} s \\ C[q'] \xrightarrow{*}_{\mathcal{A}} s' \end{cases} \Rightarrow (s \in \tilde{s'})\right)$$

by induction on the height of the constrained term. Let $q, q' \in Q$ such that $q \in \tilde{q'}$ and C a constrained term such that $C[q] \xrightarrow{*}_{\mathcal{A}} s$ and $C[q'] \xrightarrow{*}_{\mathcal{A}} s'$.

C **of height** 0: **Either** $C \in Q$: Hence $\exists q" \in Q$ such that $C = q"$. $C[q] = C[q'] = q"$ hence $s = s' = q"$. Finally $s \in \tilde{s'}$;
Or $C = x$: $C[q] = q$ and $C[q'] = q'$ hence $s = q$ and $s' = q'$. Finally $s \in \tilde{s'}$ since $q \in \tilde{q'}$.

Induction hypothesis: Let $k \in \mathbb{N}$. Let us suppose that the property is true for all constrained term C of height less than or equal to k. Let C be a constrained term of height k+1. There exists $f \in \Sigma_n, c \in CE'_n, (C_i)_{i\in[n]}$ constrained terms such that $C = f_c(C_1,\ldots,C_n)$. According to induction hypothesis, $\forall i \in [n], C_i[q] \xrightarrow{*}_{\mathcal{A}} q_i$ and $C_i[q'] \xrightarrow{*}_{\mathcal{A}} q'_i$ with $q_i \in \tilde{q'_i}$.

$(C_i)_{i\in[n]}$ satisfies the equality constraints of c. Moreover $\mathcal{A}$ is deterministic hence $\forall k,l \in [n]$ such that $c \Rightarrow (x_k = x_l)$, we have $q_k = q_l$ and $q'_k = q'_l$ since $C_k = C_l$. We deduce $(q_i)_{i\in[n]}$ and $(q'_i)_{i\in[n]}$ satisfies the equality constraints of c. Hence since $\mathcal{A}$ is normalized-complete, there exists $f(q_1,\ldots,q_n)[c] \rightarrow s \in \mathcal{R}$ and $f(q'_1,\ldots,q'_n)[c] \rightarrow s' \in \mathcal{R}$.
Moreover $\forall i \in [n], q_i \in \tilde{q'_i}$. We deduce from the Corollary 9 that $s \in \tilde{s'}$.

At the beginning of the execution of EQUIV, $P = \{F, Q \setminus F\}$, hence:

$$\forall q \in F, \forall q' \in Q, (q' \in \tilde{q}) \Rightarrow (q' \in F) \tag{2}$$

since at each step of the algorithm qPq'. We deduce that $\forall q, q' \in Q\ (q \not\equiv_{\mathcal{A}} q') \Rightarrow (q' \notin \tilde{q})$ which ends the proof of Lemma 10.

Let us now define the automaton $\mathcal{A}_m$. Let us denote $\tilde{q}$ the equivalence class of a state q w.r.t. $\equiv_{\mathcal{A}}$. Let $\mathcal{A}_m = (\Sigma, Q_m, F_m, \mathcal{R}_m)$ defined as follows:

- Q_m is the set of equivalence classes of $\equiv_{\mathcal{A}}$.
- $F_m = \{\tilde{q} \mid q \in F\}$.
- $\mathcal{R}_m = \{f(\tilde{q_1},\ldots,\tilde{q_n})[c] \rightarrow \tilde{q} \mid \forall i \in [n]\ \exists q'_i \in \tilde{q_i}, \exists q' \in \tilde{q}$ such that $f(q'_1,\ldots,q'_n)[c] \rightarrow q' \in \mathcal{R}\}$.

We prove now that $\mathcal{A}_m$ is a normalized-complete $REC_{\neq}$ automaton (Lemma 11) and that $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}_m)$ (Lemma 12).

Lemma 11. *$\mathcal{A}_m$ is a normalized-complete $REC_{\neq}$ automaton.*

Proof. First we prove that $\mathcal{A}_m$ is deterministic. Let $f(\tilde{q_1},\ldots,\tilde{q_n})[c] \rightarrow \tilde{q} \in \mathcal{R}_m$ and $f(\tilde{q_1},\ldots,\tilde{q_n})[c] \rightarrow \tilde{s} \in \mathcal{R}_m$. According to the definition of $\mathcal{R}_m$:

- $\forall i \in [n]$, $\exists q'_i \in \tilde{q_i}, \exists q' \in \tilde{q}$ such that $f(q'_1,\ldots,q'_n)[c] \rightarrow q' \in \mathcal{R}$.
- $\forall i \in [n]$, $\exists q''_i \in \tilde{q_i}, \exists s' \in \tilde{s}$ such that $f(q''_1,\ldots,q''_n)[c] \rightarrow s' \in \mathcal{R}$.

$\forall i \in [n]\ q''_i \in \tilde{q'_i}$ hence according to Lemma 8, $s' \in \tilde{q'}$. Then $\tilde{q} = \tilde{s}$ since $\tilde{q} = \tilde{q'}$, $\tilde{s} = \tilde{s'}$ and $q' = \tilde{s'}$. Finally $\mathcal{A}_m$ is deterministic. Let us now prove that $\mathcal{A}_m$ is normalized-complete. Let $f \in \Sigma_n$, $\tilde{q_1},\ldots,\tilde{q_n} \in Q_m$ and $c \in CE'_n$ such that $(\tilde{q_i})_{i\in[n]}$ satisfies the equality contraints of c.
Let $(q'_i)_{i\in[n]}$ such that $\forall i \in [n]\ q'_i \in \tilde{q_i}$ and $\forall k,l \in [n]\ (c \Rightarrow (x_k = x_l)) \Rightarrow (q'_k = q'_l)$. The last condition is possible since $\forall k,l \in [n]\ (c \Rightarrow (x_k = x_l)) \Rightarrow (\tilde{q_k} = \tilde{q_l})$ and $(\tilde{q_i})_{i\in[n]}$ satisfies the equality contraints of c. $(q'_i)_{i\in[n]}$ satisfies the equality contraints of c and $\mathcal{A}$ is complete hence $\exists f(q'_1,\ldots,q'_n)[c] \rightarrow q \in \mathcal{R}$. Hence $f(\tilde{q'_1},\ldots,\tilde{q'_n})[c] \rightarrow \tilde{q} \in \mathcal{R}_m$ according to the EQUIV algorithm. Moreover $\forall i \in [n]$, we have $\tilde{q_i} = \tilde{q'_i}$ hence $f(\tilde{q_1},\ldots,\tilde{q_n})[c] \rightarrow \tilde{q} \in \mathcal{R}_m$.
Finally, we deduce $\mathcal{A}_m$ is a normalized-complete $REC_{\neq}$ automaton which ends the proof of Lemma 11.

Lemma 12. *$\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}_m)$.*

Proof. First we can prove by induction on the height of t that $\forall t \in T_\Sigma, \forall q \in Q, (t \xrightarrow{*}_{\mathcal{A}} q) \Rightarrow (t \xrightarrow{*}_{\mathcal{A}_m} \tilde{q})$. We deduce that $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{A}_m)$.
Then we deduce $\mathcal{L}(\mathcal{A}_m) \subseteq \mathcal{L}(\mathcal{A})$ from the property (2) and the following property:

$$\forall t \in T_\Sigma, \forall q \in Q, (t \xrightarrow{*}_{\mathcal{A}_m} \tilde{q}) \Rightarrow (\exists q' \in \tilde{q} \text{ such that } t \xrightarrow{*}_{\mathcal{A}} q').$$

We deduce that $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}_m)$ which ends the proof of Lemma 12.

Remark 13. We can prove easily that $\forall q \in Q_m, \mathcal{L}_{\mathcal{A}_m}(q)$ is infinite and that $\forall q, q' \in Q_m, (q \equiv_{\mathcal{A}_m} q') \Leftrightarrow (q = q')$.

4.4 Characterization

Let $\mathcal{A}$ be a normalized-complete $REC_{\neq}$ automaton. According to the Section 4.3, we can consider automata satisfying properties of Remark 13. We give now a necessary and sufficient condition for the language recognized by $\mathcal{A}$ to be recognizable.

Proposition 14. *Let $\mathcal{A} = (\Sigma, Q, F, \mathcal{R})$ be a normalized-complete $REC_{\neq}$ automaton such that for each state q of $\mathcal{A}$, $\mathcal{L}_{\mathcal{A}}(q)$ is infinite and such that $\forall q, q' \in Q$, $(q \equiv_{\mathcal{A}} q') \Leftrightarrow (q = q')$. Then $\mathcal{L}(\mathcal{A})$ is recognizable if and only if for all rules $f(q_1, \ldots, q_n)[c] \to q$, $f(q_1, \ldots, q_n)[c'] \to q'$ of $\mathcal{R}$, we have $q = q'$.*

In order to prove Proposition 14, we need some technical lemmas. First, since the language recognized by each state of $\mathcal{A}$ is infinite, we prove that we can "instantiate" each constrained term to a ground term. In fact we prove (Definition 15 and Lemma 16) that we can associate with each constrained term over $\Sigma \cup Q$ a constrained term over Σ without occurence of x by replacing each occurence of a state q by an element of $\mathcal{L}_{\mathcal{A}}(q)$ and each occurence of x by an element of an infinite set of ground terms.

Definition 15. Let C be a constrained term. We denote:

- $\mathcal{V}(C)$ the set of variable positions of C: $\mathcal{V}(C) = \{p \in \mathcal{P}os(C) \mid C(p) = x\}$.
- $\mathcal{S}(C)$ the set of state positions of C: $\mathcal{S}(C) = \{p \in \mathcal{P}os(C) \mid C(p) \in Q\}$.
- For each $q \in Q$, $\mathcal{S}(C)(q) = \{p \in \mathcal{S}(C) \mid C(p) = q\}$.

Lemma 16. *Let C be a constrained term over $\Sigma \cup Q$ and T be an infinite set of terms of T_Σ. There exists a constrained term C' over Σ without occurence of x such that:*

- $\forall p \in \mathcal{P}os(C) \setminus (\mathcal{V}(C) \cup \mathcal{S}(C))$, $C'(p) = C(p)$,
- *Each variable of C is replaced by a constrained term associated with an element of T i.e. $\forall p \in \mathcal{V}(C), \exists t \in T$, $C'|_p = \mathrm{lab}_t$,*
- *Each state of C is replaced by a constrained term associated with an element of the language recognized by the state i.e. $\forall q \in Q, \forall p \in \mathcal{S}(C)(q), \exists t \in \mathcal{L}_{\mathcal{A}}(q), C'|_p = \mathrm{lab}_t$,*

where lab_t denotes for each term t the constrained term over Σ obtained from t, i.e. $\forall p \in \mathcal{P}os(t)$, if $t(p) = f \in \Sigma_n$, $n > 0$, then $\mathrm{lab}_t(p) = f_c$ with c the full constraint satisfied by $(t|_{pi})_{i \in [n]}$, else $\mathrm{lab}_t(p) = t(p)$.

Proof. Let C be a constrained term over $\Sigma \cup Q$ and T be an infinite set of terms of T_Σ. First let us deduce from the full constraint expressions of each position of C, full constraint expressions between the positions of C where the variable x occurs and between the positions of C where the same state occurs.
In fact if we consider the positions where the variable x occurs (positions of $\mathcal{V}(C)$), we express all the equalities between these positions imposed by the constraints of C. When none equality is imposed between two positions we impose a disequality since :

- Constraints impose only equalities between brothers hence between terms whose positions have the same length.
- According to the definition of constrained terms, equalities are only imposed between equal terms in a constrained term.

We can do the same for positions of $\mathcal{S}(C)(q)$ for each q of Q.
More formally, for each position p of C such that $C(p) \in \Sigma'$, we denote $\mathrm{cont}_C(p)$ the constraint obtained by projection from Σ' onto CE'_n and we define $\forall p \in \mathcal{S}(C) \cup \mathcal{V}(C)$ a variable z_p. We denote $c_{\mathcal{V}(C)}$ the full constraint expression over $(z_p)_{p \in \mathcal{V}(C)}$ and $\forall q \in Q, c_{\mathcal{S}(C)(q)}$ the full constraint expression over $(z_p)_{p \in \mathcal{S}(C)(q)}$ defined as follows:

1. We express the equalities imposed by the constraints:
   $\forall p \in \mathcal{P}os(C)$, $\forall i,j$ $(\mathrm{cont}_C(p) \Rightarrow (x_i = x_j)) \Rightarrow \forall \gamma$ such that $z_{pi\gamma}$ defined
   $$\begin{pmatrix} z_{pi\gamma} \in \mathcal{V}(C) \Rightarrow (z_{pi\gamma} = z_{pj\gamma}) \in c_{\mathcal{V}(C)} \\ z_{pi\gamma} \in \mathcal{S}(C)(q), q \in Q \Rightarrow (z_{pi\gamma} = z_{pj\gamma}) \in c_{\mathcal{S}(C)(q)} \end{pmatrix}$$
2. We apply the transitive closure to express all equalities:
   $(z_{p_1} = z_{p_2} \wedge z_{p_2} = z_{p_3}) \in c_{\mathcal{V}(C)} \Rightarrow (z_{p_1} = z_{p_3}) \in c_{\mathcal{V}(C)}$.
   $\forall q \in Q$, $(z_{p_1} = z_{p_2} \wedge z_{p_2} = z_{p_3}) \in c_{\mathcal{S}(C)(q)} \Rightarrow (z_{p_1} = z_{p_3}) \in c_{\mathcal{S}(C)(q)}$.
3. $\forall p, p' \in \mathcal{V}(C), p \neq p', (z_p = z_{p'}) \notin c_{\mathcal{V}(C)} \Rightarrow (z_p \neq z_{p'}) \in c_{\mathcal{V}(C)}$;
4. $\forall q \in Q, \forall p, p' \in \mathcal{S}(C)(q), p \neq p', (z_p = z_{p'}) \notin c_{\mathcal{S}(C)(q)} \Rightarrow (z_p \neq z_{p'}) \in c_{\mathcal{S}(C)(q)}$.

Since T is infinite and $\forall q \in Q$, $\mathcal{L}_\mathcal{A}(q)$ is infinite, there exists $(t_p)_{p \in \mathcal{V}(C)} \in T$ and $\forall q \in Q$, $(t_p)_{p \in \mathcal{S}(C)(q)} \in \mathcal{L}_\mathcal{A}(q)$ such that:

1. $\forall p \in \mathcal{V}(C) \cup \mathcal{S}(C)$, t_p is of height strictly greater than height of C and strictly greater than height of terms of the set $\{t_{p'} \mid p' \in \mathcal{V}(C) \cup \mathcal{S}(C)$, length of p' strictly less than length of $p\}$.
2. $\forall p \in \mathcal{V}(C), \forall p' \in \mathcal{S}(C)$, $t_p \neq t_{p'}$.
3. $\forall q, q', q \neq q', \forall p \in \mathcal{S}(C)(q), \forall p' \in \mathcal{S}(C)(q')$, $t_p \neq t_{p'}$.
4. $(t_p)_{p \in \mathcal{V}(C)}$ satisfies $c_{\mathcal{V}(C)}$.
5. $\forall q \in Q, (t_p)_{p \in \mathcal{S}(C)(q)}$ satisfies $c_{\mathcal{S}(C)(q)}$.

Let us remark that point point3 is satisfied for all families of terms since $\mathcal{A}$ is deterministic. Let C' be the term of $T_{\Sigma'}$ defined as follows:

- $\forall p \in \mathcal{P}os(C) \setminus (\mathcal{V}(C) \cup \mathcal{S}(C))$ $C'(p) = C(p)$;
- $\forall p \in \mathcal{V}(C) \cup \mathcal{S}(C)$ $C'|_p = \mathrm{lab}_{t_p}$.

For each $p \in \mathcal{V}(C)$, $p' \in \mathcal{S}(C)$, constraints of C impose $z_p \neq z_{p'}$ since $C(p) \neq C(p')$. This constraint is satisfied by lab_{t_p} and $\text{lab}_{t_{p'}}$ according to previous points 1 and 2. Similarly for each $p \in \mathcal{S}(C)(q)$, $p' \in \mathcal{S}(C)(q')$, $q \neq q'$, constraints of C impose $z_p \neq z_{p'}$. This constraint is satisfied by lab_{t_p} and $\text{lab}_{t_{p'}}$ according to previous points 1 and 3. We deduce that C' is a constrained term over Σ without occurence of x which ends the proof of Lemma 16.

Let us now prove that we can "instantiate" each constrained term over $\Sigma \cup Q$ to a constrained term over Σ by replacing each occurence of a state q by an element of $\mathcal{L}_{\mathcal{A}}(q)$ (Definition 17 and Lemma 18); similarly, given an infinite set of ground term T, we can "instantiate" each constrained term over Σ by replacing each occurence of x by a constrained term associated with an element of T (Lemma 19).

Definition 17. Let C be a constrained term over $\Sigma \cup Q$. A state-instance of C is a constrained term obtained from C, replacing each state q by a constrained term $\text{lab}_t, t \in \mathcal{L}_{\mathcal{A}}(q)$.

Lemma 18. *There exists a state-instance of each constrained term.*

Proof. Let C be a constrained term and C' be a constrained term obtained from C according to Lemma 16. Let C'' be the constrained term defined by

- $\forall p \in \mathcal{P}os(C) \setminus \mathcal{V}(C)$, $C''(p) = C'(p)$;
- $\forall p \in \mathcal{V}(C)$, $C''|_p = x$.

C" is obviously a state-instance of C which ends the proof of Lemma 18.

Let us remark that when C' is a state-instance of a constrained term C, then $\forall q \in Q, (C[q] \xrightarrow{*}_{\mathcal{A}} s \Rightarrow C'[q] \xrightarrow{*}_{\mathcal{A}} s)$.

Lemma 19. *Let C be a constrained term over Σ and T be an infinite set of terms of T_Σ. There exists $(t_p)_{p \in \mathcal{V}(C)} \in T$ such that C' defined by*

- $\forall p \in \mathcal{P}os(C) \setminus \mathcal{V}(C), C'(p) = C(p)$;
- $\forall p \in \mathcal{V}(C), C'_{|p} = lab_{t_p}$,

is a constrained term.

This lemma is an immediate corrolary of Lemma 16. Let us now prove that the condition of Proposition 14 is necessary.

Lemma 20. *Let us suppose that there exists two rules of $\mathcal{R}$, $f(q_1, \ldots, q_n)[c] \to q$ and $f(q_1, \ldots, q_n)[c'] \to q'$ such that $c \neq c'$ and $q \neq q'$. Then $\mathcal{L}(\mathcal{A})$ is not recognizable.*

Proof. Let us suppose that $\mathcal{L}(\mathcal{A})$ is a regular tree language: there exists $\mathcal{B} = (\Sigma, Q, F, \Delta)$ a deterministic and complete bottom-up tree automaton recognizing it. For each $q \in Q$, we denote $\mathcal{L}_{\mathcal{B}}(q)$ the set of terms t of T_Σ such that $t \xrightarrow{*}_{\mathcal{B}} q(t)$. Let us recall the following basic property:

Property 21. $\forall C \in \mathcal{C}^n(\Sigma), \forall q \in Q, \forall (t_i)_{i\in[n]} \in \mathcal{L}_{\mathcal{B}}(q), \forall (t'_i)_{i\in[n]} \in \mathcal{L}_{\mathcal{B}}(q)$

$$(C[t_1,\ldots,t_n] \in \mathcal{L}(\mathcal{B}) \Leftrightarrow C[t'_1,\ldots,t'_n] \in \mathcal{L}(\mathcal{B})).$$

The sketch of proof is the following: we construct two terms, saying t_1 and t_2 such that t_1 belongs to $\mathcal{L}(\mathcal{A})$ and t_2 does not. Furthermore, t_1 and t_2 will differ only on some positions p where $t_1(p) = t_2(p) = f$ but subterms at these positions in t_1 satisfy the constraint c while in t_2, subterms at the same positions satisfy the constraint c'.
From t_1 and t_2 we deduce a general context C_g, intuitively the common prefix of t_1 and t_2, such that there exists $q_{\mathcal{B}}$ state of $\mathcal{B}$, $(u_i)_{i\in[n]}$ and $(u'_i)_{i\in[n]}$ terms of $\mathcal{L}_{\mathcal{B}}(q_{\mathcal{B}})$, such that $C_g[(u_p)] \in \mathcal{L}(\mathcal{A})$ and $C_g[(u'_p)] \notin \mathcal{L}(\mathcal{A})$. This will contradict Property 21 since we supposed that $\mathcal{L}(\mathcal{A})$ is recognizable.

Since $\mathcal{A}$ is complete, we can suppose without loss of generality that c and c' differ only by the splitting of a set, i.e. $\exists (E_k)_{k\in K}, I, J \subseteq [n]$ such that:

$$c = ((E_k)_{k\in K}, I \cup J),\ card(c) = k+1;$$
$$c' = ((E_k)_{k\in K}, I, J),\ card(c') = k+2.$$

$q \neq q'$ hence $q \not\equiv_{\mathcal{A}} q'$. We deduce that there exists a constrained term C over $\Sigma \cup Q$ such that $(C[q] \xrightarrow{*}_{\mathcal{A}} s \in F \Leftrightarrow C[q'] \xrightarrow{*}_{\mathcal{A}} s' \notin F)$. We stand that $s \in F$ and according to Lemma 18, there exists $\bar{C}$ a state-instance of C. $\bar{C}[q] \xrightarrow{*}_{\mathcal{A}} s$ since $C[q] \xrightarrow{*}_{\mathcal{A}} s$ and $\bar{C}[q'] \xrightarrow{*}_{\mathcal{A}} s'$ since $C[q'] \xrightarrow{*}_{\mathcal{A}} s'$.

Let us consider the constrained term $F_1 = f_c(s_1,\ldots,s_n)$ where $\forall k \in I \cup J, s_k = x$ and $\forall k \notin I \cup J, s_k = q_k$. Lemma 18 ensures the existence of a state-instance F'_1 of F_1. Then $F'_1[q_I] \xrightarrow{*}_{\mathcal{A}} q$ since $F_1[q_I] \xrightarrow{*}_{\mathcal{A}} q$.
Let C_1 be the constrained term $\bar{C}[F'_1]$ and q_I be the unique state present in the rule r at positions belonging to $I \cup J$. The run on $C_1[q_I]$ leads to the final state s since $C_1[q_I] = \bar{C}[F'_1[q_I]] \xrightarrow{*}_{\mathcal{A}} \bar{C}[q] \xrightarrow{*}_{\mathcal{A}} s$.

The constrained term F_2 is obtained from F_1 by replacing the root symbol f_c by $f_{c'}$. Hence, F_2 and F_1 have the same projection onto $T_{\Sigma}(\{x\})$. From Lemma 18 there exists F'_2 a state-instance of F_2. F'_2 is choosen in such a way that root subterms at the same position $k \notin I \cup J$ in F'_1 and F'_2 are identical (remember that c and c' only differ by the splitting of $I \cup J$ into I and J). Then $F'_2[q_I] \xrightarrow{*}_{\mathcal{A}} q'$ since $F_2[q_I] \xrightarrow{*}_{\mathcal{A}} q'$.
In the same way as previously, C_2 denotes $\bar{C}[F'_2]$. Let us notice that F'_1 (resp. C_1) and F'_2 (resp. C_2) have the same projection onto $T_{\Sigma}(\{x\})$. The run on $C_2[q_I]$ leads to the non final state s' since $C_2[q_I] = \bar{C}[F'_2[q_I]] \xrightarrow{*}_{\mathcal{A}} \bar{C}[q'] \xrightarrow{*}_{\mathcal{A}} s'$.

As we supposed that $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{B})$ and as $\mathcal{L}_{\mathcal{A}}(q_I)$ is infinite then there exists $q_{\mathcal{B}}$ state of $\mathcal{B}$ such that the set $T = \mathcal{L}_{\mathcal{A}}(q_I) \cap \mathcal{L}_{\mathcal{B}}(q_{\mathcal{B}})$ is infinite.
As T is infinite, and according to Lemma 19, there exist terms $(u_p)_{p\in\mathcal{V}(C_1)} \in T$ such that C'_1 defined by

- $\forall p \in \mathcal{P}os(C_1) \setminus \mathcal{V}(C_1), C'_1(p) = C_1(p)$;
- $\forall p \in \mathcal{V}(C_1), C'_1|_p = lab_{u_p}$,

is a constrained term. As $\forall p, lab_{u_p} \xrightarrow{*}_{\mathcal{A}} q_I$, the run of this constrained term is the final state s. The term t_1, projection of C'_1 onto T_Σ satisfies $t_1 \in \mathcal{L}(\mathcal{A})$.
In the same way, there exist terms $(u'_p)_{p \in \mathcal{V}(C_2)} \in T$ such that C'_2 defined by

- $\forall p \in \mathcal{P}os(C_2) \setminus \mathcal{V}(C_2), C'_2(p) = C_2(p)$;
- $\forall p \in \mathcal{V}(C_2), C'_2|_p = lab_{u'_p}$,

is a constrained term and the run of C'_2 is the non final state s'. As $\mathcal{A}$ is deterministic, t_2, the projection of C'_2 over T_Σ does not belong to $\mathcal{L}(\mathcal{A})$.

Let C_g be the projection of C_1 onto $T_\Sigma(\{x\})$ (which is the same as the projection of C_2). C_g is a context -without labels- over a single variable x.
We replace each occurence of x in C_g by distinct new variables: it results a context C'_g over distinct new variables $(x_p)_{p \in \mathcal{V}(C_g)}$ defined by

- $\forall p \in \mathcal{P}os(C_g) \setminus \mathcal{V}(C_g), C'_g(p) = C_g(p)$;
- $\forall p \in \mathcal{V}(C_g), C'_g(p) = x_p$.

We can prove that $t_1 \in \mathcal{L}(\mathcal{A}) = C_g[(u_p)]$ and $t_2 = C_g[(u'_p)]$. Moreover, $\forall p, u_p \xrightarrow{*}_{\mathcal{B}} q_{\mathcal{B}}$ and $u'_p \xrightarrow{*}_{\mathcal{B}} q_{\mathcal{B}}$, which contradicts the Property 21 since $t_1 \in \mathcal{A}$ and $t_2 \notin \mathcal{A}$. We deduce that $\mathcal{L}(\mathcal{A})$ is not recognizable, which ends the proof of Lemma 20.

Let us now prove that the condition of Proposition 14 is sufficient.

Lemma 22. *Let us suppose that for all rules of $\mathcal{R}$, $f(q_1, \ldots, q_n)[c] \to q$ and $f(q_1, \ldots, q_n)[c'] \to q'$, we have $q = q'$. Then $\mathcal{L}(\mathcal{A})$ is recognizable and we can compute a tree automaton recognizing $\mathcal{L}(\mathcal{A})$.*

Proof. Let $\mathcal{B} = (\Sigma, Q, F, \Delta)$ be the tree automaton whose set of rules Δ is defined by: $\forall f \in \Sigma_n, \forall (q_i)_{i \in [n]} \in Q$, $f(q_1, \ldots, q_n) \to q \in \Delta$ where q is defined by a rule $f(q_1, \ldots, q_n)[c] \to q$ of $\mathcal{R}$ (q is unique according to hypothesis of the lemma). We easily prove that $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{B})$. Hence $\mathcal{L}(\mathcal{A})$ is recognizable which ends the proofs of Lemma 22 and of Proposition 14.

Let $\mathcal{A} = (\Sigma, Q, F, \mathcal{R})$ be a normalized-complete $REC_{\neq}$ automaton such that for each state q of $\mathcal{A}$, $\mathcal{L}_{\mathcal{A}}(q)$ is infinite. According to Remark 13 and Proposition 14, we deduce that the recognizability problem of $\mathcal{L}(\mathcal{A})$ is decidable. Finally, according to Section 4.1, we deduce the following theorem:

Theorem 23. *The recognizability problem in the class $REC_{\neq}$ is decidable.*

5 Conclusion

We proved here that recognizability problem is decidable in the class $REC_{\neq}$. It implies e.g. the decidability of recognizability of $\Phi(\mathcal{L})$ where Φ is a quasi-algebraic tree homomorphism (i.e. variables occurr at depth one in a letter's image) and $\mathcal{L}$ a recognizable language.
It provides also a rather simple algorithm for testing recognizability of the set of normal forms (resp. of the set of direct descendants of a recognizable language) for some subclasses of rewrite systems (like shallow ones).

Furthermore, the notions we define here -like constrained terms- could perhaps be extended and help to answer the two following open problems:
Is recognizability decidable in the class of reduction automata?
Can we decide whether the homomorphic image of a recognizable tree language is recognizable?

References

1. A. Arnold. Le théorème de transversale rationnelle dans les arbres. *Mathematical System Theory*, 13:275–282, 1980.
2. B. Bogaert, F. Seynhaeve, and S. Tison. The recognizability problem for tree automata with comparisons between brothers. Technical Report IT. 317, Laboratoire d'Informatique Fondamentale de Lille, Nov. 1998.
3. B. Bogaert and S. Tison. Equality and disequality constraints on direct subterms in tree automata. In P. Enjalbert, A. Finkel, and K. W. Wagner, editors, *9th Annual Symposium on Theoretical Aspects of Computer Science*, volume 577 of *Lecture Notes in Computer Science*, pages 161–171, 1992.
4. A. Caron, H. Comon, J. Coquidé, M. Dauchet, and F. Jacquemard. Pumping, cleaning and symbolic constraints solving. In *Proceedings, International Colloquium Automata Languages and Programming*, volume 820 of *Lecture Notes in Computer Science*, pages 436–449, 1994.
5. A. Caron, J. Coquidé, and M. Dauchet. Encompassment properties and automata with constraints. In C. Kirchner, editor, *Proceedings. Fifth International Conference on Rewriting Techniques and Applications*, volume 690 of *Lecture Notes in Computer Science*, pages 328–342, 1993.
6. H. Comon, M. Dauchet, R. Gilleron, S. Tison, and M. Tommasi. Tree automata techniques and applications, 1998. available through WWW from http://l3ux02.univ-lille3.fr/tata.
7. M. Dauchet, A.-C. Caron, and J.-L. Coquidé. Reduction properties and automata with constraints. *Journal of Symbolic Computation*, 20:215–233, 1995.
8. M. Dauchet and S. Tison. Réduction de la non-linéarite des morphismes d'arbres. Technical Report IT-196, Laboratoire d'Informatique Fondamentale de Lille, Université des Sciences et Technologies de Lille, Villeneuve d'Ascq, France, 1990.
9. F. Gécseg and M. Steinby. *Tree Automata*. Akademiai Kiado, 1984.
10. T. Genet. Decidable approximations of sets of descendants and sets of normal forms. volume 1379 of *Lecture Notes in Computer Science*, pages 151–165, Tsukuba, 1998.
11. J.-P. Jouannaud and E. Kounalis. Automatic proofs by induction in theories without constructors. *Information and Computation*, 82(1):1–33, July 1989.
12. G. Kucherov and M. Tajine. Decidability of regularity and related properties of ground normal form languages. In *Proceedings of 3rd International Workshop on Conditional Rewriting Systems*, pages 150–156, 1992.
13. J. Mongy. *Transformation de noyaux reconnaissables d'arbres. Forêts RATEG*. PhD thesis, Laboratoire d'Informatique Fondamentale de Lille, Université des Sciences et Technologies de Lille, Villeneuve d'Ascq, France, 1981.
14. S. Vágvölgyi and R. Gilleron. For a rewrite system it is decidable whether the set of irreducible ground terms is recognizable. *Bulletin of the European Association of Theoretical Computer Science*, 48:197–209, Oct. 1992.
15. J. Waldmann. Normalization of s-terms is decidable. volume 1379 of *Lecture Notes in Computer Science*, pages 138–150, Tsukuba, 1998.

A Theory of "May" Testing for Asynchronous Languages

Michele Boreale[1] Rocco De Nicola[2] Rosario Pugliese[2]

[1]Dipartimento di Scienze dell'Informazione, Università di Roma "La Sapienza"
[2]Dipartimento di Sistemi e Informatica, Università di Firenze

Abstract. Asynchronous communication mechanisms are usually a basic ingredient of distributed systems and protocols. For these systems, asynchronous may-based testing seems to be exactly what is needed to capture safety and certain security properties. We study may testing equivalence focusing on the asynchronous versions of CCS and π-calculus. We start from an operational testing preorder and provide finitary and fully abstract trace-based interpretations for it, together with complete inequational axiomatizations. The results throw light on the differences between synchronous and asynchronous systems and on the weaker testing power of asynchronous observations.

1 Introduction

Distributed systems often rely on asynchronous communication primitives for exchanging information. Many properties of these systems can be conveniently expressed and verified by means of behavioural equivalences. In particular, *may* testing [11] seems to be exactly what is needed for reasoning about safety properties. In this respect, an assumption of asynchrony can often play a crucial role.

As an example, consider a trivial communication protocol with two users A and B sharing a private channel c. The protocol requires that A uses c to send a bit of information m to B, then B receives two messages on channels a and b, finally B sends, on channel d, the message received on a. The ordering of the inputs on a and b depends on the message received on c. In π-calculus we can formulate this protocol as follows (the meaning of the various operators is the usual one; in particular, $(\nu\, c\,)$ stands for creation of a local channel c):

$$\begin{aligned} A &= \overline{c}m \\ B &= c(x).([x=0]a(y).b(z).\overline{d}y + [x=1]b(z).a(y).\overline{d}y) \\ S &= (\nu\, c\,)(A \mid B) \end{aligned}$$

Secrecy, i.e. the ability to keep a datum secret, is an important property which one might want to check of this protocol: externally, it should not be possible to guess message m from the behaviour of the whole system S. Following [2], this property can be formalized by requiring that the behaviour of the protocol should not depend on the bit that A sends to B: in other words, processes $S[0/m]$ and $S[1/m]$ should be equivalent. The intended equivalence is here the

W. Thomas (Ed.): FOSSACS'99, LNCS 1578, pp. 165-179, 1999.

one induced by may testing: a process may pass a 'test' performed by an external observer if and only if the other process may. If one interprets 'passing a test' as 'revealing a piece of information', then equivalent processes *may* reveal externally the same information. Now, it is easy to see that an observer could tell $S[0/m]$ and $S[1/m]$ apart via *synchronous* communication on a and b (*traffic analysis*). However, $S[0/m]$ and $S[1/m]$ *are* equivalent in a truly asynchronous scenario, in which no ordering on the arrival of outgoing messages is guaranteed.

It is therefore important to have a full understanding of may-semantics in an asynchronous setting. We shall consider asynchronous variants of CCS and π-calculus: in these models, the communication medium can be understood as a *bag* of output actions (messages), waiting to be consumed by corresponding input actions. This is reminiscent of the Linda approach [13]. In [7], we have provided an observers-independent characterization of the asynchronous testing preorders. Here, we use this characterization as a starting point for defining a "finitary" trace-based model and a complete axiomatization for the may testing preorder.

When modelling asynchronous processes, the main source of complications is the non-blocking nature of output primitives. It is demanded that processes be *receptive*, i.e. that they be able to receive all messages sent by the environment at any time. A simple approach to this problem leads to models where all possible inputs (i.e. outputs from the environment) at any stage are explicitly described. As a result, infinitary descriptions are obtained even for simple, non–recursive, processes. For example, according to [16], the operational description of the null process $\mathbf{0}$ is the same as that of $recX.a.(\overline{a} \mid X)$, where a stands for any input action, $\overline{a}$ is its complementary output and rec is the recursion operator. Similarly, [5] presents a trace-based model that permits arbitrary "gaps" in traces to take into account any external influence on processes behaviour.

Differently from [16], we build on the usual operational semantics of the language, which just describes what the process intentions are at any stage, and we take advantage of a preorder, $\preceq$, between sequences of actions (traces). The intuition behind $\preceq$ is that whenever a trace s may lead to a successful interaction with the environment and $s' \preceq s$, then s' may lead to success as well. It turns out that, when comparing two processes, only their "minimal" traces need to be taken into account. This leads to a model that assigns finite denotations to finite processes. More precisely, the interpretation of the may preorder ($\sqsubseteq_{\approx_m}$) suggested by the model is as follows: $P \sqsubseteq_{\approx_m} Q$ if, consuming the same messages, Q can produce at least the same messages as P.

Building on the above mentioned preorder over traces, we provide a complete (in-)equational axiomatization for asynchronous CCS that relies on the laws:

$$\texttt{(A1)} \quad a.b.P \sqsubseteq b.a.P \qquad \text{and} \qquad \texttt{(A2)} \quad a.(\overline{a} \mid P) \sqsubseteq P\,.$$

These two laws are specific to asynchronous testing and are not sound for the synchronous may preorder [11]. The completeness proof relies on the existence of canonical forms directly inspired by the finitary trace–based model.

We develop both the model and the axiomatization first for asynchronous CCS, and then for asynchronous π-calculus. The simpler calculus is sufficient to

isolate the key issues of asynchrony. Indeed, both the trace interpretation and the axiomatization for π-calculus are dictated by those for CCS.

The rest of the paper is organized as follows. Section 2 introduces asynchronous CCS and the may–testing preorder. Section 3 and 4 present a fully abstract trace–based interpretation of processes and a complete proof system for finite processes, respectively. In Section 5 the results of the previous sections are extended to π-calculus. The final section contains a few concluding remarks and a brief discussion of related work.

2 Asynchronous CCS

In this section we present syntax, operational and testing semantics of asynchronous CCS (ACCS, for short) [7]. It differs from standard CCS because only guarded choices are used and output guards are not allowed. The absence of output guards "forces" asynchrony; it is not possible to define processes that causally depend on output actions.

Syntax We let $\mathcal{N}$, ranged over by $a, b, \ldots$, be an infinite set of *names* used to model input actions and $\overline{\mathcal{N}} = \{\overline{a} \mid a \in \mathcal{N}\}$, ranged over by $\overline{a}, \overline{b}, \ldots$, be the set of *co–names* that model outputs. $\mathcal{N}$ and $\overline{\mathcal{N}}$ are disjoint and are in bijection via the *complementation* function $(\overline{\cdot})$; we define: $\overline{(\overline{a})} = a$. We let $\mathcal{L} = \mathcal{N} \cup \overline{\mathcal{N}}$ be the set of *visible actions*, and let $l, l', \ldots$ range over it. We let $\mathcal{L}_\tau = \mathcal{L} \cup \{\tau\}$, where τ is a distinct action, for the set of all *actions* or *labels*, ranged over by μ. We shall use $A, B, L, \ldots$, to range over subsets of $\mathcal{L}$. We let $\mathcal{X}$, ranged over by $X, Y, \ldots$, be a countable set of *process variables*.

Definition 1. The set of *ACCS terms* is generated by the grammar:

$$E ::= \overline{a} \;\mid\; \textstyle\sum_{i\in I} g_i.E_i \;\mid\; E_1 \mid E_2 \;\mid\; E\backslash L \;\mid\; E\{f\} \;\mid\; X \;\mid\; recX.E$$

where $g_i \in \mathcal{N} \cup \{\tau\}$, and $f : \mathcal{N} \to \mathcal{N}$, called *relabelling function*, is injective and such that $\{l \mid f(l) \neq l\}$ is finite. We extend f to $\mathcal{L}$ by letting $f(\overline{a}) = \overline{f(a)}$. We let $\mathcal{P}$, ranged over by P, Q, etc., denote the set of *closed* and *guarded* terms or *processes* (i.e. those terms where every occurrence of any agent variable X lies within the scope of some $recX.\_$ and $\sum$ operators).

In the sequel, $\sum_{i\in\{1,2\}} g_i.E_i$ will be abbreviated as $g_1.E_1 + g_2.E_2$, $\sum_{i\in\{1\}} g_i.E_i$ as $g_1.E_1$ and $\sum_{i\in\emptyset} g_i.E_i$ as $\mathbf{0}$; we will also write g for $g.\mathbf{0}$. As usual, we write $E[F/X]$ for the term obtained by replacing each free occurrence of X in E by F (with possible renaming of bound process variables). We write $\mathrm{n}(P)$ to denote the set of visible actions occurring in P.

Operational Semantics The labelled transition system $(\mathcal{P}, \mathcal{L}_\tau, \xrightarrow{\mu})$ in Figure 1 defines the operational semantics of the language.

As usual, we use $\Longrightarrow$ or $\stackrel{\epsilon}{\Longrightarrow}$ to denote the reflexive and transitive closure of $\stackrel{\tau}{\longrightarrow}$ and use $\stackrel{s}{\Longrightarrow}$ (resp. $\stackrel{s}{\longrightarrow}$) for $\Longrightarrow \stackrel{l}{\longrightarrow} \stackrel{s'}{\Longrightarrow}$ (resp. $\stackrel{l}{\longrightarrow} \stackrel{s'}{\longrightarrow}$) when

$s = ls'$. Moreover, we write $P \stackrel{s}{\Longrightarrow}$ for $\exists P' : P \stackrel{s}{\Longrightarrow} P'$ ($P \stackrel{s}{\longrightarrow}$ and $P \stackrel{\tau}{\longrightarrow}$ will be used similarly). We will call *language* generated by P the set $L(P) = \{s \in \mathcal{L}^* \mid P \stackrel{s}{\Longrightarrow}\}$. We say that a process P is *stable* if $P \stackrel{\tau}{\not\longrightarrow}$.

$$\texttt{AR1}\ \sum_{i\in I} g_i.P_i \stackrel{g_j}{\longrightarrow} P_j \quad j \in I \qquad \texttt{AR2}\ \overline{a} \stackrel{\overline{a}}{\longrightarrow} \mathbf{0}$$

$$\texttt{AR3}\ \frac{P \stackrel{\mu}{\longrightarrow} P'}{P\{f\} \stackrel{f(\mu)}{\longrightarrow} P'\{f\}} \qquad \texttt{AR4}\ \frac{P \stackrel{\mu}{\longrightarrow} P'}{P\backslash L \stackrel{\mu}{\longrightarrow} P'\backslash L} \quad \text{if } \mu \notin L \cup \overline{L}$$

$$\texttt{AR5}\ \frac{P \stackrel{\mu}{\longrightarrow} P'}{P \mid Q \stackrel{\mu}{\longrightarrow} P' \mid Q} \qquad \texttt{AR6}\ \frac{P[recX.P/X] \stackrel{\mu}{\longrightarrow} P'}{recX.P \stackrel{\mu}{\longrightarrow} P'}$$

$$\texttt{AR7}\ \frac{P \stackrel{l}{\longrightarrow} P', \quad Q \stackrel{\overline{l}}{\longrightarrow} Q}{P \mid Q \stackrel{\tau}{\longrightarrow} P' \mid Q'}$$

Fig. 1. Operational semantics of ACCS (symmetric of rule `AR5` omitted)

May Semantics We are now ready to instantiate on ACCS the general framework of [11, 15] to obtain the may preorder and equivalence. In the sequel, *observers*, ranged over by O, are ACCS processes that can additionally perform a distinct *success* action ω.

Definition 2. $P \stackrel{\sqsubseteq}{\sim}_m Q$ iff for every observer O, $P \mid O \stackrel{\omega}{\Longrightarrow}$ implies $Q \mid O \stackrel{\omega}{\Longrightarrow}$.

We will use $\simeq_m$ to denote the equivalence obtained as the kernel of the preorder $\stackrel{\sqsubseteq}{\sim}_m$ (i.e. $\simeq_m = \stackrel{\sqsubseteq}{\sim}_m \cap \stackrel{\sqsubseteq}{\sim}_m^{-1}$).

Universal quantification on observers makes it difficult to work with the operational definition of the may preorder; an alternative characterization is on demand. In the synchronous case, this characterization is simply *trace inclusion* (see, e.g., [11, 15]). In [7], by taking advantage of a preorder over single traces, we proved that in case of asynchronous communication a weaker condition is required; we summarize these results below.

Definition 3. Let $\preceq$ be the least preorder over $\mathcal{L}^*$ preserved under trace composition and satisfying the following laws

$$\texttt{TO1}\quad \epsilon \preceq a \qquad \texttt{TO2}\quad la \preceq al \qquad \texttt{TO3}\quad \epsilon \preceq a\overline{a}$$

The intuition behind the the laws in Definition 3 is that, whenever a process interacts with its environment by performing a sequence of actions s, an interaction is possible also if the process performs any $s' \preceq s$. To put it differently, if the environment offers $\overline{s}$, then it also offers any $\overline{s'}$ s.t. $s' \preceq s$.

More specifically, law `TO1` (*deletion*) says that process inputs cannot be enforced. For example, we have $\overline{b}c \preceq a\overline{b}c$: if the environment offers the sequence $\overline{a}b\overline{c}$, then it also offers $b\overline{c}$, as there can be no causal dependence of $b\overline{c}$ upon the output $\overline{a}$. Law `TO2` (*postponement*) says that observations of process inputs can

be delayed. For example, we have that $\overline{b}ac \preceq a\overline{b}c$. Indeed, if the environment offers $\overline{a}b\overline{c}$ then it also offers $b\overline{ac}$. Finally, law TO3 (*annihilation*) allows the environment to internally consume of complementary actions, e.g. $\overline{b} \preceq a\overline{a}\overline{b}$. Indeed, if the environment offers $\overline{a}ab$ it can internally consume $\overline{a}$ and a and offer b.

Definition 4. For processes P and Q, we write $P \ll_m Q$ iff whenever $P \stackrel{s}{\Longrightarrow}$ then there exists s' such that $s' \preceq s$ and $Q \stackrel{s'}{\Longrightarrow}$.

Theorem 5. For all processes P and Q, $P \sqsubseteq_{\sim m} Q$ iff $P \ll_m Q$.

One can easily prove that $\sqsubseteq_{\sim m}$ is a pre–congruence; the proof relies on the coincidence between $\sqsubseteq_{\sim m}$ and $\ll_m$ (however, the case of parallel composition is best dealt with by relying on the definition of $\sqsubseteq_{\sim m}$).

3 A Finitary Trace-based Model

A fully abstract set-theoretic interpretation for $\sqsubseteq_{\sim m}$ can be obtained by interpreting each P as the set of traces $[\![P]\!]_m = \{s \mid \text{there is } s' \in L(P) : s' \preceq s\}$ and then ordering interpretations by set inclusion. However, this naive interpretation is not satisfactory, because it includes infinitely many traces even for finite processes; for instance, $[\![\mathbf{0}]\!]_m = \{\epsilon, a, a\overline{a}, a\overline{a}a, \ldots, b, b\overline{b}, \ldots\}$.

To obtain a non–redundant interpretation, we shall "minimize" the language of a process P, $L(P)$, w.r.t. the trace preorder $\preceq$. In the sequel, we use $[s]$ to denote the $\preceq$-equivalence class of s, i.e. the set $\{s' : s' \preceq s \text{ and } s \preceq s'\}$.

Definition 6.

- Consider a set D of $\preceq$-equivalence classes. We say that D is a *denotation* if whenever $[s], [s'] \in D$ and $s \preceq s'$ then $[s] = [s']$. We call $\mathcal{D}$ the set of all denotations.
- $\mathcal{D}$ is ordered by setting: $D_1 \leq D_2$ iff for each $[s] \in D_1$ there is $[s'] \in D_2$ such that $s' \preceq s$.

In words, a denotation D is a set of $\preceq$-equivalence classes which are *minimal* elements of D.

Lemma 7. $(\mathcal{D}, \leq)$ is a partial order.

Definition 8. For each P, we interpret P as the denotation

$$[\![P]\!]_m \stackrel{\text{def}}{=} \{[s] : s \in L(P) \text{ and for each } s' \in L(P) : s' \preceq s \text{ implies } [s] = [s']\}.$$

Example 1.

1. If $P \stackrel{\text{def}}{=} a.(\overline{a} \,|\, b)$, we have $L(P) = \{\epsilon, a, a\overline{a}, ab, a\overline{a}b, ab\overline{a}\}$ and that ϵ is minimal in $L(P)$ (by TO1–TO3), hence $[\![a.(\overline{a} \,|\, b)]\!]_m = [\![\mathbf{0}]\!]_m = \{[\epsilon]\}$.

2. If $P \stackrel{\text{def}}{=} \overline{a} \mid b.\overline{c}$, then $L(P) = \{\epsilon, \overline{a}, \overline{a}b, \overline{a}b\overline{c}, b, b\overline{a}, b\overline{a}\overline{c}, b\overline{c}, b\overline{c}\overline{a}\}$. The set of $\preceq$-minimal traces of $L(P)$ is $\{\epsilon, \overline{a}, b\overline{c}, \overline{a}b\overline{c}, b\overline{c}\overline{a}\}$ and $[\![P]\!]_m = \{[\epsilon], [\overline{a}], [b\overline{c}], [\overline{a}b\overline{c}], [b\overline{c}\overline{a}]\}$.
3. If $P \stackrel{\text{def}}{=} \overline{a} \mid b.\overline{c}$ and $Q \stackrel{\text{def}}{=} a.P$, then $[\![P]\!]_m = \{[\epsilon], [\overline{a}], [b\overline{c}], [\overline{a}b\overline{c}], [b\overline{c}\overline{a}]\}$ and $[\![Q]\!]_m = \{[\epsilon], [ab\overline{c}], [a\overline{a}b\overline{c}], [ab\overline{c}\overline{a}]\}$; hence $[\![Q]\!]_m \leq [\![P]\!]_m$.

Lemma 9. Let C be a non–empty set of $\preceq$-equivalence classes. Then C has minimal elements (w.r.t. the obvious ordering $[s'] \leq [s]$ iff $s' \preceq s$).

Theorem 10. $P \stackrel{\sqsubseteq}{\sim}_m Q$ if and only if $[\![P]\!]_m \leq [\![Q]\!]_m$ in $\mathcal{D}$.

Proof: We use the alternative characterization $\ll_m$ of $\stackrel{\sqsubseteq}{\sim}_m$. Suppose that $P \ll_m Q$; we show that $[\![P]\!]_m \leq [\![Q]\!]_m$ in $\mathcal{D}$. Let $[s] \in [\![P]\!]_m$, with $P \stackrel{s}{\Longrightarrow}$. Then there is s' s.t. $Q \stackrel{s'}{\Longrightarrow}$ and $s' \preceq s$. Choose now $[s_0]$ which is minimal for the set $\{[s''] : s'' \in L(Q) \text{ and } s'' \preceq s'\}$, and which exists by virtue of Lemma 9. By definition of $[\![\cdot]\!]_m$, $[s_0] \in [\![Q]\!]_m$, and moreover $s_0 \preceq s$. The converse implication can be proven similarly. □

It is possible to give a "concrete" representation of equivalence classes.

Proposition 11. Let $s_1 = m_1 M_1 \cdots m_n M_n$, $n \geq 0$, be any trace, where, for $1 \leq i \leq n$, m_i (resp M_i) is a trace containing only inputs (resp. outputs). Suppose that $s_2 \preceq s_1$ and $s_1 \preceq s_2$. Then s_2 is of the form $m'_1 M_1 \cdots m'_n M_n$, where, for $1 \leq i \leq n$, m'_i is a permutation of m_i.

The above proposition allows one to consider equivalence classes of traces as sequences where *multisets* of input actions alternate with sequences of output actions. This model can be further optimized. For example, when defining $[\![\cdot]\!]_m$ it is possible to enrich the theory of $\preceq$ with a commutativity law for outputs ($\overline{a}\overline{b} \preceq \overline{b}\overline{a}$); this permits viewing sequences of outputs as multisets and yields smaller denotations of processes. For instance, the denotation of process P in Example 1 would reduce to $\{[\epsilon], [\overline{a}], [b\overline{c}], [\overline{a}b\overline{c}]\}$. A similar optimization will be used in the definition of canonical traces, in the next section.

4 A Proof System for ACCS

In this section we define a proof system for ACCS and prove that it is sound and complete with respect to $\stackrel{\sqsubseteq}{\sim}_m$ for finite (without recursion) processes.

The proof system, that we call $\mathcal{A}$, is based on the in-equational laws in Table 1 plus the usual inference rules for reflexivity, transitivity and substitutivity in any context. We use G to range over guarded sums. Given two guarded sums $G = \sum_{i \in I} g_i.P_i$ and $G' = \sum_{j \in J} g_j.P_j$, we define $G + G'$ as $\sum_{k \in I \cup J} g_k.P_k$. Each equation $P = Q$ is an abbreviation for the pair of inequations $P \sqsubseteq Q$ and $Q \sqsubseteq P$. We write $P \sqsubseteq_{\mathcal{A}} Q$ ($P =_{\mathcal{A}} Q$) to indicate that $P \sqsubseteq Q$ ($P = Q$) can be derived within the proof system $\mathcal{A}$.

C1	$G + G = G$	
P1	$P \mid \mathbf{0} = P$	
P2	$P \mid Q = Q \mid P$	
P3	$P \mid (Q \mid R) = (P \mid Q) \mid R$	
EXP	Let $G = \sum_{i \in I} g_i.P_i$ and $G' = \sum_{j \in J} g'_j.P'_j$; then: $G \mid G' = \sum_{i \in I} g_i.(P_i \mid G') + \sum_{j \in J} g'_j.(G \mid P'_j)$	
R1	$(\sum_{i \in I} g_i.P_i)\{f\} = \sum_{i \in I} f(g_i).P_i\{f\}$	
R2	$(P \mid Q)\{f\} = P\{f\} \mid Q\{f\}$	
R3	$\overline{a}\{f\} = f(\overline{a})$	
H1	$(\sum_{i \in I} g_i.P_i)\backslash L = \sum_{i \in I \wedge g_i \notin L \cup \overline{L}} g_i.P_i\backslash L$	
H2	$(P \mid Q)\backslash L = P \mid Q\backslash L$	$L \cap \mathrm{n}(P) = \emptyset$
H3	$(P\backslash L_1)\backslash L_2 = P\backslash L_1 \cup L_2$	
H4	$(\overline{a} \mid g.P)\backslash a = g.(\overline{a} \mid P)\backslash a$	$g \neq a$
H5	$(\overline{a} \mid g.P)\backslash a = P\backslash a$	$g = a$
T1	$\overline{a} \mid \sum_{i \in I} g_i.P_i = \sum_{i \in I} \tau.(\overline{a} \mid g_i.P_i)$	
T2	$g.\sum_{i \in I} g_i.P_i = \sum_{i \in I} g.g_i.P_i$	
T3	$P = \tau.P$	
T4	$G \sqsubseteq G + G'$	
T5	$a.(\overline{b} \mid P) \sqsubseteq \overline{b} \mid a.P$	
T6	$P \sqsubseteq \overline{a} \mid a.P$	
A1	$a.b.P \sqsubseteq b.a.P$	
A2	$a.(\overline{a} \mid P) \sqsubseteq P$	

Table 1. Laws for ACCS

Laws A1 and A2 differentiate asynchronous from synchronous may testing: they are not sound for the synchronous may preorder [11]. In particular, law A1 states that processes are insensitive to the arrival ordering of messages from the environment, while law A2 states that any execution of P that depends on the availability of a message $\overline{a}$ is worse than P itself, even if $\overline{a}$ is immediately re-issued. The other laws in Table 1 are sound also for the synchronous may testing [11]. The laws in Table 1 can be easily proven sound by taking advantage of the preorder $\ll_m$.

Let us now consider some derived laws, among which (D1) $a.P \sqsubseteq_{\mathcal{A}} P$ and (D2) $\mathbf{0} \sqsubseteq_{\mathcal{A}} \overline{a}$. Law D2 follows immediately from law T4. The inequality D1 can be derived by first noting that from D2 it follows $P \sqsubseteq_{\mathcal{A}} \overline{a} \mid P$, which implies $a.P \sqsubseteq_{\mathcal{A}} a.(\overline{a} \mid P)$; now apply A2. In particular, we have that $a \sqsubseteq_{\mathcal{A}} \mathbf{0}$. From $\mathbf{0} \sqsubseteq_{\mathcal{A}} P$, for any P (a consequence of T4), and $a.\overline{a} =_{\mathcal{A}} a.(\overline{a} \mid \mathbf{0}) \sqsubseteq_{\mathcal{A}} \mathbf{0}$ (law A2), we get $a.\overline{a} =_{\mathcal{A}} \mathbf{0}$.

For proving completeness of the proof system, we shall rely on the existence of canonical forms for processes, which are *unique* up to associativity and

commutativity of summation and parallel composition and up to permutation of consecutive input actions. Uniqueness is a result of independent interest, because it leads to unique (and rather compact) representatives for equivalence classes of processes. The canonical form of a process will be obtained by minimizing its set of traces via a trace preorder, that extends $\preceq$ with a commutativity law for output actions.

Definition 12. Let $\preceq_{|}$ be the least preorder over traces induced by the laws TO1–TO3 plus law: (TO4) $\overline{a}\overline{b} \preceq \overline{b}\overline{a}$.

Of course, $\preceq$ is included in $\preceq_{|}$.

Definition 13 (canonical forms).

- Given $s \in Act^*$, the process $t(s)$ is defined by induction on s as follows: $t(\epsilon) \stackrel{\text{def}}{=} \mathbf{0}$, $t(as') \stackrel{\text{def}}{=} a.t(s')$ and $t(\overline{a}s') \stackrel{\text{def}}{=} \overline{a} \,|\, t(s')$.
- Consider $A \subseteq_{\text{fin}} \mathcal{L}^*$. We say that A is:
  - *complete* if whenever $t(r) \stackrel{s}{\Longrightarrow}$, for $r \in A$, then there is $s' \in A$ s.t. $s' \preceq_{|} s$;
  - *minimal* if whenever $s, s' \in A$ and $s' \preceq_{|} s$ then $s' = s$.
- A *canonical form* is a process of the form $\sum_{s \in A - \{\epsilon\}} \tau.t(s)$, for some $A \subseteq_{\text{fin}} \mathcal{L}^*$ which is both complete and minimal.

Note that a complete set of traces always contains the empty trace ϵ. The proof of uniqueness of canonical forms can be decomposed into three simple lemmata.

Lemma 14. If $t(s) \stackrel{s'}{\Longrightarrow}$ then $t(s') \sqsubseteq_{\mathcal{A}} t(s)$.

Proof: The proof proceeds by induction on the length of s. The most interesting case is when $s = \overline{a}s_0$, for some s_0; hence $t(s) = \overline{a} \,|\, t(s_0)$. Then there are two cases for s': either $t(s_0) \stackrel{s'}{\Longrightarrow}$, and then the thesis follows from $P \sqsubseteq_{\mathcal{A}} \overline{a} \,|\, P$ (by D2) and induction hypothesis, or $s' = \sigma\overline{a}\rho$, with $t(s_0) \stackrel{\sigma\rho}{\Longrightarrow}$, for some traces σ and ρ. In the latter case, we get from the induction hypothesis that $t(\sigma\rho) \sqsubseteq_{\mathcal{A}} t(s_0)$; hence $\overline{a} \,|\, t(\sigma\rho) \sqsubseteq_{\mathcal{A}} \overline{a} \,|\, t(s_0) = t(s)$; from repeated applications of P2) and T5, we get $t(s') = t(\sigma\overline{a}\rho) \sqsubseteq_{\mathcal{A}} \overline{a} \,|\, t(\sigma\rho)$, and hence the thesis. □

Lemma 15. Let $C_1 \stackrel{\text{def}}{=} \sum_{s \in A - \{\epsilon\}} \tau.t(s)$ and $C_2 \stackrel{\text{def}}{=} \sum_{r \in B - \{\epsilon\}} \tau.t(r)$ be canonical forms such that $C_1 \sqsubseteq\!\!\!\sim_m C_2$. Then for each $s \in A$ there is $r \in B$ such that $r \preceq_{|} s$.

Proof: Let $s \in A$. Then $C_1 \stackrel{s}{\Longrightarrow}$, thus, since $C_1 \ll_m C_2$, there is s' s.t. $C_2 \stackrel{s'}{\Longrightarrow}$ and $s' \preceq s$. This implies, by completeness of B, that there is $r \in B$ such that $r \preceq_{|} s'$. Since $s' \preceq s$, we obtain that $r \preceq_{|} s$. □

We write $P_1 =_{AC} P_2$ if $P_1 =_{\mathcal{A}} P_2$ can be derived using only the laws C2–C3, P2–P3 and A1. For the proof of the following lemma, just note that whenever s_1 and s_2 are $\preceq_{|}$-equivalent, then only laws TO2 and TO4 can be used to derive $s_1 \preceq_{|} s_2$ and $s_2 \preceq_{|} s_1$.

Lemma 16. If $s_1 \preceq_| s_2$ and $s_2 \preceq_| s_1$ then $t(s_1) =_{AC} t(s_2)$.

Theorem 17 (uniqueness). Let C_1 and C_2 be canonical forms such that $C_1 \simeq_m C_2$. Then $C_1 =_{AC} C_2$.

PROOF: Suppose $C_1 = \sum_{s \in A-\{\epsilon\}} \tau.t(s)$ and $C_2 = \sum_{r \in B-\{\epsilon\}} \tau.t(r)$. We prove that for each $s \in A$ there is $r \in B$ s.t. $s \preceq_| r$ and $r \preceq_| s$, by which the result will follow by Lemma 16 and by symmetry. Suppose that $s \in A$. Since $C_1 \sqsubseteq\!\!\!\sim_m C_2$, by Lemma 15, we deduce that there is $r \in B$ s.t. $r \preceq_| s$. But since $C_2 \sqsubseteq\!\!\!\sim_m C_1$ as well, we deduce the existence of $s' \in A$ with $s' \preceq_| r$, hence $s' \preceq_| r \preceq_| s$. By minimality of A we deduce that $s = s' \preceq_| r$. □

Example 2. Consider $P \stackrel{\text{def}}{=} \tau.(\overline{a} \mid b.\overline{b}) + \tau.b.(\overline{a} \mid \overline{b})$. To get the canonical form of P, we first compute the language of P and obtain the complete set $\{\epsilon, \overline{a}, b, \overline{a}b, b\overline{a}, b\overline{b}, ab\overline{b}, b\overline{a}\overline{b}, b\overline{b}\overline{a}\}$. Then we minimize, thus finding the minimal set $\{\epsilon, \overline{a}\}$, which is also complete. Thus $\tau.\overline{a}$ is the canonical form of P.

We proceed now to prove completeness of the proof system.

Lemma 18 (absorption). If $s' \preceq_| s$ then $t(s) \sqsubseteq_{\mathcal{A}} t(s')$.

PROOF: We prove the thesis by induction on the number n of times the laws T01–T04 are used to derive $s' \preceq_| s$. The proof relies on the laws D1, D2, A2 and P2. As an example, we analyze the base case ($n = 1$), when $s' \preceq_| s$ is derived with one application of T03. This means that $s' = \sigma a\overline{a}\rho$ and $s = \sigma\rho$, for some a and some traces σ and ρ. Now, note that whenever $s = s_1 s_2$ then $t(s) = t(s_1)[t(s_2)]$, where the latter term is obtained by replacing the single occurrence of $\mathbf{0}$ in $t(s_1)$ with $t(s_2)$. Therefore, by congruence of $\sqsubseteq_{\mathcal{A}}$ and law A2, we get:

$$t(s) = t(\sigma)[a.(\overline{a} \mid t(\rho))] \sqsubseteq_{\mathcal{A}} t(\sigma)[t(\rho)] = t(s')\,.$$ □

Lemma 19. For each P there exists a canonical form C s.t. $P =_{\mathcal{A}} C$.

PROOF: By induction on P and using the laws in Table 1 it is easy to show that P is provably equivalent to some process $C_1 = \sum_{s \in A_1 - \{\epsilon\}} \tau.t(s)$, for some set A_1. Consider now the following two facts:

1. Whenever $t(s) \stackrel{s'}{\Longrightarrow}$ then $t(s) =_{\mathcal{A}} \tau.t(s) + \tau.t(s')$.
2. Let A be a complete set. Suppose that there are $s, s' \in A$ s.t. $s \preceq_| s'$ and $s \neq s'$. Then: (a) $A - \{s'\}$ is complete, and (b) $\sum_{r \in A-\{\epsilon\}} \tau.t(r) =_{\mathcal{A}} \sum_{r \in A-\{\epsilon, s'\}} \tau.t(r)$.

(1 is a consequence of Lemma 14; 2 derives from the definition of complete set and, for part (b), of Lemma 18 and law C1).

By repeatedly applying 1, we can 'saturate' A_1, thus proving C_1 equivalent to a summation C_2 over a complete set A_2. Then, by repeatedly applying 2, we can remove redundant traces in A_2, thus proving C_2 equivalent to a summation over a complete and minimal set of traces. □

Theorem 20 (completeness). For finite ACCS processes P and Q, $P \underset{\sim}{\sqsubseteq}_m Q$ implies $P \sqsubseteq_{\mathcal{A}} Q$.

PROOF: Lemma 19 allows us to assume that both P and Q are in canonical form: $P \stackrel{\text{def}}{=} \sum_{s \in A-\{\epsilon\}} \tau.t(s)$ and $Q \stackrel{\text{def}}{=} \sum_{r \in B-\{\epsilon\}} \tau.t(r)$. It is sufficient to show that for each $s \in A$ there is $r \in B$ s.t. $t(s) \sqsubseteq_{\mathcal{A}} t(r)$, by which the thesis will follows thanks to the law T4. But this fact follows by Lemmata 15 and 18. □

5 The π-calculus

In this section we discuss the extensions of our theory to the asynchronous variant of π-calculus [16, 8, 14, 1].

Syntax and semantics We assume existence of a countable set $\mathcal{N}$ of *names* ranged over by $a, b, \ldots, x, \ldots$. Processes are ranged over by P, Q and R. The syntax of asynchronous π–calculus contains the operators of inaction, output action, guarded summation, restriction, parallel composition, matching and replication:

$$P ::= \overline{a}b \quad | \quad \sum_{i \in I} \alpha_i.P_i \quad | \quad \nu\, a\, P \quad | \quad P_1 \,|\, P_2 \quad | \quad [a = b]P \quad | \quad !\, P$$

where α is an *input action* $a(b)$ or a *silent action* τ. We adopt for the sum operator the same shorthands as for ACCS. *Free names* and *bound names* of a process P, written $\mathrm{fn}(P)$, and $\mathrm{bn}(P)$ respectively, arise as expected; the *names* of P, written $\mathrm{n}(P)$ are $\mathrm{fn}(P) \cup \mathrm{bn}(P)$. We shall consider processes up to α-equivalence. Thus α-equivalent processes have the same transitions and all bound names are always assumed to be different from each other and from the free names. The tilde $\tilde{\cdot}$ will be used to denote tuples of names; when convenient, we shall regard a tuple simply as a set. We omit the definition of operational semantics (see e.g. [1]), but remind that labels on transitions (*actions*), ranged over by μ, can be of four forms: τ (interaction), ab (input at a of b), $\overline{a}b$ (output at a of b) or $\overline{a}(b)$ (bound output at a of b). Functions $\mathrm{bn}(\cdot)$, $\mathrm{fn}(\cdot)$ and $\mathrm{n}(\cdot)$ are extended to actions as expected: in particular, $\mathrm{bn}(\mu) = b$ if $\mu = \overline{a}(b)$ and $\mathrm{bn}(\mu) = \emptyset$ otherwise.

The definition of the may preorder over the π-calculus, $\underset{\sim}{\sqsubseteq}_m$, is formally the same as for ACCS. Due to the presence of matching (see e.g. [1]), $\underset{\sim}{\sqsubseteq}_m$ is not preserved by input prefix.

The trace preorder We extend the operational semantics of the π-calculus with the following rule: if $P \xrightarrow{ab} P'$ and $b \notin \mathrm{fn}(P)$ then $P \xrightarrow{a(b)} P'$. The new kind of action $a(b)$ is called *bound input*; we extend $\mathrm{bn}(\cdot)$ to bound inputs by letting $\mathrm{bn}(a(b)) = \{b\}$. Below, we shall use $\mathcal{L}_\pi$ to denote the set of all visible (non-τ) actions, including bound inputs, and let θ range over it. Given a trace $s \in \mathcal{L}_\pi^*$, we say that s is *normal* if, whenever $s = s'.\theta.s''$ (the dot . stands for trace composition), for some s', θ and s'', then $\mathrm{bn}(\theta)$ does not occur in s' and $\mathrm{bn}(\theta)$ is different from any other bound name occurring in s''. Functions

$\mathrm{bn}(\cdot)$ and $\mathrm{fn}(\cdot)$ are extended to normal traces as expected. We consider normal traces up to α-equivalence. The set of normal traces over $\mathcal{L}_\pi$ is denoted by $\mathcal{T}$ and ranged over by s. From now on, *we shall work with normal traces only.* A complementation function on $\mathcal{T}$ is defined by setting $\overline{a(b)} \stackrel{\text{def}}{=} \overline{a}(b)$, $\overline{\overline{a}b} \stackrel{\text{def}}{=} \overline{a}b$, $\overline{\overline{a}b} \stackrel{\text{def}}{=} ab$ and $\overline{\overline{a}(b)} \stackrel{\text{def}}{=} a(b)$; note that $\overline{\overline{s}} = s$.

P1	$s.s' \preceq s.\theta.s'$	if θ is an input action and $\mathrm{bn}(\theta) \cap \mathrm{n}(s') = \emptyset$
P2	$s.\theta'.\theta.s' \preceq s.\theta.\theta's'$	if θ is an input action and $\mathrm{bn}(\theta) \cap \mathrm{n}(\theta') = \emptyset$
P3	$s.s' \preceq s.\theta.\overline{a}b.s'$	if $\theta = ab$ or $(\theta = a(b)$ and $b \notin \mathrm{n}(s'))$
P4	$s.\overline{a}c.(s'\{c/b\}) \preceq s.\overline{a}(b).s'$	

Fig. 2. Trace ordering laws over $\mathcal{T}$.

The presence of bound names requires a slightly different definition of the trace preorder $\preceq$, which is given below.

Definition 21. Let $\preceq_0$ the least binary relation induced by the laws in Figure 2: $\preceq$ is the reflexive and transitive closure of $\preceq_0$.

Rules P1, P2, P3 are the natural extensions to asynchronous π-calculus of the rules for ACCS. Here, some extra attention has to be paid to bound names: an output action declaring a new name (bound output) cannot be postponed after those actions that use that name. As an example, action $\overline{a}(b)$ cannot be postponed after $b(c)$, in any execution of the observer $\nu\, b\, (\overline{a}b \mid b(c).O)$. Accordingly, in the observed process, an input action receiving the new name, $a(b)$, cannot be postponed after output actions at b.

Rule P4 is specific to π-calculus, and is linked to the impossibility for observers to fully discriminate between free and bound outputs. Informally, rule P4 states that if a bound (hence new) name is "acceptable" for an observer, then any public name is acceptable as well. Rule P4 would disappear if we extended the language with the *mismatch* ($[a \neq b]P$) operator, considered e.g. in [6], which permits a full discrimination between free and bound outputs.

The definition of $\ll_m$ for the π-calculus relies on the trace preorder $\preceq$ and remains formally unchanged w.r.t. ACCS. In [7], we prove that $\ll_m$ and $\underset{\sim}{\sqsubset}_m$ coincide for the π-calculus. All the results obtained for ACCS about the trace–based model carry over smoothly to the π-calculus.

The proof system A sound and complete proof system for $\underset{\sim}{\sqsubset}_m$ over the finite (without replication) part of the language can be obtained by "translating" the proof system for ACCS into π-calculus, and then adding four new laws, as done in Table 2. I1 replaces the substitutivity rule for input prefix, M1 and M2 are concerned with matching, and S1 is related to the law P4 for $\preceq$.

We write $P \sqsubset_\pi Q$ if the inequality $P \sqsubset Q$ is derivable within the system of Table 2. Soundness of the system is straightforward. Completeness requires an

I1	if for each $b \in \mathrm{fn}(P,Q)$ $P\{b/x\} \sqsubseteq Q\{b/x\}$ then $a(x).P \sqsubseteq a(x).Q$	
M1	$[a=b]P = \mathbf{0}$	$a \neq b$
M2	$[a=a]P = P$	
C1	$G + G = G$	
P1	$P \mid \mathbf{0} = P$	
P2	$P \mid Q = Q \mid P$	
P3	$P \mid (Q \mid R) = (P \mid Q) \mid R$	
EXP	Let $G = \sum_{i\in I} \alpha_i.P_i$ and $G' = \sum_{j\in J} \alpha'_j.P'_j$, where each α_i (resp. α'_j) does not bind free names of G' (resp. G). Then: $G \mid G' = \sum_{i\in I} \alpha_i.(P_i \mid G') + \sum_{j\in J} \alpha'_j.(G \mid P'_j)$	
H1	$(\nu\, \widetilde{b})(\sum_{i\in I} \alpha_i.P_i) = \sum_{i\in I \wedge \mathrm{n}(\alpha_i)\cap \widetilde{b}=\emptyset} \alpha_i.(\nu\, \widetilde{b})P_i$	
H2	$(\nu\, \widetilde{b})(P \mid Q) = P \mid (\nu\, \widetilde{b})Q$	$\widetilde{b} \cap \mathrm{n}(P) = \emptyset$
H3	$(\nu\, a)(\overline{a}b \mid \alpha.P) = \alpha.(\nu\, a)(\overline{a}b \mid P)$	$a \notin \mathrm{n}(\alpha)$
H4	$(\nu\, a)(\overline{a}b \mid a(c).P) = (\nu\, a)(P\{b/c\})$	
T1	$\overline{a}b \mid \sum_{i\in I} \alpha_i.P_i = \sum_{i\in I} \tau.(\overline{a}b \mid \alpha_i.P_i)$	
T2	$\alpha.\sum_{i\in I} \alpha_i.P_i = \sum_{i\in I} \alpha.\alpha_i.P_i$	
T3	$P = \tau.P$	
T4	$G \sqsubseteq G + G'$	
T5	$a(c).(\overline{b}d \mid P) \sqsubseteq \overline{b}d \mid a(c).P$	$c \neq b$, $c \neq d$
T6	$P\{b/c\} \sqsubseteq \overline{a}b \mid a(c).P$	
A1	$a(c).b(d).P \sqsubseteq b(d).a(c).P$	$c \neq b$, $c \neq d$
A2	$a(c).(\overline{a}c \mid P) \sqsubseteq P$	$c \notin \mathrm{n}(P)$
S1	$(\nu\, c)P \sqsubseteq P\{b/c\}$	

Table 2. Laws for the asynchronous π-calculus

appropriate definition of canonical form. This implies extending $\preceq$ via commutativity for output actions.

Definition 22. Let $\preceq_{|}$ be the trace preorder over $\mathcal{T}$ induced by laws P1–P4 *plus* the laws:

- (P5) $s.\theta.\theta'.s' \preceq s.\theta'.\theta.s'$ if $\mathrm{bn}(\theta) \cap \mathrm{fn}(\theta') = \emptyset$ and $\mathrm{bn}(\theta') \cap \mathrm{fn}(\theta) = \emptyset$;
- (P6) $s.\overline{a}(b).\overline{c}b.s' \preceq s'.\overline{c}(b).\overline{a}b.s$ if $c \neq b$.

Definition 23 (canonical forms). Let s be a normal trace. The process $t(s)$ is defined by induction on s as follows: $t(\epsilon) \stackrel{\mathrm{def}}{=} \mathbf{0}$, $t(\overline{a}(b).s') \stackrel{\mathrm{def}}{=} \nu\, b\,(\overline{a}b \mid t(s'))$, $t(\overline{a}b.s') \stackrel{\mathrm{def}}{=} \overline{a}b \mid t(s')$, $t(a(c).s') \stackrel{\mathrm{def}}{=} a(c).t(s')$ and $t(ab.s') \stackrel{\mathrm{def}}{=} a(x).[x=b]t(s')$ (x fresh).

Modulo the new definitions of $t(s)$ and of $\preceq_{|}$, the definitions of *complete* set, of *minimal* set and of *canonical form* remain formally as in Definition 13.

Lemma 24. If $t(s) \stackrel{s'}{\Longrightarrow}$ then $t(s') \sqsubseteq_\pi t(s)$.

PROOF: The proof parallels that of Lemma 14. We analyze only the case when $s = \overline{a}(b).s_0$, hence $t(s) = \nu\, b\, (\overline{a}b \mid t(s_0))$. There are four possible cases for s' depending on how the execution of actions in $t(s_0)$ and action $\overline{a}b$ are interleaved.

1. $t(s_0) \stackrel{s'}{\Longrightarrow}$ (action $\overline{a}b$ is not fired at all);
2. $s' = \sigma.\overline{a'}(b).\rho$ and $t(s_0) \stackrel{\sigma.\overline{a'}b.\rho}{\Longrightarrow}$;
3. $s' = \sigma.\overline{a}(b).\rho$ and $t(s_0) \stackrel{\sigma.\rho}{\Longrightarrow}$;
4. $s' = \sigma_1.\overline{a'}(b).\sigma_2.\overline{a}b.\rho$ and $t(s_0) \stackrel{\sigma_1.\overline{a'}b.\sigma_2.\rho}{\Longrightarrow}$.

For case 1, the thesis follows from induction hypothesis. We analyze now case 4, because 2 and 3 are easier. By induction hypothesis, $t(\sigma_1.\overline{a'}b.\sigma_2.\rho) \sqsubseteq_{\mathcal{A}} t(s_0)$, hence

$$T \stackrel{\text{def}}{=} \nu\, b\, (\overline{a}b \mid t(\sigma_1.\overline{a'}b.\sigma_2.\rho)) \sqsubseteq_\pi \nu\, b\, (\overline{a}b \mid t(s_0)) = t(s).$$

On the other hand, by repeatedly applying T5 and P2, we can push $\overline{a}b$ rightward inside T and get $\nu\, b\, t(\sigma_1.\overline{a'}b.\sigma_2.\overline{a}b.\rho) \sqsubseteq_\pi T$. Finally, since $b \notin \mathrm{n}(\sigma_1)$, we can push $\nu\, b$ rightward (using H1 and H2) until it reaches $\overline{a'}b$, to get $t(s') \sqsubseteq_\pi T$, and the thesis. □

Lemma 25. If $s' \preceq_{|} s$ then $t(s) \sqsubseteq_{\mathcal{A}} t(s')$.

PROOF: The thesis is proven by induction on the number n of times the laws P1–P6 are used to derive $s' \preceq_{|} s$. As an example, we analyze the base case ($n = 1$), when $s' \preceq_{|} s$ is derived with one application of P3. In particular, consider the case $s' = \sigma.ab.\overline{a}b.\rho$ and $s = \sigma\rho$, for some a, b and some traces σ and ρ. For any P and fresh x, we have that $a(x).[x = b](\overline{a}b \mid P) \sqsubseteq_{\mathcal{A}} a(x).(\overline{a}x \mid P)$ (use rule I1 and laws M1 and M2). This inequality can be proven under any substitution σ for the names in $\mathrm{fn}(P) \cup \{a, b\}$, hence under any context. From this and A2, we get:

$$t(s) = t(\sigma)[a(x).[x = b](\overline{a}b \mid t(\rho))] \sqsubseteq_{\mathcal{A}} t(\sigma)[a(x).(\overline{a}x \mid t(\rho))] \sqsubseteq_{\mathcal{A}} t(\sigma)[t(\rho)] = t(s').$$

□

The proof of uniqueness of canonical forms remains essentially unchanged. The proof of existence of provably equivalent canonical forms requires the following derived laws:

(1) $a(y).[b = c]P =_\pi \tau.[b = c]a(y).P + \tau.a(y)$ if $y \notin \{b, c\}$, and
(2) $a(b).[b = c]P =_\pi a(b).[b = c]P\{c/b\}$.

These are used to accommodate matching, when initially proving that P is equivalent to a summation of $t(s)$'s; then, the proof proceeds formally unchanged. Given the existence and the uniqueness of canonical forms, the actual proof of completeness remains essentially unchanged.

Theorem 26 (completeness). For finite π-calculus processes P and Q, $P \sqsubseteq\!\!\!\!\sim_m Q$ implies $P \sqsubseteq_\pi Q$.

6 Conclusions and Related Works

In this paper, we have studied a may testing semantics for two asynchronous variants of CCS and π-calculus. For both calculi we have proposed a finitary trace–based interpretation of processes and a complete inequational proof system.

Recently, there have been various proposals of models of asynchronous processes. Two main approaches have been followed to this purpose. They differ in the way (non–blocking) output actions are modelled. The asynchronous variants of ACP [4], CSP [17] and LOTOS [21] introduce external buffers in correspondence of output channels. This makes outputs non–blocking and immediately executable, while preserving the orderings between different output actions. Within the same group we can place the work on the actors foundation [3]. Differently, the asynchronous variants of π-calculus [16, 8, 14, 1] and CCS [20, 12, 9] model output prefix $\overline{a}.P$ as a parallel composition $\overline{a} \mid P$, i.e. output actions are independent processes. The communication medium is rendered as a bag of messages, which is directly represented within the syntax as a parallel composition of output actions.

In the past, all these formalisms have been equipped with observational semantics based on bisimulation or failures, but very few denotational or equational characterizations have been studied. A notable exception is the work by de Boer, Palamidessi and their collaborators. On one hand, in [5], they propose a trace-based model for a variant of failure semantics, on the other, in [4], they provide axiomatizations that rely on state operators and explicitly model evolution of buffers. Other studies deal with languages that fall in the first group of asynchronous formalisms and propose set of laws that help to understand the proposed semantics, but do not offer complete axiomatizations [21, 3]. For those languages that model outputs by means of processes creation, the only paper that presents an axiomatization is [1]. There, a complete axiomatization of strong bisimilarity for asynchronous π-calculus is proposed, but the problem of axiomatizing weak (τ-forgetful) variants of the equivalence is left open.

A paper closely related to ours is the recent [10]. There, for a variant of asynchronous CCS, the authors present a complete axiomatization of *must* testing semantics, which is more appropriate for reasoning about liveness properties. No finitary model is presented and the problem of extending the results to the asynchronous π-calculus is left open.

Acknowledgments. Five anonymous referees provided valuable suggestions; Istituto di Elaborazione dell'Informazione in Pisa made our collaboration possible.

References

1. R.M. Amadio, I. Castellani, D. Sangiorgi. On Bisimulations for the Asynchronous π–calculus. *CONCUR'96*, *LNCS* 1119, pp.147-162, Springer, 1996.

2. M. Abadi, A.D. Gordon: A calculus for cryptographic protocols: The Spi calculus. Proc. *4th ACM Confeence on Computer and Communication Security*, ACM Press, 1997.
3. G.A. Agha, I.A. Mason, S.F. Smith, C.L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7:1-72, 1997.
4. F.S. de Boer, J.W. Klop, C. Palamidessi. Asynchronous Communication in Process Algebra. *LICS'92*, IEEE Computer Society Press, pp. 137-147, 1992.
5. F.S. de Boer, J.N. Kok, C. Palamidessi, J.J.M.M. Rutten. The Failure of Failures in a Paradigm for Asynchronous Communication. *CONCUR'91*, *LNCS* 527, pages 111-126, Springer, 1991.
6. M. Boreale, R. De Nicola. Testing Equivalence for Mobile Systems. *Information and Computation*, 120: 279-303, 1995.
7. M. Boreale, R. De Nicola, R. Pugliese. Asynchronous Observations of Processes. *FoSSaCS'98*, *LNCS* , Springer, 1998.
8. G. Boudol. Asynchrony in the π–calculus (note). Rapport de Recherche 1702, INRIA Sophia–Antipolis, 1992.
9. N. Busi, R. Gorrieri, G-L. Zavattaro. A process algebraic view of Linda coordination primitives. Technical Report UBLCS-97-05, University of Bologna, 1997.
10. I. Castellani, M. Hennesy. Testing Theories for Asynchronous Languages. Proc. *FSTTCS*, *LNCS* , to appear Dec. 1998.
11. R. De Nicola, M.C.B. Hennessy. Testing Equivalence for Processes. *Theoretical Computers Science*, 34:83-133, 1984.
12. R. De Nicola, R. Pugliese. A Process Algebra based on Linda. *COORDINATION'96*, *LNCS* 1061, pp.160-178, Springer, 1996.
13. D. Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80-112, 1985.
14. M. Hansen, H. Huttel, J. Kleist. Bisimulations for Asynchronous Mobile Processes. *In Proc. of the Tblisi Symposium on Language, Logic, and Computation*, 1995.
15. M.C.B. Hennessy. *Algebraic Theory of Processes.* The MIT Press, 1988.
16. K. Honda, M. Tokoro. An Object Calculus for Asynchronous Communication. *ECOOP'91*, *LNCS* 512, pp.133-147, Springer, 1991.
17. H. Jifeng, M.B. Josephs, C.A.R. Hoare. A Theory of Synchrony and Asynchrony. *Proc. of the IFIP Working Conf. on Programming Concepts and Methods*, pp.446-465, 1990.
18. R. Milner. The Polyadic π-calculus: A Tutorial. Technical Report, University of Edinburgh, 1991.
19. J. Parrow, D. Sangiorgi. Algebraic theories for name-passing calculi. *Information and Computation*, 120(2):174–197, 1995.
20. R. Pugliese. A Process Calculus with Asynchronous Communications. 5th Italian Conference on Theoretical Computer Science, (A. De Santis, ed.), pp.295-310, World Scientific, 1996.
21. J. Tretmans. A formal approach to conformance testing. Ph.D. Thesis, University of Twente, 1992.

A Nondeterministic Polynomial-Time Unification Algorithm for Bags, Sets and Trees

Evgeny Dantsin[1] and Andrei Voronkov[2]

[1] Steklov Institute of Mathematics at St.Petersburg
[2] Computing Science Department, Uppsala University

Abstract. Unification in logic programming deals with tree-like data represented by terms. Some applications, including deductive databases, require handling more complex values, for example finite sets or bags (finite multisets). We extend unification to the combined domain of bags, sets and trees in which bags and sets are generated by constructors similar to the list constructor. Our unification algorithm is presented as a nondeterministic polynomial-time algorithm that solves equality constraints in the spirit of the Martelli and Montanari algorithm.

1 Introduction

Logic programming languages deal with tree-like data represented by terms. Some applications require to handle other kinds of data such as finite sets or bags (finite multisets). For example, this problem arises in databases: relational query languages typically deal with tuples of atomic values and extension to more complex values is required. Various kinds of complex values in databases and logic programming have been considered in many papers, including [2, 3, 31, 16, 46, 43, 32, 15, 1, 27, 26, 22, 20, 48].

In this paper we extend unification, the core mechanism of logic programming, to handle bags and sets. When bags and sets are represented with using the union operations, bag and set unification is a particular case of AC1- and ACI1-unification, which has been extensively studied [44, 25, 24, 10, 33, 18]. Recently, a number of unification algorithms have been introduced for various domains of bags and sets built with the bag and set constructors similar to the list constructor used in functional and logic programming [21, 22, 45, 9].

We contribute to this area by introducing a new unification algorithm for the *combined* domain of bags, sets and trees. The main novelty of our algorithm is that it is a *nondeterministic polynomial-time* algorithm. The algorithm is formalized as a nondeterministic algorithm that solves systems of equations in the spirit of the Martelli and Montanari unification algorithm [38], i.e. it uses a collection of rules that transform systems into equivalent ones. The algorithm is don't-care nondeterministic with respect to the choice of applicable rules. Nondeterministic branches lead to unifiers represented by systems in solved form (triangle form). These unifiers form a complete set of unifiers of the input system, its cardinality is at most $2^{O(n \log n)}$ where n is the input size. Thus, we obtain a

W. Thomas (Ed.): FOSSACS'99, LNCS 1578, pp. 180-196, 1999.

single-exponential upper bound on the cardinality of a minimal complete set of unifiers in our domain (cf. a double-exponential lower bound for AC unification in the domain with the bag union [30]).

The paper is organized as follows. In Section 2 we extend the Herbrand universe by adding terms that represent bags and sets. Thus, the extended domain (denoted by $\mathcal{HU}^+$) contains bags, sets and trees. Values in this domain are untyped, for example one can construct a bag whose members are sets and trees. The semantics of logic programs over $\mathcal{HU}^+$ is defined in Section 2 too. We show that logic programming over $\mathcal{HU}^+$ is powerful enough to represent all computable predicates on this domain.

Section 3 is the main section of this paper. It contains a description of our algorithm and theorems asserting that the algorithm is sound, complete and runs in nondeterministic polynomial time. Since NP-hardness is proved easily [28], we obtain that unification over $\mathcal{HU}^+$ is NP-complete. It follows from these theorems that the algorithm yields complete sets of unifiers and their cardinalities are at most $2^{O(n \log n)}$. This bound is tight. In this section we also describe a number of important special cases in which our algorithm is optimal, i.e. it gives minimal complete sets of unifiers. Due to space reasons, we do not include proofs in this paper, they can be found in our technical report [19]. Also, this report contains many examples that illustrate the algorithm as well as some related notions.

In Section 4 we briefly sketch some related results and directions of further research. In particular, we discuss bag and set unification in the context of AC and ACI unification. We also compare our algorithm with other known algorithms. Some extensions and applications of our results are discussed too.

There are several aspects of handling complex values that are beyond the scope of this paper. We do not consider the introduction of the object structure on $\mathcal{HU}^+$ and we do not discuss questions like object identity. We do not consider semantics of negation. Our algorithm can be optimized in some ways but we do not discuss such optimizations here.

2 The combined domain of bags, sets and trees

There are several possibilities to define data models that deal with bags, finite sets and trees. In this section we choose a particular data model, some variations are considered in [19]. For brevity we say "set" instead of "finite set" in the context of this data model. "Bag" is a synonym for "finite multiset".

The Herbrand universe with bags and sets. We extend the Herbrand universe by adding the bag and set constructors. The definition is parametrized by a set $\mathcal{F}$ of *function symbols*. As usual, symbols in $\mathcal{F}$ have non-negative arities. *Constants* are function symbols of arity 0. Intuitively, constants represent atomic values, like integers or strings, Function symbols of arity ≥ 1 are viewed as *tree constructors* that are used to construct complex values from existing ones.

More precisely, given a set $\mathcal{F}$ of function symbols, the *Herbrand universe with bags and sets*, denoted $\mathcal{HU}^+$, is defined inductively as follows.

1. Any constant in $\mathcal{F}$ belongs to $\mathcal{HU}^+$. These constants are called *atomic values*.
2. If $v_1, \ldots, v_n \in \mathcal{HU}^+$, where $n \geq 0$, then the bag consisting of $v_1, \ldots, v_n$ belongs to $\mathcal{HU}^+$. This bag is denoted by $\{\!|v_1, \ldots, v_n|\!\}$ and called a *bag value*.
3. If $v_1, \ldots, v_n \in \mathcal{HU}^+$, where $n \geq 0$, then the set $\{v_1, \ldots, v_n\}$ belongs to $\mathcal{HU}^+$. This set is called a *set value*.
4. If $f \in \mathcal{F}$ has arity $n \geq 1$ and $v_1, \ldots, v_n \in \mathcal{HU}^+$, then the expression $f(v_1, \ldots, v_n)$ belongs to $\mathcal{HU}^+$. This expression represents the tree whose root has n children; the root is labeled by f and the children are $v_1, \ldots, v_n$. The expression $f(v_1, \ldots, v_n)$ is called a *tree value*.

Thus, $\mathcal{HU}^+$ consists of untyped values. For example, the set $\{\{\!|1, 1|\!\}, \{2\}, 3, f(4)\}$ contains a bag, a set, an atomic value and a tree.

Equality on $\mathcal{HU}^+$ is defined inductively as follows.

1. Two atomic values are equal if they coincide.
2. Bag values $\{\!|u_1, \ldots, u_m|\!\}$ and $\{\!|v_1, \ldots, v_n|\!\}$ are equal if $m = n$ and there is a permutation ρ of the sequence $1, \ldots, m$ such that u_i is equal to $v_{\rho(i)}$ for all i.
3. Set values $\{u_1, \ldots, u_m\}$ and $\{v_1, \ldots, v_n\}$ are equal if each u_i is equal to some v_j, and vice versa, each v_j is equal to some u_i.
4. Tree values $f(u_1, \ldots, u_m)$ and $g(v_1, \ldots, v_n)$ are equal if f coincides with g and u_i is equal to v_i for all i.
5. No other equalities hold on $\mathcal{HU}^+$.

Terms and their sorts. In order to represent elements of $\mathcal{HU}^+$, we introduce the corresponding notion of a term. We assume that $\{\!|\!|\!\}$ and $\{\}$ are two constants foreign to $\mathcal{F}$. They represent the empty bag and the empty set respectively. Two binary function symbols $\{\!|\,|\!\}$ and $\{\,|\,\}$ are assumed to be foreign to $\mathcal{F}$ too. They are used to construct bags and sets. Namely, if a term s represents an arbitrary element of $\mathcal{HU}^+$ and a term t represents a bag then the term $\{\!|s\,|\,t|\!\}$ represents the bag formed by appending the element corresponding to s to the bag corresponding to t. Similarly, a term $\{s\,|\,t\}$ represents the set formed by adding the element represented by s to the set represented by t.

All terms will be divided into three sorts: sort of bags $\mathfrak{b}$, sort of sets $\mathfrak{s}$ and the universal sort $\mathfrak{u}$. We assume that there are three kinds of variables called *bag variables*, *set variables* and *universal variables*. Terms and their sorts are defined as follows.

1. A bag variable is a term of sort $\mathfrak{b}$. A set variable is a term of sort $\mathfrak{s}$. A universal variable is a term of sort $\mathfrak{u}$.
2. The constant $\{\!|\!|\!\}$ is a term of sort $\mathfrak{b}$. The constant $\{\}$ is a term of sort $\mathfrak{s}$. Any constant in $\mathcal{F}$ is a term of sort $\mathfrak{u}$.
3. If s is an arbitrary term and t is a term of sort $\mathfrak{b}$ then $\{\!|s\,|\,t|\!\}$ is a term of sort $\mathfrak{b}$. If s is an arbitrary term and t is a term of sort $\mathfrak{s}$ then $\{s\,|\,t\}$ is a term of sort $\mathfrak{s}$. Any expression of the form $f(t_1, \ldots, t_n)$, where $f \in \mathcal{F}$ and $t_1, \ldots, t_n$ are arbitrary terms, is a term of sort $\mathfrak{u}$.
4. No other terms can be formed.

We also introduce the partial order $\sqsubseteq$ on the terms. For any terms s, t, we write $s \sqsubseteq t$ if s and t have the same sort or t is of sort $\mathfrak{u}$.

Equations and solutions. A mapping ν from the set of all variables to $\mathcal{HU}^+$ is called a *valuation* if ν maps every bag variable to a bag value and every set variable to a set value. Universal variables can be mapped to any values. We extend valuations from variables to terms as follows. Let ν be a valuation and r be a non-variable term. The value $\nu(r)$ is defined inductively:

1. $\nu(\{|\,|\})$ is the bag value $\{|\,|\}$. If r is $\{|s \mid t|\}$ and $\nu(t)$ is a bag value $\{|v_1, \ldots, v_n|\}$, then $\nu(r)$ is the bag value $\{|\nu(s), v_1, \ldots, v_n|\}$.
2. $\nu(\{\})$ is the set value $\{\}$. If r is $\{s \mid t\}$ and $\nu(t)$ is a set value $\{v_1, \ldots, v_n\}$, then $\nu(r)$ is the set value $\{\nu(s), v_1, \ldots, v_n\}$.
3. If r is $f(v_1, \ldots, v_n)$, where $n \geq 0$, then $\nu(r)$ is $f(\nu(v_1), \ldots, \nu(v_n))$.

By this definition, the value $\nu(r)$ for a ground term r remains the same for all ν. Hence, elements of $\mathcal{HU}^+$ can be alternatively defined as equivalence classes of ground terms by the equality relation.

By an *equation* we mean any expression $s = t$, where s, t are terms. A *solution* to an equation is a valuation ν such that $\nu(s)$ is equal to $\nu(t)$. A finite set of equations is also called a *system (of equations)* or an *equality constraint*. A valuation ν is a *solution to a system* if ν is a solution to every equation in the system. We fix some system that has no solution and denote this system by $\bot$.

Logic programs over $\mathcal{HU}^+$ and their semantics. The notion of a *Horn clause* is defined as usual. We assume that atoms in Horn clauses are not equalities. A *logic program over $\mathcal{HU}^+$* is a finite set of Horn clauses. A *Herbrand model of a logic program L over $\mathcal{HU}^+$* is any model $\mathfrak{M}$ of L such that (i) the carrier set of $\mathfrak{M}$ is $\mathcal{HU}^+$ and (ii) each ground term is interpreted in $\mathfrak{M}$ by itself. As usual, a Herbrand model can be identified with the set of ground atoms true in this model. Thus we can redefine a Herbrand model $\mathfrak{M}$ as a set of ground non-equality atoms (meaning the set of ground non-equality atoms true in $\mathfrak{M}$).

For two Herbrand models $\mathfrak{M}_1$ and $\mathfrak{M}_2$, we write $\mathfrak{M}_1 \subseteq \mathfrak{M}_2$ if the set of all ground atoms true in $\mathfrak{M}_1$ is a subset of all ground atoms true in $\mathfrak{M}_2$. It is not difficult to prove that any logic program over $\mathcal{HU}^+$ has the least Herbrand model (with respect to $\subseteq$). This statement is basically a straightforward generalization of the standard facts of logic programming theory [35, 8].

We say that a logic program L over $\mathcal{HU}^+$ *defines a relation R on $\mathcal{HU}^+$* if for some predicate P in L, the predicate P is interpreted as R in the least Herbrand model of L.

There are several ways of defining a procedural semantics of logic programs over $\mathcal{HU}^+$. The standard way is to modify SLD-resolution, namely, instead of SLD-resolution using substitutions, we can use *constraint SLD-resolution* defined similar to [36, 37]. In particular, constraint SLD-resolution over $\mathcal{HU}^+$ uses unification over $\mathcal{HU}^+$, i.e. solving equality constraints over $\mathcal{HU}^+$.

Logic programs and computability on $\mathcal{HU}^+$. Since elements of $\mathcal{HU}^+$ can be obviously represented as strings of symbols, we can speak of computable (aka recursively enumerable) relations on $\mathcal{HU}^+$. The following theorem shows that our constructors are enough to represent any computable relation on $\mathcal{HU}^+$.

Theorem 1. *A relation R on $\mathcal{HU}^+$ is recursively enumerable if and only if there exists a logic program L that defines R.*

The proof can be easily obtained by adapting the proof of [49] or other similar proofs (e.g., [6]).

3 Unification algorithm

In this section we introduce a unification algorithm for $\mathcal{HU}^+$. The algorithm is presented as a nondeterministic algorithm that transforms an input system of equations into an output system in *solved form* (defined below). The work of the algorithm is based on repeated applications of *transformation rules*. The output systems (all nondeterministic results) are equivalent to the input systems in the sense described in Theorem 3.

3.1 Some definitions and notation

Notation for bags and sets. Like the logic programming notation for lists, we write $\{|s_1, \ldots, s_n \,|\, t|\}$ and $\{|s_1, \ldots, s_n|\}$ for terms $\{|s_1 \mid \ldots \{|s_n \mid t|\} \ldots |\}$ and $\{|s_1 \mid \ldots \{|s_n \mid \{|\,|\}|\} \ldots |\}$ respectively, and similar for sets. Letters x, y, z, u, v, w with or without indices are used to denote variables. Capital letters $X, Y, \ldots$ (possibly with indices) stand for sequences of variables. For example, if X and Y stand for $x_1, \ldots, x_k$ and $y_1, \ldots, y_m$ respectively then $\{X, Y \,|\, u\}$ and $\{X, Y\}$ denote $\{x_1, \ldots, x_k, y_1, \ldots, y_m \,|\, u\}$ and $\{x_1, \ldots, x_k, y_1, \ldots, y_m\}$. When we use notation $\{|X \,|\, s|\}$ and $\{X \,|\, s\}$, where s is a variable, we assume that X is non-empty. The *length* of a sequence X, i.e. the number of its members, is denoted by $|X|$. If X and Y are $x_1, \ldots, x_n$ and $y_1, \ldots, y_n$ respectively, $X = Y$ stands for $x_1 = y_1, \ \ldots, \ x_n = y_n$. Sometimes, we shall use capital letters joined by the set union $\cup$, for example $X \cup Y$. In this case, we mean by such an expression the sequence obtained by appending the sequence Y to the sequence X and removing duplicates. For example, if X is x, x, y and Y is y, y, z then $X \cup Y$ denotes the sequence x, y, z.

Reduction using $x = y$. Let x and y be variables appearing in a system S. The following transformation of S is called *reduction using $x = y$*. First, remove $x = y$ and $y = x$ (if any) from S. Then do the following:

1. If x and y are the same variable, do nothing.
2. Otherwise, if one of x, y is of sort $\mathfrak{b}$ and the other is of sort $\mathfrak{s}$, transform S into $\bot$.
3. Otherwise, if $y \sqsubseteq x$, replace x by y in all equations in S and add $x = y$ to S.
4. Otherwise (in this case $x \sqsubseteq y$), replace y by x in all equations in S and add $y = x$ to S.

Isolated equations. Let S be a system containing an equation $x = t$. This equation is called *isolated* in S if (i) $t \sqsubseteq x$ and (ii) x has exactly one occurrence in S (namely, the occurrence in the left-hand side of $x = t$). Note that if $x = y$ belongs to S and is not isolated in S, then reduction using $x = y$ changes $x = y$ to an isolated equation $x = y$ or $y = x$.

Simple equations. We define a *bag equation* as an equation $x = \{|Y \mid z|\}$, where x and z are bag variables and Y is a sequence of variables. Similarly, $x = \{Y \mid z\}$, where x and z are set variables and Y is a sequence of variables, is called a *set equation*. We define a *simple equation* as any of the following equations: (i) a bag equation, (ii) a set equation, (iii) an equation $x = f(x_1, \ldots, x_n)$, where $n \geq 0$ and $x, x_1, \ldots, x_n$ are variables.

Systems without duplication. Let S be a system consisting of only simple or isolated equations. We call S a *system of simple or isolated equations without duplication* if S contains no pair of equations $x = t$ and $y = t$ such that t is not a variable.

Lemma 1. *Any system S can be transformed in polynomial time to a system S' that satisfies the following conditions:*

1. *S' is a system of simple or isolated equations without duplication.*
2. *Any solution to S' is also a solution to S.*
3. *For any solution ν to S, there is a solution ν' to S' such that ν and ν' coincide on all variables of S.*

Proof. We shall only sketch the transformation.

1. Get rid of non-variable terms in left sides of equations by introducing new variables. An equation $s = t$ is replaced by two equations $x = s$ and $x = t$, where x is a new universal variable.

2. Variable abstraction. Get rid of non-simple equations $x = t$, where t is not a variable, by introducing new variables. For example, the equation $x = \{\{\}, f(a), b\}$ is replaced by five equations $x = \{y, z, v \mid y\}$, $y = \{\}$, $z = f(u)$, $u = a$ and $v = b$, where z, u, v are new universal variables, y is a new set variable.

3. To get rid of duplications, for any pair of equations $x = t$ and $y = t$ remove $x = t$ and apply reduction using $x = y$.

4. Now every non-simple equation is an equation between two variables $x = y$. If this equation is not isolated, apply reduction using $x = y$.

From now on we deal only with systems of simple or isolated equations without duplication.

Graphs associated with systems. To describe the algorithm, we associate with any system S a directed graph denoted by $\mathcal{G}_S$ and called *the graph of S*. The nodes of this graph are all bag and set variables occurring in S. If S contains a bag equation $x = \{|Y \mid z|\}$ or a set equation $x = \{Y \mid z\}$, then the graph $\mathcal{G}_S$ contains an edge from x to z labeled by Y.

Final variables. A bag variable x is said to be *final in* S if $\mathcal{G}_S$ contains no edge coming from x. A set variable x is called *final in* S if whenever $\mathcal{G}_S$ contains a path from x to another node y, it also contains a path from y to x.

Bag and set cycles. We define a *bag cycle* as a bag equation $x = \{\!|Y \mid x|\!\}$ or a sequence of bag equations

$$x_1 = \{\!|Y_1 \mid x_2|\!\},\ x_2 = \{\!|Y_2 \mid x_3|\!\},\ \ldots,\ x_m = \{\!|Y_m \mid x_1|\!\},$$

where $m \geq 2$ and $x_1, \ldots, x_m$ are pairwise different variables. A *set cycle* is defined similarly.

Bag and set extensions. Assume that a system S contains a sequence of bag equations

$$y_1 = \{\!|X_1 \mid y_2|\!\},\ y_2 = \{\!|X_2 \mid y_3|\!\},\quad \ldots,\quad y_m = \{\!|X_m \mid y_{m+1}|\!\}$$

where $m \geq 1$ and y_{m+1} is a final variable. We say that the bag equation

$$y_1 = \{\!|X_1, \ldots, X_m \mid y_{m+1}|\!\}$$

is an *extension* of the equation $y_1 = \{\!|X_1 \mid y_2|\!\}$.

Similarly, if S contains a sequence of set equations

$$y_1 = \{X_1 \mid y_2\},\ y_2 = \{X_2 \mid y_3\},\quad \ldots,\quad y_m = \{X_m \mid y_{m+1}\},$$

where $m \geq 1$ and y_{m+1} is a final variable, then an *extension* of the set equation $y_1 = \{X_1 \mid y_2\}$ is defined as the equation

$$y_1 = \{X_1 \cup \ldots \cup X_m \mid y_{m+1}\}.$$

Note that if a bag equation has no extension then S has a bag cycle. Every set equation has an extension.

Reduction using $X = Y$. Let X and Y be sequences $x_1, \ldots, x_n$ and $y_1, \ldots, y_n$ of variables respectively. Informally, reduction using $X = Y$ is successive reductions using $x_1 = y_1, \ldots, x_n = y_n$. More precisely, denote the system S by S_0 and denote the system $\{X = Y\}$ by E_0. Define S_i and E_i for $1 \leq i \leq n$ as follows. Let $u = v$ be any equation in E_{i-1}. Then S_i is obtained from S_{i-1} by reduction using $u = v$ and E_i is obtained from E_{i-1} by reduction using $u = v$ and removing $u = v$ or $v = u$. *Reduction using* $X = Y$ is defined as the transformation replacing S by S_n.

Correspondences between sequences. Let X and Y be sequences of variables. We define *correspondences* between X and Y, namely correspondences of two kinds called *bag correspondences* and *set correspondences*. A *bag correspondence* between X and Y is defined as an equivalence relation on $X \cup Y$ such that for every equivalence class R, the variables of R satisfy the following condition:

The total number of their occurrences in X is equal to the total number of their occurrences in Y.

For example, let X and Y be x, x, y, z and x, y, y, u respectively. Then there are three bag correspondences between X and Y. The first consists of two equivalence classes $\{x, y\}$ and $\{u, z\}$, the second consists of $\{x, u\}$ and $\{y, z\}$, and the third consists of one class $\{x, y, z, u\}$. Obviously, there exists a bag correspondence between X and Y if and only if X and Y have the same length.

A bag correspondence is called *minimal* if no equivalence class can be split into proper disjoint subsets such that the resulting equivalence relation remains a bag correspondence. In the above example, the first and second correspondences are minimal and the third one is not minimal. Note that the minimality can be checked in polynomial time by reducing this problem to the unary version of the knapsack problem, i.e. the version in which weights and values are given in unary notation (see e.g. [41]). Details of the reduction will be given in a full version of the paper.

A *set correspondence* between X and Y is an equivalence relation on $X \cup Y$ such that for every equivalence class R, we have

R contains at least one variable of X and at least one variable of Y.

Like the case of bag correspondences, a set correspondence is called *minimal* if no equivalence class can be split into proper disjoint subsets such that the resulting equivalence relation remains a set correspondence. In this case, it means that each equivalence class contains either exactly one variable of X or exactly one variable of Y.

For example, let X and Y be x, x, y, z and x, u, v, w respectively. Then the equivalence relation consisting of classes $\{x, u\}$, $\{y, v\}$ and $\{z, w\}$ is a minimal set correspondence between X and Y. The relation consisting of $\{x, y, u\}$ and $\{z, v, w\}$ is a set correspondence but it is not minimal.

Let E be any set of equations $x_i = y_j$ where x_i is in X and y_j is in Y. By $\sim_E$ we denote the smallest equivalence relation on $X \cup Y$ containing all pairs (x, y) such that $(x = y) \in E$. We say that E is a (minimal) bag or set correspondence between X and Y if such is $\sim_E$. For example, the set $\{x = u, y = v, z = w\}$ is a minimal set correspondence between X and Y that denote x, x, y, z and x, u, v, w respectively.

3.2 Rules

Rule 1 (Tree Decomposition). This rule can be applied to a system S if S contains two distinct equations $z = f(x_1, \ldots, x_n)$ and $z = f(y_1, \ldots, y_n)$, where $f \in \mathcal{F}$ and $n \geq 0$. Remove the equation $z = f(x_1, \ldots, x_n)$ from S and reduce S using $x_1 = y_1, \ldots, x_n = y_n$. (This rule is close to the term decomposition rule of [38]).

Rule 2 (Bag Decomposition). The rule can be applied to a system S if the system contains bag equations $z = s$ and $z = t$, where s and t are different. If at least one of $z = s$ and $z = t$ has no extension then transform S into $\bot$ (the system contains a bag cycle). Otherwise, choose any extensions of $z = s$ and $z = t$. Remove $z = s$ and $z = t$ from S and consider two possible cases.

1. *The extensions have the forms* $z = \{\!|X \,|\, u|\!\}$ *and* $z = \{\!|Y \,|\, u|\!\}$. If $|X| \neq |Y|$ then transform S into $\bot$. Otherwise, add to S the equation $z = \{\!|Y \,|\, u|\!\}$. Don't-know nondeterministically generate a minimal bag correspondence E between X and Y and reduce S using E.
2. *The extensions have the forms* $z = \{\!|X \,|\, u|\!\}$ *and* $z = \{\!|Y \,|\, v|\!\}$, *where the variables* u *and* v *are different.* Don't-know nondeterministically divide X into disjoint parts X_1 and X_2 (one of them may be empty). Similarly, Don't-know nondeterministically divide Y into Y_1 and Y_2. Informal comment: X_1 and Y_1 are intended to coincide as bags, X_2 and Y_2 are intended to have no common elements. The division is required to satisfy the following conditions. First, the parts X_1 and Y_1 have the same length, i.e. $|X_1| = |Y_1|$. Second, X_2 and Y_2 contain no common variables. Then S is transformed as follows.
   (a) Add the equation $z = \{\!|X_2, Y_1, Y_2 \,|\, w|\!\}$, where w is a new bag variable.
   (b) If X_2 is non-empty, add the equation $v = \{\!|X_2 \,|\, w|\!\}$. Otherwise, reduce S using $v = w$.
   (c) If Y_2 is non-empty, add the equation $u = \{\!|Y_2 \,|\, w|\!\}$. Otherwise, reduce S using $u = w$.
   (d) If X_1 and Y_1 are non-empty, don't-know nondeterministically generate a minimal bag correspondence E between X_1 and Y_1. Reduce S using E.

Rule 3 (Set Decomposition). The rule can be applied to a system S if the system contains set equations $z = s$ and $z = t$, where s and t are different. If z is a final variable, do the following. Suppose that s is $\{X \,|\, u\}$ and t is $\{Y \,|\, v\}$. Replace $z = s$ and $z = t$ by the equation $z = \{X \cup Y \,|\, z\}$ and reduce the system using $z = u, z = v$. Otherwise, choose any extensions of $z = s$ and $z = t$. Remove $z = s$ and $z = t$ from S and consider two possible cases.

1. *The extensions have the forms* $z = \{X \,|\, u\}$ *and* $z = \{Y \,|\, u\}$. Don't-know nondeterministically divide X into disjoint parts X_1 and X_2. Similarly, divide Y into Y_1 and Y_2. Informally: X_1 and Y_1 coincide as sets, X_2 and Y_2 have no common members. If one of the parts X_1 and Y_1 is empty then the other is required to be empty too. In addition, like the case of bags, X_2 and Y_2 are required to contain no common variables. Consider two cases.
   (a) *Both* X_1 *and* Y_1 *are empty.* Add the equation $u = \{X_2 \cup Y_2 \,|\, u\}$ and reduce S using $z = u$.
   (b) *Both* X_1 *and* Y_1 *are non-empty.* Don't-know nondeterministically generate a minimal set correspondence E between X_1 and Y_1. Then transform S as follows.
      i. Add the equation $z = \{Y_1 \,|\, u\}$.
      ii. If at least one of X_2 and Y_2 is non-empty, add $u = \{X_2 \cup Y_2 \,|\, u\}$.

iii. Reduce S using E.

2. *The extensions have the forms* $z = \{X \mid u\}$ *and* $z = \{Y \mid v\}$*, where the variables* u *and* v *are different.* Don't-know nondeterministically divide X into disjoint parts X_1 and X_2 and divide Y into Y_1 and Y_2. As above, the division is required to satisfy the following conditions. First, if one of the parts X_1 and Y_1 is empty then the other is empty too. Second, X_2 and Y_2 contain no common variables. Consider two cases.
   (a) *Both* X_1 *and* Y_1 *are empty.* Add the equations $z = \{X_2 \cup Y_2 \mid w\}$, $u = \{Y_2 \mid w\}$ and $v = \{X_2 \mid w\}$, where w is a new variable.
   (b) *Both* X_1 *and* Y_1 *are non-empty.* Transform S as follows.
      i. Add equation $z = \{X_2 \cup Y_1 \cup Y_2 \mid w\}$, where w is a new set variable.
      ii. If X_2 is non-empty, add the equation $v = \{X_2 \mid w\}$. Otherwise reduce S using $v = w$.
      iii. If Y_2 is non-empty, add the equation $u = \{Y_2 \mid w\}$. Otherwise reduce S using $u = w$.
      iv. Don't-know nondeterministically generate a minimal set correspondence E between X_1 and Y_1 and reduce S using E.

Rule 4 (Function Failure). If S contains equations $x = f(x_1, \ldots, x_m)$ and $x = g(y_1, \ldots, y_n)$ where $m, n \geq 0$ and f and g are distinct function symbols, transform S into $\perp$.

Rule 5 (Type Failure). If the system S contains an equation $x = t$ such that $t \not\sqsubseteq x$, transform S into $\perp$.

3.3 Algorithms

Solved form. A system S of equations is said to be *in solved form* if no rule is applicable to S. In particular, $\perp$ is in solved form. It is easy to see that a system S is in solved form if and only if (i) S does not contain two different equations $x = s$ and $x = t$, and (ii) for any equation $x = t$ in S, we have $t \sqsubseteq x$.

Transformation algorithm. *The transformation algorithm* don't-care nondeterministically chooses Rules 1–5 and applies them to an input system until no rule is applicable. Thus, the transformation algorithm is a nondeterministic algorithm that transforms any input system into a system in solved form.

Variable dependency graph. We introduce one more graph associated with a system S. The *variable dependency graph* of S is the graph whose nodes are variables occurring in S and whose edges are defined as follows. There is an edge coming from x to y if S contains at least one of the following equations:

$x = f(y_1, \ldots, y_n)$, where $f \in \mathcal{F}$ and y is one of $y_1, \ldots, y_n$;
$x = \{\!|y_1, \ldots, y_n \mid z|\!\}$ such that y is one of $y_1, \ldots, y_n, z$;
$x = \{y_1, \ldots, y_n \mid z\}$ such that y is one of $y_1, \ldots, y_n$.

Occur-check algorithm. The occur-check algorithm is applied to a system S in solved form. If the variable dependency graph of S has a cycle, S is transformed into $\bot$. Otherwise, the algorithm does not change S.

Unification algorithm. The unification algorithm is the composition of the transformation algorithm and the occur-check algorithm.

Theorem 2 (running time). *The unification algorithm runs in nondeterministic polynomial time.*

Theorem 3 (soundness and completeness). *Let $S_1, \ldots, S_n$ be all nondeterministic results of the application of the unification algorithm to a system S. Then*

1. *Any solution to S_i is also a solution to S.*
2. *For any solution ν to S, there is a solution ν_i to some S_i such that ν and ν_i coincide on all variables of S.*

It follows from these theorems and NP-hardness [28] that the unifiability problem for $\mathcal{HU}^+$ is NP-complete (as we note in Section 4 below, this fact also follows from results on AC and ACI unification).

Usually, unification problems are formulated in terms of finding *unifiers*, i.e. substitutions that make two terms equal. A set Σ of unifiers of a system S is said to be *complete* if for every unifier of S, the set Σ contains a more general unifier of S. It is straightforward to extract unifiers from our algorithm. Namely, every nondeterministic branch leads to either $\bot$ or a system in solved form. Such a system represents a unifier called a *resulting unifier* for S, for details see [19]. The completeness of our algorithm provides that all resulting unifiers for S form a complete set of unifiers of S. Since every branch gives us at most one unifier, we obtain an upper bound on the minimal cardinality of a complete set of unifiers.

Theorem 4 (upper bound). *For any system S, the set of all resulting unifiers for S is a complete set of unifiers. An upper bound on its cardinality is $2^{O(n \log n)}$ where n is the size of S.*

It follows from [9] and [19] that $2^{O(n \log n)}$ is also a lower bound. To establish this lower bound, it is enough to consider unification for flat sets. Thus, $2^{O(n \log n)}$ is a tight bound.

Minimality. We say that a complete set Σ of unifiers is *minimal* if for every pair of unifiers in Σ, none of them is more general than the other. A unification algorithm is said to be *optimal for a system S* if the algorithm yields a minimal complete set of unifiers of S. Our algorithm (as well as all other known algorithms) is not optimal in general, but it is optimal for a number of important special cases. First, consider the following equations on sets:

$$\{s_1, \ldots, s_m \mid x\} = \{t_1, \ldots, t_n \mid x\} \tag{1}$$

$$\{s_1, \ldots, s_m \mid x\} = \{t_1, \ldots, t_n \mid y\} \tag{2}$$

$$\{s_1, \ldots, s_m \mid x\} = \{t_1, \ldots, t_n\} \tag{3}$$

$$\{s_1, \ldots, s_m\} = \{t_1, \ldots, t_n\} \tag{4}$$

where x, y are different variables not occurring in $s_1, \ldots, s_m, t_1, \ldots, t_n$, and $s_1, \ldots, s_m, t_1, \ldots, t_n$ are either variables or ground terms without occurrences of bag or set constructors, not necessarily different. Systems (1)–(4)are not, in general, systems of simple or isolated equations without duplication, and there are several ways of transforming them into such systems.

Assuming that equations (1)–(4) are preprocessed using the algorithm presented in Lemma 1, we obtain

Theorem 5. *The unification algorithm is optimal for any system consisting of one equation of the form* (1)–(4).

The proof is not difficult and is based on properties of minimal set correspondences. Note that (1)–(4) contain all equations considered in [9], for which optimality of a different algorithm is proved.

Similarly, using properties of minimal bag correspondences we can prove optimality for special cases of bag equation. Consider the following equations on bags:

$$\{|s_1, \ldots, s_m \mid x|\} = \{|t_1, \ldots, t_n \mid x|\} \tag{5}$$
$$\{|s_1, \ldots, s_m \mid x|\} = \{|t_1, \ldots, t_n \mid y|\} \tag{6}$$
$$\{|s_1, \ldots, s_m \mid x|\} = \{|t_1, \ldots, t_n|\} \tag{7}$$
$$\{|s_1, \ldots, s_m|\} = \{|t_1, \ldots, t_n|\} \tag{8}$$

with the same conditions as for (1)–(4).

Assuming that equations (5)–(8) are preprocessed using the algorithm presented in Lemma 1, we obtain

Theorem 6. *The unification algorithm is optimal for any system consisting of one equation of the form* (5)–(8).

Note that having systems without duplication is esential for the optimality. For example, our algorihtm is optimal when the equation $\{|x, c|\} = \{|y, c|\}$ is translated into $\{|x, z|\} = \{|y, z|\}, z = c$ but is not optimal when this equation is translated into $\{|x, z|\} = \{|y, u|\}, z = c, u = c$.

4 Related results

AC and ACI unification. AC and ACI unification is more general than bag and set unification. First results relevant to bag and set unification appeared in the automated deduction community as results on AC- and ACI-unification algorithms [44, 28, 25, 10, 33, 18]. These algorithms deal with the first three of the following equality axioms:

$$(A) \quad (x \cup y) \cup z = x \cup (y \cup z)$$
$$(C) \quad x \cup y = y \cup x$$
$$(I) \quad x \cup x = x$$
$$(1) \quad x \cup \{\} = x$$

All the four axioms give an axiomatization of finite sets in the signature consisting of the set union $\cup$ and the empty set $\{\}$. By removing (I), we obtain an axiomatization of bags in the signature consisting of the bag union and the empty bag. Both axiomatizations are complete in the sense that every valid equation on sets or bags is a logical consequence of these axiomatizations. It is known that both AC1-unifiability with constants is NP-complete: NP-hardness for a very simple special case is proved in [28] and inclusion in NP is proved in [10]. ACI1-unifiability with constant can be solved in polynomial time [29]. It follows from the results on combination of unification algorithms [11, 42, 17] that for the theory combining AC1 and ACI1, the unifiability problem is in NP [12, Theorem 5.2].

The influence of results on AC1 and ACI1-unification on unification problems with bag and set constructors was largely ignored (for which we are also to blame, see the preliminary version of this paper [19]). For example, one can read in [9]:

> "by dealing with nested sets we can solve set-unification problems that cannot be expressed using ACI unification; for instance $\{x, \{y, \{\emptyset, z\}\}\} = \{\{z\}, w\}$."

However, it is not hard to see that the set constructor can be defined from the union $\cup$ by using the additional singleton set constructor $\{\dots\}$. Indeed, we have $\{x \mid y\} = \{x\} \cup y$. Thus, the unification problem for $\mathcal{HU}^+$ (as well as for the domains of [22, 45, 21]) can be implemented as unification in the theory combining AC1 and ACI1. This fact has been noted for example in [7] and in [43]. In particular, by [12, Theorem 5.2], the unifiability in the domain combining bags, sets and trees is in NP.

There are various motivations for using the signature with the bag and set constructors instead of the union. Our motivation is explained by Theorem 1: bag and set constructors are enough to express any computable function on the universe with bags and sets. In addition, known AC1- and ACI1-unification algorithms adapted for $\mathcal{HU}^+$ are too complex compared to our algorithm (for example, they use solutions to systems of Diophantine equations and one should also count the complexity added by the techniques of combining unification algorithms). In the signature with the union, there is a double-exponential lower bound on cardinalities of minimal complete sets of unifiers for bag equations [30]. Using bag and set constructors instead of the union, our algorithm gives a single-exponential upper bound even for the combined domain. This bound is new and does not follows from other results in the literature.

Other domains for bags and finite sets. Our unification algorithm can be modified in a straightforward way to deal with other data models for bags and sets. In particular, [19] defines a *typed universe* (in the spirit of [1] or [34]) and a *universe of colored bags and colored sets* (similar to [22]). The corresponding modifications of the algorithm are sketched too.

Complexity of nonrecursive logic programs with bags and sets. As it is shown in [20], if solvability of equations over a domain is in NP, then

the query evaluation problem for nonrecursive Horn clause logic programs over this domain is in NEXP (see [20] for precise definitions). Therefore, the query evaluation problem for such programs over bags and/or sets and/or trees is in NEXP (as it follows from another result of [20], this problem is also NEXP-hard). This bound does not hold for nonrecursive logic programs with negation: in this case the query evaluation problem may be nonelementary or even undecidable already for domains with sets [47].

Comparison with other algorithms. Our approach is close to the approach of [22] where a set unification algorithm has been proposed. However, incorporating bags in a logic programming language is stated as an open problem in [22, page 34]. Also, there is much in common with other known unification algorithms for bags and sets built with the bag and set constructors [21–23, 45, 9]. It is natural to compare them in important special cases of flat bags and sets, for example when solving equations of the form

$$\{x_1, \ldots, x_n \mid x\} = \{y_1, \ldots, y_m \mid y\},$$

where $x_1, \ldots, x_n, x, y_1, \ldots, y_m, y$ are variables. As it was mentioned, in this case the minimal cardinality of complete sets of unifiers may be exponential.

Set unification in [22] is not optimal in this case. In [45] the special case of flat sets is treated in detail. The algorithm of [45] tries to take care of information about unifiability of elements of sets. This idea is interesting and natural, but unfortunately the algorithm of [45] does not work for embedded sets, despite the claim to do so. For the flat case, the algorithm of [45] may be better than all known algoritms in the number of computed unifiers, but it is not clear how to modify it for embedded sets (for example, because it checks unifiability of subterms and uses *most specific generalizations* that do not exist for sets). The set unification algorithm of [9] also uses optimizations for the flat case. It is proved that the algorithm computes minimal complete sets of unifiers for a number of special cases of flat sets. All these special cases are covered by cases (1)–(4). We achieve minimality by using minimal bag and set correspondences which is a new idea and allows us to get an optimal algorithm both for cases (1)–(4) of sets and (5)–(8) of bags. Note that the algorithms of [45, 9] apply substitutions explicitly and thus use exponential space, though we guess they can be modified into nondeterministic polynomial-time ones.

It seems that optimal algorithms for flat bags have not been considered in other papers. For example, the algorithm of [23] is not optimal for the bag equation $\{|x, x \mid y|\} = \{|x \mid z|\}$, while our algorithm is optimal for such equations (case (6) of Theorem 6). Although [23] asserts that

> "The axiomatizations presented can easily be combined in order to obtain axiomatic theories capable to deal with any subset of the collection of proposed structures. Moreover, the unification algorithms presented in the next section can easily be merged to solve the unification problem relative to such combined context",

no details are given.

Directions of further research. It is interesting to consider constraint logic programming over bags and finite sets which uses more powerful primitives than just the bag and set constructors. For examples, what is the complexity of constraint satisfaction when we also have primitives like $\cup$ or $\subseteq$? Set constraints have recently received a considerable attention in connection with program verification, but mostly for infinite sets and constraints that are less relevant to databases or logic programs, see e.g. [5, 14, 39, 13] and the survey [40]).

Also, it is interesting to consider a suitable representation of graphs (up to isomorphism) and the complexity of unification over graphs. This may be useful for dealing with semistructured data [4].

Acknowledgements

The first author was supported by grants from the Swedish Royal Academy of Sciences, the Swedish Institute, INTAS, and RFBR. The second author was supported by a TFR grant. We thank Franz Baader and Klaus Schulz for their comments on the combination of unification algorithms.

References

1. S. Abiteboul and C. Beeri. The power of languages for the manipulation of complex values. *VLDB Journal*, 4:727–794, 1995.
2. S. Abiteboul and S. Grumbach. Col: A logic-based language for complex objects. In J. Schmidt, S. Ceri, and M. Missikoff, editors, *Advances in Database Technology - EDBT'88. Proceedings of the International Conference on Extending Database Technology*, volume 303 of *Lecture Notes in Computer Science*, pages 271–293, Venice, Italy, March 1988. Springer Verlag.
3. S. Abiteboul and S. Grumbach. A rule-based language with functions and sets. *ACM Transactions on Database Systems*, 16(1):1–30, 1991.
4. S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener. The Lorel query language for semistructured data. *Journal of Digital Libraries*, 1(1), 1996.
5. A. Aiken. Set constraints: Results, applications and future directions. In A. Borning, editor, *Proceedings of the Workshop on Principles and Practice of Constraint Programming*, volume 874 of *Lecture Notes in Computer Science*, pages 326–335. Springer Verlag, 1994.
6. Andréka and Németi. A generalized completeness of Horn clause logic seen as a programming language. *Acta Cybernetica*, 4:3–10, 1978.
7. V.M. Antimirov, A. Degtyarev, V.S. Procenko, A. Voronkov, and M.V. Zakharjashchev. Mathematical logic in programming (in Russian). In Yu.I. Yanov and M.V. Zakharjashchev, editors, *Mathematical Logic in Programming*, pages 331–407. Mir, Moscow, 1991.
8. K.R. Apt. Logic programming. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Methods and Semantics, chapter 10, pages 493–574. Elsevier Science, Amsterdam, 1990.
9. P. Arenas-Sánchez and A. Dovier. A minimality study for set unification. *Journal of Functional and Logic Programming*, 1997(7), December 1997.

10. F. Baader and W. Büttner. Unification in commutative and idempotent monoids. *Theoretical Computer Science*, 56:345–352, 1988.
11. F. Baader and K.U. Schulz. Unification in the union of disjoint equational theories: Combining decision procedures. In D. Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 50–65, Saratoga Springs, NY, USA, June 1992.
12. F. Baader and K.U. Schulz. Unification in the union of disjoint equational theories: Combining decision procedures. *Journal of Symbolic Computations*, 21:211–243, 1996.
13. L. Bachmair and H. Ganzinger. Set constraints with intersection. In G. Winskel, editor, *Proc. IEEE Conference on Logic in Computer Science (LICS)*, pages 362–372. IEEE Computer Society Press, 1997.
14. L. Bachmair, H. Ganzinger, and U. Waldmann. Set constraints are the monadic class. In *Proc. IEEE Conference on Logic in Computer Science (LICS)*, pages 75–83. IEEE Computer Society Press, 1993.
15. C. Beeri and Y. Kornatzky. A logical query language for hypermedia systems. *Information Sciences*, 77:1–38, 1994.
16. C. Beeri, S. Naqvi, O. Shmueli, and S. Tsur. Set constructors in a logic database language. *Journal of Logic Programming*, 10:181–232, 1991.
17. A. Boudet. Combining unification algorithms. *Journal of Symbolic Computations*, 16:597–626, 1993.
18. A. Boudet, E. Contejean, and C. Marcé. AC-complete unification and its application to theorem proving. In H. Ganzinger, editor, *7th International Conference on Rewriting Techniques and Applications*, volume 1103 of *Lecture Notes in Computer Science*, pages 18–32. Springer Verlag, 1996.
19. E. Dantsin and A. Voronkov. Bag and set unification. UPMAIL Technical Report 150, Uppsala University, Computing Science Department, November 1997.
20. E. Dantsin and A. Voronkov. Complexity of query answering in logic databases with complex values. In S. Adian and A. Nerode, editors, *Logical Foundations of Computer Science. 4th International Symposium, LFCS'97*, volume 1234 of *Lecture Notes in Computer Science*, pages 56–66, Yaroslavl, Russia, July 1997.
21. A. Dovier. *Computable Set Theory and Logic Programming.* PhD thesis, Universita' degli Studi di Pisa, dip. di Informatica, March 1996.
22. A. Dovier, E.G. Omodeo, E. Pontelli, and G. Rossi. {log}: A language for programming in logic with finite sets. *Journal of Logic Programming*, 28(1):1–44, 1996.
23. A. Dovier, A. Policriti, and G. Rossi. Integrating lists, multisets and sets in a logic programming framework. In F. Baader and K.U. Schulz, editors, *Frontiers of Combining Systems*, pages 303–321. Kluwer, 1996.
24. F. Fages. Associative-commutative unification. *Journal of Symbolic Computations*, 3(3), 1987.
25. Fortenbacher. An algebraic approach to unification under associativity and commutativity. *Journal of Symbolic Computations*, 3(3):217–229, 1987.
26. S. Grumbach and V. Vianu. Tractable query languages for complex object databases. *Journal of Computer and System Sciences*, 51(2):149–167, 1995.
27. P. Kanellakis, G.M. Kuper, and P.Z. Revesz. Constraint query languages. *Journal of Computer and System Sciences*, 51:26–52, 1995.
28. D. Kapur and P. Narendran. NP-completeness of the set unification and matching problems. In J. Siekmann, editor, *Proc. 8th CADE*, volume 230 of *Lecture Notes in Computer Science*, pages 489–495, 1986.

29. D. Kapur and P. Narendran. Complexity of unification problems with associative-commutative operators. *Journal of Automated Reasoning*, 9(2):261–288, 1992.
30. D. Kapur and P. Narendran. Double-exponential complexity of computing a complete set of *AC*-unifiers. In *Proc. IEEE Conference on Logic in Computer Science (LICS)*. IEEE Computer Society Press, 1992.
31. G.M. Kuper. Logic programming with sets. *Journal of Computer and System Sciences*, 41:44–64, 1990.
32. N. Leone and P. Rullo. Ordered logic programming with sets. *Journal of Logic and Computation*, 3(6):621–642, 1993.
33. P. Lincoln and J. Christian. Adventures in associative-commutative unification (a summary). *Journal of Logic and Computation*, 8(1/2):217–240, 1989.
34. M. Liu. Relationlog: a typed extension to datalog with sets and tuples. *Journal of Logic Programming*, 35(1):1–30, 1998.
35. J.W. Lloyd. *Foundations of Logic Programming (2nd edition)*. Springer Verlag, 1987.
36. M.J. Maher. A CLP view of logic programming. In *Proc. Conf. on Algebraic and Logic Programming*, volume 632 of *Lecture Notes in Computer Science*, pages 364–383, October 1992.
37. M.J. Maher. A logic programming view of CLP. In *International Conference on Logic Programming*, pages 737–753, 1993.
38. A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, 1982.
39. D.A. McAllister, R. Givan, C. Witty, and D. Kozen. Tarskian set constraints. In *Proc. IEEE Conference on Logic in Computer Science (LICS)*, pages 138–147, New Brunswick, New Jersey, July 1996. IEEE Computer Society Press.
40. L. Pacholski and A. Podelski. Set constraints: a pearl in research on constraints. In G. Smolka, editor, *Proceedings of the Third International Conference on Constraint Programming*, volume 1330 of *Lecture Notes in Computer Science*. Springer Verlag, 1997.
41. C. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.
42. M. Schmidt-Schauß. Unification in a combination of arbitrary disjoint equational theories. *Journal of Symbolic Computations*, 1990. Special issue on Unification.
43. O. Shmueli, S. Tsur, and C. Zaniolo. Compilation of set terms in the logic data language (LDL). *Journal of Logic Programming*, 12(1):89–119, 1992.
44. M.E. Stickel. A complete unification algorithm for associative-commutative functions. *Journal of the Association for Computing Machinery*, 28(3):423–434, 1981.
45. F. Stolzenburg. Membership constraints and complexity in logic programming with sets. In F. Baader and K.U. Schulz, editors, *Frontiers of Combining Systems*, pages 285–302. Kluwer, 1996.
46. K. Vadaparty. On the power of rule-based languages with sets. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 26–36, 1991.
47. S. Vorobyov and A. Voronkov. Complexity of nonrecursive logic programs with complex values. In *PODS'98*, pages 244–253, Seattle, Washington, 1998. ACM Press.
48. A. Voronkov. Logic programming with bounded quantifiers. In A. Voronkov, editor, *Logic Programming*, volume 592 of *Lecture Notes in Artificial Intelligence*, pages 486–514. Springer Verlag, 1992.
49. A. Voronkov. On computability by logic programs. *Annals of Mathematics and Artificial Intelligence*, 15(3,4):437–456, 1995.

Categorical Models of Explicit Substitutions

Neil Ghani, Valeria de Paiva, and Eike Ritter

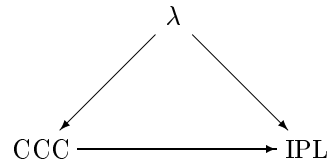
University of Birmingham, England

Abstract. Indexed categories provide models of cartesian calculi of explicit substitutions. However, these structures are inherently non-linear and hence cannot be used to model linear calculi of explicit substitution. This paper replaces indexed categories with pre-sheaves, thus providing a categorical semantics covering both the linear and cartesian cases. We further justify our models by proving soundness and completeness results.

1 Introduction

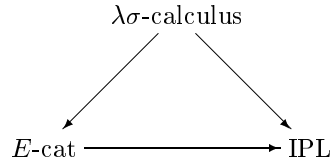
Functional programming languages are based on the λ-calculus and model computation by β-reduction $(\lambda x.t)u \Rightarrow t[u/x]$. This process can duplicate the redexes in u and hence is highly inefficient from the implementational perspective. Abstract machines avoid this problem by reducing terms in an environment — the contraction of a β-redex creates a new substitution which is added to the existing environment and only evaluated when needed. In order to study such machines, *calculi of explicit substitutions* incorporate substitutions directly into the syntax of the λ-calculus rather than treating them as meta-theoretic operations.

Category theory aids the design of abstract machines [7,16] by providing a semantics for explicit substitutions based upon the Curry-Howard triangle relating typed λ-calculi, intuitionistic logic and their categorical models. The best known example relates the simply typed λ-calculus, the positive fragment of intuitionistic propositional logic (IPL) and cartesian closed categories (CCC's).



Indexed categories seem to provide the correct semantic framework for cartesian calculi of explicit substitutions by interpreting substitutions in the base, terms in the fibres and the application of a substitution to a term via re-indexing. Indexed categories also arise in the semantics of dependent types [8] and also in models of the simply-typed λ-calculus where not all objects are exponentiable [12]. Unfortunately indexed categories cannot be used as models of linear calculi of explicit substitution as identities cannot be defined in the fibres. Dropping the identities from our models leads to what we call E-categories and hence our triangle now looks like:

W. Thomas (Ed.): FOSSACS'99, LNCS 1578, pp. 197-211, 1999.



where the $\lambda\sigma$-calculus [1](a simply-typed λ-calculus with explicit substitution) is derived as the internal language of E-categories — this ensures that we have a Curry-Howard triangle. As we shall see, E-categories are essential in generalising the monoidal adjunction semantics of linear logic to cover explicit substitutions.

Now we turn to linear logic. The Curry-Howard correspondence between the linear λ-calculus, the standard categorical models and intuitionistic linear logic is described, for example, in [4]. These categorical models are essentially symmetric monoidal closed categories (SMCCs) with extra structure to model the modality !. However, categorical combinators based on SMCCs ([13] [14]) are not adequate for modelling resource allocations, the idea behind linear functional languages.

Linear categorical abstract machines are designed to implement linear logic and we require linear analogues of the modifications described above. In particular, we want a linear λ-calculus extended with explicit substitutions, a categorical model for the calculus and a Curry-Howard relationship between them. The calculus appears in full in [10] and this paper concentrates on the more refined categorical models for the linear λ-calculus extended with explicit substitutions.

Indexed categories cannot be used as models of linear calculi of explicit substitution as they are an inherently non-linear structure. Asking that the fibres form a category requires identities which in turn corresponds to weakening which is not admissible in linear λ-calculi. Hence we alter the notion of an indexed category to a presheaf (*i.e.*, a functor with **Set** as codomain rather than **Cat**), and call this structure a *linear context-handling category. Cartesian context-handling categories* analogously model cartesian calculi of explicit substitutions.

The tensor, unit and linear implication are then modelled by adding natural isomorphisms to the "fibres" of linear context handling categories — we call these structures L-categories. Modelling contexts by structure in the base and the logical connectives by structure in the fibres distinguishes our models from the usual SMCC's where the same semantic structure is used to model both the behaviour of contexts and the tensor connective. Similarly, intuitionistic implications and conjunctions are modelled by adding structure to the fibres of a cartesian context handling category obtaining the previously mentioned E-categories. The modalities of linear logic are modelled by a monoidal adjunction between (the bases of) an L-category and an E-category.

2 Context Handling Categories

The traditional categorical semantics of explicit substitutions is based on *indexed categories*, ie a *base* category $\mathcal{B}$ and a contravariant functor $E\colon \mathcal{B}^{op} \to \mathbf{Cat}$. The objects of $\mathcal{B}$ model the contexts, the morphisms of $\mathcal{B}$ interpret the explicit substitutions and the fibres interpret the types and terms of the calculus. Unfortunately, indexed categories do not generalise to the linear setting as the

identity on A in a fibre $E(\Gamma)$ corresponds to the non-linear typing judgement $\Gamma, x{:}\, A \vdash x{:}\, A$ [16, 8]. This paper replaces indexed categories with pre-sheaves, thus providing a categorical semantics covering both the linear and cartesian cases. That is, we change the codomain of E from **Cat** to **Set**, thus removing the identities from the fibres. As motivate for our definition, consider the most primitive form of a linear or cartesian calculus of explicit substitutions. Such a calculus has the following components:

- **Types**: A set of types $\mathcal{T}$.
- **Contexts**: Contexts are obtained by "glueing", in a linear or cartesian manner, variable-type pairs $(x{:}\, A)$ together.
- **Substitutions**: Given contexts Γ and Δ, there is a collection of *explicit substitutions* which are judgements of the form $\Gamma \vdash f : \Delta$
- **Terms**: Given a context Γ and type $B \in \mathcal{T}$, there is a collection of *terms* which are judgements $\Gamma \vdash t : B$. Applying a substitution $\Gamma \vdash f : \Delta$ to a term $\Delta \vdash t : A$ results in another term $\Gamma \vdash f * t{:}\, A$.

These properties can be captured by a presheaf $L{:}\, \mathcal{B}^{op} \longrightarrow \mathbf{Set}^{\mathcal{T}}$ with additional structure to capture the formation and behaviour of explicit substitutions:

Definition 1 *Let $\mathcal{B}$ be a (symmetric) monoidal category with distinguished collection of objects $\mathcal{T} \subseteq |\mathcal{B}|$. A* linear context handling category *is a functor $L{:}\, \mathcal{B}^{op} \to \mathbf{Set}^{\mathcal{T}}$ such that for each $A \in \mathcal{T}$ there exists a natural isomorphism*

$$\mathtt{Sub}_A{:}\, L(-)_A \cong \mathsf{Hom}_{\mathcal{B}}(-, A){:}\, \mathtt{Term}_A$$

Each component of Definition 1 corresponds to part of the description of a term assignment system given previously:

- **The Base** $\mathcal{B}$: The base $\mathcal{B}$ of a context handling category models contexts as objects and substitutions as morphisms — types are treated semantically as singleton contexts [1]. That $\mathcal{B}$ forms a category means that substitutions can be sequentially composed and there is an identity substitution. Contexts and substitutions can be put into parallel and there is a an empty context — these features are described by the monoidal structure on $\mathcal{B}$.
- **The Functor** L: The functor L associates to each context Γ and type A a set, written $L(\Gamma)_A$, which we think of as the terms of type A in context Γ. Given a substitution $f : \Gamma \to \Delta$, and any type A, the contravariance of L gives a function $L(f)_A{:}\, L(\Delta)_A \to L(\Gamma)_A$. This re-indexing is exactly what is used to model the application of a substitution to a term.
- **The Natural Transformations** $\mathtt{Sub}_A$ **and** $\mathtt{Term}_A$**:** $\mathtt{Sub}_A$ describes the formation of new substitutions by converting elements in the fibres to morphisms in the base, ie taking a term t and constructing the substitution $\langle t/x \rangle$ [2]. By the Yoneda-Lemma, $\mathtt{Term}_{\mathtt{A}}$ can be replaced by elements $\mathtt{Var}_A \in$

[1] although we simplify notation by sometimes writing A for $x : A$

[2] where x is the variable associated to the singleton context A

$L(A)_A$ and $\mathtt{Term}_{\mathtt{A}}(f)$ is then given by $f * \mathtt{Var}_{\mathtt{A}}$ — thus $\mathtt{Term}_A$ evaluates a substitution at a variable. The condition that $\mathtt{Sub}$ and $\mathtt{Term}$ are natural isomorphisms is then replaced by the equations

$$\mathtt{Sub}_A(t) * \mathtt{Var}_A = t \qquad \mathtt{Sub}_A(\mathtt{Var}_A) = \mathsf{Id}$$

In order to model a calculus of *cartesian* contexts we use *cartesian context handling* categories whose definition only differs in requiring the monoidal structure in the base is actually a product so that weakening and contraction can be interpreted. This notion of a cartesian handling of contexts is implicit in most of the work on categorical modelling of higher-order typed calculi [8] [11].

Definition 2 *Let $\mathcal{B}$ be a cartesian category with distinguished collection of objects $\mathcal{T} \subseteq |\mathcal{B}|$. A* cartesian context handling category *is a functor $E\colon \mathcal{B}^{op} \to \mathbf{Set}^{\mathcal{T}}$ such that for each $A \in \mathcal{T}$ there exists a natural isomorphism*

$$\mathtt{Sub}_A\colon E(-)_A \cong \mathsf{Hom}_{\mathcal{B}}(-, A)\colon \mathtt{Term}_A$$

Notation: We use $\Gamma, \Delta, \cdots$ as generic objects in $\mathcal{B}$, $f, g, \ldots$ as generic morphisms in $\mathcal{B}$ and $A, B, C, \ldots$, as generic elements of $\mathcal{T}$. We write $f*-$ for $E(f)$ (or $L(f)$) for the functor on morphisms. When $\mathcal{B}$ is monoidal the unit is denoted $[\,]$, the tensor product of objects $\Gamma_1, \ldots, \Gamma_n$ is denoted $(\Gamma_1, \ldots, \Gamma_n)$ and similarly the tensor product of two morphisms f and g is written (f, g). In addition, if $\mathcal{B}$ is cartesian, we write Fst and Snd for the two projections.

Should $\mathtt{Sub}$ and $\mathtt{Term}$ be isomorphisms? The equation $\mathtt{Sub}_A(t) * \mathtt{Var}_A = t$ formalises our understanding that $x[t/x] = t$. However, the other equation, namely that if f is a substitution for the variable x, then $f = (f * x)/x$, does not carry the same force, and *intensional* definitions requiring only a retraction $\mathtt{Term}_A \circ \mathtt{Sub}_A = \mathsf{Id}$ could be considered. The situation would be analogous to the intensionality of function spaces: In the same way as two functions are not intensionally equal if they produce the same result when applied to the same arguments, two substitutions are not intensionally equal if applied to the same variable they produce the same result. The formal definition is

Definition 3 *An intensional (cartesian) context category consists of the same data as a (cartesian) context handling category but the natural transformations $\mathtt{Sub}_A$ and $\mathtt{Term}_A$ need only form a retraction $\mathtt{Term}_A \circ \mathtt{Sub}_A = \mathsf{Id}$.*

The next lemma proves that an intensional context handling category where $\mathtt{Sub}_A(\mathtt{Var}_A) = \mathsf{Id}$ is actually extensional. Since we think of $\mathtt{Sub}_A(\mathtt{Var}_A)$ as the substitution $[x/x]$, this is rather a mild condition and dropping it, as intensional structures require, seems counter-intuitive.

Proposition 4 *Let L be an intensional context handling category with types $\mathcal{T}$. If for all $A \in \mathcal{T}$, $\mathtt{Sub}_A(\mathtt{Var}_A) = \mathsf{Id}_A$, then L is a linear context handling category.*

Proof. If $A \in \mathcal{T}$, then we show $\mathtt{Sub}_A \circ \mathtt{Term}_A = \mathsf{Id}$. So let $f : \Gamma \to A$. Then

$$f = f; \mathsf{Id} = f; \mathtt{Sub}_A(\mathtt{Var}_A) = \mathtt{Sub}(f * \mathtt{Var}_A) = \mathtt{Sub} \circ \mathtt{Term}(f)$$

where the first equality holds by definition, the second by assumption, the third is the naturality of $\mathtt{Sub}_A$ and the fourth by definition.

The natural isomorphism between $\mathsf{Hom}(-, A)$ and $E(-)_A$ (or $L(-)_A$ in the linear setting) has several consequences. Firstly, the fibres of a context handling category are determined up to isomorphism by the base of the category. Secondly, all substitutions are *extensional*, ie morphisms are determined by their effects on terms: $f = g$ iff for all terms t, $f * t = g * t$. Finally, models of λ-calculi based on context handling categories can be compared with the standard categorical models by constructing an "internal category" from the fibres. The objects of this category are the elements of $\mathcal{T}$ and the set of morphisms from A to B is the fibre $E(A)_B$. The identity on A is the term $\mathtt{Var}_A$ and the composition of two morphisms $t \in E(A)_B$ and $s \in E(B)_C$ is given by $\mathtt{Sub}(t) * s$. Clearly this internal category is isomorphic to the full subcategory of $\mathcal{B}$ whose objects are $\mathcal{T}$.

3 The Cartesian Model

Context handling categories model the basic features of explicit substitutions, namely the ability to form substitutions from terms, put them in parallel and apply them to terms. This structure is insufficient to model a calculus of explicit substitutions as no mention is made of the connectives. We now consider a canonical extension of the simply typed λ-calculus with explicit substitutions and the extra structure required to model it. Our presentation varies slightly from the original [1], eg we use names rather than De Bruijn numbers.

3.1 The $\lambda\sigma$-calculus

The types of the $\lambda\sigma$-calculus are ground types, the unit type 1, function types $A \to B$ and product types $A \times B$. The raw expressions of $\lambda\sigma$ are:

$$t ::= x \mid \lambda x{:}\, A.t \mid tt \mid \langle t, t\rangle \mid \pi_i t \mid \bullet \mid f * t$$
$$f ::= \langle\rangle \mid \langle f, t/x\rangle \mid f; f$$

where x is a variable. The term $f * t$ represents the application of the explicit substitution f to the term t and $\bullet$ represents the canonical element of the unit type. The substitution $\langle\rangle$ should be thought of as a substitution of variables for themselves, while $\langle f, t/x\rangle$ represents the parallel composition of the substitution f with the substitution of the term t for the variable x. Finally, $f; g$ represents the composition of the substitutions f and g and models iterated substitution.

Contexts are lists $x_1{:}\, A_1, \ldots x_n{:}\, A_n$ where the x's are distinct variables and the A's are types — the domain of the context is $\{x_1, \ldots, x_n\}$ and we write $\Gamma \subseteq \Gamma'$ if the domain of Γ is contained in the domain of Γ'. The $\lambda\sigma$-calculus has term judgements $\Gamma \vdash t : A$ and substitution judgements $\Gamma \vdash f : \Delta$ — these judgements are generated by the inference rules of Table 1. The inference rules for declaring variables and the introduction and elimination rules for function spaces and conjunctions are standard. All the free variables of t are bound in $f * t$ and similarly all the free variables of g are bound in $f; g$. For a full presentation of the meta-theory of $\lambda\sigma$ see [1, 15].

Table 1. Typing Judgements for the $\lambda\sigma$-calculus

– Term Judgements

$$\frac{x{:}A \text{ declared in } \Gamma}{\Gamma \vdash x{:}A} \qquad \frac{\Gamma \vdash f : \Delta \qquad \Delta \vdash t : A}{\Gamma \vdash f * t : A}$$

$$\frac{\Gamma, x{:}A \vdash t{:}B}{\Gamma \vdash \lambda x{:}A.t{:}A \to B} \qquad \frac{\Gamma \vdash t{:}A \to B \qquad \Gamma \vdash u{:}A}{\Gamma \vdash tu{:}B}$$

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash u : B}{\Gamma \vdash \langle t, u\rangle : A \times B} \qquad \frac{\Gamma \vdash t : A_1 \times A_2}{\Gamma \vdash \pi_i(t) : A_i} \qquad \frac{}{\Gamma \vdash \bullet : 1}$$

– Substitution Judgements

$$\frac{\Gamma' \subseteq \Gamma}{\Gamma \vdash \langle\rangle : \Gamma'} \qquad \frac{\Gamma \vdash f{:}\Delta \qquad \Gamma \vdash t{:}A}{\Gamma \vdash \langle f, t/x\rangle : \Delta, x : A} \qquad \frac{\Gamma \vdash f{:}\Delta \qquad \Delta \vdash g{:}\Psi}{\Gamma \vdash f; g : \Psi}$$

where in the second rule x is not in the domain of Δ.

3.2 Modelling the $\lambda\sigma$-calculus in an E-category

Cartesian context handling categories model the behaviour of explicit substitutions, eg their formation from terms and their application to other terms. We now add extra structure to cartesian context handling categories to model the types of the $\lambda\sigma$-calculus. Since these types define new terms, and terms are interpreted in the fibres, this extra structure is defined on the fibres:

Definition 5 *An* E-category *is a cartesian handling category* $E{:}\mathcal{B}^{op}\to\mathbf{Set}^{\mathcal{T}}$ *with a distinguished type* $1 \in \mathcal{T}$ *and such that for two types* $A, B \in \mathcal{T}$*, there are types* $A \Rightarrow B, A \times B \in \mathcal{T}$*. In addition* 1 *is terminal in* $\mathcal{B}$*, and there are isomorphisms, natural in* Γ *between* $E((\Gamma, A))_B$ *and* $E(\Gamma)_{A\Rightarrow B}$ *as well as between* $E(\Gamma)_A \times E(\Gamma)_B$ *and* $E(\Gamma)_{A\times B}$*.*

Definition 5 implies that $A \times B$ is isomorphic to the product of A and B in $\mathcal{B}$, namely (A, B) — this is consistent with our philosophy that the semantics of context concatenation and the connective $\times$ are, although related, conceptually distinct. Similarly the type 1 is isomorphic to the empty context []. E-categories also provide a theory of equality judgements for the $\lambda\sigma$-calculus which are of the form $\Gamma \vdash t = t'$ and $\Gamma \vdash f = f'$ — these are given in Table 2 and used in proving soundness and completeness.

Similar structures to our E-categories have been considered in the literature. Jacobs [12] defines a λ1-category as an indexed category $\mathcal{B}^{op}\to\mathbf{Cat}$ such that $\mathcal{B}$ has finite products; morphisms in the fibre from A to B are morphisms $\Gamma \times A$ to B in the base category together with the condition that the fibration defined by the indexed category has $\mathcal{T}$-products. Such a λ1-category is an extensional E-category where the fibres are categories and not sets and where the isomorphism between substitutions and terms is the identity.

Table 2. Equality Judgements for the $\lambda\sigma$-calculus

(1)	$\langle\rangle ; f = f$	(2)	$\langle\rangle * t = t$
(3)	$(f;g);h = f;(g;h)$	(4)	$(f;g) * t = f * (g * t)$
(5)	$\langle f, t/x\rangle * x = t$	(6)	$\langle f, t/x\rangle * y = f * y$
(7)	$\dfrac{\Gamma \vdash f : \_}{\Gamma \vdash f = \langle\rangle}$	(8)	$f;\langle g, t/x\rangle = \langle f;g, (f * t)/x\rangle$
(9)	$\dfrac{\Gamma \vdash t : 1}{\Gamma \vdash t = \bullet}$	(10)	$\dfrac{\Gamma \vdash \langle\rangle : \Gamma' \qquad \Gamma' = x_1 : X_1, \ldots, x_n : X_n}{\Gamma \vdash \langle\rangle = \langle x_1/x_1, \ldots, x_n/x_n\rangle}$
(11)	$t = \lambda x : A.tx$	(12)	$(\lambda x : A.t)u = \langle\langle\rangle, u/x\rangle * t$
(13)	$f * (tu) = (f * t)(f * u)$	(14)	$f * \lambda x : A.t = \lambda y : A.\langle f, y/x\rangle * t$

where in equation 2, $x \notin \mathtt{FV}(t)$

***E*-categories and CCC's** Since the $\lambda\sigma$-calculus contains the λ-calculus, every model of the $\lambda\sigma$-calculus should contain a model of the λ-calculus, ie every E-category should contain an underlying CCC. In addition, one of the key-meta-theoretic properties of the $\lambda\sigma$-calculus is that every $\lambda\sigma$-term is provably equal to a λ-term. The semantic counterpart to this is that every CCC should extend to an E-category. The following theorem makes this relationship clear.

Theorem 6 *(i) Let $E : \mathcal{B}^{op} \to \mathbf{Set}^{\mathcal{T}}$ be an extensional E-category. Then the full subcategory of $\mathcal{B}$ generated by $\mathcal{T}$ is a CCC.*

(ii) Let $\mathcal{C}$ be a CCC and $\mathcal{T}$ be the set of objects of $\mathcal{C}$. Define a functor $E : \mathcal{C}^{op} \to \mathbf{Set}^{\mathcal{T}}$ by $E(\_)_\Delta = \mathsf{Hom}_{\mathcal{C}}(\_, \Delta)$, then E is an extensional E-category.

(iii) If $E : \mathcal{B}^{op} \to \mathbf{Set}^{\mathcal{T}}$ is an extensional E-category, then the E-category constructed in (ii) from the CCC constructed in (i) is isomorphic to the restriction of E to the full subcategory of $\mathcal{B}$ generated by $\mathcal{T}$.

Soundness and Completeness We now prove that we can model the $\lambda\sigma$-calculus in any E-category.

Theorem 7 *Let $E : \mathcal{B}^{op} \to \mathbf{Set}^{\mathcal{T}}$ be any E-category. Then there is a canonical interpretation map $[\![\_]\!]$ which assigns to any term of the $\lambda\sigma$-calculus with set $\mathcal{T}$ of base types an element of a fibre and assigns to every substitution a morphism.*

Proof. The types of the λ-calculus are interpreted as elements of $\mathcal{T}$ and, using the product structure of $\mathcal{B}$, this extends to an interpretation of contexts as objects of $\mathcal{B}$. We now define $[\![t]\!]$ by induction over the structure of t:

- Variable are interpreted by $\pi * \mathtt{Var}_A$ where π is a projection in the base
- λ-abstraction, application, product and projections are interpreted via the natural transformations occurring in definition 5. Finally the application of a substitution to a term is modelled by re-indexing.
- The substitution $\langle\rangle$ is interpreted as a projection in $\mathcal{B}$ and $[\![f;g]\!]$ is the composition of $[\![f]\!]$ and $[\![g]\!]$. Finally, $[\![\langle f, t/x\rangle]\!] = \langle [\![f]\!], \mathtt{Sub}([\![t]\!])\rangle$, where the right-hand side uses pairing in $\mathcal{B}$.

That $[\![.]\!]$ respects the equality judgements of the $\lambda\sigma$-calculus relies on proving by induction on t that $[\![t[\boldsymbol{s}/\boldsymbol{x}]]\!] = \langle \mathtt{Sub}([\![s_1]\!]), \ldots, \mathtt{Sub}([\![s_n]\!])\rangle * [\![t]\!]$.

E-categories form a complete class of models for the $\lambda\sigma$-calculus.

Theorem 8 *Let $\Gamma \vdash f : \Gamma'$ and $\Gamma \vdash f' : \Gamma'$ be $\lambda\sigma$-substitution judgements and $[\![\_]\!]$ be the interpretation function of Theorem 7. If for every E-category, $[\![f]\!] = [\![f']\!]$, then $\Gamma \vdash f = f' : \Gamma'$ is provable.*

Proof. The proof is by the standard term-model construction. We sketch this in three stages: (i) first we construct the base; (ii) next we construct the pre-sheaf structure; and (iii) we finally describe the natural transformations `Term` and `Sub`.

- **The Base Category $\mathcal{B}$**: In the term model, the base has as objects contexts and equivalence classes of substitutions as morphisms. The identity will be $\langle\rangle$ while composition is given by ;. Equations (1), (3) and (10) ensure that this structure does indeed define a category.
- **Cartesian Structure on $\mathcal{B}$:** By equation (7), the empty context is terminal in $\mathcal{B}$. On objects, the product structure is context concatenation while pairing is defined via the $\langle\rangle$-combinator. Universality follows from equation (10).
- **The Pre-sheaf:** The functor E maps a context Γ and singleton context $(x : A)$ to the set of equivalence classes of terms of type A in context Γ. On morphisms $E(f)$ maps a term t to the term $f * t$ and E is a functor, ie E preserves identities and composition, by equations (2) and (4)
- **The Natural Transformations `Term` and `Sub`:** The natural transformation `Term` maps $f : \Gamma \to (x : A)$ to $f * x$, while `Sub` maps an element of $E(\Gamma)_{x:A}$, ie a term t, to the substitution $\langle t/x\rangle$. Naturality of `Term` follows from equation (4), while naturality of `Sub` is implicit in equation (8). Equations (5) and (10) imply `Term` and `Sub` are isomorphisms.

4 The Multiplicative Structure

What extra structure must be added to a linear context handling category to model the $(I, \otimes, \multimap)$ connectives from linear logic? Following the definition of E-categories, we may try adding natural isomorphisms to the fibres of linear context handling categories. While this works for the linear function space, there is a complication for the tensor and unit. Recall from section 3 that the structure used to interpret product types is defined on the fibres and then induces an isomorphism in the base between the contexts $z : A \times B$ and $x : A, y : B$. However, imposing structure on the fibres of linear context handling category does not induce such an isomorphism and hence we require one explicitly.

Formally, a L-category requires an object $I \in \mathcal{T}$ to model the unit and binary operations $\otimes$ and $\multimap$ to model the tensor and linear implication. As argued before, I will be isomorphic to the unit $[]$ of the monoidal structure of $\mathcal{B}$. Similarly, $\otimes$ will not be equal to the tensor of $\mathcal{B}$ but will be isomorphic to it.

Definition 9 *An L-category is a linear context-handling category $(\mathcal{B}, \mathcal{T})$ st:*

(i) *There is a type $I \in \mathcal{T}$, and given $A, B \in \mathcal{T}$, there are types $A \otimes B$ and $A \multimap B$.*
(ii) *For every type A and B, there are isomorphisms $n_I : [\,] \cong I : n_I^{-1}$ and $n_\otimes : (A, B) \cong A \otimes B : n_\otimes^{-1}$.*
(iii) *Given types A, B and C, there is a Γ-natural isomorphism between $L((\Gamma, A))_B$ and $L(\Gamma)_{A \multimap B}$.*

Theorem 6 generalises to the linear setting, ie every L-category has an underlying SMCC and every SMCC generates an L-category. This reflects the fact that a linear calculus of explicit substitutions contains an underlying linear λ-calculus and every term is equal to a term of the underlying linear λ-calculus.

Theorem 10 *(i) Let $L : \mathcal{B}^{op} \to \mathbf{Set}^{\mathcal{T}}$ be an L-category. Then the full subcategory of $\mathcal{B}$ generated by $\mathcal{T}$ is a symmetric monoidal closed category.*
(ii) Let $\mathcal{C}$ be any symmetric monoidal closed category with objects $\mathcal{T}$. The functor $L : \mathcal{C}^{op} \to \mathbf{Set}^{\mathcal{T}}$ defined by $L(\_)_\Delta = \mathsf{Hom}_{\mathcal{C}}(\_, \Delta)$, is an L-category.
(iii) If $L : \mathcal{B}^{op} \to \mathbf{Set}^{\mathcal{T}}$ is an L-category, then the L-category constructed in (ii) from the underlying SMC defined in (i) is naturally isomorphic to the restriction of L to the full subcategory of $\mathcal{B}$ generated by $\mathcal{T}$.

Proof. The same proof as for Theorem 6 works.

L-categories provide models for a linear λ-calculus with the $(\otimes, I, \multimap)$-type structure extended with explicit substitutions — we call this calculus the monoidal $\lambda\sigma$-calculus. Formally, the raw expressions are

$$t ::= x \mid \lambda x{:}A.t \mid tu \mid t \otimes t \mid \bullet \mid f * t \mid \texttt{let}\ t\ \texttt{be}\ p\ \texttt{in}\ t$$
$$f ::= \langle\rangle \mid \langle f, t/x\rangle \mid f; f \mid \texttt{let}\ t\ \texttt{be}\ p\ \texttt{in}\ f$$

where x is a variable and p is of the form $\bullet, x \otimes y$. The calculus contains the usual terms of the linear λ-calculus, substitution constructs we have already seen and finally there are two new forms of substitution given by `let`-expressions. That is, not only do we have terms of the form $\texttt{let}\ t\ \texttt{be}\ p\ \texttt{in}\ u$ (where u is a term) but also substitutions of the form $\texttt{let}\ t\ \texttt{be}\ p\ \texttt{in}\ f$ (where f is a substitution). These `let`-expressions ensure the context $z : A \otimes B$ is isomorphic to the context $x : A, y : B$ as required by the definition of an L-category. Formally, the typing judgements are of the form $\Gamma \vdash t : A$ and $\Gamma \vdash f : \Gamma'$ and are given in Table 3, while the equality judgements for the calculus are given in Table 4. The η-equations are derived from Ghani's adjoint rewriting [9].

Theorem 11 *Let $L : \mathcal{B}^{op} \to \mathbf{Set}^{\mathcal{T}}$ be a L-category. Then there is an interpretation map $[\![.]\!]$ sending terms of the monoidal $\lambda\sigma$-calculus with ground types $\mathcal{T}$ to elements of the fibres and substitutions to morphisms.*

Proof. The proof is similar to that of Theorem 7. Variables are interpreted by the elements $\mathtt{Var}_A$, while $\langle\rangle$ is interpreted via the identity in the base, parallel composition via the tensor on $\mathcal{B}$ and sequential composition via composition in the base. The isomorphism $A \otimes B \to (A, B)$ is used to interpret both the term

Table 3. Typing Judgements of the Monoidal $\lambda\sigma$-calculus

The term judgements of are

$$\frac{}{x\colon A \vdash x\colon A} \qquad \frac{\Gamma, x\colon A \vdash t\colon B}{\Gamma \vdash \lambda x\colon A.t\colon A \multimap B} \qquad \frac{\Gamma_1 \vdash t\colon A \multimap B \qquad \Gamma_2 \vdash u\colon A}{\Gamma \vdash tu\colon B}$$

$$\frac{\Gamma_1 \vdash f\colon \Gamma_2 \qquad \Gamma_2 \vdash t\colon A}{\Gamma_1 \vdash f * t\colon A} \qquad \frac{}{\_ \vdash \bullet\colon I} \qquad \frac{\Gamma_1 \vdash t\colon I \qquad \Gamma_2 \vdash u\colon A}{\Gamma \vdash \texttt{let}\ t\ \texttt{be}\ \bullet\ \texttt{in}\ u\colon A}$$

$$\frac{\Gamma_1 \vdash t\colon A \quad \Gamma_2 \vdash u\colon B}{\Gamma \vdash t \otimes u\colon A \otimes B} \qquad \frac{\Gamma_1 \vdash u\colon A \otimes B \qquad \Gamma_2, x\colon A, y\colon B \vdash t\colon C}{\Gamma \vdash \texttt{let}\ u\ \texttt{be}\ x \otimes y\ \texttt{in}\ t\colon C}$$

The substitution judgements are

$$\frac{}{\Gamma \vdash \langle\rangle : \Gamma} \qquad \frac{\Gamma_1 \vdash f\colon \Gamma_2 \qquad \Gamma_2 \vdash g\colon \Gamma_3}{\Gamma_1 \vdash f; g\colon \Gamma_3} \qquad \frac{\Gamma_1 \vdash f : \Gamma' \qquad \Gamma_2 \vdash t\colon A}{\Gamma \vdash \langle f, t/x\rangle\colon \Gamma', x\colon A}$$

$$\frac{\Gamma_1 \vdash t\colon I \qquad \Gamma_2 \vdash f\colon \Gamma'}{\Gamma \vdash \texttt{let}\ t\ \texttt{be}\ \bullet\ \texttt{in}\ f\colon \Gamma'} \qquad \frac{\Gamma_1 \vdash u\colon A \otimes B \qquad \Gamma_2, x\colon A, y\colon B \vdash f\colon \Gamma'}{\Gamma \vdash \texttt{let}\ u\ \texttt{be}\ x \otimes y\ \texttt{in}\ f\colon \Gamma'}$$

The rules for substitutions assume x, y are fresh and, where applicable, Γ_1, Γ_2 are disjoint and Γ is any permutation of Γ_1, Γ_2.

$\texttt{let}\ u\ \texttt{be}\ x \otimes y\ \texttt{in}\ t$ and the substitution $\texttt{let}\ u\ \texttt{be}\ x \otimes y\ \texttt{in}\ f$. Similarly the corresponding isomorphism for the unit is used to interpret the other $\texttt{let}$-expression. The verification that the map $[\![t]\!]$ respects equality judgements relies on a substitution lemma similar to that of Theorem 7.

Theorem 12 *L-categories form a complete class of models for the monoidal $\lambda\sigma$-calculus.*

Proof. A term model is constructed with contexts as objects and equivalence classes of substitutions as morphisms. The pre-sheaf structure is added as in Theorem 8 and, finally, we get an L-category from the inverse morphisms

$$x : X, y : Y \vdash \langle (x \otimes y)/z\rangle : z : A \otimes B$$
$$z : A \otimes B \vdash \texttt{let}\ z\ \texttt{be}\ x \otimes y\ \texttt{in}\ \langle x/x, y/y\rangle : (x\colon X, y\colon Y)$$

Similarly, I is isomorphic to the empty context.

5 The Modalities

The standard categorical model of the modalities of linear logic is via a co-Kleisli construction [17] [6]. Benton [5] proposes the equivalent LNL-categories consisting of a monoidal adjunction between a cartesian closed category (CCC) and a symmetric monoidal closed category (SMCC). The adaptation of this approach to our framework is more succinct and hence used here.

Definition 13 *An* $!L$-category *is an L-category $L\colon \mathcal{B}^{op} \to \mathbf{Set}^{\mathcal{T}}$ together with an E-category $E : \mathcal{C}^{op} \to \mathbf{Set}^{\mathcal{S}}$ and monoidal adjunction $F \dashv G : \mathcal{C} \to \mathcal{B}$ such that if $A \in \mathcal{S}$, then $FA \in \mathcal{T}$, and conversely, if $B \in \mathcal{T}$, then $GB \in \mathcal{S}$.*

Table 4. Equality Judgements of the Monoidal $\lambda\sigma$-calculus

Let @ be either ; or $*$ depending whether h is a term or a substitution.

– β- and η-equality:

$$(\lambda x{:}\,A.t)u = \langle u/x\rangle * t \qquad \lambda x : A.tx = x$$
$$\texttt{let } v{\otimes}u \texttt{ be } x{\otimes}y \texttt{ in } h = \langle v/x, u/y\rangle * h \qquad \texttt{let } \bullet \texttt{ be } \bullet \texttt{ in } h = h$$
$$\frac{\Gamma_1 \vdash u : I \qquad \Gamma_2, z : I \vdash f : \Gamma'}{\Gamma \vdash f[u/z] = \texttt{let } u \texttt{ be } \bullet \texttt{ in } \langle \bullet/z\rangle; f} \qquad \frac{\Gamma_1 \vdash u : A{\otimes}B \qquad \Gamma_2, z : A{\otimes}B \vdash f : \Gamma'}{\Gamma \vdash f[u/z] = \texttt{let } u \texttt{ be } x{\otimes}y \texttt{ in } \langle (x{\otimes}y)/z\rangle; f}$$

– Application of Substitutions:

$$\langle\rangle; f = f \qquad \langle\rangle * t = t \qquad (f;g);h = f;(g;h)$$
$$\langle f, t/x\rangle * x = t \qquad \langle f, t/y\rangle * x = f * x \qquad (f;g) * t = f * (g * t)$$
$$\frac{\Gamma \vdash f : -}{\Gamma \vdash f = \langle\rangle} \qquad \frac{\Gamma \vdash \langle\rangle : \Gamma \qquad \Gamma = x_i{:}\,X_i}{\Gamma \vdash \langle\rangle = \langle x_i/x_i\rangle}$$

– If $f = \langle t_1/x_1, \ldots, t_n/x_n\rangle$ then f_t is f restricted to the free variables of t.

$$f; \langle g, t/x\rangle = \langle f_g; g, (f_t * t)/x\rangle$$
$$f * (u \otimes v) = (f_u * u) \otimes (f_v * v) \qquad f * \bullet = \bullet$$
$$f * \lambda y{:}\,A.u = \lambda z{:}\,A.\langle f, z/y\rangle * u \qquad f * uv = (f_u * u)(f_v * v)$$
$$f@\texttt{let } t \texttt{ be } p \texttt{ in } h = \texttt{let } (f_t * t) \texttt{ be } p \texttt{ in } f_h@h$$

As in Theorem 6 LNL-categories embed into $!L$-categories and vice versa.

Theorem 14 *(i) If* $(L{:}\,\mathcal{B}^{op}{\to}\mathbf{Set}^{\mathcal{T}}, E{:}\,\mathcal{C}^{op}{\to}\mathbf{Set}^{\mathcal{S}})$ *is a* $!L$*-category, the full subcategory of* $\mathcal{B}$ *defined by* $\mathcal{T}$ *and of* $\mathcal{C}$ *defined by* $\mathcal{S}$ *is a LNL-category.*

(ii) Let $F \dashv G : \mathcal{C}{\to}\mathcal{B}$ *be a monoidal adjunction between a cartesian closed category* $\mathcal{C}$ *with objects* $\mathcal{S}$ *and a symmetric monoidal closed category* $\mathcal{B}$ *with objects* $\mathcal{T}$*. If we define functors* $E{:}\,\mathcal{C}^{op}{\to}\mathbf{Set}^{\mathcal{S}}$ *by* $E(\_)_\Delta = \mathsf{Hom}_{\mathcal{C}}(\_, \Delta)$ *and* $L{:}\,\mathcal{B}^{op}{\to}\mathbf{Set}^{\mathcal{T}}$ *by* $L(\_)_\Delta = \mathsf{Hom}_{\mathcal{B}}(\_, \Delta)$ *then* (L, E) *is an* $!L$*-category.*

(iii) If $(L{:}\,\mathcal{B}^{op}{\to}\mathbf{Set}^{\mathcal{T}}, E{:}\,\mathcal{C}^{op}{\to}\mathbf{Set}^{\mathcal{S}})$ *be a* $!L$*-category, then the* $!L$*-category constructed in (ii) from the monoidal adjunction constructed in (i) is isomorphic* [3] *to the original* L*-category.*

5.1 xDILL - A Linear Calculus of Explicit Substitutions

We now extend the monoidal $\lambda\sigma$-calculus with !-types and prove that $!L$-categories form a sound and complete class of models for this calculus. Underlying our extended calculus is Barber's DILL [3] — hence we call our calculus xDILL [10]. We use DILL because it incorporates the semantic separation of linear and non-linear contexts directly within the syntax although we could have started from Bierman's linear λ-calculus. Formally, the types of xDILL are base types, unit, function, tensor and !-types and the raw expressions are

$$t ::= x \mid \lambda x{:}\,A.t \mid tu \mid t \otimes t \mid \bullet \mid\ !t \mid f * t \mid \texttt{let } t \texttt{ be } p \texttt{ in } t$$
$$f ::= \langle\rangle \mid \langle f, t/x_I\rangle \mid \langle f, t/x_L\rangle \mid f; f \mid \texttt{let } t \texttt{ be } p \texttt{ in } f$$

[3] in the obvious component-wise sense

where x is a variable and p is of the form $\bullet, x \otimes y$ or $!x$. Like DILL, xDILL contains both linear and intuitionistic variables and hence has zoned contexts of the form $\Gamma|\Delta$. Weakening and contraction are only permitted for variables declared in Γ and the !-type constructor controls the interaction between the intuitionistic and linear zones of a context, thus allowing terms of !-type to be copied and discarded. As in section 4, the `let`-expressions must be generalised to *substitutions as well as terms*. Formally, the typing judgements of xDILL are of the form $\Gamma|\Delta \vdash t : A$ and $\Gamma|\Delta \vdash f : \Gamma'|\Delta'$ and are given in Table 5, while the equality judgements of xDILL as presented in Table 6.

Proposition 15 *There is a canonical interpretation* $[\![\_]\!]$ *of xDILL with base types in* $\mathcal{T}$ *in a* $!L$*-category* $(L\colon \mathcal{B}^{op} \to \mathbf{Set}^{\mathcal{T}}, E\colon \mathcal{C}^{op} \to \mathbf{Set}^{\mathcal{S}})$ *with monoidal adjunction* $F \dashv G : \mathcal{C} \to \mathcal{B}$.

Proof. Firstly interpret the types in $\mathcal{T}$ — for !-types set $[\![!A]\!] = FG[\![A]\!]$. This gives an interpretation of xDILL contexts using the monoidal structure of $\mathcal{B}$

$$[\![\Gamma|\Delta]\!] = (FG([\![A_1]\!]), \cdots, FG([\![A_n]\!]), [\![B_1]\!], \cdots, [\![B_m]\!])$$

where $\Gamma = x_1\colon A_1, \ldots, x_n\colon A_n$ and $\Delta = y_1\colon B_1, \ldots, y_m\colon B_m$. Now any xDILL term judgement $\Gamma|\Delta \vdash t : A$ is interpreted as an element of $L([\![\Gamma|\Delta]\!])_{[\![A]\!]}$ and any xDILL substitution judgement $\Gamma|\Delta \vdash f : \Gamma'|\Delta'$ is interpreted as a $\mathcal{B}$-map $[\![f]\!] : [\![\Gamma|\Delta]\!] \to [\![\Gamma'|\Delta']\!]$. This map $[\![\_]\!]$ is defined inductively, eg

$$\begin{aligned}
[\![!t]\!] &= \delta_\Gamma * m_\Gamma * FG(\mathtt{Sub}(t)) * \mathtt{Var}_A \\
[\![\mathtt{let}\ t\ \mathtt{be}\ !x\ \mathtt{in}\ u]\!] &= (\mathsf{Id}, \mathtt{Sub}([\![t]\!]), \mathsf{Id}) * [\![u]\!] \\
[\![\mathtt{let}\ t\ \mathtt{be}\ !x\ \mathtt{in}\ f]\!] &= (\mathsf{Id}, \mathtt{Sub}([\![t]\!]), \mathsf{Id}); [\![f]\!]
\end{aligned}$$

where $\delta_\Gamma : (!X_1, \ldots, !X_n) \to (!!X_1, \ldots, !!X_n)$ is derived via the co-multiplication of the comonad on $\mathcal{B}$ and $m_\Gamma : (!!X_1, \ldots, !!X_n) \to !(!X_1, \ldots, !X_n)$ is derived from the monoidal transformation $!X, !Y \to !(X, Y)$.

Completeness of $!L$-categories as models of xDILL is proven by constructing a term model. We only define the structure involved and (mostly) omit the (lengthy, but routine) verification that the structure has the required properties. First the functor $L\colon \mathcal{B}^{op} \to \mathbf{Set}^{\mathcal{T}}$ is defined. The objects of $\mathcal{B}$ are contexts $\Gamma|\Delta$ and morphisms are substitution judgements $\Gamma|\Delta \vdash f : \Gamma'|\Delta'$. Context union makes $\mathcal{B}$ monoidal. Next define $\mathcal{T}$ to be the set of xDILL types, $L(\Gamma|\Delta)_A$ to be the judgements $\Gamma \vdash |\Delta \vdash t : A$, $L(f)(t)$ to be $f * t$, $\mathtt{Var}_A$ to be a canonical variable and set $\mathtt{Sub}(t)$ to be the substitution $\langle t/x \rangle$. This makes L a linear context handling category. Section 4 shows how to turn L into a L-category.

Now we turn to the modalities. The category $\mathcal{C}$ has as objects contexts Γ and morphisms $\mathcal{C}(\Gamma, \Delta)$ are tuples of judgements $\Gamma|\_ \vdash t : A_i$ where Δ is the context $x_1 : A_1, \ldots, x_n : A_n$. Note that in $\mathcal{C}$ there are no `let`-substitutions — this corresponds exactly to the syntactic restrictions on term substitutions that arise in the meta-theory of xDILL [10]. Composition in $\mathcal{C}$ is given by substitution with tuples of variables forming the identities. $\mathcal{S}$ is the set of types and define

Table 5. xDILL Typing Judgements

The term judgements of xDILL are

$$\Gamma, x\colon A, \Gamma' | \_ \vdash x\colon A \qquad \frac{\Gamma_1|\Delta_1 \vdash f\colon \Gamma_2|\Delta_2 \qquad \Gamma_2|\Delta_2 \vdash t\colon A}{\Gamma_1|\Delta_1 \vdash f * t\colon A}$$

$$\Gamma|x\colon A \vdash x\colon A \qquad \frac{\Gamma|\Delta, x\colon A \vdash t\colon B}{\Gamma|\Delta \vdash \lambda x\colon A.t\colon A \multimap B} \qquad \frac{\Gamma|\Delta_1 \vdash t\colon A \multimap B \qquad \Gamma|\Delta_2 \vdash u\colon A}{\Gamma|\Delta \vdash tu\colon B}$$

$$\Gamma|\_ \vdash \bullet\colon I \qquad \frac{\Gamma|\Delta_1 \vdash t\colon I \qquad \Gamma|\Delta_2 \vdash u\colon A}{\Gamma|\Delta \vdash \texttt{let } t \texttt{ be } \bullet \texttt{ in } u\colon A}$$

$$\frac{\Gamma|\Delta_1 \vdash t\colon A \quad \Gamma|\Delta_2 \vdash u\colon B}{\Gamma|\Delta \vdash t \otimes u\colon A \otimes B} \qquad \frac{\Gamma|\Delta_1 \vdash u\colon A \otimes B \qquad \Gamma|\Delta_2, x\colon A, y\colon B \vdash t\colon C}{\Gamma|\Delta \vdash \texttt{let } u \texttt{ be } x \otimes y \texttt{ in } t\colon C}$$

$$\frac{\Gamma|\_ \vdash t\colon A}{\Gamma|\_ \vdash !t\colon !A} \qquad \frac{\Gamma|\Delta_1 \vdash t\colon !A \qquad \Gamma, x\colon A|\Delta_2 \vdash u\colon B}{\Gamma|\Delta \vdash \texttt{let } t \texttt{ be } !x \texttt{ in } u\colon B}$$

The substitution judgements of xDILL are

$$\frac{\Gamma' \subseteq \Gamma}{\Gamma|\Delta \vdash \langle\rangle : \Gamma'|\Delta} \qquad \frac{\Gamma_1|\Delta_1 \vdash f\colon \Gamma_2|\Delta_2 \qquad \Gamma_2|\Delta_2 \vdash g\colon \Gamma_3|\Delta_3}{\Gamma_1|\Delta_1 \vdash f; g\colon \Gamma_3|\Delta_3}$$

$$\frac{\Gamma|\Delta \vdash f : \Gamma'|\Delta' \qquad \Gamma|\_ \vdash t\colon A}{\Gamma|\Delta \vdash \langle f, t/x_I\rangle\colon \Gamma', x\colon A|\Delta'} \qquad \frac{\Gamma|\Delta_1 \vdash f : \Gamma'|\Delta' \qquad \Gamma|\Delta_2 \vdash t\colon A}{\Gamma|\Delta \vdash \langle f, t/x_L\rangle\colon \Gamma'|\Delta', x\colon A}$$

$$\frac{\Gamma|\Delta_1 \vdash t\colon I \qquad \Gamma|\Delta_2 \vdash f\colon \Gamma'|\Delta'}{\Gamma|\Delta \vdash \texttt{let } t \texttt{ be } \bullet \texttt{ in } f\colon \Gamma'|\Delta'} \qquad \frac{\Gamma|\Delta_1 \vdash t\colon !A \qquad \Gamma, x\colon A|\Delta_2 \vdash f\colon \Gamma'|\Delta'}{\Gamma|\Delta \vdash \texttt{let } t \texttt{ be } !x \texttt{ in } f\colon \Gamma'|\Delta'}$$

$$\frac{\Gamma|\Delta_1 \vdash u\colon A \otimes B \qquad \Gamma|\Delta_2, x\colon A, y\colon B \vdash f\colon \Gamma'|\Delta'}{\Gamma|\Delta \vdash \texttt{let } u \texttt{ be } x \otimes y \texttt{ in } f\colon \Gamma'|\Delta'}$$

The rules for substitutions assume x, y are fresh and, where applicable, Δ_1, Δ_2 are disjoint and Δ is a permutation of Δ_1, Δ_2.

$E\colon \mathcal{C}^{op} \to \mathbf{Set}^{\mathcal{T}}$ by setting $E(\Gamma)_A$ to be the set of typing judgements $\Gamma|\_ \vdash t : A$. This makes E a cartesian context handling category. E can be made into an E-category using Girard's decomposition of intuitionistic function spaces $A \to B$ into linear function spaces $!A \multimap B$.

Finally we construct a $!L$-category. This is greatly simplified by observing that $\mathcal{B}$ is naturally isomorphic to the full subcategory $\mathcal{B}_0$ whose objects are $\_|x : A$ — again these isomorphisms use the `let`-substitutions of xDILL. Thus we define a monoidal adjunction $F \dashv G : \mathcal{C} \to \mathcal{B}_0$ which then extends to a monoidal adjunction on $\mathcal{B}$. The functor F is given by $F(\Gamma) = \_|z : (!X_1 \otimes \cdots \otimes !X_n)$, where $\Gamma = x_1 : X_1, \ldots, x_n : X_n$ and z is some canonical choice of variable. To define F on morphisms, let $\Gamma|\_ \vdash t_j : Y_j$. Then since there is an isomorphism $\iota^{-1} : F(\Gamma) \to \Gamma$, there are judgements $F(\Gamma) \vdash \iota^{-1}; !t_j : !Y_j$. Hence $F(t_1, \ldots t_n) = \langle(\iota^{-1}; !t_1 \otimes \cdots \otimes \iota^{-1}; !t_m)/x\rangle$. We define G on objects by $G(\_|x : A) = z : A$ — this makes G right-adjoint to F as the required natural isomorphism on sets of derivations follows from the isomorphism in $\mathcal{B}$ between Γ and $F(\Gamma)$. Moreover one can show that we have the required additional data to form a $!L$-category. Hence we have shown the following Theorem:

Theorem 16 *The term model is a $!L$-category.*

Table 6. Equality Judgements for xDILL

Let @ be either ; or $*$ depending whether h is a term or a substitution.

– β- and η-equality:

$$(\lambda x{:}\,A.t)u = \langle u/x\rangle * t \qquad \lambda x : A.tx = x$$

$$\texttt{let } v\otimes u \texttt{ be } x\otimes y \texttt{ in } h = \langle v/x, u/y\rangle * h \qquad \texttt{let } \bullet \texttt{ be } \bullet \texttt{ in } h = h$$

$$\frac{\Gamma|\Delta_1 \vdash u : A\otimes B \qquad \Gamma|\Delta_2, z : A\otimes B \vdash f : \Gamma'|\Delta'}{\Gamma|\Delta \vdash f[u/z] = \texttt{let } u \texttt{ be } x\otimes y \texttt{ in } \langle (x\otimes y)/z\rangle; f} \qquad \texttt{let } !u \texttt{ be } !x \texttt{ in } t = \langle u/x\rangle * t$$

$$\frac{\Gamma|\Delta_1 \vdash u{:}I \qquad \Gamma|\Delta_2, z : I \vdash f : \Gamma'|\Delta'}{\Gamma|\Delta \vdash f[u/z] = \texttt{let } u \texttt{ be } \bullet \texttt{ in } \langle \bullet/z\rangle; f} \qquad \frac{\Gamma|\Delta_1 \vdash u{:}!A \qquad \Gamma|\Delta_2 z :!A \vdash f : \Gamma'|\Delta'}{\Gamma|\Delta \vdash f[u/z] = \texttt{let } u \texttt{ be } !x \texttt{ in } \langle !x/z\rangle; f}$$

– Application of Substitutions:

$$\langle\rangle; f = f \qquad \langle\rangle * t = t \qquad (f;g);h = f;(g;h)$$

$$\langle f, t/x\rangle * x = t \qquad \langle f, t/y\rangle * x = f * x \qquad (f;g) * t = f * (g * t)$$

$$\frac{\Gamma|\Delta \vdash f : -|-}{\Gamma|\Delta \vdash f = \langle\rangle} \qquad \frac{\Gamma|\Delta \vdash \langle\rangle : \Gamma'|\Delta' \qquad \Gamma' = x_i{:}\,X_i \qquad \Delta' = y_i{:}\,Y_j}{\Gamma|\Delta \vdash \langle\rangle = \langle x_i/x_i, y_j/y_j\rangle}$$

– If f is of the form $\langle t_1/x_1, \ldots, t_n/x_n\rangle$ and f_t is f restricted to the free variables of t.

$$f;\langle g, t/x\rangle = \langle f_g; g, (f_t * t)/x\rangle \qquad f*!u = !(f * u)$$

$$f * (u \otimes v) = (f_u * u) \otimes (f_v * v) \qquad f * \bullet = \bullet$$

$$f * \lambda y{:}\,A.u = \lambda z{:}\,A.\langle f, z/y\rangle * u \qquad f * uv = (f_u * u)(f_v * v)$$

$$f@\texttt{let } t \texttt{ be } p \texttt{ in } h = \texttt{let } (f_t * t) \texttt{ be } p \texttt{ in } f_h@h$$

6 Summary and Discussion

We have modularly defined new categorical models for λ-calculi extended with explicit substitutions. We took our intuitions from indexed category theory but had to make alterations so as to accommodate linear calculi in the same framework as cartesian calculi. We have also related these models to the well-established categorical models for their underlying λ-calculi and proved appropriate soundness and completeness results.

Recapitulating from the introduction, the reason for describing these models is our goal of designing an abstract machine based on the linear lambda-calculus that is conceptually clean (and easy to prove correct!). These models have already been used to derive a linear lambda-calculus with explicit substitutions [10] and an abstract machine, which has been implemented by Alberti [2].

However there are two questions which remain unresolved and require further research. Firstly we have been unable to find concrete instances of our proposed models. Secondly, and perhaps more importantly, our definition of a context handling category $L{:}\,\mathcal{B}^{op}\to\mathbf{Set}^{\mathcal{T}}$ distinguishes between isomorphic entities, eg the functor $L(A)$ and the hom-functor $\mathsf{Hom}_{\mathcal{B}}(-, A)$. This goes somewhat against the grain of category theory which tends to regard isomorphic structures as being indistinguishable. However, were we to take the alternative approach of identifying the functor L with the hom-functors and dropping the transformations Sub

and `Term`, while maintaining a Curry-Howard correspondence, this would entail dropping the crucial combinator which forms substitutions from terms from the associated calculus of explicit substitutions. Hence we keep the functor L and the natural isomorphisms `Sub` and `Term` in Definition 1.

We would like to thank Peter Dybjer, Martin Hofmann, Andrea Schalk and Martin Hyland for discussions on the subject of this paper.

References

1. Martin Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jaques Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, 1991.
2. F.J Alberti. An abstract machine based on linear logic and explicit substitutions. Master's thesis, School of Computer Science, University of Birmingham, 1997.
3. A. Barber and G. Plotkin. Dual intuitionistic linear logic. Technical report, LFCS, University of Edinburgh, 1997.
4. N. Benton, G. Bierman, V. de Paiva, and M. Hyland. A term calculus for intuitionistic linear logic. In M. Bezem and J. F. Groote, editors, *Typed Lambda Calculi and Applications*, volume 664 of *Lecture Notes in Computer Science*, pages 75–90. Springer Verlag, 1993.
5. Nick Benton. A mixed linear and non-linear logic: Proofs, terms and models. In *Proceedings of Computer Science Logic '94, Kazimierz, Poland*. Lecture Notes in Computer Science No. 933, Berlin, Heidelberg, New York, 1995.
6. Gavin Bierman. *On Intuitionistic Linear Logic*. Phd-thesis, University of Cambridge, 1994. Also available as Technical Report No. 346.
7. Guy Cousineau, Pierre-Louis Curien, and Michel Mauny. The categorical abstract machine. *Science of Computer Programming*, 8:173–202, 1987.
8. Thomas Ehrhard. A categorical semantics of constructions. In *Third Annual Symposium on Logic in Computer Science*, pages 264–273. IEEE, 1988.
9. N. Ghani. *Adjoint Rewriting*. PhD thesis, University of Edinburgh, 1995.
10. N. Ghani, V. de Paiva, and E. Ritter. Linear explicit substitutions. In *Proc. of Westapp'98*, 1998. Full version submitted for publication.
11. J. M. E. Hyland and A. M. Pitts. The theory of constructions: Categorical semantics and topos theoretic models. *Contemporary Mathematics*, 92:137–198, 1989.
12. Bart Jacobs. Simply typed and untyped lambda calculus revisited. In Michael Fourman, Peter Johnstone, and Andrew Pitts, editors, *Applications of Categories in Computer Science*, LMS Lecture Note Series 177, pages 119–142. CUP, 1992.
13. Yves Lafont. The linear abstract machine. *Theoretical Computer Science*, 59:157–180, 1988.
14. Ian Mackie. Lilac: A functional programming language based on linear logic. *Journal of Functional Programming*, 4(4):395–433, 1994.
15. E. Ritter and V. de Paiva. On explicit substitution and names (extended abstract). In *Proc. of ICALP'97*, LNCS 1256, pages 248–258, 1997.
16. Eike Ritter. Categorical abstract machines for higher-order lambda calculi. *Theoretical Computer Science*, 136(1):125–162, 1994.
17. R. A. G. Seely. Linear logic, *-autonomous categories and cofree algebras. *Contemporary Mathematics*, 92, 1989.

Equational Properties of Mobile Ambients

Andrew D. Gordon[1] and Luca Cardelli[2]

[1] Microsoft Research, http://research.microsoft.com/~adg
[2] Microsoft Research, http://www.luca.demon.co.uk

Abstract. The ambient calculus is a process calculus for describing mobile computation. We develop a theory of Morris-style contextual equivalence for proving properties of mobile ambients. We prove a context lemma that allows derivation of contextual equivalences by considering contexts of a particular limited form, rather than all arbitrary contexts. We give an activity lemma that characterizes the possible interactions between a process and a context. We prove several examples of contextual equivalence. The proofs depend on characterizing reductions in the ambient calculus in terms of a labelled transition system.

1 Motivation

This paper develops tools for proving equations in the ambient calculus.

In earlier work [6], we introduced the ambient calculus by adding *ambients*—mobile, hierarchical protection domains—to a framework for concurrency extracted from the π-calculus [12]. The ambient calculus is an abstract model of mobile computation, including both mobile software agents and mobile hardware devices. The calculus models access control as well as mobility. For example, a process may move into or out of a particular ambient only if it possesses the appropriate capability.

This paper focuses on behavioural equivalence of mobile ambients. In particular, we study a form of Morris' contextual equivalence [14] for ambients and develop some proof techniques. Our motivation is to prove a variety of equations. Some of these equations express and confirm some of the informal principles we had in mind when designing the calculus. As in other recent work [1, 2], some of the equations establish security properties of systems modelled within the calculus.

The inclusion of primitives for mobility makes the theory of the ambient calculus more complex than that of its ancestor, the π-calculus. The main contribution of this paper is to demonstrate that some standard tools—a labelled transition system, a context lemma, and an activity lemma—may be recast in the setting of the ambient calculus. Moreover, the paper introduces a new technique—based on what we call the hardening relation—for factoring the definition of the labelled transition system into a set of rules that identify the individual processes participating in a transition, and a set of rules that express how the participant processes interact.

W. Thomas (Ed.): FOSSACS´99, LNCS 1578, pp. 212-226, 1999.

We begin, in Section 2, by reviewing the syntax and reduction semantics of the ambient calculus. The semantics consists of a structural congruence relation $P \equiv Q$ (which says that P may be structurally rearranged to yield Q) and a reduction relation $P \rightarrow Q$ (which says that P may evolve in one step of computation to yield Q).

We introduce contextual equivalence $P \simeq Q$ in Section 3. We define a predicate, $P \Downarrow n$, which means intuitively that an observer may eventually detect an ambient named n at the top-level of the process P. Then we define $P \simeq Q$ to mean that, whenever P and Q are placed within an arbitrary context constructed from the syntax of the calculus, any observation made of P may also be made of Q, and vice versa. We give examples of pairs of processes that are equivalent and examples of pairs that are inequivalent.

In Section 4, we describe some techniques for proving contextual equivalence. We introduce a second operational semantics for the ambient calculus based on a hardening relation and a labelled transition system. The hardening relation identifies the subprocesses of a process that may participate in a computation step. We use the hardening relation both for defining the labelled transition system and for characterizing whether an ambient of a particular name is present at the top-level of a process. Our first result, Theorem 1, asserts that the τ-labelled transition relation and the reduction relation are the same, up to structural congruence. So our two operational semantics are equivalent. The labelled transition system is useful for analyzing the possible evolution of a process, since we may read off the possible labelled transitions of a process by inspecting its syntactic structure. Our second result, Theorem 2 is a context lemma that allows us to prove contextual equivalence by considering a limited set of contexts, known as harnesses, rather than all arbitrary contexts. A harness is a context with a single hole that is enclosed only within parallel compositions, restrictions, and ambients. The third result of this section, Theorem 3, is an activity lemma that elaborates the ways in which a reduction may be derived when a process is inserted into a harness: either the process reduces by itself, or the harness reduces by itself, or there is an interaction between the harness and the process.

We exercise these proof techniques on examples in Section 5, and conclude in Section 6.

2 The Ambient Calculus (Review)

We briefly describe the syntax and semantics of the calculus. We assume there are infinite sets of *names* and *variables*, ranged over by m, n, p, q, and x, y, z, respectively. The syntax of the ambient calculus is based on categories of *expressions* and *processes*, ranged over by M, N, and P, Q, R, respectively. The calculus inherits a core of concurrency primitives from the π-calculus: a restriction $(\nu n)P$ creates a fresh name n whose scope is P; a composition $P \mid Q$ behaves as P and Q running in parallel; a replication $!P$ behaves as unboundedly many replicas of P running in parallel; and the inactive process $\mathbf{0}$ does nothing. We augment these π-calculus processes with primitives for mobility—ambients,

$n[P]$, and the exercise of capabilities, $M.P$—and primitives for communication—input, $(x).P$, and output, $\langle M \rangle$.

Here is an example process that illustrates the new primitives for mobility and communication:

$$m[p[\mathit{out}\ m.\mathit{in}\ n.\langle M \rangle]] \mid n[\mathit{open}\ p.(x).Q]$$

The effect of the mobility primitives in this example is to move the ambient p out of m and into n, and then to open it up. The input $(x).Q$ may then consume the output $\langle M \rangle$ to leave the residue $m[] \mid n[Q\{x \leftarrow M\}]$. We may regard the ambients m and n in this example as modelling two machines on a network, and the ambient p as modelling a packet sent from m to n. Next, we describe the semantics of the new primitives in more detail.

An ambient $n[P]$ is a boundary, named n, around the process P. The boundary prevents direct interactions between P and any processes running in parallel with $n[P]$, but it does not prevent interactions within P. Ambients may be nested, so they induce a hierarchy. For example, in the process displayed above, the ambient named m is a parent of the ambient named p, and the ambients named m and n are siblings.

An action $M.P$ exercises the capabilities represented by M, and then behaves as P. The action either affects an enclosing ambient or one running in parallel. A capability is an expression derived from the name of an ambient. The three basic capabilities are *in* n, *out* n, and *open* n. An action *in* $n.P$ moves its enclosing ambient into a sibling ambient named n. An action *out* $n.P$ moves its enclosing ambient out of its parent ambient, named n, to become a sibling of the former parent. An action *open* $n.P$ dissolves the boundary of an ambient $n[Q]$ running in parallel; the outcome is that the residue P of the action and the residue Q of the opened ambient run in parallel. In general, the expression M in $M.P$ may stand for a finite sequence of the basic capabilities, which are exercised one by one. Finite sequences are built up using concatenation, written $M.M'$. The empty sequence is written ϵ.

The final two process primitives allow communication of expressions. Expressions include names, variables, and capabilities. An output $\langle M \rangle$ outputs the expression M. An input $(x).P$ blocks until it may consume an output running in parallel. Then it binds the expression being output to the variable x, and runs P. In $(x).P$, the variable x is bound; its scope is P. Inputs and outputs are local to the enclosing ambient. Inputs and outputs may not interact directly through an ambient boundary. Hence we may think of there being an implicit input/output channel associated with each ambient.

We formally specify the syntax of the calculus as follows:

Expressions and processes:

$M, N ::=$	expressions	$P, Q, R ::=$	processes
x	variable	$(\nu n)P$	restriction
n	name	$\mathbf{0}$	inactivity
in M	can enter M	$P \mid Q$	composition

$out\ M$	can exit M	$!P$	replication
$open\ M$	can open M	$M[P]$	ambient
ϵ	null	$M.P$	action
$M.M'$	path	$(x).P$	input
		$\langle M \rangle$	output

In situations where a process is expected, we often write just M as a shorthand for the process $M.\mathbf{0}$. We often write just $M[]$ as a shorthand for the process $M[\mathbf{0}]$. We write $(\nu\vec{p})P$ as a shorthand for $(\nu p_1)\cdots(\nu p_k)P$ where $\vec{p} = p_1, \ldots, p_k$.

We let $fn(M)$ and $fv(M)$ be the sets of *free names* and *free variables*, respectively, of an expression M. Similarly, $fn(P)$ and $fv(P)$ are the sets of *free names* and *free variables* of a process P. If a phrase ϕ is an expression or a process, we write $\phi\{x \leftarrow M\}$ and $\phi\{n \leftarrow M\}$ for the outcomes of capture-avoiding substitutions of the expression M for each free occurrence of the variable x and the name n, respectively, in ϕ. We identify processes up to consistent renaming of bound names and variables.

We formally define the operational semantics of ambient calculus in the chemical style, using structural congruence and reduction relations:

Structural Congruence: $P \equiv Q$

$P \mid Q \equiv Q \mid P$	$P \equiv P$
$(P \mid Q) \mid R \equiv P \mid (Q \mid R)$	$Q \equiv P \Rightarrow P \equiv Q$
$!P \equiv P \mid !P$	$P \equiv Q, Q \equiv R \Rightarrow P \equiv R$
$(\nu n)(\nu m)P \equiv (\nu m)(\nu n)P$	
$n \notin fn(P) \Rightarrow (\nu n)(P \mid Q) \equiv P \mid (\nu n)Q$	$P \equiv Q \Rightarrow (\nu n)P \equiv (\nu n)Q$
$n \neq m \Rightarrow (\nu n)m[P] \equiv m[(\nu n)P]$	$P \equiv Q \Rightarrow P \mid R \equiv Q \mid R$
$P \mid \mathbf{0} \equiv P$	$P \equiv Q \Rightarrow\ !P \equiv\ !Q$
$(\nu n)\mathbf{0} \equiv \mathbf{0}$	$P \equiv Q \Rightarrow M[P] \equiv M[Q]$
$!\mathbf{0} \equiv \mathbf{0}$	$P \equiv Q \Rightarrow M.P \equiv M.Q$
$\epsilon.P \equiv P$	$P \equiv Q \Rightarrow (x).P \equiv (x).Q$
$(M.M').P \equiv M.M'.P$	

Reduction: $P \rightarrow Q$

$n[in\ m.P \mid Q] \mid m[R] \rightarrow m[n[P \mid Q] \mid R]$	$P \rightarrow Q \Rightarrow P \mid R \rightarrow Q \mid R$
$m[n[out\ m.P \mid Q] \mid R] \rightarrow n[P \mid Q] \mid m[R]$	$P \rightarrow Q \Rightarrow (\nu n)P \rightarrow (\nu n)Q$
$open\ n.P \mid n[Q] \rightarrow P \mid Q$	$P \rightarrow Q \Rightarrow n[P] \rightarrow n[Q]$
$\langle M \rangle \mid (x).P \rightarrow P\{x \leftarrow M\}$	$P' \equiv P, P \rightarrow Q, Q \equiv Q' \Rightarrow P' \rightarrow Q'$

For example, the process displayed earlier has the following reductions:

$$\begin{aligned} m[p[out\ m.in\ n.\langle M \rangle]] \mid n[open\ p.(x).P] &\rightarrow m[] \mid p[in\ n.\langle M \rangle] \mid n[open\ p.(x).P] \\ &\rightarrow m[] \mid n[p[\langle M \rangle] \mid open\ p.(x).P] \\ &\rightarrow m[] \mid n[\langle M \rangle \mid (x).P] \\ &\rightarrow m[] \mid n[P\{x \leftarrow M\}] \end{aligned}$$

The syntax allows the formation of certain processes that may not participate in any reductions, such as the action $n.P$ and the ambient $(inn)[P]$. The presence of these nonsensical processes is harmless as far as the purposes of this paper are concerned. They may be ruled out by a simple type system [7].

This concludes our brief review of the calculus. An earlier paper [6] explains in detail the motivation for our calculus, and gives several programming examples.

3 Contextual Equivalence

Morris-style contextual equivalence [14] (otherwise known as may-testing equivalence [8]) is a standard way of saying that two processes have the same behaviour: two processes are contextually equivalent if and only if they admit the same elementary observations whenever they are inserted inside any arbitrary enclosing process. In the setting of the ambient calculus, we shall define contextual equivalence in terms of observing the presence, at the top-level of a process, of an ambient whose name is not restricted.

Let us say that a process P *exhibits a name* n just if P is a process with a top-level ambient named n, that is not restricted:

Exhibition of a Name: $P \downarrow n$

$$P \downarrow n \quad \triangleq \quad \text{there are } \vec{m},\, P',\, P'' \text{ with } n \notin \{\vec{m}\} \text{ and } P \equiv (\nu \vec{m})(n[P'] \mid P'')$$

Let us say that a process P *converges to a name* n just if after some number of reductions, P exhibits n:

Convergence to a Name: $P \Downarrow n$

$$\text{(Conv Exh)}\ \frac{P \downarrow n}{P \Downarrow n} \qquad \text{(Conv Red)}\ \frac{P \to Q \qquad Q \Downarrow n}{P \Downarrow n}$$

Next, let a *context*, $\mathcal{C}()$, be a process containing zero or more holes. We write a hole as $()$. We write $\mathcal{C}(P)$ for the outcome of filling each of the holes in the context $\mathcal{C}$ with the process P. Variables and names free in P may become bound in $\mathcal{C}(P)$. For example, if $P = n[\langle x \rangle]$ and $\mathcal{C}() = (\nu n)(x).()$, the variable x and the name n have become bound in $\mathcal{C}(P) = (\nu n)(x).n[\langle x \rangle]$. Hence, we do not identify contexts up to renaming of bound variables and names.

Now, we can formally define contextual equivalence of processes:

Contextual Equivalence: $P \simeq Q$

$$P \simeq Q \quad \triangleq \quad \text{for all contexts } \mathcal{C}() \text{ and names } n,\, \mathcal{C}(P) \Downarrow n \Leftrightarrow \mathcal{C}(Q) \Downarrow n$$

The following two propositions state some basic properties enjoyed by contextual equivalence. Let a relation $\mathcal{R}$ be a *precongruence* if and only if, for all

P, Q, and $\mathcal{C}()$, if $P\ \mathcal{R}\ Q$ then $\mathcal{C}(P)\ \mathcal{R}\ \mathcal{C}(Q)$. If, in addition, $\mathcal{R}$ is reflexive, symmetric, and transitive, we say it is a *congruence*. For example, the structural congruence relation has these properties. Moreover, by a standard argument, so has contextual equivalence:

Proposition 1. *Contextual equivalence is a congruence.*

Structural congruence preserves exhibition of or convergence to a name, and hence is included in contextual equivalence:

Lemma 1. *Suppose $P \equiv Q$. If $P \downarrow n$ then $Q \downarrow n$. Moreover, if $P \Downarrow n$ then $Q \Downarrow n$ with the same depth of inference.*

Proposition 2. *If $P \equiv Q$ then $P \simeq Q$.*

The following two examples illustrate that to show that two processes are contextually inequivalent, it suffices to find a context that distinguishes them.

Example 1. If $m \neq n$ then $m[] \not\simeq n[]$.

Proof. Consider the context $\mathcal{C}() = ()$. Since $\mathcal{C}(m[]) \equiv m[]$, we have $\mathcal{C}(m[]) \downarrow m$. By (Conv Exh), $\mathcal{C}(m[]) \Downarrow m$. On the other hand, the process $n[]$ has no reductions, and does not exhibit m. Hence, we cannot derive $\mathcal{C}(n[]) \Downarrow m$. □

Example 2. If $m \neq n$ then $\mathit{open}\ m.\mathbf{0} \not\simeq \mathit{open}\ n.\mathbf{0}$.

Proof. Let $\mathcal{C}() = m[p[]] \mid ()$. Then $\mathcal{C}(\mathit{open}\, m.\mathbf{0}) \Downarrow p$ but not $\mathcal{C}(\mathit{open}\, n.\mathbf{0}) \Downarrow p$. □

On the other hand, it is harder to show that two processes are contextually equivalent, since one must consider their behaviour when placed in an arbitrary context. For example, consider the following contextual equivalence:

Example 3. $(\nu n)(n[] \mid \mathit{open}\ n.P) \simeq P$ if $n \notin \mathit{fn}(P)$.

The restriction of the name n in the process $(\nu n)(n[] \mid \mathit{open}\, n.P)$ implies that no context may interact with this process until it has reduced to P. Therefore, we would expect the equation to hold. But to prove this and other equations formally we need some further techniques, which we develop in the next section. We return to Example 3 in Section 5.

4 Tools for Proving Contextual Equivalence

The tools we introduce are relations and theorems that help prove contextual equivalence.

4.1 A Hardening Relation

In this section, we define a relation that explicitly identifies the top-level subprocesses of a process that may be involved in a reduction. This relation, the *hardening* relation, takes the form,

$$P > (\nu p_1, \ldots, p_k)\langle P' \rangle P''$$

where the phrase $(\nu p_1, \ldots, p_k)\langle P' \rangle P''$ is called a *concretion*. We say that P' is the *prime* of the concretion, and that P'' is the *residue* of concretion. Both P' and P'' lie in the scope of the restricted names $p_1, \ldots, p_k$. The intuition is that the process P, which may have many top-level subprocesses, may harden to a concretion that singles out a prime subprocess P', leaving behind the residue P''. By saying that P' has a top-level occurrence in P, we mean that P' is a subprocess of P not enclosed within any ambient boundaries. In the next section, we use the hardening relation to define an operational semantics for the ambient calculus in terms of interactions between top-level occurrences of processes.

Concretions were introduced by Milner in the context of the π-calculus [10]. For the ambient calculus, we specify them as follows, where the prime of the concretion must be an action, an ambient, an input, or an output:

Concretions:

$C, D ::=$	concretions
$(\nu \vec{p})\langle M.P \rangle Q$	action, $M \in \{in\ n, out\ n, open\ n\}$
$(\nu \vec{p})\langle n[P] \rangle Q$	ambient
$(\nu \vec{p})\langle (x).P \rangle Q$	input
$(\nu \vec{p})\langle \langle M \rangle \rangle Q$	output

The order of the bound names $p_1, \ldots, p_k$ in a concretion $(\nu p_1, \ldots, p_k)\langle P' \rangle P''$ does not matter and they may be renamed consistently. When $k = 0$, we may write the concretion as $(\nu)\langle P' \rangle P''$.

We now introduce the basic ideas of the hardening relation informally. If P is an action *in* $n.Q$, *out* $n.Q$, *open* $n.Q$, an ambient $n[Q]$, an input $(x).Q$, or an output $\langle M \rangle$, then P hardens to $(\nu)\langle P \rangle \mathbf{0}$. Consider two processes P and Q. If either of these hardens to a concretion, then their composition $P \mid Q$ may harden to the same concretion, but with the other process included in the residue of the concretion. For example, if $P > (\nu)\langle P_1 \rangle P_2$ then $P \mid Q > (\nu)\langle P_1 \rangle (P_2 \mid Q)$. If a process P hardens to a concretion, then the replication $!P$ may harden to the same concretion, but with $!P$ included in the residue of the concretion—a replication is not consumed by hardening. Finally, if a process P hardens to a concretion C, then the restriction $(\nu n)P$ hardens to a concretion written $\overline{(\nu n)}C$, which is the same as C but with the restriction (νn) included either in the list of bound names, the prime, or the residue of C. We define $\overline{(\nu n)}C$ by:

Restricting a concretion: $\overline{(\nu n)}C$ **where** $C = (\nu \vec{p})\langle P_1 \rangle P_2$ **and** $n \notin \{\vec{p}\}$

(1) If $n \in fn(P_1)$ then:

(a) If $P_1 = m[P_1']$, $m \neq n$, and $n \notin fn(P_2)$, let $\overline{(\nu n)}C \triangleq (\nu \vec{p})\langle m[(\nu n)P_1']\rangle P_2$.
(b) Otherwise, let $\overline{(\nu n)}C \triangleq (\nu n, \vec{p})\langle P_1\rangle P_2$.
(2) If $n \notin fn(P_1)$ let $\overline{(\nu n)}C \triangleq (\nu \vec{p})\langle P_1\rangle(\nu n)P_2$.

Next, we define the hardening relation by the following:

Hardening: $P > C$

$$\frac{M \in \{in\ n,\ out\ n,\ open\ n\}}{M.P > (\nu)\langle M.P\rangle \mathbf{0}} \quad \text{(Harden Action)} \qquad \frac{P > C}{\epsilon.P > C} \quad (\text{Harden } \epsilon) \qquad \frac{M.(N.P) > C}{(M.N).P > C} \quad (\text{Harden .})$$

$$\frac{}{n[P] > (\nu)\langle n[P]\rangle \mathbf{0}} \quad \text{(Harden Amb)} \qquad \frac{}{(x).P > (\nu)\langle (x).P\rangle \mathbf{0}} \quad \text{(Harden Input)} \qquad \frac{}{\langle M\rangle > (\nu)\langle \langle M\rangle\rangle \mathbf{0}} \quad \text{(Harden Output)}$$

$$\frac{P > (\nu \vec{p})\langle P'\rangle P''}{P \mid Q > (\nu \vec{p})\langle P'\rangle (P'' \mid Q)} \quad \text{(Harden Par 1) (for } \{\vec{p}\} \cap fn(Q) = \varnothing) \qquad \frac{Q > (\nu \vec{q})\langle Q'\rangle Q''}{P \mid Q > (\nu \vec{q})\langle Q'\rangle (P \mid Q'')} \quad \text{(Harden Par 2) (for } \{\vec{q}\} \cap fn(P) = \varnothing)$$

$$\frac{P > (\nu \vec{p})\langle P'\rangle P''}{!P > (\nu \vec{p})\langle P'\rangle (P'' \mid !P)} \quad \text{(Harden Repl)} \qquad \frac{P > C}{(\nu n)P > \overline{(\nu n)}C} \quad \text{(Harden Res)}$$

For example, the process $P = (\nu p)(\nu q)(n[p[]] \mid q[])$ may harden in two ways:

$$P > (\nu)\langle n[(\nu p)p[]]\rangle (\nu q)(\mathbf{0} \mid q[])$$
$$P > (\nu q)\langle q[]\rangle (\nu p)(n[p[]] \mid \mathbf{0})$$

The next two results relate hardening and structural congruence.

Lemma 2. *If $P > (\nu \vec{p})\langle P'\rangle P''$ then $P \equiv (\nu \vec{p})(P' \mid P'')$.*

Proposition 3. *If $P \equiv Q$ and $Q > (\nu \vec{r})\langle Q'\rangle Q''$ and then there are P' and P'' with $P > (\nu \vec{r})\langle P'\rangle P''$, $P' \equiv Q'$, and $P'' \equiv Q''$.*

These results follow from inductions on the derivations of $P > (\nu \vec{p})\langle P'\rangle P''$ and $P \equiv Q$, respectively. Using them, we may characterize exhibition of a name independently of structural congruence:

Proposition 4. *$P \downarrow n$ if and only if $P > (\nu \vec{p})\langle n[P']\rangle P''$ and $n \notin \{\vec{p}\}$.*

Now, we can show that the hardening relation is image-finite:

Lemma 3. *For all P, $\{C : P > C\}$ is finite.*

The proof of this lemma is by induction on the structure of P, and suggests a procedure for the enumerating the set $\{C : P > C\}$. Given Proposition 4, it follows that the predicate $P \downarrow n$ is decidable.

4.2 A Labelled Transition System

The labelled transition system presented in this section allows for an analysis of the possible reductions from a process P in terms of the syntactic structure of P. The definition of the reduction relation does not directly support such an analysis, because of the rule $P' \equiv P, P \to Q, Q \equiv Q' \Rightarrow P' \to Q'$, which allows for arbitrary structural rearrangements of a process during the derivation of a reduction.

We define a family of transition relations $P \xrightarrow{\alpha} Q$, indexed by a set of labels, ranged over by $\alpha ::= \tau \mid \mathit{in}\ n \mid \mathit{out}\ n \mid \mathit{open}\ n$. An M-transition $P \xrightarrow{M} Q$ means that the process P has a top-level process exercising the capability M; these transitions are defined by the rule (Trans Cap) below. A τ-transition $P \xrightarrow{\tau} Q$ means that P evolves in one step to Q; these transitions are defined by the other rules below.

Labelled transitions: $P \xrightarrow{\alpha} P'$ **where** $\alpha ::= \tau \mid \mathit{in}\ n \mid \mathit{out}\ n \mid \mathit{open}\ n$

(Trans Amb)
$$\frac{P > (\nu\vec{p})\langle n[Q]\rangle P' \quad Q \xrightarrow{\tau} Q'}{P \xrightarrow{\tau} (\nu\vec{p})(n[Q'] \mid P')}$$

(Trans Cap)
$$\frac{P > (\nu\vec{p})\langle M.P'\rangle P'' \quad \mathit{fn}(M) \cap \{\vec{p}\} = \varnothing}{P \xrightarrow{M} (\nu\vec{p})(P' \mid P'')}$$

(Trans In) (where $\{\vec{r}\} \cap \mathit{fn}(n[Q]) = \varnothing$ and $\{\vec{r}\} \cap \{\vec{p}\} = \varnothing$)
$$\frac{P > (\nu\vec{p})\langle n[Q]\rangle R \quad Q \xrightarrow{in\ m} Q' \quad R > (\nu\vec{r})\langle m[R']\rangle R''}{P \xrightarrow{\tau} (\nu\vec{p},\vec{r})(m[n[Q'] \mid R'] \mid R'')}$$

(Trans Out) (where $n \notin \{\vec{q}\}$)
$$\frac{P > (\nu\vec{p})\langle n[Q]\rangle P' \quad Q > (\nu\vec{q})\langle m[R]\rangle Q' \quad R \xrightarrow{out\ n} R'}{P \xrightarrow{\tau} (\nu\vec{p})((\nu\vec{q})(m[R'] \mid n[Q']) \mid P')}$$

(Trans Open)
$$\frac{P > (\nu\vec{p})\langle n[Q]\rangle P' \quad P' \xrightarrow{open\ n} P''}{P \xrightarrow{\tau} (\nu\vec{p})(Q \mid P'')}$$

(Trans I/O) (where $\{\vec{q}\} \cap \mathit{fn}(\langle M\rangle) = \varnothing$)
$$\frac{P > (\nu\vec{p})\langle\langle M\rangle\rangle P' \quad P' > (\nu\vec{q})\langle(x).P''\rangle P'''}{P \xrightarrow{\tau} (\nu\vec{p})(P' \mid (\nu\vec{q})(P''\{x{\leftarrow}M\} \mid P'''))}$$

The rules (Trans In), (Trans Out), and (Trans Open) derive a τ-transition from an M-transition. We introduced the M-transitions to simplify the statement of these three rules. (Trans I/O) allows for exchange of messages. (Trans Amb) is a congruence rule for τ-transitions within ambients.

Given its definition in terms of the hardening relation, we may analyze the transitions derivable from any process by inspection of its syntactic structure. This allows a structural analysis of the possible reductions from a process, since the τ-transition relation corresponds to the reduction relation as in the following theorem, where $P \xrightarrow{\tau} \equiv Q$ means there is R with $P \xrightarrow{\tau} R$ and $R \equiv Q$.

Theorem 1. *$P \to Q$ if and only if $P \xrightarrow{\tau} \equiv Q$.*

As corollaries of Theorem 1 and Lemma 4, we get that the transition system is image-finite, and that the reduction relation is image-finite up to structural congruence:

Lemma 4. *For all P and α, the set $\{R : P \xrightarrow{\alpha} R\}$ is finite.*

Lemma 5. *For all P, the set $\{\{R : Q \equiv R\} : P \to Q\}$ is finite.*

4.3 A Context Lemma

The context lemma presented in this section is a tool for proving contextual equivalence by considering only a limited set of contexts, rather than all contexts. Many context lemmas have been proved for a wide range of calculi, starting with Milner's context lemma for the combinatory logic form of PCF [9].

Our context lemma is stated in terms of a notion of a *harness*:

Harnesses

$H ::=$	harnesses
$-$	process variable
$(\nu n)H$	restriction
$P \mid H$	left composition
$H \mid Q$	right composition
$n[H]$	ambient

Harnesses are analogous to the evaluation contexts found in context lemmas for some other calculi. Unlike the contexts of Section 3, harnesses are identified up to consistent renaming of bound names. We let $fn(H)$ and $fv(H)$ be the sets of names and variables, respectively, occurring free in a harness H. There is exactly one occurrence of the process variable $-$ in any harness. If H is an harness, we write $H\{P\}$ for the outcome of substituting the process P for the single occurrence of the process variable $-$. Names restricted in H are renamed to avoid capture of free names of P. For example, if $H = (\nu n)(- \mid \mathit{open}\ n)$ then $H\{n[]\} = (\nu n')(n[] \mid \mathit{open}\ n')$ for some $n' \neq n$.

Let a *substitution*, σ, be a list $x_1 \leftarrow M_1, \ldots, x_k \leftarrow M_k$, where the variables x_1, $\ldots$, x_k are pairwise distinct, and $fv(M_i) = \varnothing$ for each $i \in 1..k$. Let $dom(\sigma) = \{x_1, \ldots, x_k\}$. Let $P\sigma$ be the process $P\{x_1 \leftarrow M_1\} \cdots \{x_k \leftarrow M_k\}$. Let a process or a harness be *closed* if and only if it has no free variables (though it may have free names).

Here is our context lemma:

Theorem 2 (Context). *For all processes P and Q, $P \simeq Q$ if and only if for all substitutions σ with $dom(\sigma) = fv(P) \cup fv(Q)$, and for all closed harnesses H and names n, that $H\{P\sigma\} \Downarrow n \Leftrightarrow H\{Q\sigma\} \Downarrow n$.*

A corollary is that for all closed processes P and Q, $P \simeq Q$ if and only if for all closed harnesses H and names n, that $H\{P\} \Downarrow n \Leftrightarrow H\{Q\} \Downarrow n$.

In general, however, we need to consider the arbitrary closing substitution σ when using Theorem 2. This is because a variable free in a process may become bound to an expression once the process is placed in a context. For example, let $P = x[n[]] \mid \mathit{open}\ y.\mathbf{0}$ and $Q = \mathbf{0}$. Consider the context $\mathcal{C}() = \langle m, m\rangle \mid (x, y).()$.

We have $\mathcal{C}(P) \Downarrow n$ but not $\mathcal{C}(Q) \Downarrow n$. So P and Q are not contextually equivalent but they do satisfy $H\{P\} \Downarrow n \Leftrightarrow H\{Q\} \Downarrow n$ for all closed H and n.

Some process calculi enjoy stronger context lemmas. Let processes P and Q be *parallel testing equivalent* if and only if for all processes R and names n, that $P \mid R \Downarrow n \Leftrightarrow Q \mid R \Downarrow n$. We might like to show that any two closed processes are contextually equivalent if and only if they are parallel testing equivalent. This would be a stronger result than Theorem 2 because it would avoid considering contexts that include ambients. Such a result is true for CCS [8], for example, but it is false for the ambient calculus. To see this, let $P = \mathit{out}\, p.\mathbf{0}$ and $Q = \mathbf{0}$. We may show that $P \mid R \Downarrow n \Leftrightarrow Q \mid R \Downarrow n$ for all n and R. Now, consider the context $\mathcal{C}() = p[m[()]]$. We have $\mathcal{C}(P) \Downarrow m$ but not $\mathcal{C}(\mathbf{0}) \Downarrow m$. So P and Q are parallel testing equivalent but not contextually equivalent.

4.4 An Activity Lemma

When we come to apply Theorem 2 we need to analyze judgments of the form $H\{P\} \downarrow n$ or $H\{P\} \rightarrow Q$. In this section we formalize these analyses.

We begin by extending the structural congruence, hardening, and reduction relations to harnesses as follows:

- Let $H \equiv H'$ hold if and only if $H\{P\} \equiv H'\{P\}$ for all P.
- Let $H > (\nu\vec{p})\langle H'\rangle Q$ hold if and only if $H\{P\} > (\nu\vec{p})\langle H'\{P\}\rangle Q$ for all P such that $\{\vec{p}\} \cap \mathit{fn}(P) = \varnothing$.
- Let $H > (\nu\vec{p})\langle Q\rangle H'$ hold if and only if $H\{P\} > (\nu\vec{p})\langle Q\rangle(H'\{P\})$ for all P such that $\{\vec{p}\} \cap \mathit{fn}(P) = \varnothing$.
- Let $H \rightarrow H'$ hold if and only if, for all P, $H\{P\} \rightarrow H'\{P\}$.

We need the following lemma about hardening:

Lemma 6. *If $H\{P\} > C$ then either:*

(1) $H > (\nu\vec{r})\langle H'\rangle R$ *and* $C = (\nu\vec{r})\langle H'\{P\}\rangle R$, *or*
(2) $H > (\nu\vec{r})\langle R\rangle H'$ *and* $C = (\nu\vec{r})\langle R\rangle(H'\{P\})$, *or*
(3) $H > (\nu\vec{r})\langle -\rangle R$, $P > (\nu\vec{p})\langle P'\rangle P''$, $C = (\nu\vec{r},\vec{p})\langle P'\rangle R'$ *with* $R' \equiv P'' \mid R$,

where in each case $\{\vec{r}\} \cap \mathit{fn}(P) = \varnothing$.

Proposition 5. *If $H\{P\} \downarrow n$ then either (1) $H\{Q\} \downarrow n$ for all Q, or (2) $P \downarrow n$, and for all Q, $Q \downarrow n$ implies that $H\{Q\} \downarrow n$.*

Proof. By Proposition 4, $H\{P\} \downarrow n$ means there are $\vec{p}$, P', P'' such that $H\{P\} > (\nu\vec{p})\langle n[P']\rangle P''$ with $n \notin \{\vec{p}\}$. Hence, the proposition follows from Lemma 6. □

Intuitively, there are two ways in which $H\{P\} \downarrow n$ can arise: either the process P exhibits the name by itself, or the harness H exhibits the name n by itself. Proposition 5 formalizes this analysis. Similarly, there are three ways in which a reduction $H\{P\} \rightarrow Q$ may arise: either (1) the process P reduces by itself, or (2) the harness H reduces by itself, or (3) there is an interaction between the process and the harness. Theorem 3 formalizes this analysis. Such a result is sometimes known as an activity lemma [15].

Theorem 3 (Activity). *$H\{P\} \to R$ if and only if:*

(Act Proc) *there is a reduction $P \to P'$ with $R \equiv H\{P'\}$, or*
(Act Har) *there is a reduction $H \to H'$ with $R \equiv H'\{P\}$, or*
(Act Inter) *there are H' and $\vec{r}$ with $\{\vec{r}\} \cap fn(P) = \varnothing$, and one of the following holds:*

(Inter In) $H \equiv (\nu\vec{r})H'\{m[- \mid R'] \mid n[R'']\}$, $P \xrightarrow{in\ n} P'$,
and $R \equiv (\nu\vec{r})H'\{n[m[P' \mid R'] \mid R'']\}$

(Inter Out) $H \equiv (\nu\vec{r})H'\{n[m[- \mid R'] \mid R'']\}$, $P \xrightarrow{out\ n} P'$,
and $R \equiv (\nu\vec{r})H'\{m[P' \mid R'] \mid n[R'']\}$

(Inter Open) $H \equiv (\nu\vec{r})H'\{- \mid n[R']\}$, $P \xrightarrow{open\ n} P'$,
and $R \equiv (\nu\vec{r})H'\{P' \mid R'\}$

(Inter Input) $H \equiv (\nu\vec{r})H'\{- \mid \langle M\rangle\}$, $P > (\nu\vec{p})\langle(x).P'\rangle P''$,
and $R \equiv (\nu\vec{r})H'\{(\nu\vec{p})(P'\{x{\leftarrow}M\} \mid P'')\}$, *with* $\{\vec{p}\} \cap fn(M) = \varnothing$

(Inter Output) $H \equiv (\nu\vec{r})H'\{- \mid (x).R'\}$, $P > (\nu\vec{p})\langle\langle M\rangle\rangle P'$,
and $R \equiv (\nu\vec{r})H'\{(\nu\vec{p})(P' \mid R'\{x{\leftarrow}M\})\}$, *with* $\{\vec{p}\} \cap fn(R') = \varnothing$

(Inter Amb) $P > (\nu\vec{p})\langle n[Q]\rangle P'$ *and one of the following holds:*

(1) $Q \xrightarrow{in\ m} Q'$, $H \equiv (\nu\vec{r})H'\{- \mid m[R']\}$, $\{\vec{p}\} \cap fn(m[R']) = \varnothing$,
and $R \equiv (\nu\vec{r})H'\{(\nu\vec{p})(P' \mid m[n[Q'] \mid R'])\}$

(2) $Q \xrightarrow{out\ m} Q'$, $H \equiv (\nu\vec{r})H'\{m[- \mid R']\}$, $m \notin \{\vec{p}\}$,
and $R \equiv (\nu\vec{r})H'\{(\nu\vec{p})(n[Q'] \mid m[P' \mid R'])\}$

(3) $H \equiv (\nu\vec{r})H'\{m[R' \mid in\ n.R''] \mid -\}$, $\{\vec{p}\} \cap fn(m[R' \mid in\ n.R'']) = \varnothing$,
and $R \equiv (\nu\vec{r})H'\{(\nu\vec{p})(n[Q \mid m[R' \mid R'']] \mid P')\}$

(4) $H \equiv (\nu\vec{r})H'\{- \mid open\ n.R'\}$, $n \notin \{\vec{p}\}$,
and $R \equiv (\nu\vec{r})H'\{(\nu\vec{p})(Q \mid P') \mid R'\}$

5 Examples of Contextual Equivalence

In this section, two examples demonstrate how we may apply Theorem 2 and Theorem 3 to establish contextual equivalence.

5.1 Opening an Ambient

We can now return to and prove Example 3 from Section 3.

Lemma 7. *If $H\{(\nu n)(n[] \mid open\ n.P)\} \Downarrow m$ and $n \notin fn(P)$ then $H\{P\} \Downarrow m$.*

Proof. By induction on the derivation of $H\{(\nu n)(n[] \mid open\ n.P)\} \Downarrow m$, with appeal to Propositions 4 and 5, and Theorems 1 and 3. □

Proof of Example 3 $(\nu n)(n[] \mid open\ n.P) \simeq P$ *if* $n \notin fn(P)$.

Proof. By Theorem 2, it suffices to prove $H\{((\nu n)(n[] \mid open\ n.P))\sigma\} \Downarrow m \Leftrightarrow H\{P\sigma\} \Downarrow m$ for all closed harnesses H and names m and for all substitutions σ with $dom(\sigma) = fv(P)$. Since the name n is bound, we may assume that $n \notin$

$fn(\sigma(x))$ for all $x \in dom(\sigma)$. Therefore, we are to prove that: $H\{(\nu n)(n[] \mid open\ n.P\sigma\} \Downarrow m \Leftrightarrow H\{P\sigma\} \Downarrow m$ where $n \notin fn(P\sigma)$.

We prove each direction separately. First, suppose that $H\{P\sigma\} \Downarrow m$. Since $(\nu n)(n[] \mid open\ n.P\sigma) \rightarrow P\sigma$, we get $H\{(\nu n)(n[] \mid open\ n.P\sigma)\} \rightarrow H\{P\sigma\}$. By (Exh Red), we get $H\{(\nu n)(n[] \mid open\ n.P\sigma)\} \Downarrow m$. Second, suppose that $H\{(\nu n)(n[] \mid open\ n.P\sigma)\} \Downarrow m$. By Lemma 7, we get $H\{P\sigma\} \Downarrow m$. □

5.2 The Perfect Firewall Equation

Consider a process $(\nu n)n[P]$, where n is not free in P. Since the name n is known neither inside the ambient $n[P]$, nor outside it, the ambient $n[P]$ is a "perfect firewall" that neither allows another ambient to enter nor to exit. The following two lemmas allow us to prove that $(\nu n)n[P]$ is contextually equivalent to $\mathbf{0}$, when $n \notin fn(P)$, which is to say that no context can detect the presence of $(\nu n)n[P]$.

Lemma 8. *If* $H\{(\nu n)n[P]\} \Downarrow m$ *and* $n \notin fn(P)$ *then* $H\{\mathbf{0}\} \Downarrow m$.

Proof. By induction on the derivation of $H\{(\nu n)n[P]\} \Downarrow m$.

(Conv Exh) Here $H\{(\nu n)n[P]\} \downarrow m$. By Proposition 5, either (1), for all Q, $H\{Q\} \downarrow m$, or (2), $(\nu n)n[P] \downarrow m$. In case (1), we have, in particular, that $H\{\mathbf{0}\} \downarrow m$. Hence, $H\{\mathbf{0}\} \Downarrow m$, by (Conv Exh). Case (2) cannot arise, since, by Proposition 4, $(\nu n)n[P] \downarrow m$ implies that $(\nu n)n[P] > (\nu \vec{p})\langle m[P']\rangle P''$ with $m \notin \{\vec{p}\}$, which is impossible.

(Conv Red) Here $H\{(\nu n)n[P]\} \rightarrow R$ and $R \Downarrow m$. By Theorem 3, one of three cases pertains:

(Act Proc) Then $(\nu n)n[P] \rightarrow P''$ with $R \equiv H\{P''\}$. By Theorem 1, there is Q with $(\nu n)n[P] \xrightarrow{\tau} Q$ and $Q \equiv P''$. Since $(\nu n)n[P] > (\nu n)\langle n[P]\rangle \mathbf{0}$ is the only hardening derivable from $(\nu n)n[P]$, the transition $(\nu n)n[P] \xrightarrow{\tau} Q$ can only be derived using (Trans Amb), with $P \xrightarrow{\tau} P'$ and $Q = (\nu n)(n[P'] \mid \mathbf{0})$. Therefore, there is a reduction $P \rightarrow P'$ and $P'' \equiv (\nu n)n[P']$. We may show that $P \rightarrow P'$ implies $fn(P') \subseteq fn(P)$, and so $n \notin fn(P')$. We have $R \equiv H\{(\nu n)n[P']\}$ with $n \notin fn(P')$. By Lemma 1, we may derive $H\{(\nu n)n[P']\} \Downarrow m$ by the same depth of inference as $R \Downarrow m$. By induction hypothesis, $H\{\mathbf{0}\} \Downarrow m$.

(Act Har) Then $H \rightarrow H'$ with $R \equiv H'\{(\nu n)n[P]\}$. By Lemma 1, we may derive $H'\{(\nu n)n[P]\} \Downarrow m$ by the same depth of inference as $R \Downarrow m$. By induction hypothesis, $H'\{\mathbf{0}\} \Downarrow m$. From $H \rightarrow H'$ we obtain $H\{\mathbf{0}\} \rightarrow H'\{\mathbf{0}\}$ in particular. By (Conv Red), we get $H\{\mathbf{0}\} \Downarrow m$.

(Act Inter) Then there are H' and $\vec{r}$ with $\{\vec{r}\} \cap fn(P) = \varnothing$ and one of several conditions must hold. Since the only hardening or transition from $(\nu n)n[P]$ is $(\nu n)n[P] > (\nu n)\langle n[P]\rangle\mathbf{0}$, only the rule (Inter Amb) applies. According to Theorem 3, there are four possibilities to consider.

(1) Here, $P \stackrel{in\ m}{\longrightarrow} P'$, $H \equiv (\nu\vec{r})H'\{- \mid m[R']\}$, $\{n\} \cap fn(m[R']) = \varnothing$, and $R \equiv (\nu\vec{r})H'\{(\nu n)(\mathbf{0} \mid m[n[P'] \mid R'])\}$. We have $R \equiv (\nu\vec{r})H'\{m[R' \mid (\nu n)n[P']]\}$ and that $n \notin fn(P')$. By Lemma 1, we get $(\nu\vec{r})H'\{m[R' \mid (\nu n)n[P']]\} \Downarrow m$ with the same depth of inference as $R \Downarrow m$. By induction hypothesis, $(\nu\vec{r})H'\{m[R' \mid \mathbf{0}]\} \Downarrow m$. Moreover, $H\{\mathbf{0}\} \equiv (\nu\vec{r})H'\{m[R' \mid \mathbf{0}]\}$, and therefore $H\{\mathbf{0}\} \Downarrow m$.

(2) Here, $P \stackrel{out\ m}{\longrightarrow} P'$, $H \equiv (\nu\vec{r})H'\{m[- \mid R']\}$, $m \notin \{n\}$, and also $R \equiv (\nu\vec{r})H'\{(\nu n)(n[P'] \mid m[\mathbf{0} \mid R'])\}$. We have $R \equiv (\nu\vec{r})H'\{m[R'] \mid (\nu n)n[P']\}$ and that $n \notin fn(P')$. By Lemma 1, we get $(\nu\vec{r})H'\{m[R'] \mid (\nu n)n[P']\} \Downarrow m$ with the same depth of inference as $R \Downarrow m$. By induction hypothesis, $(\nu\vec{r})H'\{m[R'] \mid \mathbf{0}\} \Downarrow m$. Moreover, $H\{\mathbf{0}\} \equiv (\nu\vec{r})H'\{m[R'] \mid \mathbf{0}\}$ and therefore $H\{\mathbf{0}\} \Downarrow m$.

The other possibilities, (3) and (4), are ruled out because the name n is restricted in the concretion $(\nu n)\langle n[P]\rangle\mathbf{0}$. □

By a similar induction, we can also prove:

Lemma 9. *If $H\{\mathbf{0}\} \Downarrow m$ then $H\{P\} \Downarrow m$.*

By combining Theorem 2, Lemmas 8 and 9, we get:

Example 4. If $n \notin fn(P)$ then $(\nu n)n[P] \simeq \mathbf{0}$.

Our first proof of this equation (which was stated in an earlier paper [6]) was by a direct quantification over all contexts. The proof above using the context lemma is simpler.

6 Conclusions

We developed a theory of Morris-style contextual equivalence for the ambient calculus. We showed that standard tools such as a labelled transition system, a context lemma, and an activity lemma, may be adapted to the ambient calculus. We introduced a new technique, based on a hardening relation, for defining the labelled transition system. We employed these tools to prove equational properties of mobile ambients.

Our use of concretions to highlight those subprocesses of a process that may participate in a computation follows Milner [10, 11], and is an alternative to the use of membranes and airlocks in the chemical abstract machine of Berry and Boudol [5]. Unlike these authors, in the definition of our transition relation we use the hardening relation, rather than the full structural congruence relation, to choose subprocesses to participate in a transition. Hardening is more convenient in some proofs, such as the proof that the labelled transition system is image-finite, Lemma 4.

In the future, it would be of interest to study bisimulation of ambients. Various techniques adopted for higher-order [13, 17] and distributed [4, 3, 16] variants of the π-calculus may be applicable to the ambient calculus.

Acknowledgement Comments by Cédric Fournet, Georges Gonthier, and Tony Hoare were helpful.

References

1. M. Abadi, C. Fournet, and G. Gonthier. Secure implementation of channel abstractions. In *Proceedings LICS'98*, pages 105–116, 1998.
2. M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*. To appear. An extended version appears as Digital Equipment Corporation Systems Research Center report No. 149, January 1998.
3. R. M. Amadio. An asynchronous model of locality, failure, and process mobility. In *Proceedings COORDINATION 97*, volume 1282 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.
4. R. M. Amadio and S. Prasad. Localities and failures. In *Proceedings FST&TCS'94*, volume 880 of *Lecture Notes in Computer Science*, pages 205–216. Springer-Verlag, 1994.
5. G. Berry and G. Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96(1):217–248, April 1992.
6. L. Cardelli and A. D. Gordon. Mobile ambients. In *Proceedings FoSSaCS'98*, volume 1378 of *Lecture Notes in Computer Science*, pages 140–155. Springer-Verlag, 1998.
7. L. Cardelli and A. D. Gordon. Types for mobile ambients. In *Proceedings POPL'99*, 1999. To appear.
8. R. De Nicola and M. C. B. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34:83–133, 1984.
9. R. Milner. Fully abstract models of typed lambda-calculi. *Theoretical Computer Science*, 4:1–23, 1977.
10. R. Milner. The polyadic π-calculus: A tutorial. Technical Report ECS–LFCS–91–180, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, October 1991.
11. R. Milner. The π-calculus. Undergraduate lecture notes, Cambridge University, 1995.
12. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, parts I and II. *Information and Computation*, 100:1–40 and 41–77, 1992.
13. R. Milner and D. Sangiorgi. Barbed bisimulation. In *Proceedings ICALP'92*, volume 623 of *Lecture Notes in Computer Science*. Springer-Verlag, 1992.
14. J. H. Morris. *Lambda-Calculus Models of Programming Languages*. PhD thesis, MIT, December 1968.
15. G. D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255, 1977.
16. J. Riely and M. Hennessy. A typed language for distributed mobile processes. In *Proceedings POPL'98*, pages 378–390, 1998.
17. D. Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms*. PhD thesis, University of Edinburgh, 1992. Available as Technical Report CST–99–93, Computer Science Department, University of Edinburgh.

Model Checking Logics for Communicating Sequential Agents*

Michaela Huhn[1] and Peter Niebert[2] and Frank Wallner[3]

[1] Institut für Rechnerentwurf und Fehlertoleranz (Prof. D. Schmid), Univ. Karlsruhe, Postf. 6980, D-76128 Karlsruhe, huhn@ira.uka.de
[2] VERIMAG, 2, av. de Vignate, 38610 Gières, France, niebert@imag.fr
[3] Institut für Informatik, Technische Universität München, D-80290 München, wallnerf@in.tum.de

Abstract. We present a model checking algorithm for $\mathcal{L}_{CSA}$, a temporal logic for *communicating sequential agents* (CSAs) introduced by Lodaya, Ramanujam, and Thiagarajan. $\mathcal{L}_{CSA}$ contains temporal modalities indexed with a local point of view of one agent and allows to refer to properties of other agents according to the *latest gossip* which is related to *local knowledge*.
The model checking procedure relies on a modularisation of $\mathcal{L}_{CSA}$ into temporal and gossip modalities. We introduce a hierarchy of formulae and a corresponding hierarchy of equivalences, which allows to compute for each formula and finite state distributed system a finite multi modal Kripke structure, on which the formula can be checked with standard techniques.

1 Introduction

A reasonable and lucid way of formally treating distributed systems is to consider them as a fixed collection of sequential components (agents) which can operate independently as well as cooperate by exchanging information. There is an increasing awareness, both in theory and practice, of the benefits of specifying the requirements of such systems by *localised*, component based formalisms, that allow to refer to properties of the individual components.

The operational models for localised specification usually consist of *local* temporal orders (sequences in the linear time case, trees in branching time) together with an interrelation between these orders, descended from communication [LRT92,Ram95]. The most established models for the linear time case are partial orders, whereas in the branching time setting, *(prime) event structures* or closely related models like *occurrence nets* [NPW80,Win87] have been recognised to be a suitable formalism. In these models, partial orders are extended by an additional conflict relation, representing the moments of choice.

* This work was partially supported by the DFG within SFB 342 (A3), and within the priority program "*Design and design methodology of embedded systems*", and by the EEC program on *Training and Mobility in Research* (TMR). The work was done while the second author was affiliated with the Univ. of Hildesheim, Germany.

W. Thomas (Ed.): FOSSACS'99, LNCS 1578, pp. 227-242, 1999.

Investigating partial order models has attained the interest of researchers for mainly two reasons: There is no distinction among computations that are equal up to possible total orderings of independent actions, which makes it a faithful and natural formalism for representing concurrency. Furthermore, restricting the attention to local states mitigates one of the most tackled difficulty of model checking, the so-called *state explosion problem*, which results from an explicit computation of the global state space of a distributed system.

For a component-oriented specification of behaviour, local linear time temporal logics have been investigated by Thiagarajan in [Thi94,Thi95] and Niebert [Nie98]. Local branching time logics were introduced in [LT87,LRT92,HNW98b]. While for the linear time case there now exist sound model checking procedures based on automata [Thi94,Nie98], only recently the model checking problem for local branching time logics has been inspected [Pen97,HNW98b].

In this paper, we investigate model checking for a local branching time logic defined by Lodaya, Ramanujam and Thiagarajan in [LRT92], here called $\mathcal{L}_{CSA}$, which is intended to specify the behaviour of *communicating sequential agents* (CSAs). It allows a component i to refer to local properties of another component j according to the *latest gossip* (in [LRT92] also called local knowledge), i.e., the most recent j-local state that causally precedes the current i-local state.

Based on net unfoldings [Eng91] and McMillan's *finite prefix* construction [McM92], we solve the model checking problem for $\mathcal{L}_{CSA}$, which remained open since [LRT92].

McMillan's prefix has successfully been applied to alleviate state explosion in many verification problems, for instance deadlock detection [McM92], and model checking S4 [Esp94], LTL [Wal98], and the distributed μ-calculus [HNW98b]. All of the previous problems principally can be solved with conventional state space exploration, but often with an exponentially higher effort.

The focus of this paper is to show decidability of model checking $\mathcal{L}_{CSA}$. Generalising the techniques of [HNW98b], we demonstrate that the unfolding approach is very suitable for model checking a wider class of local logics, for which previously the problem appeared to be too difficult.

Technically, we proceed as follows: We lift the semantics of $\mathcal{L}_{CSA}$ from CSAs onto net unfoldings, and factorise the net unfolding with respect to an equivalence relation satisfying two key properties: It is a congruence for the $\mathcal{L}_{CSA}$-specification to be checked, and it has finite index. Via this factorisation, the $\mathcal{L}_{CSA}$ model checking problem can be transformed into a model checking problem for a multi modal logic on a finite transition system constructed upon a modified McMillan prefix, using the defined equivalence relation as cutoff condition. With an appropriate interpretation of the $\mathcal{L}_{CSA}$ modalities, standard model checking algorithms, e.g. [CES86], can be applied on this transition system.

The approach follows the lines of [HNW98b], but whereas the focus there was to derive an algorithm for calculating the transition system, the main difficulty here is to develop an appropriate equivalence relation. The modalities of the distributed μ-calculus of [HNW98b] are purely future oriented, while the past and also the gossip modalities of $\mathcal{L}_{CSA}$ may lead to rather complex patterns

within the past of a configuration. As a consequence, the coarsest equivalence preserving *all* $\mathcal{L}_{CSA}$ properties has non-finite index and it is not possible to construct a single (finite-state) transition system representing all $\mathcal{L}_{CSA}$ properties of a particular finite state distributed system. However, a single $\mathcal{L}_{CSA}$ formula has a limited power of referring to the past so that we can construct an equivalence depending on the formula. For this purpose, we introduce a syntactic hierarchy of formulae and a corresponding equivalence hierarchy. The construction of these equivalences and the proof of their soundness are both complex, and the resulting model checking complexity of the construction given here is high.

The technical presentation of the paper relies on notions from Petri net theory, mainly to correspond directly to McMillan's prefix. Note however, that the entire method can easily be restated for other formalisms, like e.g. asynchronous automata, coupled finite state machines, and so forth.

The paper is structured as follows. In Section 2 we introduce distributed net systems, and their unfoldings as semantic model of branching behaviour. In Section 3 we introduce the logic $\mathcal{L}_{CSA}$ and our slightly generalised version $\mathcal{L}$. In Section 4 we present McMillan's finite prefix, and parameterise its definition by an abstract equivalence relation. Then we develop an appropriate equivalence for $\mathcal{L}$. In Section 5 we use this equivalence to compute a finite state transition system, on which the model checking problem for $\mathcal{L}$ can be solved by conventional model checkers. In Section 6, we discuss our results and indicate future work.

2 Distributed net systems and their unfoldings

Petri nets. Let P and T be disjoint, finite sets of *places* and *transitions*, generically called *nodes*. A *net* is a triple $N = (P, T, F)$ with a *flow relation* $F \subseteq (P \times T) \cup (T \times P)$. The *preset* of a node x is defined as ${}^\bullet x := \{y \in P \cup T \mid yFx\}$ and its *postset* as $x^\bullet := \{y \in P \cup T \mid xFy\}$. The preset (resp. postset) of a set X of nodes is the union of the presets (resp. postsets) of all nodes in X.

A *marking* of a net is a mapping $M : P \to \mathbb{N}_0$. If $M(p) = n$, we say that p contains n *tokens* at M. A *net system* $\Sigma = (N, M_0)$ consists of a net N, and an *initial marking* M_0. The marking M *enables* the transition t if every place in the preset of t contains at least one token. In this case the transition can *occur*. If t occurs, it removes one token from each place $p \in {}^\bullet t$ and adds one token to each place $p' \in t^\bullet$, yielding a new marking M'. We denote this occurrence by $M \xrightarrow{t} M'$. If there exists a chain $M_0 \xrightarrow{t_1} M_1 \xrightarrow{t_2} \ldots \xrightarrow{t_n} M_n$ for $n \geq 0$, then the marking M_n is a *reachable marking*.

We will restrict our attention to *1-safe net systems*, in which every reachable marking M puts at most one token on each place, and thus can be identified by the subset of places that contain a token, i.e., $M \subseteq P$.

In the last years, 1-safe net systems have become a significant model [CEP95]. In [NRT90] it has been shown that an instance of 1-safe nets, called *Elementary Net Systems*, correspond to other models of concurrency, such as (Mazurkiewicz) traces and prime event structures. They can naturally be interpreted as a synchronised product of several finite automata, and thus they are frequently used

as a convenient formalism for modelling distributed systems. In the following we will exploit this compositional view by considering the notion of *locations*.

Distributed net systems. Let us introduce the formalism for describing distributed systems. Clearly, the *behaviour* of our models shall resemble the *Communicating Sequential Agents* of [LRT92]. This means, a system consists of several distributed, autonomous agents, which mutually communicate. Each of the agents shall behave strictly sequentially, and non-deterministically.

Let Σ be a 1-safe net system, and t, t' two transitions of Σ. A marking M *concurrently* enables t and t' if M enables t, and $(M \setminus {}^\bullet t)$ enables t'. We call Σ *sequential* if no reachable marking concurrently enables two transitions.

Let $\{\Sigma_i = (P_i, T_i, F_i, M_i^0) \mid i \in Loc\}$ be a family of 1-safe, sequential net systems (called *agents*, or *components*) with pairwise disjoint sets P_i of places, indexed by a finite set Loc of *locations*. Note that the sets of transitions are not necessarily disjoint. In fact, we will interpret the execution of a transition that is common to several agents as a *synchronous communication* action of these agents, i.e., the communication capabilities are given by the common execution of joint transitions. Formally, a *distributed net system* $\Sigma_{Loc} = (N, M_0)$ is defined as the union of its components Σ_i:

$$P = \bigcup_{i \in Loc} P_i\,, \qquad T = \bigcup_{i \in Loc} T_i\,, \qquad F = \bigcup_{i \in Loc} F_i\,, \qquad M_0 = \bigcup_{i \in Loc} M_i^0\,.$$

Clearly, Σ_{Loc} is again 1-safe. The *location* $loc(x)$ of a node x is defined by $loc(x) := \{i \in Loc \mid x \in P_i \cup T_i\}$. A simple distributed net system consisting of two components is depicted in Fig. 1.

In [LRT92] also *asynchronous* communication (message passing) is considered. However, in general this yields systems with infinitely many states, making an algorithmic, state space based approach to model checking impossible. To model the asynchronous setting, we can assume some finite-state communication mechanism like e.g. bounded channels or buffers, which can easily be defined within the presented framework by considering a buffer as an agent of its own, (synchronously) communicating with both the agents that communicate (asynchronously) via this buffer.

Net unfoldings. As a partial order semantics of the behaviour of a distributed net system, we consider *net unfoldings*, also known as *branching processes*. They contain information about both concurrency and conflict.

Two nodes x, x' of a net (P, T, F) are *in conflict*, denoted $x \# x'$, if there exist two distinct transitions t, t' such that ${}^\bullet t \cap {}^\bullet t' \neq \emptyset$, and $(t, x), (t', x')$ belong to the reflexive, transitive closure of F. If $x \# x$, we say x *is in self-conflict*.

An *occurrence net* [NPW80] is a net $N' = (B, E, F)$ with the following properties: (1) for every $b \in B$, $|{}^\bullet b| \leq 1$, (2) the irreflexive transitive closure $<$ of F is well-founded and acyclic, i.e., for every node $x \in B \cup E$, the set $\{y \in B \cup E \mid y < x\}$ is finite and does not contain x, and (3) no element $e \in E$ is in self-conflict. The reflexive closure $\leq$ of $<$ is a partial order, called *causality relation*. In occurrence nets we speak of *conditions* and *events* instead of places and transitions, respectively. $Min(N')$ denotes the minimal elements of N' w.r.t. $\leq$.

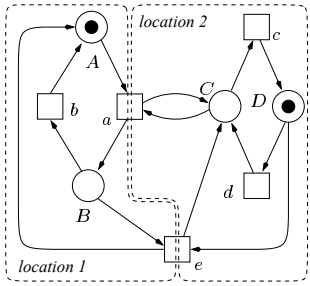


Fig. 1. Distributed net

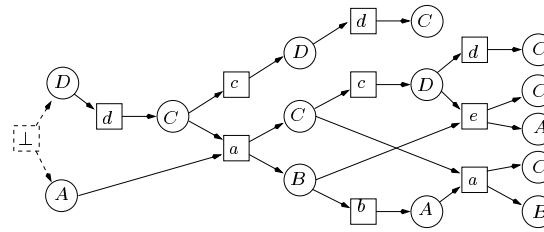


Fig. 2. Branching process

Given two nets N_1, N_2, the mapping $h : P_1 \cup T_1 \to P_2 \cup T_2$ is called a *homomorphism* if $h(P_1) \subseteq P_2, h(T_1) \subseteq T_2$, and for every $t \in T_1$ the restriction of h to ${}^\bullet t$, denoted $h|_{{}^\bullet t}$, is a bijection between ${}^\bullet t$ and ${}^\bullet h(t)$, and analogous for $h|_{t^\bullet}$.

A *branching process* [Eng91] of a net system $\Sigma = (N, M_0)$ is a pair $\beta = (N', \pi)$ where $N' = (B, E, F)$ is an occurrence net and $\pi : N' \to N$ is a homomorphism, such that the restriction of π to $Min(N')$ is a bijection between $Min(N')$ and M_0, and additionally for all $e_1, e_2 \in E$: if $\pi(e_1) = \pi(e_2)$ and ${}^\bullet e_1 = {}^\bullet e_2$ then $e_1 = e_2$. Loosely speaking, we unfold the net N to an occurrence net N', such that each node x of N' refers to node $\pi(x)$ of N. Two branching processes β_1, β_2 of Σ are *isomorphic* if there exists a bijective homomorphism $h : N_1 \to N_2$ such that the composition $\pi_2 \circ h$ equals π_1. In [Eng91] it is shown that each net system Σ has a unique maximal branching process up to isomorphism, which we call the *unfolding* of Σ, and denote by $Unf_\Sigma = (N', \pi)$.

In distributed net systems, the location $loc(x)$ of a node x of N' is given by $loc(x) = loc(\pi(x))$. By $E_i := \{e \in E \mid i \in loc(e)\}$, we denote the set of *$i$-events*.

Let $N'' = (B'', E'', F'')$ be a subnet of N', such that $e \in E''$ implies $e' \in E''$ for every $e' < e$, and $B'' = Min(N') \cup E''^\bullet$, and let π'' be the restriction of π onto the nodes of N''. We call $\beta'' = (N'', \pi'')$ a *prefix* of Unf_Σ. Fig. 2 shows a prefix of the infinite unfolding of the net system drawn in Fig. 1.

Configurations and Cuts. For the remainder of the section, let us fix the unfolding $Unf_\Sigma = (N', \pi)$ of the distributed net system Σ with $N' = (B, E, F)$.

A *configuration* $C \subseteq E$ is a causally downward-closed, conflict-free set of events, i.e., $\forall e \in C$: if $e' \leq e$ then $e' \in C$, and $\forall e, e' \in C : \neg(e \# e')$. A finite configuration describes the initial part of a computation of the system. If we understand the *states* of the system as moments in time, then configurations represent the *past* (by exhibiting all the events that have occurred so far, and the causal structure among them), as well as the *present* and the *future*, as formalised in the following.

Two nodes of N' are *concurrent* if they are neither in conflict nor causally related. A set $B' \subseteq B$ of conditions of N' is called a *cut* if B' is a maximal set of pairwise concurrent conditions. Every finite configuration C determines a cut $Cut(C) := (Min(N') \cup C^\bullet) \setminus {}^\bullet C$. The corresponding set $\pi(Cut(C)) \subseteq P$ of

places is a reachable marking of Σ, denoted by $\mathcal{M}(C)$ and called *the state of* C. Notice that for every reachable marking M of Σ, there exists a (not necessarily unique) finite configuration with state M. We will often identify configurations with their state. Given a configuration C and a disjoint set E' of events, we call $C \oplus E'$ an *extension* of C if $C \cup E'$ is a configuration.

Let $\uparrow C := \{x \in (B \cup E) \mid \exists b \in Cut(C).\ b \leq x \quad \text{and} \quad \forall y \in C.\ \neg(x \# y)\}$. The *(branching) future* of a configuration C is given by the branching process $\beta(C) := (N'_C, \pi_C)$, where N'_C is the unique subnet of N' whose set of nodes is $\uparrow C$, and π_C is the restriction of π onto the nodes of N'_C. Let us call two configurations $\mathcal{M}$*-equivalent*, denoted $C \equiv_{\mathcal{M}} C'$, if $\mathcal{M}(C) = \mathcal{M}(C')$. It is easy to show that if $C \equiv_{\mathcal{M}} C'$ then there exists an isomorphism $I_C^{C'}$ from $\beta(C)$ to $\beta(C')$. It induces a mapping from the extensions of C onto the extensions of C', mapping $C \oplus E'$ onto $C' \oplus I_C^{C'}(E')$, which are again $\mathcal{M}$-equivalent.

Local states and views. The notion of *local state* arises by considering configurations that are determined by single events. For an event e, we call the set $\downarrow e := \{e' \in E \mid e' \leq e\}$ the *local configuration of* e. It is indeed a configuration, because no event is in self-conflict. If $e \in E_i$ is an i-event, we consider $\downarrow e$ to be an *i-local state*. It determines the local past of component i, as well as the local past of every component that communicated with i so far — directly, or indirectly via other components.

In distributed net systems, we define the *i-view* $\downarrow^i C$ of a configuration C as $\downarrow^i C := \{e \in C \mid \exists e_i \in (C \cap E_i).\ e \leq e_i\}$. Notice that the sequentiality of the components implies that for each $i \in Loc$, the i-events form a tree in *Unf*, i.e., in each configuration the i-events are totally ordered. Thus, the i-view of C is the local configuration of the unique, causally maximal i-event in C. Intuitively, $\downarrow^i C$ can be understood as the most recent i-local configuration that the whole system is aware of in the (global) configuration C. The i-view of a local configuration $\downarrow e$ is written as $\downarrow^i e$. Note that $\downarrow^i e = \downarrow e$ iff $i \in loc(e)$. We will interpret the empty configuration as the local configuration of a virtual event $\perp$, which can be seen as *initial* event with empty preset and $Min(N')$ as postset. We assume the set of events of Unf_Σ to contain this virtual event, $\perp \in E$, and set $loc(\perp) := Loc$.

Let $\mathcal{C}_{loc}(Unf)$ denote the set of local configurations of *Unf* (abbreviated $\mathcal{C}_{loc}$ if *Unf* is clear), and let $\mathcal{C}_{loc}^i := \{\downarrow e \mid e \in E_i\}$ be the set of i-local configurations.

Correspondence of CSAs and unfoldings. Originally in [LRT92], the entire formalism relies on CSAs, a subclass of prime event structures. We note that net unfoldings as presented here, directly correspond to *rooted* CSAs. The differences are only technical. For details of this correspondence, cf. [HNW98a].

3 Temporal Logic for Communicating Sequential Agents

In [LRT92], Lodaya, Ramanujam, and Thiagarajan defined and axiomatised the temporal logic $\mathcal{L}_{CSA}$ that allows to express properties referring to the *latest gossip* of the agents in a distributed system. Let us give a brief idea of the logic, related to unfoldings of distributed net systems. For details, cf. [LRT92].

Basically, $\mathcal{L}_{CSA}$ consists of propositional logic. Additionally, it provides two temporal operators $\Diamond_i$, resp. $\Diamond\!\!\!\!-_i$, for each $i \in Loc$, referring to the *local* future, resp. local past, of agent i. All formulae are interpreted exclusively on the *local* configurations of a given unfolding.

Intuitively, $\Diamond\!\!\!\!-_i\, \varphi$ holds at $\downarrow e$ if some i-local configuration in the past of e satisfies φ. When e is a j-event, this can be read as "agent j has at its local state $\downarrow e$ enough gossip information to assert that φ was true in the past in agent i".

The local configuration $\downarrow e$ satisfies $\Diamond_i \varphi$ iff some i-local configuration in the i-local future of $\downarrow e$ satisfies φ, i.e., if there is some configuration $\downarrow e'$ $(e' \in E_i)$ such that $\downarrow e' \supseteq \downarrow^i e$ and $\downarrow e'$ satisfies φ. For $e \in E_j$, this can be read as "at the j-local state where e has just occurred, agent j has enough gossip information about agent i to assert that φ may hold eventually in i".

Typical specifications are properties like $\Diamond_i(x_i \to \bigwedge_{j \in Loc} \Diamond_j x_j)$: "whenever x_i holds in i, then agent i knows that x_j may hold eventually in all other agents j". For more detailed examples, cf. [LRT92].

A generalised syntax – $\mathcal{L}$. We now introduce a slightly extended language in which the temporal modalities $\Diamond, \Diamond\!\!\!\!-$ are separated from the gossip modality $@\,i:$. The separation yields a higher degree of modularity in the technical treatment and also saves redundant indices in nested formulae residing at a single location. The abstract syntax of $\mathcal{L}$ is

$$\varphi \quad ::= \quad p \mid \neg\varphi \mid \varphi \vee \varphi \mid \Diamond\varphi \mid \Diamond\!\!\!\!-\varphi \mid @\,i\colon \varphi$$

where p ranges over a set of atomic propositions, and i over Loc. We require that every occurrence of a temporal modality lies within the scope of a gossip modality. The operators $\Diamond$ and $\Diamond\!\!\!\!-$ are now seen as temporal future and past modalities within a single location, which is determined by the next enclosing gossip modality $@\,i:$. For example, $\Diamond_i \Box_j \varphi$ will be written as $@\,i : \Diamond\, @\,j : \Box\varphi'$ in our syntax. Formally, the connection to the original $\mathcal{L}_{CSA}$ syntax is given in [HNW98a].

Like in $\mathcal{L}_{CSA}$, formulae are interpreted at local configurations only. The models of $\mathcal{L}$ are unfoldings of distributed net systems. The interpretation of the atomic propositions relies on the state function $\mathcal{M}$, i.e., we identify the atomic propositions with the set P of places of the system under consideration (without loosing expressive power by this convention), and evaluate a proposition at configuration $\downarrow e$ according to $\mathcal{M}(\downarrow e)$.

Formally, we define two satisfaction relations: a global relation $\models$, defined for the local configurations of *arbitrary* locations, and for each agent $i \in Loc$ a local relation $\models_i$, exclusively defined for the *i-local* configurations. These relations are inductively defined as follows:

$$\begin{array}{llll}
\downarrow e \models p & \text{iff } p \in \mathcal{M}(\downarrow e) & \downarrow e \models \varphi \vee \psi & \text{iff } \downarrow e \models \varphi \text{ or } \downarrow e \models \psi \\
\downarrow e \models \neg\varphi & \text{iff } \downarrow e \not\models \varphi & \downarrow e \models @\,i\colon \varphi & \text{iff } \downarrow^i e \models_i \varphi \\
\\
\downarrow e \models_i p & \text{iff } p \in \mathcal{M}(\downarrow e) & \downarrow e \models_i \varphi \vee \psi & \text{iff } \downarrow e \models_i \varphi \text{ or } \downarrow e \models_i \psi \\
\downarrow e \models_i \neg\varphi & \text{iff } \downarrow e \not\models_i \varphi & \downarrow e \models_i \Diamond\!\!\!\!-\,\varphi & \text{iff } \exists e' \in E_i\,.\, e' \le e \text{ and } \downarrow e' \models_i \varphi \\
\downarrow e \models_i @\,j\colon \varphi & \text{iff } \downarrow^j e \models_j \varphi & \downarrow e \models_i \Diamond\varphi & \text{iff } \exists e' \in E_i\,.\, e' \ge e \text{ and } \downarrow e' \models_i \varphi
\end{array}$$

We say that the system Σ satisfies a formula φ if the empty configuration $\downarrow\!\perp$ of Unf_Σ satisfies φ, i.e., if $\downarrow\!\perp \models \varphi$. The *future fragment* $\mathcal{L}^+$ of $\mathcal{L}$ consists of all formulae without past-operator $\diamondsuit$.

4 Factorisation of the Unfolding

In general, the unfolding of a net system is infinite, even if the net is finite-state. Therefore, many model checking algorithms cannot directly be applied on a modal logic defined over the unfolding. One way to overcome this problem is to look for a factorisation of the unfolding by a decidable equivalence relation $\equiv$ that is finer than the distinguishing power of the formula to be evaluated, i.e., $C \equiv C'$ shall imply $C \models \varphi \Leftrightarrow C' \models \varphi$. The second requirement on $\equiv$ is that a set of representatives of its finitely many equivalence classes and a representation of the (transition) relations between the classes can be computed effectively. Then we can decide $C \models \varphi$ on *Unf* by transferring the question to the model checking problem $(C/\equiv) \models \varphi$ on $(Unf/\equiv, \longrightarrow)$.

The finite prefix. The first construction of an appropriate finite factorisation was given by McMillan [McM92]. He showed how to construct a finite prefix of the unfolding of a safe, i.e. finite-state, net system in which every reachable marking is represented by some cut. In terms of temporal logic, his approach means to consider formulae of the type $\Diamond\psi$ where $\Diamond$ is "global reachability" and ψ is a boolean combination of atomic propositions P. The key to the construction is that if the prefix contains several events with $\mathcal{M}$-equivalent local configurations, then their futures are isomorphic, i.e., they cannot be distinguished by the logic. Consequently, only one of them needs to be explored further, while the others become *cutoff* events. The *finite prefix Fin* is that initial part of the unfolding, that contains no causal successor of any cutoff, i.e., an event e' belongs to *Fin* iff no event $e < e'$ is a cutoff.

In general, the formal definition of a cutoff requires two crucial relations on configurations: An instance of the equivalence relation $\equiv$, and a partial order $\prec$. On the one hand, this partial order shall ensure that the expanded prefix contains a representative for each equivalence class. On the other hand, it shall guarantee that the prefix remains finite. The requirements for an *adequate* partial order $\prec$ (in conjunction with $\mathcal{M}$-equivalence) were examined very detailed in [ERV96]. They are as follows: it must be well-founded, it must respect set inclusion ($C \subset C'$ implies $C \prec C'$), and it must be preserved under *finite extensions*, i.e., if $C \equiv C'$ and $C \prec C'$ then $C \oplus E' \prec C' \oplus I_C^{C'}(E')$.

Such an adequate partial order is particularly useful, if it is *total*, such that for each two equivalent local configurations $\downarrow e \equiv \downarrow e'$ either e or e' can be discriminated as a cutoff. For 1-safe nets, a total order satisfying the above requirements was defined in [ERV96], yielding a minimal prefix.

In [McM92,ERV96] just $\mathcal{M}$-equivalence is considered. In conjunction with an adequate order $\prec$, the definition of *Fin* guarantees that each reachable marking is represented by the state of a configuration contained in *Fin*.

It was already observed in [HNW98b] that refining $\mathcal{M}$-equivalence yields an *extended prefix*, which – although being possibly larger than the prefix of [McM92,ERV96] – allows to apply a standard μ-calculus model checker for a location based modal logic called the *distributed μ-calculus*. We defined an equivalence $\equiv_{\mathcal{M}\text{-}loc}$ by $\downarrow e \equiv_{\mathcal{M}\text{-}loc} \downarrow e'$ iff $\downarrow e \equiv_{\mathcal{M}} \downarrow e'$ and $loc(e) = loc(e')$, and proved that $\equiv_{\mathcal{M}\text{-}loc}$-equivalence equals the distinguishing power of the distributed μ-calculus.

Generalised cutoffs. Now we look for more general conditions on equivalence relations that ensure that all equivalence classes can be computed by a prefix construction. Let us call a decidable equivalence relation $\equiv$ on configurations of *Unf* to be *adequate* if it refines $\mathcal{M}$-equivalence and has finite index. I.e., $C \equiv C'$ implies $C \equiv_{\mathcal{M}} C'$ and $\equiv$ has only finitely many equivalence classes on *Unf*. We give a generalised definition of a *cutoff* event by

$$e \in E \text{ is a } \textit{cutoff} \quad \text{iff} \quad \exists e' \in E, \text{ such that } \downarrow e' \equiv \downarrow e \text{ and } \downarrow e' \prec \downarrow e$$

where $\equiv$ is an adequate equivalence relation and $\prec$ is an adequate partial order. The *finite prefix Fin* constructed for $\equiv$ is given by the condition: e' belongs to *Fin* iff no event $e < e'$ is a cutoff. It is obvious from the cutoff definition that *Fin* constructed for $\equiv$ contains a representative for each $\equiv$-class of *Unf*.

Proposition 1. *The prefix Fin constructed for an adequate $\equiv$ is finite.*

An adequate equivalence finer than $\mathcal{L}$. In difference to S4 as used in [Esp94] and the distributed μ-calculus in [HNW98b], an equivalence finer than the distinguishing power of $\mathcal{L}$ has infinite index. However, by each finite set of $\mathcal{L}$-formulae we can only distinguish finitely many classes of configurations. Thus we can hope for a model checking procedure following the outline from the beginning of the section, if we find an equivalence which is at least as discriminating as the Fisher-Ladner-closure of a $\mathcal{L}$-formula φ, because this is the set of formulae relevant for model checking φ on *Unf*. First, we need some technical definitions.

Let us denote the *gossip-past-depth* of a formula $\varphi \in \mathcal{L}$ by $gpd(\varphi)$. It shall count how often in the evaluation of φ we have to change the local view or to go back into the local past. The inductive definition is

$$\begin{array}{ll} gpd(p) = 1 & gpd(\neg\varphi) = gpd(\varphi) \\ gpd(\varphi \vee \psi) = \max\{gpd(\varphi), gpd(\psi)\} & gpd(\Diamond\varphi) = gpd(\varphi) \\ gpd(@\, i\colon \varphi) = gpd(\varphi) + 1 & gpd(\Diamond\!\!\!\!-\, \varphi) = gpd(\varphi) + 1 \end{array}$$

Now we are ready to define the crucial equivalence relation $\equiv_i^n$, which is the basis for model checking $\mathcal{L}$. It is parameterised by a natural number n (which will be the gossip-past-depth of a given formula) and by a location i (at which the formula is interpreted). Formally, we define $\equiv_i^n \subseteq \mathcal{C}_{loc}^i \times \mathcal{C}_{loc}^i$ to be the coarsest equivalence relation satisfying:

$$\begin{array}{l} \downarrow e \equiv_i^0 \downarrow f \text{ implies } \forall p \in P_i\,.\ p \in \mathcal{M}(\downarrow e) \Leftrightarrow p \in \mathcal{M}(\downarrow f) \\ \downarrow e \equiv_i^1 \downarrow f \text{ implies } \forall j,k \in Loc\,.\ \downarrow^j e \subseteq \downarrow^k e \Leftrightarrow \downarrow^j f \subseteq \downarrow^k f \end{array}$$

and for all $n \geq 0$ moreover

$$\begin{aligned} \downarrow e \equiv_i^{n+1} \downarrow f \;\text{ implies }\; & \forall j \in Loc\,.\, \downarrow^j e \;\equiv_j^n\; \downarrow^j f \\ (*)\ \textit{and}\ \; & \forall e' \in (\downarrow e \cap E_i)\,.\, \exists f' \in (\downarrow f \cap E_i)\,.\, \downarrow e' \equiv_i^n \downarrow f' \\ \textit{and}\ \; & \forall f' \in (\downarrow f \cap E_i)\,.\, \exists e' \in (\downarrow e \cap E_i)\,.\, \downarrow e' \equiv_i^n \downarrow f' \end{aligned}$$

The first condition is an i-localised version of $\mathcal{M}$-equivalence. The second one refers to the *latest information* concerning agents other than i, and the third condition inductively lifts the equivalence with respect to the levels of the gossip-past-depth. Let us briefly collect some important facts about the equivalence.

Observation 2. *The equivalence relation $\equiv_i^n$ is decidable and of finite index for every $n \geq 0$. Furtheron, $\equiv_i^{n+1}$ is refining $\equiv_i^n$, i.e., $\equiv_i^{n+1} \subseteq \equiv_i^n$ for all n. Finally, it respects $\mathcal{M}$-equivalence, i.e., $\downarrow e \equiv_i^n \downarrow f$ implies $\mathcal{M}(\downarrow e) = \mathcal{M}(\downarrow f)$ for all $n > 0$.*

Remark 3. Note that the last two lines of the third condition after (*) can be omitted if we restrict ourselves to the (still very useful) sublanguage $\mathcal{L}^+$, yielding considerable savings: With this condition, the number of equivalence classes of $\equiv_i^n$ may grow non-elementarily with n, forbidding any consideration of practicability, whereas without this condition the number of equivalence classes grows exponentially with n.

The most important property of the equivalence used in the proof of the main result is that it is preserved by local successors, as stated in Lemma 4.

Lemma 4. *Let $e \leq e'$, and $f \leq f'$ be i-events, such that $\downarrow e \equiv_i^n \downarrow f$, and let I be the isomorphism from $\beta(\downarrow e)$ onto $\beta(\downarrow f)$. If $f' = I(e')$ then also $\downarrow f' \equiv_i^n \downarrow e'$.*

Proof. This the most involved proof, and a main result of the paper. Please note that (for reasons of readability) the proof given here only deals with the pure future fragment $\mathcal{L}^+$ of the logic $\mathcal{L}$, i.e. the third condition of the definition of the equivalence relation $\equiv_i^n$ has to be read without the last two lines after the (*). For the (even more involved) proof for the full logic $\mathcal{L}$, i.e., inclusive the condition (*) of the $\equiv_i^n$ definition, see [HNW98a].

Let us define some notions and notations: Since we will often talk about a number of view changes in sequence, we introduce "paths" through the locations of the system: Let $\sigma = l_1 l_2 \ldots l_n$ be a sequence of locations (called *location path*), i.e., $l_j \in Loc$ for all $1 \leq j \leq n$. Given any configuration C, we define $\downarrow^\sigma C := \downarrow^{l_1}(\downarrow^{l_2}(\ldots(\downarrow^{l_n} C)\ldots))$. We set $\downarrow^\varepsilon e := \downarrow e$, where ε is the (empty) sequence of length 0. Note that a location path may include repetitions, i.e., $l_i = l_j$ for $i \neq j$ is allowed. Given an event g and some location path σ, we denote by g_σ the event that determines the σ-view of $\downarrow g$, i.e., $\downarrow^\sigma g = \downarrow g_\sigma$.

Now let $e \leq e'$ and $f \leq f'$ be events of E_i, and $n \geq 1$, as in the assumptions of the Lemma. First of all, we note that the required isomorphism I exists because $\equiv_i^n$-equivalence implies $\mathcal{M}$-equivalence.

We have to show $\downarrow f' \equiv_i^n \downarrow e'$. A key observation is the following: for every location path σ, it holds that if $e'_\sigma \not\leq e$ then $I(e'_\sigma) = f'_\sigma \not\leq f$.

This is the basis for the induction on $m \leq n$: for each sequence σ of length $n-m$ with $e'_\sigma \not\leq e$ (and also $f'_\sigma \not\leq f$), it holds that $\downarrow e'_\sigma \equiv^m_j \downarrow f'_\sigma$, where j is either the first location occurring in the sequence σ (if $n > m$), or $j := i$ (if $n = m$ and the empty sequence ε is the only sequence of length $n-m$). In the latter case, $\downarrow^i e' = \downarrow e'$ (because $e' \in E_i$), and $\downarrow^i f' = \downarrow f'$, we thus obtain $\downarrow e' \equiv^n_i \downarrow f'$ as required. The induction relies on a case analysis according to the following cases: $m = 0$, $n = m = 1$, $n = m > 1$, $n > m = 1$, and finally $n > m > 1$.

• For $m = 0$ we have to show that $\downarrow e'_\sigma \equiv^0_j \downarrow f'_\sigma$. This is clear, because $I(e'_\sigma) = f'_\sigma \in E_j$ and thus the j-local part of the markings of $\downarrow e'_\sigma$ and $\downarrow f'_\sigma$ coincide, because $\pi(e'_\sigma)^\bullet = \pi(f'_\sigma)^\bullet$.

• For $n = m = 1$ we have to show that $\downarrow e \equiv^1_i \downarrow f$ implies $\downarrow e' \equiv^1_i \downarrow f'$, i.e., (1) for all $j \in Loc$: $\downarrow^j e' \equiv^0_j \downarrow^j f'$, and (2) for all $j, k \in Loc$: $e'_j \leq e'_k$ iff $f'_j \leq f'_k$.
If $e'_j \leq e$ then $\downarrow e' \setminus \downarrow e$ contains no j-event, which means that $e'_j = e_j$ and similarly $f'_j = f_j$, so (1) follows easily. If $e'_j \not\leq e$ then also $f'_j \not\leq f$, in which case $\downarrow^j e' \equiv^0_j \downarrow^j f'$ follows by induction.
So consider (2). Let $j, k \in Loc$. We show that $e'_j \leq e'_k$ iff $f'_j \leq f'_k$, using a similar case analysis. If $e'_j, e'_k \not\leq e$, then the isomorphism preserves the order. If $e'_j, e'_k \leq e$, then $e'_j = e_j$ and $e'_k = e_k$, (and similarly $f'_j = f_j$, $f'_k = f_k$), and so the order is inherited from the corresponding local views of $\downarrow e$ and $\downarrow f$, which by assumption match. The third case is $e'_j \leq e$, but $e'_k \not\leq e$, and thus similarly $f'_j \leq f$, but $f'_k \not\leq f$. Since this is the most sophisticated argument and used also in the other cases, the situation is illustrated in Figure 3. $e'_j \leq e$ implies $e'_j = e_j$. Now we choose an $l \in Loc$, such that $e_j \leq e_l \leq e'_k$, and moreover e_l is (causally)

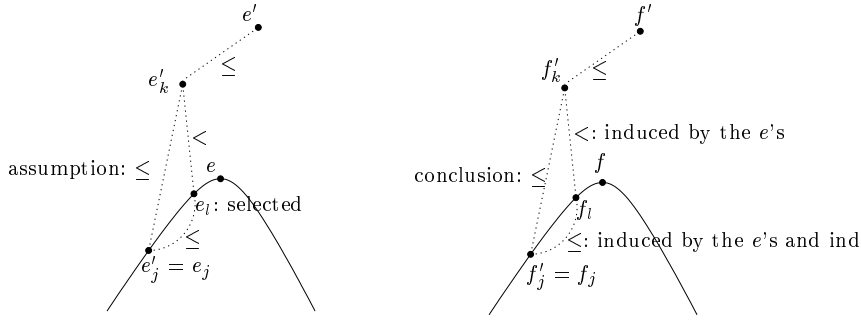


Fig. 3. Situation: $e'_k \not\leq e$ and $e'_j \leq e$

maximal with this respect. For at least one of the possible choices of l, there exists an event $e'' \in E_l$, such that $e'' \in (\downarrow^k e' \setminus \downarrow e)$. By the isomorphism, we have that $I(e'') = f'' \in (\downarrow^k f' \setminus \downarrow f)$. By assumption on the equivalence of e and f we can conclude $f'_j = f_j \leq f_l \leq f'_l \leq f'_k$, i.e., $\downarrow^j f' \subseteq \downarrow^k f'$ as desired.

• For $n = m > 1$ the reasoning is similar to the case $n = m = 1$, except that the argument for the gossip aspect of the equivalence is not needed.

• For $n > m = 1$, let $\sigma = (i'\sigma')$ be a sequence of length $n-1$ with $e'_\sigma \not\leq e$.

Again, we have to show $\downarrow^\sigma e' \equiv^1_{i'} \downarrow^\sigma f'$. Let $j \in Loc$. For the case of $e'_{j\sigma} \not\leq e$ the $\equiv^0_j$-equivalence is a consequence of $I(e'_{j\sigma}) = f'_{j\sigma}$. For $e'_{j\sigma} \leq e$ there exists again an $l \in Loc$ with $e'_{j\sigma} \leq e_l \leq e'_\sigma$, so that e_l is maximal in this respect, and as above we also obtain $f'_{j\sigma} \leq f_l \leq f'_\sigma$. Moreover, in this case it holds that $e_{jl} = e'_{j\sigma}$ and similarly $f_{jl} = f'_{j\sigma}$. By assumption, we have $\downarrow(e_{jl}) \equiv^{n-2}_j \downarrow(f_{jl})$, and because of $n \geq 2$, in particular $\downarrow(e_{jl}) \equiv^0_j \downarrow(f_{jl})$, as desired.
The argument concerning the relative orders of j-views and k-views of e_σ and e'_σ is the same as for the case of $n = m = 1$.
• For $n > m > 1$ let σ be of length $n - m$, such that σ has j as first element, and such that $e'_\sigma \not\leq e$, and similarly $f'_\sigma \not\leq f$. We have to show that for each $k \in Loc$ it holds that $\downarrow^k e'_\sigma \equiv^{m-1}_k \downarrow^k f'_\sigma$. For $e'_{k\sigma} \not\leq e$ and similarly $f'_{k\sigma} \not\leq f$ this follows from the induction hypothesis. For $e'_{k\sigma} \leq e$ there exists (again) a location l, such that $e'_{k\sigma} \leq e_l \leq e'_\sigma$ and e_l is causally maximal in this respect. Then $\downarrow^k e'_\sigma = \downarrow^k e_l \equiv^{n-2}_k \downarrow^k f_l = \downarrow^k f'_\sigma$, where $n - 2 \geq m - 1$, so that the desired claim follows from the observation $\equiv^{\bar{n}+\bar{m}}_k \subseteq \equiv^{\bar{n}}_k$. □

Theorem 5. *Let φ be an $\mathcal{L}$-formula of gossip-past-depth n, and let $e, f \in E_i$ with $\downarrow e \equiv^n_i \downarrow f$. Then $\downarrow e \models_i \varphi$ iff $\downarrow f \models_i \varphi$.*

Proof. By structural induction on φ: For atomic propositions, note that $\downarrow e \equiv^1_i \downarrow f$ implies $\downarrow e \equiv_{\mathcal{M}} \downarrow f$ (cf. Observation 2), and hence $\downarrow e \models_i p$ iff $\downarrow f \models_i p$. The induction for boolean connectives is obvious.

For $gpd(\Diamond\varphi) = gpd(\varphi) = n$ let $\downarrow e \models_i \Diamond\varphi$ and $\downarrow e \equiv^n_i \downarrow f$. We have to show that also $\downarrow f \models_i \Diamond\varphi$ (all other cases follow by symmetry). By definition, there exists $e' \geq e$ with $e' \in E_i$ and $\downarrow e' \models_i \varphi$. By Lemma 4 the event $f' = I(e') \in E_i$ obtained from the isomorphism I due to the $\mathcal{M}$-equivalence of $\downarrow e$ and $\downarrow f$ satisfies $f \leq f'$ and $\downarrow e' \equiv^n_i \downarrow f'$. By induction, $\downarrow f' \models_i \varphi$ and finally $\downarrow f \models_i \Diamond\varphi$.

Now let $\varphi = @j : \psi$ with $gpd(\varphi) = gpd(\psi) + 1 = n$. If $\downarrow e \models_i \varphi$ then $\downarrow^j e \models_j \psi$, and by definition $\downarrow^j e \equiv^{n-1}_j \downarrow^j f$. Thus, by induction, $\downarrow^j f \models_j \psi$, and finally $\downarrow f \models_i \varphi$.

Finally, let $\varphi = \Diamond\!\!\!-\, \psi$, with $gpd(\psi) = n-1$, and $\downarrow e \models_i \Diamond\!\!\!-\, \psi$, i.e., there exists an event $e' \in E_i$, s.t. $e' \leq e$ and $\downarrow e' \models_i \psi$. Due to the third condition (*), there exists an $f' \in E_i$, s.t. $f' \leq f$ and $\downarrow f' \equiv^{n-1}_i \downarrow e'$. Hence, by induction, also $\downarrow f' \models_i \psi$, and thus $\downarrow f \models_i \varphi$. □

Based on the local equivalences, we define an adequate equivalence relation for the construction of a finite prefix by $\downarrow e \equiv^n \downarrow f$ iff $loc(e) = loc(f)$ and $\downarrow e \equiv^n_i \downarrow f$ for all $i \in loc(e)$. The next and last step to transfer the $\mathcal{L}$ model checking problem from the unfolding to an equivalent model checking problem over a finite structure is the definition of the transitions between the $\equiv^n$-equivalence classes of *Unf*. This is done in the next section.

5 Model checking

In this section we propose a verification technique for $\mathcal{L}$. Following the lines of [HNW98b], we will sketch a reduction of a given instance of the problem to a suitable input for well investigated model checkers like e.g. [CES86].

Let us consider a distributed net system Σ and an $\mathcal{L}$-formula φ of gossip-past-depth n. We have shown so far how to construct a finite prefix *Fin* of the unfolding Unf_Σ that contains representatives for all $\equiv_i^n$ equivalence classes. Now we want to compute a finite, multi modal Kripke structure on the representatives that is equivalent to Unf_Σ with respect to the evaluation of φ. What is missing are the transitions between the representatives.

Computing a finite Kripke structure. Let $n \in \mathbb{N}$, and $Unf_\Sigma = (N', \pi)$ with $N' = (B, E, F)$ be fixed, and let $\equiv^n$ be the equivalence relation used for the construction of *Fin*. The state space $\mathcal{S}_n$ of the desired Kripke structure consists of one representative of each $\equiv^n$ equivalence class. Note that by using the adequate *total* partial order $\prec$ of [ERV96], these representatives are unique, and so the state space is given by $\mathcal{S}_n := \{\downarrow e \mid e \in Fin \text{ and } e \text{ is not a cutoff}\}$. If the used order $\prec$ is not total, we fix one non-cutoff (resp. its local configuration) of the prefix as the representative of each $\equiv^n$ equivalence class. For every local configuration $\downarrow e$ of Unf_Σ, let $rep(\downarrow e) \in \mathcal{S}_n$ denote the unique representative.

Now let us consider the transitions of the Kripke structure. We introduce a transition relation for each of the modalities of the logic. Let $\downarrow e, \downarrow f \in \mathcal{S}_n$.

$$\begin{aligned}
\downarrow e \xrightarrow{\Diamond i}_n \downarrow f \quad &\text{iff} \quad e, f \in E_i \ \text{ and } \ \exists f' \in E_i \,.\, f' \geq e \ \wedge \ rep(\downarrow f') = \downarrow f \\
\downarrow e \xrightarrow{@j} \downarrow f \quad &\text{iff} \quad e \in E_i,\ f \in E_j \ \wedge \ \downarrow^j e = \downarrow f \\
\downarrow e \xrightarrow{\Diamond i} \downarrow f \quad &\text{iff} \quad e, f \in E_i \ \wedge \ f \leq e
\end{aligned}$$

Note that the definitions of $\xrightarrow{@j}$ and $\xrightarrow{\Diamond i}$ rely on the fact that the set of configurations in *Fin* (and thus also in $\mathcal{S}_n$) is downward closed, i.e., the j-view of any element of $\mathcal{S}_n$ is again in $\mathcal{S}_n$ for every j, and of course past configurations as well. On the whole, we obtain the multi modal Kripke structure $\mathcal{T}_n = (\mathcal{S}_n, \{ \xrightarrow{\Diamond i}_n , \xrightarrow{@i} , \xrightarrow{\Diamond i} \mid i \in Loc\}, \downarrow\bot)$ with root $\downarrow\bot$.

As a corollary to Theorem 5 we obtain the following characterisation of the semantics of $\mathcal{L}$ formulae over $\mathcal{T}_n$:

Corollary 6. *Let $\varphi \in \mathcal{L}$ be a formula of gossip-past-depth $m \leq n$, and let $\downarrow e \in \mathcal{S}_n$ be an i-local configuration, i.e., $e \in E_i$.*

1. *If $\varphi = \Diamond\psi$ then $\downarrow e \models_i \varphi$ iff $\exists\ \downarrow f \in \mathcal{S}_n$ with $\downarrow e \xrightarrow{\Diamond i}_n \downarrow f$ and $\downarrow f \models_i \psi$.*
2. *If $\varphi = @j{:}\psi$ then $\downarrow e \models_i \varphi$ iff $\exists\ \downarrow f \in \mathcal{S}_n$ with $\downarrow e \xrightarrow{@j} \downarrow f$ and $\downarrow f \models_j \psi$.*
3. *If $\varphi = \Diamond\!\!\!\!\cdot\ \psi$ then $\downarrow e \models_i \varphi$ iff $\exists\ \downarrow f \in \mathcal{S}_n$ with $\downarrow e \xrightarrow{\Diamond i} \downarrow f$ and $\downarrow f \models_i \psi$.*

Proof. (1) follows from the semantics of $\Diamond$ and the fact that by construction of $\mathcal{T}_n$ for any pair of states $\downarrow f'$ and $\downarrow f = rep(\downarrow f')$, we have that $\downarrow f \models_i \varphi$ iff $\downarrow f' \models_i \varphi$ for any formula φ with $gpd(\varphi) = m \leq n$. (2) and (3) are trivial. □

Thus, if we are able to actually compute (the transitions of) $\mathcal{T}_n$ then we can immediately reduce the model checking problem of $\mathcal{L}$ to a standard model checking problem over finite transition systems, applying e.g. [CES86].

Computing the transitions $\downarrow e \xrightarrow{@j} \downarrow f$ in $\mathcal{T}_n$ is trivial: $\downarrow f = \downarrow^j e$. Similarly computing the $\xrightarrow{\Diamond i}$ *successors* of $\downarrow e$ is very easy. It is more difficult to compute the transitions $\downarrow e \xrightarrow{\Diamond i}_n \downarrow f$, if only *Fin* is given. To achieve this, we use a modified version of the algorithm proposed in [HNW98b].

An algorithm to compute the $\xrightarrow{\diamond i}_n$ transitions. We assume in the following that the algorithm for constructing the prefix *Fin* uses a total, adequate order $\prec$. The construction of *Fin* provides some useful structural information: each cutoff e has a *corresponding* event e^0, such that $\downarrow e^0 \equiv^n \downarrow e$, and $\downarrow e^0 \prec \downarrow e$. Clearly, we choose $rep(\downarrow e) := \downarrow e^0$ for each cutoff e, and for non-cutoffs f, we set $rep(\downarrow f) := \downarrow f$. For technical reasons, we extend the definition of $\xrightarrow{\diamond i}_n$: we define $C \xrightarrow{\diamond i}_n \downarrow e$ for any local *or* global configuration $C \subseteq \downarrow e'$, with $rep(\downarrow e') = \downarrow e$ and $e, e' \in E_i$. The construction of *Fin* also provides a function *shift**, which maps any configuration $C = C_1$ of Unf_Σ containing some cutoff, onto a configuration $shift^*(C) = C_m$ not containing a cutoff, hence being present in *Fin*. This function works by repeatedly applying $C_{k+1} := \downarrow e_k^0 \oplus I_{\downarrow e_k}^{\downarrow e_k^0}(C_k \setminus \downarrow e_k)$ with $e_k \in C_k$ being a cutoff of Fin, and e_k^0 being its corresponding, equivalent event. This iterative application terminates, because the sequence $C_1, C_2, ..$ decreases in the underlying (well-founded) order $\prec$. Obviously, this function implies the existence of an isomorphism I between $\beta(C)$ and $\beta(shift^*(C))$, which is the composition of the isomorphisms $I_{\downarrow e_k}^{\downarrow e_k^0}$ induced by the chosen cutoff events. Moreover, $shift^*(\downarrow e) \prec \downarrow e$ for any $e \in \beta(C)$, and hence for any e for which $C \xrightarrow{\diamond i}_n \downarrow e$.

The most important part of the algorithm (cf. Fig. 4) is the recursive procedure *successors* which, when called *from the top level* with a pair $(\downarrow e, i)$, returns the $\xrightarrow{\diamond i}_n$-successors of $\downarrow e$ in the finite structure. More generally, *successors* performs a depth first search through pairs (C, i), where C is an arbitrary, not necessarily local configuration not containing a cutoff and i is a location. It determines the subset of local configurations in $\mathcal{S}_n$ that represent the $\xrightarrow{\diamond i}_n$-successors of C. Formally, $\downarrow e \in successors(C, i)$ iff there exists $\downarrow e'$ in *Unf*, which is $\equiv^n$-equivalent to $\downarrow e$, and $C \xrightarrow{\diamond i}_n \downarrow e'$.

Proposition 7. *Compute_Multi_Modal_Kripke_Structure computes the* $\xrightarrow{\diamond i}_n$ *-,* $\xrightarrow{\Diamond i}$ *-, and* $\xrightarrow{@j}$ *-transitions.*

The proof can be found in [HNW98a]. Note that at top level, *successors* is always called with a *local* configuration $\downarrow e$ as parameter, but the extension of $\downarrow e$ with cutoffs requires that we can also handle global configurations. In this paper, we focus on decidability but not on efficiency. For heuristics on efficiency improvements we refer the reader to [HNW98b].

6 Conclusion

We have shown the decidability of the model checking problem for $\mathcal{L}$, a location based branching-time temporal logic including temporal and gossip modalities. The method is based on a translation of the modalities over net unfoldings (or prime event structures) into transitions of a sequential transition system, for which established model checkers for sequential logics can be applied.

While the method as presented is non elementary for the full logic $\mathcal{L}$, the restriction to the future fragment $\mathcal{L}^+$ has "only" exponential complexity but still allows to express interesting properties.

type Vertex = {C: Configuration; i: Location; pathmark: **bool**; *(* for dfs *)* }

$prefix\_successors(C,i) = \{rep(\downarrow e) \mid \downarrow e \in \mathcal{S}_n \ \wedge \ C \xrightarrow{\Diamond i}_n \downarrow e\}$
$compatible\_cutoffs(C) = \{e \mid e \text{ is cutoff and } \downarrow e \cup C \text{ is a configuration in } Fin\}$

```
proc successors(C, i): ConfigurationSet;
{   var result: ConfigurationSet;          (* result accumulator for current vertex *)
    Vertex v := findvertex(C,i);           (* lookup in hash table, if not found then *)
                                           (* create new vertex with pathmark=false *)
    if   v.pathmark then return ∅; fi      (* we have closed a cycle *)
    result := prefix_successors(C, i);     (* directly accessible successors *)
    v.pathmark:=true;                      (* put vertex on path *)
    for e_c ∈ compatible_cutoffs(C) do     (* find successors outside Fin behind e_c *)
        result := result ∪ successors(shift*(C ∪ ↓e_c), i);
    od ;
    v.pathmark:=false;                     (* take vertex from path *)
    return result;
}

proc Compute_Multi_Modal_Kripke_Structure;
{   InitializeTransitionSystem(T_n, Fin);  (* extract state space from Fin *)
    for ↓e ∈ S_n, i ∈ Loc do
        add transition ↓e --◎i--> ↓^i e;
    for i ∈ Loc, ↓e, ↓f ∈ S_n ∩ C^i_loc, ↓f ⊆ ↓e do
        add transition ↓e --◈i--> ↓f;
        for ↓e' ∈ successors(↓e,i) do
            add transition ↓e --◇i-->_n ↓e';
        od
    od
}
```

Fig. 4. The conceptual algorithm to compute the transitions of $\mathcal{T}_n$.

We also hope that the presented results can be used as a methodological approach to model checking temporal logics of *causal knowledge* [Pen98].

The main difficulty, the solution of which is also the major contribution of the paper, was to find an adequate equivalence relation on local states that allows to construct a finite transition system containing a representative for each class of equivalent local states. If the method really is to be applied, then refinements of the equivalence bring it closer to the logical equivalence and thus leading to a smaller index will be crucial. We believe that the potential for such improvements is high at the price of much less understandable definitions.

For the treatment of past an alternative and potentially more efficient approach in the line of [LS95] – elimination of past modalities in CTL – might come to mind, but the techniques used there can at least not directly be transferred to $\mathcal{L}_{CSA}$ because of the intricate interaction between past and gossip modalities.

References

[CEP95] A. Cheng, J. Esparza, and J. Palsberg, *Complexity results for 1-safe nets*, Theoretical Computer Science (1995), no. 147, 117–136.

[CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla, *Automatic verification of finite-state concurrent systems using temporal logic specifications*, ACM Transactions on Programming Languages and Systems **8** (1986), no. 2, 244–263.

[Eng91] J. Engelfriet, *Branching processes of Petri nets*, Acta Informatica **28** (1991), 575–591.

[ERV96] J. Esparza, S. Römer, and W. Vogler, *An Improvement of McMillan's Unfolding Algorithm*, Proc. of TACAS '96, LNCS, vol. 1055, Springer, 1996, 87–106.

[Esp94] J. Esparza, *Model checking using net unfoldings*, Science of Computer Programming **23** (1994), 151–195.

[HNW98a] M. Huhn, P. Niebert, and F. Wallner, *Model checking gossip modalities*, Technical Report 21/98, Univ. Karlsruhe, Fakultät für Informatik, Aug. 1998.

[HNW98b] M. Huhn, P. Niebert, and F. Wallner, *Verification based on local states*, Proc. of TACAS '98, LNCS, vol. 1384, Springer, 1998, 36–51.

[LRT92] K. Lodaya, R. Ramanujam, and P.S. Thiagarajan, *Temporal logics for communicating sequential agents: I*, Int. Journal of Foundations of Computer Science **3** (1992), no. 2, 117–159.

[LS95] F. Laroussinie and P. Schnoebelen, *A hierarchy of temporal logics with past*, Theoretical Computer Science **148** (1995), 303–324.

[LT87] K. Lodaya and P.S. Thiagarajan, *A modal logic for a subclass of event structures*, Automata, Languages and Programming, LNCS, vol. 267, Springer, 1987, 290–303.

[McM92] K.L. McMillan, *Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits*, Proc. of CAV '92, LNCS, vol. 663, Springer, 1992, 164–174.

[Nie98] Peter Niebert, *A temporal logic for the specification and verification of distributed behaviour*, PhD thesis, Institut für Informatik, Universität Hildesheim, March 1998.

[NPW80] M. Nielsen, G. Plotkin, and G. Winskel, *Petri nets, event structures and domains*, Theoretical Computer Science **13** (1980), no. 1, 85–108.

[NRT90] M. Nielsen, G. Rozenberg, and P.S. Thiagarajan, *Behavioural notions for elementary net systems*, Distributed Computing **4** (1990), no. 1, 45–57.

[Pen97] W. Penczek, *Model-Checking for a Subclass of Event Structures*, Proc. of TACAS '97, LNCS, vol. 1217, 1997.

[Pen98] W. Penczek, *Temporal logic of causal knowledge*, Proc. of WoLLiC '98, 1998.

[Ram95] R. Ramanujam, *A Local Presentation of Synchronizing Systems*, Structures in Concurrency Theory, Workshops in Computing, 1995, 264–279.

[Thi94] P.S. Thiagarajan, *A Trace Based Extension of PTL*, Proc. of 9th LICS, 1994.

[Thi95] P.S. Thiagarajan, *A Trace Consistent Subset of PTL*, Proc. of CONCUR '95, LNCS, vol. 962, Springer, 1995, 438–452.

[Wal98] F. Wallner, *Model checking LTL using net unfoldings*, Proc. of CAV '98, LNCS, vol. 1427, Springer, 1998.

[Win87] Glynn Winskel, *Event structures*, Advances in Petri Nets, LNCS, vol. 255, Springer, 1987.

A Complete Coinductive Logical System for Bisimulation Equivalence on Circular Objects*

Marina Lenisa

Laboratory for the Foundations of Computer Science
University of Edinburgh, Scotland.
lenisa@dcs.ed.ac.uk

Abstract. We introduce a *coinductive* logical system *à la* Gentzen for establishing *bisimulation* equivalences on *circular non-wellfounded regular* objects, inspired by work of Coquand, and of Brandt and Henglein. In order to describe circular objects, we utilize a typed language, whose coinductive types involve disjoint sum, cartesian product, and finite powerset constructors. Our system is shown to be complete with respect to a maximal fixed point semantics. It is shown to be complete also with respect to an equivalent final semantics. In this latter semantics, terms are viewed as points of a coalgebra for a suitable endofunctor on the category Set^* of *non-wellfounded* sets. Our system subsumes an axiomatization of regular processes, alternative to the classical one given by Milner.

Introduction

In recent years, considerable energy has been devoted towards the development of simple principles and techniques for understanding, defining and reasoning on infinite and circular objects, such as streams, exact reals, processes, and other *lazy data types* ([Mil83, MPC86, Tal90, Coq94, Gim95, BM96, Fio96]). Structural induction trivially fails on infinite and non-wellfounded objects. It can be applied only in rather contrived ways, and always indirectly, often utilizing inefficient implementations of these objects, e.g. streams as inductively defined functions on natural numbers. Elaborate mathematical theories, such as domain theory ([Plo85]) and metric semantics ([BV96]), can be used, of course, to support rigorous treatment of such objects. But an ideal framework should allow to deal with infinite computational objects in a *natural, operationally based, implementation-independent* way, without requiring any heavy mathematical overhead.

Systems based on *coinductive definitions* and *coinduction proof principles* appear to be a good starting point for developing such a framework. See e.g. [Coq94, HL95, BM96, Fio96, Len96, Pit96, Rut96, Tur96, Len98] for various approaches to infinite objects based on coinduction. Coinductive techniques are natural, in that infinite and circular objects and concepts often arise in

* Work supported by Esprit Working Group "Types", MURST'97 Cofin. "Sistemi Formali..." grant, TMR Linear FMRX-CT98-0170.

W. Thomas (Ed.): FOSSACS'99, LNCS 1578, pp. 243-257, 1999.

connection with a *maximal fixed point* construction of some kind. Moreover, they can be justified often simply by elementary set-theoretical means, see e.g. [Acz88, Len98]. In many situations, simple categorical concepts, such as those of Final Semantics ([Acz88, RT93, Len96, Rut96, Len98]), are enough to achieve a substantial generality. In this context infinite objects are described as terms of F-coalgebras for suitable functors F's.

In this paper, inspired by the seminal work of Coquand ([Coq94, Gim94]), we make a first step towards the formulation of a *simple coinductive logical* system for reasoning on infinite circular objects, generalizing [BH97]. In particular, we present a system *à la* Gentzen $\mathcal{S}_{co}$ for capturing bisimulation equivalences on non-wellfounded *regular* (rational) objects, i.e. objects which have only a *finite* number of non-isomorphic subobjects. In order to describe the objects, we make use of an elementary typed language. Types are defined using the constructors $+$ (disjoint sum), $\times$ (cartesian product), $\mathcal{P}_f$ (finite powerset), and the higher order binding constructor ν (maximal fixed point). Objects are defined only by constructors and recursive definitions. Differently from Coquand, we do not consider functional types or term destructors. Many infinite recursive objetcs usually dealt with in lazy programming can be easily seen to have a formal counterpart in our typed language.

The crucial ingredient in the formulation of our logical system are rules whose conclusion can be used as auxiliary hypothesis in establishing the premises. In a sense, our system can be viewed as a system for infinitely regressive proofs. As remarked earlier, it is inspired by the technique for dealing with coinductive types in Intuitionistic Type Theories, introduced in [Coq94], where *infinitely proofs* are handled by means of the *guarded induction principle*. This technique, originally developed for predicative systems, was later extended by Giménez to impredicative systems, [Gim94, Gim95]. Our system can be seen as a partial attempt to an elementary reconstruction of that approach, in such a way that it can be reconciled with other, more classical, syntactical approaches to circular objects ([Mil84, Acz88, BH97]). Our work seems to be related in particular with [Gim94], where Coquand's principle of guarded induction is shown to be complete with respect to the traditional principle of coinduction, in a type theoretic setting.

This paper generalizes [BH97], where a coinductive axiomatization of the type (in)equality for a simple first order language of regular recursive types is provided. The types considered in [BH97] are terms for denoting regular binary trees.

In order to give external independent justifications to our system, we consider two different, but equivalent, semantics. The first is a fixed point semantics, the latter is based on the Final Semantics paradigm ([Acz88, RT93, Tur96, Len98]).

The fixed point semantics is defined by introducing, for each type σ, a corresponding bisimulation equivalence $\approx_\sigma$ on the set T^0_σ of closed terms typable with σ. This family of equivalences is defined as the greatest fixed point of a monotone operator Φ, and it can be viewed as the "intended semantics". One of the main technical results in this paper is the fact that the system $\mathcal{S}_{co}$ axiomatizes

completely the bisimulation equivalences $\approx_\sigma$, for all type σ. The correctness of $\mathcal{S}_{co}$ is proved by coinduction, i.e. by showing that the family of relations axiomatized by $\mathcal{S}_{co}$ on closed terms typable with σ is a Φ-bisimulation. The completeness proof exploits the fact that the terms that we consider are regular.

In order to give the categorical semantics, we define a "universal" functor F, involving constructors corresponding to each of the type constructors. Then we show how to endow the family of closed typable terms $\{T^0_\sigma\}_{\sigma \in Type}$ with a structure of F-coalgebra, in such a way that the greatest F-bisimulation on the coalgebra of terms coincides with the family of bisimulation equivalences $\{\approx_\sigma\}_{\sigma \in Type}$. This yields a final semantics for our typed language. Another technical result of this paper is the fact that the categorical semantics coincides with the fixed point semantics. For simplicitly, we work in the category Set^* of *non-wellfounded sets* and set-theoretic functions. In this context final coalgebras of many functors are maximal fixpoints. Non-wellfounded sets are elements of a Universe *à la* Zermelo-Fraenkel in which the Foundation Axiom is replaced by the Antifoundation Axiom X_1 of Forti and Honsell [FH83] (or by the Antifoundation Axiom AFA of [Acz88]).

Our system, when restricted to the type of CCS-like processes, can be viewed as a logical system for establishing strong equivalence of processes, alternative to the classical axiomatic system of Milner, [Mil84].

The paper is organized as follows. In Section 1, we introduce the syntax for types and terms, and the system for establishing correct typing judgements. We introduce also the fixed point semantics as a family of bisimulation equivalences $\{\approx_\sigma\}_\sigma$. In Section 2, we introduce the coinductive formal system $\mathcal{S}_{co}$ *à la* Gentzen, and we show that, for all closed type σ, this system axiomatizes the bisimulation equivalence $\approx_\sigma$ on T^0_σ. In Section 3, we define a "universal" functor F on the category Set^*, and we endow the set of closed typable terms with a coalgebra structure for the functor F. Moreover, we show that the system $\mathcal{S}_{co}$ axiomatizes the largest F-bisimulation on the coalgebra of closed typable terms. Final remarks and directions for future work appear in Section 4.

The author is grateful to Peter Aczel, Furio Honsell, and the anonymous referees for useful comments.

1 Types and Terms

In this section we introduce a finite language for infinite objects.

Definition 1.1 (Types). Let $TVar$ be a set of type variables. The set of types $Type$ is defined by

$$\sigma \ ::= \ X \ \mid \ K_1 \ \mid \ldots \mid \ K_n \ \mid \ \sigma + \sigma \ \mid \ \sigma \times \sigma \ \mid \ \mathcal{P}_f(\sigma) \ \mid \ \nu X.\sigma \ ,$$

where $X \in TVar$, the symbols $K_1, \ldots, K_n$ denote constant types, $+$, $\times$, $\mathcal{P}_f()$ are disjoint sum, cartesian product, and finite powerset type constructors. The coinductive type $\nu X.\sigma$ is considered always to be guarded, i.e. all the free occurrences of the variable X in σ are within the scope of a type constructor.

In the type $\nu X.\sigma$, the occurrences of the variable X in σ are *bound*. An occurrence of the variable X in σ is *free* if it is not bound.

Remark 1.2. For simplicitly, in the definition of types we have considered only binary product and binary disjoint sum, but we could have considered, more in general, n-ary products and n-ary disjoint sums, for $n \geq 0$.

Definition 1.3 (Terms). Let Var be a set of variables. The set of terms $Term$ is defined by

$$t \ ::= \ x \ \mid \ c_j^i \ \mid \ i_1(t) \ \mid \ i_2(t) \ \mid < t, t > \mid \ [t, \ldots, t] \ \mid \ rec\, x.t \ \mid \ in(t) \ ,$$

where $x \in Var$, $\{C_j \equiv \{c_j^i \mid i \in I_j\}\}_{j \leq n}$ are sets of constants, $[\ldots]$ denotes the multiset term constructor, $i_1(\ )$, $i_2(\ )$ are the left and right injections in the disjoint sum, $<\ ,\ >$ is the pairing constructor, $in(\ )$ is the unfolding constructor, and the term $rec\, x.t$ is required to be guarded, i.e. all the free occurrences of the variable x in t are within the scope of one of the following term constructors: $i_1(\ ), i_2(\ ), <\ ,\ >, [\ldots]$.
Let $Term^0$ denote the set of closed terms.

We take terms to be equal up to permutation of elements in multisets. The constructor $in(\ )$ is introduced in order to obtain a typing system in which the shape of the type determines the form of the terms typable with that type (see Definition 1.4 and Lemma 1.5 below).

In the syntax defined above, the non-deterministic process constructor $+$ of CCS-like concurrent languages ([Mil83]) is subsumed by the $[\ldots]$ constructor.

The terms which we are interested in are those typable as follows:

Definition 1.4. Let $\mathcal{S}_{type}$ be the following formal typing system for deriving judgements of the shape $\Delta \vdash t : \sigma$, where the environment Δ is a partial function from Var to $Type$.

$$\frac{}{\Delta, x : \sigma \vdash_{type} x : \sigma} \quad (var)$$

$$\frac{}{\Delta \vdash_{type} c_j^i : K_j} \quad (const)$$

$$\frac{\Delta \vdash_{type} t : \sigma_1}{\Delta \vdash_{type} i_1(t) : \sigma_1 + \sigma_2} \quad (+_1)$$

$$\frac{\Delta \vdash_{type} t : \sigma_2}{\Delta \vdash_{type} i_2(t) : \sigma_1 + \sigma_2} \quad (+_2)$$

$$\frac{\Delta \vdash_{type} t_1 : \sigma_1 \quad \Delta \vdash_{type} t_2 : \sigma_2}{\Delta \vdash_{type} < t_1, t_2 > : \sigma_1 \times \sigma_2} \quad (\times)$$

$$\frac{\{\Delta \vdash_{type} t_i : \sigma\}_{i=1,\ldots,n}}{\Delta \vdash_{type} [t_1, \ldots, t_n] : \mathcal{P}_f(\sigma)} \quad ([\,])$$

$$\frac{\Delta, x : \nu X.\sigma \vdash_{type} t : \nu X.\sigma \quad rec\, x.t \text{ guarded}}{\Delta \vdash_{type} rec\, x.t : \nu X.\sigma} \quad (\nu)$$

$$\frac{\Delta \vdash_{type} t : \sigma[\nu X.\sigma/X]}{\Delta \vdash_{type} in(t) : \nu X.\sigma} \quad (fold)$$

Lemma 1.5. *Let $t \in Term \setminus Var$ be such that $\Delta \vdash_{type} t : \sigma$. Then*

$$\begin{array}{ll}
\sigma \equiv K_j & \Longleftrightarrow t \in C_j \\
\sigma \equiv \sigma_1 + \sigma_2 & \Longleftrightarrow \exists j \in \{1,2\}.\ \exists t'.\ (t \equiv i_j(t')\ \&\ \Delta \vdash_{type} t' : \sigma_j) \\
\sigma \equiv \sigma_1 \times \sigma_2 & \Longleftrightarrow \exists t_1, t_2.\ (t \equiv < t_1, t_2 >\ \&\ \forall j = 1,2.\ \Delta \vdash_{type} t_j : \sigma_j) \\
\sigma \equiv \mathcal{P}_f(\sigma_1) & \Longleftrightarrow \exists n \geq 0.\ \exists t_1, \ldots, t_n.\ (t \equiv [t_1, \ldots, t_n]\ \& \\
 & \qquad \forall i = 1, \ldots, n.\ \Delta \vdash_{type} t_i : \sigma_1) \\
\sigma \equiv \nu X.\sigma_1 & \Longleftrightarrow \exists n \geq 0.\ \exists t'.\ (t \equiv rec\, x_1 \ldots rec\, x_n.\, in(t')\ \& \\
 & \qquad \Delta, x_1 : \nu X.\sigma_1, \ldots, x_n : \nu X.\sigma_1 \vdash_{type} t' : \sigma_1[\nu X.\sigma_1/X]).
\end{array}$$

The following Substitution Lemma can be easily proved by induction on derivations.

Lemma 1.6 (Substitution).

$$\Delta, x : \tau \vdash_{type} t : \sigma\ \&\ \ \Delta \vdash_{type} t' : \tau\ \implies\ \Delta \vdash_{type} t[t'/x] : \sigma\ .$$

The following notation will be useful in the sequel:

Notation Let $\sigma \in Type$.

- Let T_σ denote the set $\{t \in Term \mid\ \exists \Delta.\ \Delta \vdash_{type} t : \sigma\}$.
- Let T^0_σ denote the set $\{t \in Term^0 \mid \vdash_{type} t : \sigma\}$.

1.1 Bisimulation Equivalence on Closed Typable Terms

In this subsection we give the intended fixed point semantics of our typed language. This takes the form of a family of *bisimulation equivalences* $\{\approx_\sigma\}_{\sigma \in Type}$, where $\approx_\sigma$ is a relation on the set of closed terms T^0_σ. The family $\{\approx_\sigma\}_\sigma$ is characterized as the greatest fixed point of the following monotone operator, whose definition clearly reflects the intended meaning of the constructors:

Definition 1.7. Let $\Phi : \Pi_{\sigma \in Type} \mathcal{P}(T^0_\sigma \times T^0_\sigma) \to \Pi_{\sigma \in Type} \mathcal{P}(T^0_\sigma \times T^0_\sigma)$ be the operator[1] defined as follows

$$\Phi(\{\mathcal{R}_\sigma\}_{\sigma \in Type}) = \{\mathcal{R}^\Phi_\sigma\}_{\sigma \in Type}\ ,$$

where the the relation $\mathcal{R}^\Phi_\sigma \subseteq T^0_\sigma \times T^0_\sigma$ is defined by

$$\begin{array}{ll}
t\ \mathcal{R}^\Phi_{K_j}\ t' & \Longleftrightarrow t \equiv t' \\
t\ \mathcal{R}^\Phi_{\sigma_1+\sigma_2}\ t' & \Longleftrightarrow \exists j \in \{1,2\}.\exists t_1, t'_1.\ (t \equiv i_j(t_1)\ \&\ t' \equiv i_j(t'_1)\ \&\ t_1\ \mathcal{R}_{\sigma_j}\ t'_1) \\
t\ \mathcal{R}^\Phi_{\sigma_1\times\sigma_2}\ t' & \Longleftrightarrow \exists t_1, t_2, t'_1, t'_2.\ (t \equiv < t_1, t_2 >\ \&\ t' \equiv < t'_1, t'_2 >\ \& \\
 & \qquad \forall j = 1,2.\ t_j\ \mathcal{R}_{\sigma_j}\ t'_j) \\
t\ \mathcal{R}^\Phi_{\mathcal{P}_f(\sigma_1)}\ t' & \Longleftrightarrow \exists m, n \geq 0. \exists t_1, \ldots, t_m, t'_1, \ldots, t'_n.\ (t \equiv [t_1, \ldots, t_m]\ \& \\
 & \qquad t' \equiv [t'_1, \ldots, t'_n]\ \& \\
 & \qquad \forall t_i \in [t_1, \ldots, t_m]\ \exists t'_j \in [t'_1, \ldots, t'_n].\ t_i\ \mathcal{R}_{\sigma_1}\ t'_j\ \& \\
 & \qquad \forall t'_j \in [t'_1, \ldots, t'_n]\ \exists t_i \in [t_1, \ldots, t_m].\ t_i\ \mathcal{R}_{\sigma_1}\ t'_j) \\
t\ \mathcal{R}^\Phi_{\nu X_1.\sigma_1}\ t' & \Longleftrightarrow \exists m, n \geq 0. \exists t_1, t'_1.\ (t \equiv rec\, x_1 \ldots rec\, x_m.in(t_1)\ \& \\
 & \qquad t' \equiv rec\, x_1 \ldots rec\, x_n.in(t'_1)\ \& \\
 & \qquad t_1[t/x_1, \ldots, t/x_m]\ \mathcal{R}_{\sigma_1[\nu X_1.\sigma_1/X_1]}\ t'_1[t'/x_1, \ldots, t'/x_n]).
\end{array}$$

[1] $\Pi_{i\in I} A_i$ denotes the infinite cartesian product of the A_i's, for $i \in I$.

The definition above can be viewed as the set-theoretical counterpart of the definition of relational structures on c.p.o.'s given by Pitts (see [Pit96]). Among the various differences between our approach and his, we point out that we allow for *nested recursion* directly at the outset in Definition 1.7, while Pitts deals with it separately.

Proposition 1.8. *The operator* $\Phi : \Pi_{\sigma \in Type} \mathcal{P}(T^0_\sigma \times T^0_\sigma) \to \Pi_{\sigma \in Type} \mathcal{P}(T^0_\sigma \times T^0_\sigma)$ *is monotone over the complete lattice* $(\Pi_{\sigma \in Type} \mathcal{P}(T^0_\sigma \times T^0_\sigma), \Pi_{\sigma \in Type} \subseteq_\sigma)$, *where* $\forall \sigma.\ \subseteq_\sigma \equiv \subseteq$.

Let us denote by $\{\approx_\sigma\}_{\sigma \in Type}$ the greatest fixed point of the operator Φ. This will be the family of *bisimulation equivalences* giving the intended semantics of our system.

The validity of the following coinduction principle follows immediately:

$$\frac{\forall \sigma \in Type.\ \mathcal{R}_\sigma \subseteq \mathcal{R}^\Phi_\sigma}{\forall \sigma \in Type.\ \mathcal{R}_\sigma \subseteq \approx_\sigma}$$

We call Φ-*bisimulation* a family $\{\mathcal{R}_\sigma\}_{\sigma \in Type}$ such that $\forall \sigma \in Type.\ \mathcal{R}_\sigma \subseteq \mathcal{R}^\Phi_\sigma$.

Notice that, using our language of types and the notion of bisimulation equivalences introduced above, we can recover the case of binary trees, and the case of non-deterministic processes with strong bisimulation equivalence. In fact, binary trees can be described as the set of terms $T^0_{\nu X.(X \times X) + \sigma_C}$, for σ_C constant type, while non-deterministic processes over a set of labels C of type σ_C can be described as the set of terms $T^0_{\nu X. \mathcal{P}_f(\sigma_C \times X)}$.

2 A Coinductive Logical System for Bisimulation Equivalence

In this section, we introduce the formal system $\mathcal{S}_{co}$, *à la* Gentzen, for proving $\sim$-equivalence between pairs of terms. We will show that $\mathcal{S}_{co}$ axiomatizes exactly, for all type σ, the bisimulation equivalence $\approx_\sigma$.

Definition 2.1. Let $\mathcal{S}_{co}$ be the following formal system for deriving judgements of the shape $< \Delta; \Gamma >\vdash_{co} t \sim t' : \sigma$, where $< \Delta; \Gamma >$ is the environment and

- Δ is a partial function from Var to $Type$;
- Γ is a multiset of the shape $[t_1 \sim t'_1 : \sigma_1, \ldots, t_n \sim t'_n : \sigma_n]$;
- Γ is *coherent* with Δ, i.e.
  $t_i \sim t'_i : \sigma_i \in \Gamma \ \Rightarrow\ (\Delta \vdash_{type} t_i : \sigma_i\ \&\ \Delta \vdash_{type} t'_i : \sigma_i)$;
- $\Delta \vdash_{type} t : \sigma\ \&\ \Delta \vdash_{type} t' : \sigma$.

The rules of $\mathcal{S}_{co}$ are the following:

$$\frac{\Delta \vdash_{type} t : \sigma \quad \Gamma \text{ coherent with } \Delta}{< \Delta; \Gamma >\vdash_{co} t \sim t : \sigma} \quad (refl)$$

$$\frac{< \Delta; \Gamma >\vdash_{co} t_1 \sim t_2 : \sigma}{< \Delta; \Gamma >\vdash_{co} t_2 \sim t_1 : \sigma} \quad (symm)$$

$$\frac{< \Delta; \Gamma >\vdash_{co} t_1 \sim t_2 : \sigma \quad < \Delta; \Gamma >\vdash_{co} t_2 \sim t_3 : \sigma}{< \Delta; \Gamma >\vdash_{co} t_1 \sim t_3 : \sigma} \quad (trans)$$

$$\frac{\Delta \vdash_{type} t : \sigma \quad \Delta \vdash_{type} t' : \sigma}{< \Delta; \Gamma, t \sim t' : \sigma >\vdash_{co} t \sim t' : \sigma} \quad (hyp)$$

$$\frac{\Delta \vdash_{type} rec\, x.t : \nu X.\sigma \quad \Gamma \text{ coherent with } \Delta}{< \Delta; \Gamma >\vdash_{co} rec\, x.t \sim t[rec\, x.t/x] : \nu X.\sigma} \quad (rec)$$

$$\frac{\{< \Delta; \Gamma >\vdash_{co} t_i \sim t'_i : \sigma_i\}_{i=1,2}}{< \Delta; \Gamma >\vdash_{co} < t_1, t_2 > \sim < t'_1, t'_2 > : \sigma_1 \times \sigma_2} \quad (\times cong)$$

$$\frac{< \Delta; \Gamma >\vdash_{co} t \sim t' : \sigma_1}{< \Delta; \Gamma >\vdash_{co} i_1(t) \sim i_1(t') : \sigma_1 + \sigma_2} \quad (+_1 cong)$$

$$\frac{< \Delta; \Gamma >\vdash_{co} t \sim t' : \sigma_2}{< \Delta; \Gamma >\vdash_{co} i_2(t) \sim i_2(t') : \sigma_1 + \sigma_2} \quad (+_2 cong)$$

$$\frac{\{< \Delta; \Gamma >\vdash_{co} t_i \sim t'_i : \sigma\}_{i=1,\ldots,n}}{< \Delta; \Gamma >\vdash_{co} [t_1, \ldots, t_n] \sim [t'_1, \ldots, t'_n] : \mathcal{P}_f(\sigma)} \quad ([\,] cong)$$

$$\frac{\Delta \vdash_{type} [t_1, \ldots, t_n, t, t'] : \mathcal{P}_f(\sigma) \quad < \Delta; \Gamma >\vdash_{co} t \sim t' : \sigma}{< \Delta; \Gamma >\vdash_{co} [t_1, \ldots, t_n, t, t'] \sim [t_1, \ldots, t_n, t] : \mathcal{P}_f(\sigma)} \quad (abs)$$

$$\frac{< \Delta; \Gamma, in(t) \sim in(t') : \nu X.\sigma >\vdash_{co} t \sim t' : \sigma[\nu X.\sigma/X]}{< \Delta; \Gamma >\vdash_{co} in(t) \sim in(t') : \nu X.\sigma} \quad (in)$$

The names given to the rules above are suggestive. In particular, the rules *(cong)* are the *congruence rules*, while rule *(abs)* is the *absorption rule*, which embodies contraction for equal terms appearing in multisets.

One can easily check, by induction on derivations, using Lemma 1.6, that the definition above is well posed, i.e.

$$< \Delta; \Gamma >\vdash_{co} t \sim t' : \sigma \implies (\Gamma \text{ coherent with } \Delta \,\&\, \Delta \vdash_{type} t : \sigma \,\&\, \Delta \vdash_{type} t' : \sigma) .$$

Notice the "coinductive" nature of the rule (in): in order to establish the equivalence $\sim$ between terms of the shape $in(t)$ and $in(t')$, we can assume, in the premise of the rule (in), the judgement that we want to prove, i.e. $in(t) \sim in(t')$.

Remark 2.2. i) In place of rule (in) in the system $\mathcal{S}_{co}$ above, one could use equivalently the following two rules

$$\frac{< \Delta; \Gamma >\vdash_{co} t \sim t' : \sigma}{< \Delta; \Gamma >\vdash_{co} in(t) \sim in(t') : \nu X.\sigma}$$

$$\frac{< \Delta; \Gamma, rec\, x.t \sim rec\, y.t' : \nu X.\sigma >\vdash_{co} t \sim t' : \sigma}{< \Delta; \Gamma >\vdash_{co} rec\, x.t \sim rec\, y.t' : \nu X.\sigma}$$

This latter presentation would emphasize Coquand's correspondence between guarded infinite objects and guarded infinite proofs, but the presentation of the system of Definition 2.1 slightly simplifies the proof of Theorem 2.10 below.

ii) When specialized to the type $\nu X.\mathcal{P}_f(\sigma_C \times X)$ of CCS non-deterministic processes, our logical system provides an alternative axiomatization of Milner's strong bisimulation ([Mil84]). The crucial difference between our system and the classical system of Milner is the absence, in $\mathcal{S}_{co}$, of a counterpart to Milner's rule for recursion, *viz*: $\dfrac{t \sim t'[t/x]}{t \sim rec\, x.t'}$ *(uniqueness)* . This rule is recovered in $\mathcal{S}_{co}$ by the coinductive rule (in), which amounts to the coinductive version of the congruence rule for the *rec* operator in Milner's system. Milner's system is a Hilbert system. Hence top-down proof search can be rather unpractical. For instance, when confronted with two terms one of which is a *rec* term, one has to guess whether to unfold the term or to use rule *(uniqueness)*. On the contrary, in $\mathcal{S}_{co}$, one can capitalize on hypotheses, and hence the structure of terms determine essentially, i.e. up-to absorption and unfolding, the top-down search of a proof (see Example 2.4 below). This informal argument will be put to use in order to show the completeness of $\mathcal{S}_{co}$ (Theorem 2.13) and its decidability.

It is immediate to see, by induction on derivations, that the following Weakening Lemma holds:

Lemma 2.3 (Weakening). *If* $< \Delta; \Gamma >\vdash_{co} t \sim t' : \sigma$ *is derivable in* $\mathcal{S}_{co}$ *and* Γ' *is coherent with* Δ*, then also* $< \Delta; \Gamma, \Gamma' >\vdash_{co} t \sim t' : \sigma$ *is derivable in* $\mathcal{S}_{co}$*.*

We illustrate now the system $\mathcal{S}_{co}$ at work.

Example 2.4. Let $t_1 \equiv rec\, x.in(< c, x >)$ and $t_2 \equiv rec\, y.in(< c, in(< c, y >) >)$, where $c \in C_j$. Then one can easily check that $\vdash_{type} t_i : \nu X.K_j \times X$, for $i = 1, 2$. Moreover, using the system $\mathcal{S}_{co}$, one can show that the two terms t_1 and t_2 are bisimilar. In fact, up to applications of the rules *(rec), (symm), (trans)*, we can build a derivation of $\vdash_{co} t_1 \sim t_2 : \nu X.K_j \times X$ as follows:

$$\dfrac{\dfrac{< \Delta; \Gamma', \Gamma >\vdash_{co} c \sim c : K_j \quad < \Delta; \Gamma', \Gamma >\vdash_{co} t_1 \sim t_2 : \nu X.K_j \times X}{< \Delta; \Gamma', \Gamma >\vdash_{co} < c, t_1 >\sim< c, t_2 > : K_j \times \nu X.K_j \times X}\ (\times cong)}{< \Delta; \Gamma >\vdash_{co} in(< c, t_1 >) \sim in(< c, t_2 >) : \nu X.K_j \times X}\ (in)$$

and

$$\dfrac{\dfrac{< \Delta; \Gamma >\vdash_{co} c \sim c : K_j \quad < \Delta; \Gamma >\vdash_{co} t_1 \sim in(< c, t_2 >) : \nu X.K_j \times X}{< \Delta; \Gamma >\vdash_{co} < c, t_1 >\sim< c, in(< c, t_2 >) > : K_j \times \nu X.K_j \times X}\ (\times cong)}{\vdash_{co} in(< c, t_1 >) \sim in(< c, in(< c, t_2 >)) : \nu X.K_j \times X}\ (in)$$

where
$\Delta \equiv \emptyset$
$\Gamma \equiv [in(< c, t_1 >) \sim in(< c, in(< c, t_2 >) >) : \nu X.K_j \times X]$
$\Gamma' \equiv [in(< c, t_1 >) \sim in(< c, t_2 >) : \nu X.K_j \times X]$.

Example 2.5 (Conway Identity). A term with $n > 0$ rec's at the top is equivalent to a term with just one rec, i.e., any term $\overline{t} \equiv rec\, x_1 \ldots rec\, x_n.in(t)$, $n > 0$, typable with $\nu X.\sigma$, for some σ, is such that

$$\exists < \Delta; \Gamma > .\ < \Delta; \Gamma >\vdash_{co} \overline{t} \sim \overline{t'} : \nu X.\sigma\ ,$$

where $\overline{t'} \equiv rec\, x.in(t[x/x_1, \ldots, x/x_n])$, and the variables $x_1, \ldots, x_n$ are replaced by the variable x, which is new in $\overline{t}$.
By rules *(rec)*, *(symm)*, *(trans)*, *(in)* (using also the Weakening Lemma), it is sufficient to show that the two terms $t[\overline{t}/x_1, \ldots, \overline{t}/x_n]$ and $t[\overline{t'}/x_1, \ldots, \overline{t'}/x_n]$ are $\sim$-equivalent. More in general, we show, by structural induction on t, that, for all $n > 0$, for all $\overline{t}_1, \overline{t'}_1, \ldots, \overline{t}_n, \overline{t'}_n$ such that $\exists \Delta \exists \tau.\ \Delta \vdash_{type} t[\overline{t}_1/x_1, \ldots, \overline{t}_n/x_n] : \tau$ and $\Delta \vdash_{type} t[\overline{t'}_1/x_1, \ldots, \overline{t'}_n/x_n] : \tau$,

$$\exists \Gamma.\ < \Delta; \Gamma > \vdash_{co} t[\overline{t}_1/x_1, \ldots, \overline{t}_n/x_n] \sim t[\overline{t'}_1/x_1, \ldots, \overline{t'}_n/x_n] : \tau\ .$$

The only non trivial case is that of $t \equiv rec\, y_1 \ldots rec\, y_m.in(\widetilde{t})$, for $m \geq 0$. But, again by rules *(rec)*, *(symm)*, *(trans)*, *(in)*, it is sufficient to prove that
$< \Delta; \Gamma > \vdash_{co} \widetilde{t}^{\overline{t}_1 \ldots \overline{t}_n t \ldots t}_{x_1 \ldots x_n y_1 \ldots y_m} \sim \widetilde{t}^{\overline{t'}_1 \ldots \overline{t'}_n t \ldots t}_{x_1 \ldots x_n y_1 \ldots y_m} : \tau'$, for a suitable τ', where
$\widetilde{t}^{\overline{t}_1 \ldots \overline{t}_n t \ldots t}_{x_1 \ldots x_n y_1 \ldots y_m} \equiv \widetilde{t}[\overline{t}_1/x_1, \ldots, \overline{t}_n/x_n, t/y_1, \ldots, t/y_m]$ and
$\widetilde{t}^{\overline{t'}_1 \ldots \overline{t'}_n t \ldots t}_{x_1 \ldots x_n y_1 \ldots y_m} \equiv \widetilde{t}[\overline{t'}_1/x_1, \ldots, \overline{t'}_n/x_n, t/y_1, \ldots, t/y_m]$.
But this follows by induction hypothesis.

The rest of this section is devoted to the proof of the fact that the system $\mathcal{S}_{co}$ axiomatizes exactly, for all type σ, the bisimulation equivalence $\approx_\sigma$. More precisely, we will prove that, for all $\sigma \in Type$ and for all $t, t' \in T^0_\sigma$,

$$\vdash_{co} t \sim t' : \sigma \iff t \approx_\sigma t'\ .$$

We will refer to the implication ($\Rightarrow$) as the correctness of the system $\mathcal{S}_{co}$ w.r.t. $\approx_\sigma$, and to the implication ($\Leftarrow$) as the completeness of the system $\mathcal{S}_{co}$ w.r.t. $\approx_\sigma$.

2.1 Correctness of $\mathcal{S}_{co}$

First we need a technical definition.

Definition 2.6. A sequent $< \Delta; t_1 \sim t'_1 : \sigma_1, \ldots, t_n \sim t'_n : \sigma_n >\vdash_{co} t_{n+1} \sim t'_{n+1} : \sigma_{n+1}$ is *completely derivable* in $\mathcal{S}_{co}$ if there exist derivations in $\mathcal{S}_{co}$ of $< \Delta; t_1 \sim t'_1 : \sigma_1, \ldots, t_{i-1} \sim t'_{i-1} : \sigma_{i-1} >\vdash_{co} t_i \sim t'_i : \sigma_i$, for all $i = 1, \ldots, n+1$.

In order to show the correctness of $\mathcal{S}_{co}$, we will prove that the following family of relations is a Φ-bisimulation:

Definition 2.7. Let $\sigma \in Type$. We define

$$\mathcal{R}^{cd}_\sigma = \{(t, t') \in T^0_\sigma \times T^0_\sigma \mid \exists < \Delta; \Gamma > .\ < \Delta; \Gamma > \vdash_{co} t \sim t' : \sigma \text{ completely derivable}\}\ .$$

The following two lemmata are instrumental.

Lemma 2.8. *Let* $< \Delta; \Gamma > \vdash_{co} t \sim t' : \sigma$ *be a completely derivable sequent. Then*

1. *If* $\sigma \equiv \sigma_1 + \sigma_2$, $t \equiv i_j(\overline{t})$ *and* $t' \equiv i_j(\overline{t}')$, *for some* $j \in \{1,2\}$, *then also* $< \Delta; \Gamma >\vdash_{co} \overline{t} \sim \overline{t}' : \sigma_j$ *is a completely derivable sequent.*
2. *If* $\sigma \equiv \sigma_1 \times \sigma_2$ *and* $t \equiv < t_1, t_2 >$, *then also* $< \Delta; \Gamma >\vdash_{co} t_i \sim t_i' : \sigma_i$, *for all* $i = 1,2$, *is a completely derivable sequent.*
3. *If* $\sigma \equiv \mathcal{P}_f(\sigma_1)$, $t \equiv [t_1, \ldots, t_m]$, $t' \equiv [t_1', \ldots, t_n']$, *then* $\forall i \in \{1, \ldots, m\}$. $\exists j \in \{1, \ldots, n\}$ *such that* $< \Delta; \Gamma >\vdash_{co} t_i \sim t_j' : \sigma_1$ *is a completely derivable sequent, and* $\forall j \in \{1, \ldots, n\}$. $\exists i \in \{1, \ldots, m\}$ *such that* $< \Delta; \Gamma >\vdash_{co} t_i \sim t_j' : \sigma_1$ *is a completely derivable sequent.*

Proof. The proof is by induction on the sum of the lengths of the derivations τ and τ_i's, where τ denotes the derivation of $< \Delta; t_1 \sim t_1' : \sigma_1, \ldots, t_n \sim t_n' : \sigma_n >\vdash_{co} t \sim t' : \sigma$ and τ_i denotes the derivation of $< \Delta; t_1 \sim t_1' : \sigma_1, \ldots, t_{i-1} \sim t_{i-1}' : \sigma_{i-1} >\vdash_{co} t_i \sim t_i' : \sigma_i$, for $i = 1, \ldots, n$. We work out in detail only the proof of item 3, the proofs of the other two items are similar.
Base Case: The only rule applied in τ is *(refl)*. The thesis follows using Lemma 1.5 and rule *(refl)*.
Induction Step: we proceed by analyzing the last rule applied in τ. If the last rule is *(refl)*, then again the thesis follows using Lemma 1.5 and rule *(refl)*. If the last rule is *(symm)* or *(hyp)*, the thesis follows immediately by induction hypothesis. If the last rule is *(trans)*, then the thesis follows by induction hypothesis, using Lemma 1.5. If the last rule is *(abs)*, the thesis follows using Lemma 1.5 and rule *(refl)*. Finally, if the last rule in τ is $([\,]cong)$, then the thesis is immediate. □

Lemma 2.9. *Let* $\sigma \in Type$. *Then*
i) For all $t \in T_\sigma^0$, $t(\mathcal{R}_\sigma^{cd})^\Phi t$.
ii) For all $t_1, t_2 \in T_\sigma^0$, $t_1(\mathcal{R}_\sigma^{cd})^\Phi t_2 \implies t_2(\mathcal{R}_\sigma^{cd})^\Phi t_1$.
iii) For all $t_1, t_2, t_3 \in T_\sigma^0$ *such that, for some* Γ, Δ, *the sequents* $< \Delta; \Gamma >\vdash_{co} t_1 \sim t_2 : \sigma$ *and* $< \Delta; \Gamma' >\vdash_{co} t_2 \sim t_3 : \sigma$ *are completely derivable,*

$$[t_1(\mathcal{R}_\sigma^{cd})^\Phi t_2 \ \&\ t_2(\mathcal{R}_\sigma^{cd})^\Phi t_3] \implies t_1(\mathcal{R}_\sigma^{cd})^\Phi t_3 \ .$$

Proof. Both items i) and ii) can be easily shown by case analysis on σ, using Lemma 1.5. Item iii) is shown by by case analysis on σ, using Lemmata 2.3 and 2.8. □

Theorem 2.10 (Correctness). *Let* $\sigma \in Type$. *For all* $t, t' \in T_\sigma^0$,

$$\vdash_{co} t \sim t' : \sigma \implies t \approx_\sigma t' \ .$$

Proof. We show that the family $\{\mathcal{R}_\sigma^{cd}\}_{\sigma \in Type}$ is a Φ-bisimulation, i.e. we have to show that $\forall \sigma$. $\mathcal{R}_\sigma^{cd} \subseteq (\mathcal{R}_\sigma^{cd})^\Phi$. We prove this by induction on the sum of the lengths of the derivations τ and τ_i's, where τ denotes the derivation of $< \Delta; t_1 \sim t_1' : \sigma_1, \ldots, t_n \sim t_n' : \sigma_n >\vdash_{co} t \sim t' : \nu X.\sigma$ and τ_i denotes the derivation of $< \Delta; t_1 \sim t_1' : \sigma_1, \ldots, t_{i-1} \sim t_{i-1}' : \sigma_{i-1} >\vdash_{co} t_i \sim t_i' : \sigma_i$, for $i = 1, \ldots, n$.
Base Case: The only rule applied in τ is *(refl)* or *(rec)*. The thesis follows from item i) of Lemma 2.9.

Induction Step: We proceed by analyzing the last rule applied in τ. If the last rule is *(refl)* or *(rec)*, then again the thesis follows from item i) of Lemma 2.9. If the last rule is *(symm)*, then the thesis is immediate by induction hypothesis, using item ii) of Lemma 2.9. If the last rule is *(trans)*, then again the thesis is immediate by induction hypothesis, using item iii) of Lemma 2.9. If the last rule in τ is one of the following $(\times cong)$, $(+_1 cong)$, $(+_2 cong)$, $([\,]cong)$, *(abs)*, then then the thesis is immediate. Finally, if the last rule in τ is *(hyp)* or *(in)*, then the thesis follows immediately from the induction hypothesis.

□

2.2 Completeness of $\mathcal{S}_{co}$

In order to show the completeness of the system $\mathcal{S}_{co}$, we need to exploit the implicit regularity of the terms expressible in our language. Namely, we introduce the notion of set of subterms of a given term.

Definition 2.11. Let $t \in T_\sigma$. The set of subterms of t, $sub(t)$, is defined by induction on t as follows:

- if $t \equiv x \in Var$ or $t \equiv c \in C$, then $sub(t) = \{t\}$;
- if $t \equiv i_j(t')$, for some $j \in \{1,2\}$, then $sub(t) = \{t\} \cup sub(t')$;
- if $t \equiv < t_1, t_2 >$, then $sub(t) = \{t\} \cup sub(t_1) \cup sub(t_2)$;
- if $t \equiv [t_1, \ldots, t_n]$, for some $n \geq 0$, then $sub(t) = \{t\} \cup \bigcup_{i=1,\ldots,n} sub(t_i)$;
- if $t \equiv in(t')$, then $sub(t) = \{t\} \cup sub(t')$;
- if $t \equiv rec\, x.t'$, then $sub(t) = \{t\} \cup \{t_1[t/x] \mid t_1 \in sub(t')\}$.

The following lemma can be immediately shown by induction on terms.

Lemma 2.12. *For all σ and for all $t \in T_\sigma$,*
i) the set $sub(t)$ is finite;
ii) $\forall t' \in sub(t).\ sub(t') \subseteq sub(t)$.

Now we are in the position of stating the Completeness Theorem for the system $\mathcal{S}_{co}$. The proof of this theorem consists in showing that, if two terms $t, t' \in T^0_\sigma$ are $\approx_\sigma$-bisimilar, then, since they have only a finite number of subterms, we can build a derivation of $\vdash_{co} t \sim t' : \sigma$ in a top-down fashion.

Theorem 2.13 (Completeness). *Let $\sigma \in Type$. For all $t, t' \in T^0_\sigma$,*

$$t \approx_\sigma t' \implies \vdash_{co} t \sim t' : \sigma\ .$$

Proof. We prove that, if $t \approx_\sigma t'$, then for all $t_1, \ldots, t_n, \overline{t} \in sub(t)$, $t'_1, \ldots, t'_n, \overline{t}' \in sub(t')$ such that $\forall i = 1, \ldots, n.\ t_i, t'_i \in T^0_{\sigma_i}$ & $t_i \approx_{\sigma_i} t'_i$, $\overline{t}, \overline{t}' \in T^0_{\overline{\sigma}}$ and $\overline{t} \approx_{\overline{\sigma}} \overline{t}'$, there exists a derivation of $t_1 \sim t'_1 : \sigma_1, \ldots, t_n \sim t'_n : \sigma_n \vdash_{co} \overline{t} \sim \overline{t}' : \overline{\sigma}$.
Suppose by contradiction that $t_1 \sim t'_1 : \sigma_1, \ldots, t_n \sim t'_n : \sigma_n \vdash_{co} \overline{t} \sim \overline{t}' : \overline{\sigma}$ is not derivable. Then we show that there exists an infinite sequence of distinct pairs of processes $t_i, t'_i \in T^0_{\sigma_i}$ such that $t_i \approx_{\sigma_i} t'_i$ and $t_i \in sub(t)$, $t'_i \in sub(t')$,

for $i = 1, \ldots, n$, which is clearly impossible because, by Lemma 2.12, $sub(t)$ and $sub(t')$ are finite. In fact, if $t_1 \sim t'_1 : \sigma_1, \ldots, t_n \sim t'_n : \sigma_n \vdash_{co} \overline{t} \sim \overline{t}' : \overline{\sigma}$ is not derivable, then we show that a sequent of the following shape is not derivable: $t_1 \sim t'_1 : \sigma_1, \ldots, t_n \sim t'_n : \sigma_n, t_{n+1} \sim t'_{n+1} : \sigma_{n+1} \vdash_{co} \widehat{t} \sim \widehat{t'} : \widehat{\sigma}$, for some $t_{n+1}, \widehat{t} \in sub(t)$, $t'_{n+1}, \widehat{t'} \in sub(t')$, such that $t_{n+1} \approx_{\sigma_{n+1}} t'_{n+1}$, $\widehat{t} \approx_{\widehat{\sigma}} \widehat{t'}$, and the hypothesis $t_{n+1} \sim t'_{n+1} : \sigma_{n+1}$ is new, in the sense that it does not appear among $t_1 \sim t'_1 : \sigma_1, \ldots, t_n \sim t'_n : \sigma_n$. This latter fact is proved by induction on the structure of $\overline{\sigma}$.
If $\overline{\sigma} \equiv K_j$, then the sequent $t_1 \sim t'_1 : \sigma_1, \ldots, t_n \sim t'_n : \sigma_n \vdash_{co} \overline{t} \sim \overline{t}' : K_j$ is immediately derivable, since $\overline{t} \approx_{K_j} \overline{t}' \Rightarrow t = t' \in C_j$.
If $\overline{\sigma} \equiv \nu X_1.\sigma_1$, then there exists $m, n \geq 0$ such that $\overline{t} \equiv rec\, x_1 \ldots rec\, x_m.in(\widetilde{t})$ and $\overline{t'} \equiv rec\, x_1 \ldots rec\, x_n.in(\widetilde{t'})$, for some terms $\widetilde{t}, \widetilde{t'}$. Then, by rule *(in)* (possibly using rules *(rec)*, *(symm)*, and *(trans)*), also $t_1 \sim t'_1 : \sigma_1, \ldots, t_n \sim t'_n : \sigma_n, in(\widetilde{t})[\overline{t}/x_1, \ldots, \overline{t}/x_m] \sim in(\widetilde{t})[\overline{t}'/x_1, \ldots, \overline{t}'/x_n] : \nu X_1.\sigma_1 \vdash_{co} \widetilde{t}[\overline{t}/x_1, \ldots, \overline{t}/x_m] \sim \widetilde{t}[\overline{t}'/x_1, \ldots, \overline{t}'/x_n] : \nu X_1.\sigma_1$ is not derivable, and the pair $in(\widetilde{t})[\overline{t}/x_1, \ldots, \overline{t}/x_m] \sim in(\widetilde{t})[\overline{t}'/x_1, \ldots, \overline{t}'/x_n] : \nu X_1.\sigma_1$ is new among $t_1 \sim t'_1 : \sigma_1, \ldots, t_n \sim t'_n : \sigma_n$, otherwise we would already have a proof of the sequent $t_1 \sim t'_1 : \sigma_1, \ldots, t_n \sim t'_n : \sigma_n \vdash_{co} \overline{t} \sim \overline{t}' : \overline{\sigma}$.
If $\overline{\sigma} \equiv \sigma_1 + \sigma_2$, then $\overline{t} \equiv i_j(\overline{t}_j)$ and $\overline{t}' \equiv i_j(\overline{t}'_j)$, for some $j \in \{1, 2\}$ and, by rule $(+_j\, cong)$ (possibly using rules *(rec)*, *(symm)*, and *(trans)*), also the sequent $t_1 \sim t'_1 : \sigma_1, \ldots, t_n \sim t'_n : \sigma_n \vdash_{co} \overline{t}_j \sim \overline{t_j}' : \sigma_j$ is not derivable. Hence we can apply the induction hypothesis to σ_j, since, by definition of $i_j(\overline{t}_j) \approx_{\sigma_1+\sigma_2} i_j(\overline{t}'_j)$, we have also $\overline{t}_j \approx_{\sigma_j} \overline{t}'_j$.
Finally, the cases $\overline{\sigma} \equiv \sigma_1 \times \sigma_2$ and $\overline{\sigma} \equiv \mathcal{P}_f(\sigma_1)$ are dealt with similarly to the previous case. □

The proof of Theorem 2.13 above is given by contradiction just for the sake of conciseness. Clearly a *constructive* proof can be easily obtained from the proof above. As a side-remark, we point out that a proof of decidability of $\sim$-equivalence can be easily obtained using the argument of the above proof.

3 Categorical Semantics

In this section we give a categorical final semantics in the style of [Acz88, RT93, Len96, Rut96, Len98] (to which we refer for further details on this topic) to our language, and we show that it captures exactly the greatest fixed point semantics of Section 1.

The interest of this categorical semantics is that it achieves a significant degree of generality, in that it subsumes naturally a great number of concrete examples of infinite objects in programming. The significance of a final semantics for a language like ours is that, contrary to the fixed point semantics, it allows us to embody as a point of a final coalgebra a canonical "minimal"representative for each equivalence class of terms. These denotations are the mathematical

counterparts of our intuitive circular objects. Notice that defining a final semantics for a language with a given notion of equivalence is not a mechanical task.

We work in the category Set^* of non-wellfounded sets and set-theoretic functions for simplicity, but we could have also worked in other categories based on sets. Denotations would have become rather obscure however. We proceed as follows. We define a "universal" endofunctor F, embodying constructors corresponding to the type constructors. Then we endow the set $\Sigma_{\sigma \in Type} T^0_\sigma$, i.e. the disjoint sum of all closed typable terms, with a structure of F-coalgebra. Finally, we show that the largest F-bisimulation on the coalgebra defined on $\Sigma_{\sigma \in Type} T^0_\sigma$ coincides with the family of bisimulation equivalences $\{\approx_\sigma\}_\sigma$ introduced in Section 1.

Our categorical semantics could be equivalently presented in the framework of categories indexed over types. But, for the sake of simplicitly, we prefer the set-theoretic setting.

For more informations on the Final Semantics paradigm see e.g. [Len98].

Definition 3.1. Let $F : Set^* \to Set^*$ be the functor defined by:

$$F(X) = \Sigma_{\sigma \in Type}(\Sigma_{j \leq n} C_j + (X + X) + (X \times X) + \mathcal{P}_f(X)) \ .$$

We endow the set $\Sigma_{\sigma \in Type} T^0_\sigma$ with a structure of F-coalgebra as follows:

Definition 3.2. Let $\alpha : \Sigma_{\sigma \in Type} T^0_\sigma \to F(\Sigma_{\sigma \in Type} T^0_\sigma)$ be the function defined by:

$$\alpha(t) = (\sigma, z) \ ,$$

where

$$z = \begin{cases} \mathrm{in}_{K_j}(c^i_j) & \text{if } A_{K_j} \\ \mathrm{in}_+(i_j(t_1)) & \text{if } A_{\sigma_1+\sigma_2} \\ \mathrm{in}_\times(t_1, t_2) & \text{if } A_{\sigma_1\times\sigma_2} \\ \mathrm{in}_{\mathcal{P}_f}([t_1, \ldots, t_m]) & \text{if } A_{\mathcal{P}_f(\sigma_1)} \\ \mathrm{in}_{K_j}(c^i_j) & \text{if } A_{\nu X.K_j} \\ \mathrm{in}_+(i_j(t_1[t/x_1, \ldots, t/x_n])) & \text{if } A_{\nu X.\sigma_1+\sigma_2} \\ \mathrm{in}_\times(t_1[t/x_1, \ldots, t/x_n], t_2[t/x_1, \ldots, t/x_n]) & \text{if } A_{\nu X.\sigma_1\times\sigma_2} \\ \mathrm{in}_{\mathcal{P}_f}([t_1[t/x_1, \ldots, t/x_n], \ldots, t_m[t/x_1, \ldots, t/x_n]]) & \text{if } A_{\nu X.\mathcal{P}_f(\sigma_1)} \end{cases}$$

where in_{K_j}, in_+, $\mathrm{in}_\times$, $\mathrm{in}_{\mathcal{P}_f}$ denote canonical injections into disjoint sum and

$$\begin{array}{ll} A_{K_j} & \equiv (\sigma \equiv K_j \wedge t \equiv c^i_j, i \in I_j) \\ A_{\sigma_1+\sigma_2} & \equiv (\sigma \equiv \sigma_1 + \sigma_2 \wedge t \equiv i_j(t_1)) \\ A_{\sigma_1\times\sigma_2} & \equiv (\sigma \equiv \sigma_1 \times \sigma_2 \wedge t \equiv < t_1, t_2 >) \\ A_{\mathcal{P}_f(\sigma_1)} & \equiv (\sigma \equiv \mathcal{P}_f(\sigma_1) \wedge t \equiv [t_1, \ldots, t_m]) \\ A_{\nu X.K_j} & \equiv (\sigma \equiv \nu X.K_j \wedge t \equiv rec\, x_1 \ldots rec\, x_n.in(c^i_j), i \in I_j) \\ A_{\nu X.\sigma_1+\sigma_2} & \equiv (\sigma \equiv \nu X.\sigma_1 + \sigma_2 \wedge t \equiv rec\, x_1 \ldots rec\, x_n.in(i_j(t_1))) \\ A_{\nu X.\sigma_1\times\sigma_2} & \equiv (\sigma \equiv \nu X.\sigma_1 \times \sigma_2 \wedge t \equiv rec\, x_1 \ldots rec\, x_n.in(< t_1, t_2 >)) \\ A_{\nu X.\mathcal{P}_f} & \equiv (\sigma \equiv \nu X.\mathcal{P}_f(\sigma_1) \wedge t \equiv rec\, x_1 \ldots rec\, x_n.in([t_1, \ldots, t_m])). \end{array}$$

Now, our goal is that of showing that the largest F-bisimulation on the coalgebra $(\Sigma_\sigma T^0_\sigma, \alpha)$, which we denote by $\sim$, coincides exactly with the family of bisimulation equivalences $\{\approx_\sigma\}_\sigma$ defined in Section 1. First of all, we recall the definition of categorical F-bisimulation:

Definition 3.3. Let $F : Set^* \to Set^*$. An F-bisimulation on the F-coalgebra (X, α_X) is a set-theoretic relation $R \subseteq X \times X$ such that there exists an arrow of Set^*, $\gamma : \mathcal{R} \to F(\mathcal{R})$, making the following diagram commute:

$$\begin{array}{ccccc} X & \xleftarrow{\pi_1} & \mathcal{R} & \xrightarrow{\pi_2} & X \\ {\scriptstyle \alpha_X}\downarrow & & \downarrow{\scriptstyle \gamma} & & \downarrow{\scriptstyle \alpha_X} \\ F(X) & \xleftarrow[F(\pi_1)]{} & F(\mathcal{R}) & \xrightarrow[F(\pi_2)]{} & F(X) \end{array}$$

The proof of the following proposition is routine:

Proposition 3.4. *The largest F-bisimulation on the coalgebra $(\Sigma_\sigma T^0_\sigma, \alpha)$ is the family $\{\approx_\sigma\}_\sigma$.*

4 Final Remarks and Directions for Future Work

In this paper, we have presented a "coinductive" axiomatization of the bisimulation equivalence on non-wellfounded regular objects. Moreover, we have shown that it is complete with respect to a maximal fixed point semantics and also to a categorical semantics. Our presentation makes use of a typed language for denoting circular terms.

We could generalize our language of terms so as to allow non-regular objects, still getting a sound axiomatization. In fact, the regularity property is crucial only for proving the completeness of our system.

There are various other promising directions for possible generalizations and extensions of the coinductive axiomatization presented in this paper.

- Categories other than the purely set-theoretical ones could be investigated. This would involve the use of a generalized notion of set-theoretic relation. In the case of c.p.o.'s, this should go in the direction of providing a formal system for expressing Pitts' relational structures ([Pit96]).
- A richer collection of types, including inductive types and the *mixed* covariant-contravariant $\to$ constructor could be considered, as well as destructors in terms.
- Coarser notions of bisimulations other than Milner's strong bisimulation could be considered, e.g. weak bisimulation and congruence, van Glabbeek-Weijland branching bisimulation, Montanari-Sassone dynamic bisimulation.
- Other coinductively defined equivalences, arising in different contexts, could be considered. E.g. equivalence of streams representing exact reals.
- Finally, it would be interesting to compare systems like $\mathcal{S}_{co}$ to other logics for bisimulations (see e.g. [Mos?]).

References

Acz88. P.Aczel. *Non-well-founded sets*, CSLI Lecture Notes **14**, Stanford 1988.

BV96. J.de Bakker, E.de Vink. *Control Flow Semantics*, Foundations of Computing Series, The MIT Press, Cambridge, 1996.

BM96. J.Barwise, L.Moss. *Vicious circles: On the mathematics of non-wellfounded phenomena*, CSLI Publications, Stanford, 1996.

BH97. M.Brandt, F.Henglein. *Coinductive axiomatization of recursive type equality and subtyping*, *TLCA'97* Conf. Proc., P.de Groote, R.Hindley, eds., Springer LNCS **1210**, 1997, 63–81.

Coq94. T.Coquand. Infinite Objects in Type Theory, Types for Proofs and Programs *TYPES-93*, Springer LNCS **806**, 1994, 62–78.

Fio96. M.Fiore. A Coinduction Principle for Recursive Data Types Based on Bisimulation, *Inf. & Comp.* **127**, 1996, 186–198.

FH83. M.Forti, F.Honsell. Set theory with free construction principles, *Ann. Scuola Norm. Sup. Pisa*, Cl. Sci. (4) **10**, 1983, 493– 522.

Gim94. E.Giménez. Codifying guarded definitions with recursive schemes, *Workshop on Types for Proofs and Programs*, P.Dybjer et al. eds., Springer LNCS **996**, 1994, 39–59.

Gim95. E.Giménez. An application of co-Inductive types in Coq: verification of the Alternating Bit Protocol, *Workshop Types Proofs and Programs*, 1995.

Gim96. E.Giménez. Un calcul de constructions infinies et son application a la verfication de systemes communicants, PhD Thesis, École normale supérieure de Lyon, December 1996.

HL95. F.Honsell, M.Lenisa. Final Semantics for Untyped Lambda Calculus, *TLCA'95* Conf. Proc., M.Dezani et al eds., Springer LNCS **902**, 1995, 249–265.

Len96. M.Lenisa. Final Semantics for a Higher Order Concurrent Language, *CAAP'96*, H.Kirchner et. al. eds., Springer LNCS **1059**, 1996, 102–118.

Len98. M.Lenisa. Themes in Final Semantics, Ph.D. Thesis **TD-6/98**, Dipartimento di Informatica, Università di Pisa, March 1998.

MPC86. N.P.Mendler, P.Panangaden, R.L.Constable. Infinite Objects in Type Theory, 1th *LICS* Conf. Proc., IEEE Computer Society Press, 1986, 249–255.

Mil83. R.Milner. Calculi for synchrony and asynchrony, *TCS* **25**, 1983, 267–310.

Mil84. R.Milner. A complete inference system for a class of regular behaviours, *J. of Computer and System Sciences* **28**, 1984, 39–466.

Mos?. L.Moss. Coalgebraic Logic, to appear in the *Annals of Pure and Applied Logic*.

Pit96. A.M.Pitts. Relational Properties of Domains, *Inf. & Comp.* **127**, 1996.

Plo85. G.Plotkin. Types and Partial Functions, Post-Graduate Lecture Notes, Department of Computer Science, University of Edinburgh, 1985.

Rut96. J.J.M.M.Rutten. Universal coalgebra: a theory of systems, Report **CS-R9652**, CWI, Amsterdam, 1996.

RT93. J.J.M.M.Rutten, D.Turi. On the Foundations of Final Semantics: Non-Standard Sets, Metric Spaces, Partial Orders, *REX* Conf. Proc., J.de Bakker et al. eds., Springer LNCS **666**, 1993, 477–530.

Tal90. C.Talcott. A Theory for Program and Data type Specification, *TCS*, Disco90 Special Issue, 1990.

Tur96. D.Turi. *Functorial Operational Semantics and its Denotational Dual*, PhD thesis, CWI, 1996.

String Languages Generated by Total Deterministic Macro Tree Transducers*

Sebastian Maneth

Department of Computer Science, Leiden University, PO Box 9512,
2300 RA Leiden, The Netherlands, E-mail: maneth@wi.leidenuniv.nl

Abstract. The class of string languages obtained by taking the yields of output tree languages of total deterministic macro tree transducers (MTTs) is investigated. The first main result is that MTTs which are linear and nondeleting in the parameters generate the same class of string languages as total deterministic top-down tree transducers. The second main result is a so called "bridge theorem"; it can be used to show that there is a string language generated by a nondeterministic top-down tree transducer with monadic input, i.e., an ET0L language, which cannot be generated by an MTT. In fact, it is shown that this language cannot even be generated by the composition closure of MTTs; hence it is also not in the IO-hierarchy.

1 Introduction

Macro tree transducers [Eng80, CF82, EV85, EM98] are a well-known model of syntax-directed semantics (for a recent survey, see [FV98]). They are obtained by combining top-down tree transducers with macro grammars. In contrast to top-down tree transducers they have the ability to handle context information. This is done by parameters.

A total deterministic macro tree transducer (for short, MTT) M realizes a translation τ_M which is a function from trees to trees. The input trees may, for instance, be derivation trees of a context-free grammar which describes the syntax of some programming language (the source language). To every input tree s (viz. the derivation tree of a source program P) M associates the tree $\tau_M(s)$. This tree may then be interpreted in an appropriate semantic domain, e.g., yielding a program in another programming language (the target language): the semantics of P. One specific, quite popular, such domain is the one of strings with concatenation as only operation. More precisely, every symbol of rank greater than zero is interpreted as concatenation and constant symbols are interpreted as letters. The interpretation of a tree t in this domain is simply its *yield* (or frontier, i.e., the string obtained from t by reading its leaves from left to right). Thus, an MTT M can be seen as a translation device from trees to strings. Taking a tree language as input it generates a formal language as output. It

* This work was supported by the EC TMR Network GETGRATS.

W. Thomas (Ed.): FOSSACS'99, LNCS 1578, pp. 258-272, 1999.

is this class of formal languages (viz. the sets of target programs that can be generated) which we investigate in this paper.

An MTT M such that each right-hand side of a rule is linear and nondeleting in the parameters, that is, every parameter occurs exactly once, will be called *simple in the parameters*. This means that M cannot copy by means of its parameters. We prove that the class of string languages generated by such MTTs equals the class of string languages generated by top-down tree transducers. Hence the parameters can be eliminated. It is known that for unrestricted MTTs this is not the case; also if we consider output tree languages, MTTs that are simple in the parameters can do more than top-down tree transducers: they can generate tree languages that have non-regular path languages, which cannot be done by top-down tree transducers. For a more severe restriction, namely, the finite copying restriction, MTTs generate the same class of string languages as finite copying top-down tree transducers (Corollary 7.10 of [EM98]).

Now consider the case that we want to prove that a certain tree language R cannot be generated (as output tree language) by any MTT. In general this is difficult for there are very few appropriate tools: there exists a pumping lemma [Küh98] for a restricted case of MTTs. If we know that the string language obtained by taking the yields of the trees in R cannot be generated by any MTT, then we immediately know that R cannot be generated by an MTT. Since there are many tree languages with the same yield language, it is much stronger to know that a string language cannot be generated by an MTT than to know this for a tree language. We present a tool which is capable of proving that certain string languages L cannot be generated by an MTT. More precisely we will show that if L is of the form $f(L')$ for some fixed operation f, then L' can be generated by an MTT which is simple in the parameters; by our first result this means that L' can be generated by a top-down tree transducer. The proof is a direct generalization of Fischer's result on IO macro grammars: in the proof of Theorem 3.4.3 in [Fis68] it is proved that if $f(L)$ is an IO macro language then L can be generated by an IO macro grammar which is simple in the parameters. The result shows that the structure of L forces it from a bigger into a smaller class; it gives a "bridge" from the bigger (viz. unrestricted MTTs) into the smaller class (viz. MTTs which are simple in the parameters). For this smaller class, i.e., the class of string languages generated by top-down tree transducers, there exists another bridge theorem into yet another smaller class (using the same operation f), namely the class of string languages generated by finite copying top-down tree transducers [ERS80]. Due to the limited copying power of this class, it only contains languages that are of linear growth (they have the "Parikh property"); thus, languages like $L_{\exp} = \{a^{2^n} \mid a \geq 0\}$ are not in this class. Altogether we get that $f(f(L'))$, where L' is a non-Parikh language (e.g., $L_{\exp}$) cannot be generated by an MTT; in fact, we prove that it cannot be generated by any composition of MTTs.

This paper is structured as follows. In Section 2 we fix some notions used throughout the paper. Section 3 recalls macro tree transducers. In Section 4 we establish our two main results. Section 5 concludes with some open problems.

2 Preliminaries

The set $\{0, 1, \dots\}$ of natural numbers is denoted by $\mathbb{N}$. The empty set is denoted by $\varnothing$. For $k \in \mathbb{N}$, $[k]$ denotes the set $\{1, \dots, k\}$; thus $[0] = \varnothing$. For a set A, A^* is the set of all strings over A. The empty string is denoted by ε and the length of a string w is denoted $|w|$. For strings $v, w_1, \dots, w_n \in A^*$ and distinct $a_1, \dots, a_n \in A$, we denote by $v[a_1 \leftarrow w_1, \dots, a_n \leftarrow w_n]$ the result of (simultaneously) substituting w_i for every occurrence of a_i in v. Note that $[a_1 \leftarrow w_1, \dots, a_n \leftarrow w_n]$ is a homomorphism on strings. For a condition P on a and w we use, similar to set notation, $[a \leftarrow w \mid P]$ to denote the substitution $[L]$, where L is the list of all $a \leftarrow w$ for which condition P holds.

For functions $f\colon A \to B$ and $g\colon B \to C$ their composition is $(f \circ g)(x) = g(f(x))$; note that the order of f and g is nonstandard. For sets of functions F and G their composition is $F \circ G = \{f \circ g \mid f \in F, g \in G\}$.

2.1 Trees

A set Σ together with a mapping $\mathrm{rank}_\Sigma\colon \Sigma \to \mathbb{N}$ is called a *ranked* set. For $k \in \mathbb{N}$, $\Sigma^{(k)}$ denotes the set $\{\sigma \in \Sigma \mid \mathrm{rank}_\Sigma(\sigma) = k\}$. We will often write $\sigma^{(k)}$ to indicate that $\mathrm{rank}_\Sigma(\sigma) = k$.

The *set of trees over* Σ, denoted by T_Σ, is the smallest set of strings $T \subseteq (\Sigma \cup \{(,), ,\})^*$ such that if $\sigma \in \Sigma^{(k)}$, $k \geq 0$, and $t_1, \dots, t_k \in T$, then $\sigma(t_1, \dots, t_k) \in T$. For $\alpha \in \Sigma^{(0)}$ we denote the tree $\alpha()$ also by α. For a set A, $T_\Sigma(A)$ denotes the set $T_{\Sigma \cup A}$, where every symbol of A has rank 0 and $\langle \Sigma, A \rangle$ denotes the ranked set $\{\langle \sigma, a \rangle^{(k)} \mid \sigma \in \Sigma^{(k)}, a \in A\}$ (if Σ is unranked, then every symbol in $\langle \Sigma, A \rangle$ is of rank zero). We fix the set of *variables* X as $\{x_1, x_2, \dots\}$ and the set of *parameters* Y as $\{y_1, y_2, \dots\}$. For $k \in \mathbb{N}$, X_k and Y_k denote the sets $\{x_1, \dots, x_k\}$ and $\{y_1, \dots, y_k\}$, respectively.

For a tree t, the string obtained by reading the labels of its leaves from left to right, called the *yield of* t, is denoted by yt. The special symbol e of rank zero will be used to denote the empty string ε (e.g., $y(\sigma(a, e)) = a$ and $ye = \varepsilon$). For a string $w = a_1 \cdots a_n$ and a binary symbol b let $\mathrm{comb}_b(w)$ denote the tree $b(a_1, b(a_2, \dots b(a_n, e) \dots))$ over $\{b^{(2)}, a_1^{(0)}, \dots, a_n^{(0)}\}$; note that $y\mathrm{comb}_b(w) = w$.

A subset L of T_Σ is called a *tree language*. The class of all *regular* (or, *recognizable*) tree languages is denoted by $REGT$ (cf., e.g., [GS97]). For a tree language L we denote by yL the string language $\{yt \mid t \in L\}$ and for a class of tree languages $\mathcal{L}$ we denote by $y\mathcal{L}$ the class of string languages $\{yL \mid L \in \mathcal{L}\}$. A relation $\tau \subseteq T_\Sigma \times T_\Delta$ is called a *tree translation* or simply translation; by $y\tau$ we denote $\{(s, yt) \mid (s, t) \in \tau\}$. For a tree language $L \subseteq T_\Sigma$, $\tau(L)$ denotes the set $\{t \in T_\Delta \mid (s, t) \in \tau \text{ for some } s \in L\}$. For a class $\mathcal{T}$ of tree translations and a class $\mathcal{L}$ of tree languages, $\mathcal{T}(\mathcal{L})$ denotes the class of tree languages $\{\tau(L) \mid \tau \in \mathcal{T}, L \in \mathcal{L}\}$ and $y\mathcal{T}$ denotes $\{y\tau \mid \tau \in \mathcal{T}\}$.

2.2 Tree Substitution and Relabelings

Note that trees are particular strings and that string substitution as defined in the beginning of this section is applicable to a tree to replace symbols of rank

zero; we refer to this type of substitution as "first order tree substitution".

Let Σ be a ranked set and let $\sigma_1, \ldots, \sigma_n$ be distinct elements of Σ, $n \geq 1$, and for each $i \in [n]$ let s_i be a tree in $T_\Sigma(Y_k)$, where $k = \mathrm{rank}_\Sigma(\sigma_i)$. For $t \in T_\Sigma$, the *second order substitution of s_i for σ_i in t*, denoted by $t[\![\sigma_1 \leftarrow s_1, \ldots, \sigma_n \leftarrow s_n]\!]$ is inductively defined as follows (abbreviating $[\![\sigma_1 \leftarrow s_1, \ldots, \sigma_n \leftarrow s_n]\!]$ by $[\![\ldots]\!]$). For $t = \sigma(t_1, \ldots, t_k)$ with $\sigma \in \Sigma^{(k)}$, $k \geq 0$, and $t_1, \ldots, t_k \in T_\Sigma$, (i) if $\sigma = \sigma_i$ for an $i \in [n]$, then $t[\![\ldots]\!] = s_i[y_j \leftarrow t_j[\![\ldots]\!] \mid j \in [k]]$ and (ii) otherwise $t[\![\ldots]\!] = \sigma(t_1[\![\ldots]\!], \ldots, t_k[\![\ldots]\!])$. For a condition P on σ and s, we use $[\![\sigma \leftarrow s \mid P]\!]$ to denote the substitution $[\![L]\!]$, where L is the list of all $\sigma \leftarrow s$ for which condition P holds.

A (deterministic) *finite state relabeling* M is a tuple $(Q, \Sigma, \Delta, F, R)$, where Q is a finite set of *states*, Σ and Δ are ranked alphabets of *input* and *output symbols*, respectively, $F \subseteq Q$ is a set of *final states*, and R is a finite set of rules such that for every $\sigma \in \Sigma^{(k)}$, $k \geq 0$, and $q_1, \ldots, q_k \in Q$, there is exactly one rule of the form $\sigma(\langle q_1, x_1\rangle, \ldots, \langle q_k, x_k\rangle) \to \langle q, \delta(x_1, \ldots, x_k)\rangle$ in R, where $q \in Q$ and $\delta \in \Delta^{(k)}$. The rules of M are used as term rewriting rules, and the rewrite relation induced by M (on $T_{\langle Q, T_\Delta\rangle \cup \Sigma}$) is denoted by $\Rightarrow_M$. The translation realized by M is $\tau_M = \{(s,t) \in T_\Sigma \times T_\Delta \mid s \Rightarrow_M^* \langle q, t\rangle, q \in F\}$. The class of all translations that can be realized by finite state relabelings is denoted by $DQRELAB$.

3 Macro Tree Transducers

A macro tree transducer is a syntax-directed translation device in which the translation of an input subtree may depend on its context. The context information is processed by parameters. We will consider total deterministic macro tree transducers only.

Definition 1. A *macro tree transducer* (for short, MTT) is a tuple $M = (Q, \Sigma, \Delta, q_0, R)$, where Q is a ranked alphabet of *states*, Σ and Δ are ranked alphabets of *input* and *output symbols*, respectively, $q_0 \in Q^{(0)}$ is the *initial state*, and R is a finite set of *rules*; for every $q \in Q^{(m)}$ and $\sigma \in \Sigma^{(k)}$ with $m, k \geq 0$ there is exactly one rule of the form $\langle q, \sigma(x_1, \ldots, x_k)\rangle(y_1, \ldots, y_m) \to \zeta$ in R, where $\zeta \in T_{\langle Q, X_k\rangle \cup \Delta}(Y_m)$.

A rule of the form $\langle q, \sigma(x_1, \ldots, x_k)\rangle(y_1, \ldots, y_m) \to \zeta$ is called the (q, σ)-rule and its right-hand side ζ is denoted by $\mathrm{rhs}_M(q, \sigma)$; it is also called a q-rule.

The rules of M are used as term rewriting rules and by $\Rightarrow_M$ we denote the *derivation relation induced by M* (on $T_{\langle Q, T_\Sigma\rangle \cup \Delta}(Y)$). The *translation realized by M*, denoted by τ_M is the total function $\{(s,t) \in T_\Sigma \times T_\Delta \mid \langle q_0, s\rangle \Rightarrow_M^* t\}$. The class of all translations that can be realized by MTTs is denoted by MTT. If for every $\sigma \in \Sigma$, $q \in Q^{(m)}$, $m \geq 0$, and $j \in [m]$, y_j occurs *exactly once* in $\mathrm{rhs}_M(q, \sigma)$ (i.e., the rules of M are linear and nondeleting in Y_m), then M is *simple in the parameters* (for short *sp*; we say, M is an $\mathrm{MTT_{sp}}$). The class of all translations that can be realized by $\mathrm{MTT_{sp}}$s is denoted by MTT_{sp}. If all states of an MTT are of rank zero, then M is called *top-down tree transducer*. The class of translations realized by top-down tree transducers is denoted by T. For top-down

tree transducers we also consider the case that for a state q and an input symbol σ there may be more than one rule of the form $\langle q, \sigma(x_1, \ldots, x_k)\rangle \to \zeta$ in R. Such a top-down tree transducer is called *nondeterministic* and the corresponding class of translations is denoted by $N\text{-}T$ (note that this is a class of relations rather than total functions). The class of translations realized by nondeterministic top-down tree transducer with monadic input (i.e., each input symbol is of rank 0 or 1) is denoted by $N\text{-}T_{\text{mon}}$.

Let us now consider an example of an MTT.

Example 1. Let $M = (Q, \Sigma, \Sigma, q_0, R)$ be the MTT_{sp} with $Q = \{q^{(2)}, q_0^{(0)}\}$, $\Sigma = \{\sigma^{(2)}, a^{(0)}, b^{(0)}\}$, and R consisting of the following rules.

$$\begin{array}{ll}
\langle q_0, \sigma(x_1, x_2)\rangle & \to \langle q, x_2\rangle(\langle q_0, x_1\rangle, \langle q_0, x_1\rangle) \\
\langle q, \sigma(x_1, x_2)\rangle(y_1, y_2) & \to \langle q, x_2\rangle(\sigma(y_1, \langle q_0, x_1\rangle), \sigma(\langle q_0, x_1\rangle, y_2)) \\
\langle q_0, a\rangle & \to a \\
\langle q, a\rangle(y_1, y_2) & \to \sigma(y_1, y_2) \\
\langle q_0, b\rangle & \to b \\
\langle q, b\rangle(y_1, y_2) & \to \sigma(y_2, y_1)
\end{array}$$

Consider the input tree $t = \sigma(a, \sigma(b, \sigma(b, b)))$. Then a derivation by M looks as follows.

$$\begin{array}{ll}
\langle q_0, t\rangle & \Rightarrow_M \langle q, \sigma(b, \sigma(b, b))\rangle(\langle q_0, a\rangle, \langle q_0, a\rangle) \\
& \Rightarrow_M^* \langle q, \sigma(b, \sigma(b, b))\rangle(a, a) \\
& \Rightarrow_M \langle q, \sigma(b, b)\rangle(\sigma(a, \langle q_0, b\rangle), \sigma(\langle q_0, b\rangle, a)) \\
& \Rightarrow_M^* \langle q, \sigma(b, b)\rangle(\sigma(a, b), \sigma(b, a)) \\
& \Rightarrow_M^* \langle q, b\rangle(\sigma(\sigma(a, b), b), \sigma(b, \sigma(b, a))) \\
& \Rightarrow_M \sigma(\sigma(b, \sigma(b, a)), \sigma(\sigma(a, b), b))
\end{array}$$

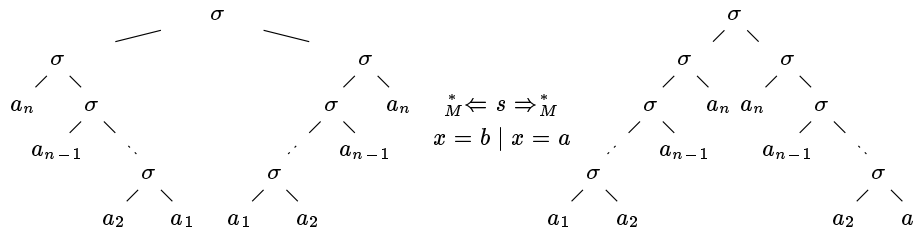


Fig. 1. Translations of M with input s for $x = b$ and $x = a$

In Fig. 1 it is shown how the translations for trees of the form

$$s = \sigma(a_1, \sigma(a_2, \ldots \sigma(a_n, x)\ldots))$$

with $a_1, \ldots, a_n \in \Sigma^{(0)}$ and $n \geq 1$ look like. If $x = a$ then $y\tau_M(s) = ww^r$ and if $x = b$ then $y\tau_M(s) = w^r w$, where $w = a_1 \cdots a_n$ and w^r denotes the reverse of w (i.e., the string $a_n a_{n-1} \cdots a_1$). Note that M is sp because both y_1 and y_2 appear exactly once in the right-hand side of each q-rule of M. □

The next lemma will be used in proofs by induction on the structure of the input tree. Let $M = (Q, \Sigma, \Delta, q_0, R)$ be an MTT. For every $q \in Q^{(m)}$ and $s \in T_\Sigma$ let the *q-translation of s*, denoted by $M_q(s)$, be the unique tree $t \in T_\Delta(Y_m)$ such that $\langle q, s\rangle(y_1, \ldots, y_m) \Rightarrow_M^* t$. Note that, for $s \in T_\Sigma$, $\tau_M(s) = M_{q_0}(s)$. The q-translations of trees in T_Σ can be characterized inductively as follows.

Lemma 2. (cf. Definition 3.18 of [EV85]) *Let $M = (Q, \Sigma, \Delta, q_0, R)$ be an* MTT. *For every $q \in Q$, $\sigma \in \Sigma^{(k)}$, $k \geq 0$, and $s_1, \ldots, s_k \in T_\Sigma$, $M_q(\sigma(s_1, \ldots, s_k)) = \mathrm{rhs}_M(q, \sigma)[\![\langle q', x_i\rangle \leftarrow M_{q'}(s_i) \mid \langle q', x_i\rangle \in \langle Q, X_k\rangle]\!]$.*

4 String Languages Generated by MTT

To prove our first main result we need the following small lemma about second order tree substitution. It says that if we are considering the yield of a tree to which a second order tree substitution is applied, then inside the substitution merely the yields of the trees that are substituted are relevant.

Lemma 3. *Let Σ be a ranked alphabet, $\gamma_1, \ldots, \gamma_n \in \Sigma$, and $t, s_1, s_1', \ldots, s_n, s_n' \in T_\Sigma(Y)$. If $ys_i = ys_i'$ for every $i \in [n]$, then*

$$y(t[\![\gamma_1 \leftarrow s_1, \ldots, \gamma_n \leftarrow s_n]\!]) = y(t[\![\gamma_1 \leftarrow s_1', \ldots, \gamma_n \leftarrow s_n']\!]).$$

Lemma 3 can be proved by straightforward induction on t. We now show how to generate by a top-down tree transducer the string language generated by an $\mathrm{MTT}_{\mathrm{sp}}$.

Lemma 4. $yMTT_{\mathrm{sp}} \subseteq y(DQRELAB \circ T)$.

Proof. Let $M = (Q, \Sigma, \Delta, q_0, R)$ be an $\mathrm{MTT}_{\mathrm{sp}}$. We will construct a finite state relabeling N and a top-down tree transducer M' such that for every $s \in T_\Sigma$, $y(\tau_{M'}(\tau_N(s))) = y\tau_M(s)$. The idea is as follows. Let $q \in Q^{(m)}$ and $s \in T_\Sigma$. Then, since M is sp, $yM_q(s)$ is of the form

$$w = w_0 y_{j_1} w_1 y_{j_2} w_2 \cdots y_{j_m} w_m,$$

where $j_1, \ldots, j_m \in [m]$ are pairwise different and $w_0, \ldots, w_m \in (\Delta^{(0)})^*$. For a string of the form w (where the w_i are arbitrary strings not containing parameters) and for $0 \leq \nu \leq m$ we denote by $\mathrm{part}_\nu(w)$ the string w_ν. For every w_ν the top-down tree transducer M' has a state (q, ν) which computes w_ν. The information on the order of the parameters, i.e., the indices $j_1, \ldots, j_m$, will be determined by the finite state relabeling N in such a way that $\sigma \in \Sigma^{(k)}$ is relabeled by $(\sigma, (\mathrm{pos}_1, \ldots, \mathrm{pos}_k))$, where for each $i \in [k]$, pos_i is a mapping associating with every $q \in Q^{(m)}$ a bijection from $[m]$ to $[m]$. For instance, if s_i equals the tree s from above, then the σ in $\sigma(s_1, \ldots, s_i, \ldots, s_k)$ is relabeled by $(\sigma, (\mathrm{pos}_1, \ldots, \mathrm{pos}_k))$ and $\mathrm{pos}_i(q)(\nu) = j_\nu$ for $\nu \in [m]$. Formally, $N = (Q_N, \Sigma, \Gamma, Q_N, R_N)$, where

- Q_N is the set of all mappings pos which associate with every $q \in Q^{(m)}$ a bijection $\mathrm{pos}(q)$ from $[m]$ to $[m]$. For convenience we identify $\mathrm{pos}(q)$ with the string $j_1 \cdots j_m$ over $[m]$, where $\mathrm{pos}(q)(i) = j_i$ for $i \in [m]$.

- $\Gamma = \{(\sigma, (\mathrm{pos}_1, \ldots, \mathrm{pos}_k))^{(k)} \mid \sigma \in \Sigma^{(k)}, k \geq 0, \mathrm{pos}_1, \ldots, \mathrm{pos}_k \in Q_N\}$.
- For every $\sigma \in \Sigma^{(k)}$, $k \geq 0$, and $\mathrm{pos}_1, \ldots, \mathrm{pos}_k \in Q_N$ let

$$\sigma(\langle \mathrm{pos}_1, x_1\rangle, \ldots, \langle \mathrm{pos}_k, x_k\rangle) \to \langle \mathrm{pos}, (\sigma, (\mathrm{pos}_1, \ldots, \mathrm{pos}_k))(x_1, \ldots, x_k)\rangle$$

be in R_N, where for every $q \in Q^{(m)}$, $\mathrm{pos}(q) = \mathrm{order}(\mathrm{rhs}_M(q, \sigma))$ and for $t \in T_{\langle Q, X_k\rangle \cup \Delta}(Y_m)$, $\mathrm{order}(t)$ is the string over $[m]$ defined recursively as follows: if $t = y_j \in Y_m$, then $\mathrm{order}(t) = j$, if $t = \delta(t_1, \ldots, t_l)$ with $\delta \in \Delta^{(l)}$, $l \geq 0$, and $t_1, \ldots, t_l \in T_{\langle Q, X_k\rangle \cup \Delta}(Y_m)$, then $\mathrm{order}(t) = \mathrm{order}(t_1) \cdots \mathrm{order}(t_l)$, and if $t = \langle q', x_i\rangle(t_1, \ldots, t_l)$ with $\langle q', x_i\rangle \in \langle Q, X_k\rangle^{(l)}$, $l \geq 0$, and $t_1, \ldots, t_l \in T_{\langle Q, X_k\rangle \cup \Delta}(Y_m)$, then $\mathrm{order}(t) = \mathrm{order}(t_{\mathrm{pos}_i(q')(1)}) \cdots \mathrm{order}(t_{\mathrm{pos}_i(q')(l)})$.

It is straightforward to show (by induction on the structure of s) that N is defined in such a way that if $\tau_N(\sigma(s_1, \ldots, s_k)) = (\sigma, (\mathrm{pos}_1, \ldots, \mathrm{pos}_k))(\tilde{s}_1, \ldots, \tilde{s}_k)$, then for every $i \in [k]$ and $q \in Q^{(m)}$,

$$yM_q(s_i) = w_0 y_{\mathrm{pos}_i(q)(1)} w_1 y_{\mathrm{pos}_i(q)(2)} w_2 \cdots y_{\mathrm{pos}_i(q)(m)} w_m,$$

for some $w_0, \ldots, w_m \in (\Delta^{(0)})^*$. In the induction step it can be shown that for $t \in T_{\langle Q, X_k\rangle \cup \Delta}(Y_m)$, $\mathrm{order}(t) = j_1 \cdots j_m$, where $j_1, \ldots, j_m \in [m]$, $yt[\![\ldots]\!] = w_0 y_{j_1} w_1 y_{j_2} w_2 \cdots y_{j_m} w_m$ for some $w_0, \ldots, w_m \in (\Delta^{(0)})^*$, and $[\![\ldots]\!] = [\![\langle q', x_i\rangle \leftarrow M_{q'}(s_i) \mid \langle q', x_i\rangle \in \langle Q, X_k\rangle]\!]$.

We now define the top-down tree transducer $M' = (Q', \Gamma, \Delta', (q_0, 0), R')$, where

- $Q' = \{(q, \nu)^{(0)} \mid q \in Q^{(m)}, 0 \leq \nu \leq m\}$,
- $\Delta' = \Delta^{(0)} \cup \{b^{(2)}, e^{(0)}\}$, where $e \notin \Delta$, and
- for every $(q, \nu) \in Q'$, $(\sigma, (\mathrm{pos}_1, \ldots, \mathrm{pos}_k)) \in \Gamma^{(k)}$, and $k \geq 0$ let the rule

$$\langle (q, \nu), (\sigma, (\mathrm{pos}_1, \ldots, \mathrm{pos}_k))(x_1, \ldots, x_k)\rangle \to \zeta$$

be in R', where $\zeta = \mathrm{comb}_b(\mathrm{part}_\nu(y(\xi[\![\_]\!])))$, $\xi = \mathrm{rhs}_M(q, \sigma)$, and $[\![\_]\!]$ is the substitution

$$[\![\langle q', x_i\rangle \leftarrow \mathrm{comb}_b(\langle (q', 0), x_i\rangle y_{\mathrm{pos}_i(q')(1)} \langle (q', 1), x_i\rangle y_{\mathrm{pos}_i(q')(2)} \cdots y_{\mathrm{pos}_i(q')(m)} \langle (q', m), x_i\rangle) \mid \langle q', x_i\rangle \in \langle Q, X_k\rangle^{(m)}]\!].$$

We now prove the correctness of M', i.e., that for every $s \in T_\Sigma$, $y(\tau_{M'}(\tau_N(s))) = y\tau_M(s)$. It follows from the next claim by taking $(q, \nu) = (q_0, 0)$.

Claim: For every $(q, \nu) \in Q'$ and $s \in T_\Sigma$, $y(M'_{(q,\nu)}(\tau_N(s))) = \mathrm{part}_\nu(yM_q(s))$.

The proof of this claim is done by induction on the structure of s. Let $s = \sigma(s_1, \ldots, s_k)$, $\sigma \in \Sigma^{(k)}$, $k \geq 0$, and $s_1, \ldots, s_k \in T_\Sigma$. Then $y(M'_{(q,\nu)}(\tau_N(s))) = y(M'_{(q,\nu)}((\sigma, (\mathrm{pos}_1, \ldots, \mathrm{pos}_k))(\tilde{s}_1, \ldots, \tilde{s}_k)))$, where $\tilde{s}_i = \tau_N(s_i)$ for all $i \in [k]$. This equals $y(\zeta[\ldots])$, where $\zeta = \mathrm{rhs}_{M'}((q, \nu), (\sigma, (\mathrm{pos}_1, \ldots, \mathrm{pos}_k)))$ and $[\ldots] = [\langle (q', \nu'), x_i\rangle \leftarrow M'_{(q',\nu')}(\tau_N(s_i)) \mid \langle (q', \nu'), x_i\rangle \in \langle Q', X_k\rangle]$. By the definition of the rules of M', $\zeta = \mathrm{comb}_b(\mathrm{part}_\nu(y(\xi[\![\_]\!])))$, where $\xi = \mathrm{rhs}_M(q, \sigma)$ and $[\![\_]\!]$ is as above. By applying y (yield) and the induction hypothesis we get $\mathrm{part}_\nu(y(\xi[\![\_]\!]))\Psi$,

where Ψ is the string substitution $[\langle(q',\nu'),x_i\rangle \leftarrow \mathrm{part}_{\nu'}(yM_{q'}(s_i)) \mid \langle(q',\nu'),x_i\rangle \in \langle Q', X_k\rangle]$. Since Ψ does not change parameters, we can move it inside the application of part_ν to get $\mathrm{part}_\nu(y(\xi[\![\_]\!])\Psi)$. If we move Ψ inside the application of y (yield) we get $\mathrm{part}_\nu(y(\xi[\![\_]\!]\Psi'))$, where Ψ' denotes the first order tree substitution of replacing $\langle(q',\nu'),x_i\rangle$ of rank zero by a tree with $\mathrm{part}_{\nu'}(yM_{q'}(s_i))$ as yield. Applying Ψ' inside of $[\![\_]\!]$ amounts to replacing $\langle q', x_i\rangle$ by a tree with yield $w = \mathrm{part}_0(yM_{q'}(s_i))y_{\mathrm{pos}_i(q')(1)} \cdots \mathrm{part}_m(yM_{q'}(s_i))$. By the correctness of the finite state relabeling N, $w = yM_{q'}(s_i)$. Since, by Lemma 3, we can put any tree with yield w in the second order substitution, taking $M_{q'}(s_i)$ we get $\mathrm{part}_\nu(y(\xi[\![\ldots]\!]))$ with $[\![\ldots]\!] = [\![\langle q', x_i\rangle \leftarrow M_{q'}(s_i) \mid \langle q', x_i\rangle \in \langle Q, X_k\rangle]\!]$. By Lemma 2 this is equal to $\mathrm{part}_\nu(yM_q(s))$ which ends the proof of the claim. □

Let us look at an example of an application of the construction in the proof of Lemma 4.

Example 2. Let M be the $\mathrm{MTT}_{\mathrm{sp}}$ of Example 1. We construct the finite state relabeling N and the top-down tree transducer M' following the construction in the proof of Lemma 4. Let $N = (Q_N, \Sigma, \Gamma, Q_N, R_N)$ be the finite state relabeling with $Q_N = \{q_{12}, q_{21}\}$, $q_{12} = \{(q_0,\varepsilon),(q,12)\}$, $q_{21} = \{(q_0,\varepsilon),(q,21)\}$, and $\Gamma = \{(\sigma,(q_{12},q_{12}))^{(2)}, (\sigma,(q_{12},q_{21}))^{(2)}, (\sigma,(q_{21},q_{12}))^{(2)}, (\sigma,(q_{21},q_{21}))^{(2)}, (a,())^{(0)}, (b,())^{(0)}\}$. The set R_N of rules of N consists of the rules

$$\begin{aligned} a &\to \langle q_{12},(a,())\rangle \\ b &\to \langle q_{21},(b,())\rangle \\ \sigma(\langle r,x_1\rangle,\langle r',x_2\rangle) &\to \langle r',(\sigma,(r,r'))(x_1,x_2)\rangle \qquad \text{for all } r,r' \in Q_N. \end{aligned}$$

Consider the tree $t = \sigma(a,\sigma(b,\sigma(b,b)))$ again. Then $\tau_N(t)$ equals

$$(\sigma,(q_{12},q_{21}))((a,()),(\sigma,(q_{21},q_{21}))((b,()),(\sigma,(q_{21},q_{21}))((b,()),(b,()))))). \qquad (*)$$

We now construct the top-down tree transducer M'. Let $M' = (Q', \Gamma, \Delta', (q_0,0), R')$ with $Q' = \{(q_0,0)^{(0)},(q,0)^{(0)},(q,1)^{(0)},(q,2)^{(0)}\}$ and $\Delta' = \Sigma^{(0)} \cup \{b^{(2)}, e^{(0)}\}$. For simplicity we write down the rules of M' as tree-to-string rules, i.e., we merely show the yield of the corresponding right-hand side. Let us consider in detail how to obtain the right-hand sides of the $((q,\nu),(\sigma,(r,q_{21})))$-rules for $0 \le \nu \le 2$ and $r \in Q_N$. Since we are only interested in the yields, we have to consider the string $v = y(\mathrm{rhs}_M(q,\sigma)[\![\_]\!])$, where $[\![\_]\!]$ is defined as in the proof of Lemma 4. This string equals

$$\underbrace{\langle(q,0),x_2\rangle\langle(q_0,0),x_1\rangle}_{\mathrm{part}_0(v)}\, y_2\, \underbrace{\langle(q,1),x_2\rangle}_{\mathrm{part}_1(v)}\, y_1\, \underbrace{\langle(q_0,0),x_1\rangle\langle(q,2),x_2\rangle}_{\mathrm{part}_2(v)}.$$

Hence, for every $r \in Q_N$ and $0 \le \nu \le 2$, $y\mathrm{rhs}_{M'}((q,\nu),(\sigma,(r,q_{21}))) = \mathrm{part}_\nu(v)$; similarly we get $y\mathrm{rhs}_{M'}((q,0),(\sigma,(r,q_{12}))) = \langle(q,0),x_2\rangle$,
$y\mathrm{rhs}_{M'}((q,1),(\sigma,(r,q_{12}))) = \langle(q_0,0),x_1\rangle\langle(q,1),x_2\rangle\langle(q_0,0),x_1\rangle$,
$y\mathrm{rhs}_{M'}((q,2),(\sigma,(r,q_{12}))) = \langle(q,2),x_2\rangle$.
The remaining rules are, for $0 \le \nu \le 2$ and $r,r' \in Q_N$,

$$\langle (q_0,0),(\sigma,(r,r'))(x_1,x_2)\rangle \to \langle (q,0),x_2\rangle\langle (q_0,0),x_1\rangle\langle (q,1),x_2\rangle \langle (q_0,0),x_1\rangle\langle (q,2),x_2\rangle$$
$$\langle (q_0,0),(a,())\rangle \to a$$
$$\langle (q_0,0),(b,())\rangle \to b$$
$$\langle (q,\nu),(a,())\rangle \to \varepsilon$$
$$\langle (q,\nu),(b,())\rangle \to \varepsilon$$

Finally, consider the derivation by M' with input tree $t' = \tau_N(t)$ (shown in $(*)$). Denote by $t'/2$ the tree $\tau_N(\sigma(b,\sigma(b,b)))$ and by $t'/22$ the tree $\tau_N(\sigma(b,b))$. Again we merely show the corresponding yields.

$$\begin{aligned}
&\langle (q_0,0),t'\rangle\\
\Rightarrow_{M'}\ &\langle (q,0),t'/2\rangle\langle (q_0,0),(a,())\rangle\langle (q,1),t'/2\rangle\langle (q_0,0),(a,())\rangle\langle (q,2),t'/2\rangle\\
\Rightarrow^*_{M'}\ &\langle (q,0),t'/22\rangle\langle (q_0,0),(b,())\rangle\, a\,\langle (q,1),t'/22\rangle\, a\,\langle (q_0,0),(b,())\rangle\langle (q,2),t'/22\rangle\\
\Rightarrow^*_{M'}\ &\langle (q,0),(b,())\rangle\, bba\,\langle (q,1),(b,())\rangle\, abb\,\langle (q,2),(b,())\rangle\\
\Rightarrow^*_{M'}\ &bbaabb. \qquad \square
\end{aligned}$$

From Lemma 4 we obtain our first main result: $\mathrm{MTT}_{\mathrm{sp}}$s and top-down tree transducers generate the same class of string languages if they take as input a class of tree languages that is closed under finite state relabelings.

Theorem 5. *Let $\mathcal{L}$ be a class of tree languages that is closed under finite state relabelings. Then $yMTT_{\mathrm{sp}}(\mathcal{L}) = yT(\mathcal{L})$.*

Proof. By Lemma 4, $yMTT_{\mathrm{sp}}(\mathcal{L}) \subseteq yT(\mathcal{L})$ and since every top-down tree transducer is an $\mathrm{MTT}_{\mathrm{sp}}$, $yT(\mathcal{L}) \subseteq yMTT_{\mathrm{sp}}(\mathcal{L})$. $\square$

Since the class $REGT$ of regular tree languages is closed under finite state relabelings (cf. Proposition 20.2 of [GS97]), we get $yMTT_{\mathrm{sp}}(REGT) = yT(REGT)$ from Theorem 5. For top-down tree transducers it is known (Theorem 3.2.1 of [ERS80] and Theorem 4.3 of [Man98b]) that $T(REGT)$ is equal to the class $OUT(T)$ of output tree languages of top-down tree transducers (i.e., taking the particular regular tree language T_Σ as input). In fact, it is shown in [Man98b] that for any class Ψ of tree translations which is closed under left composition with "semi-relabelings", which are particular linear top-down tree translations, $\Psi(REGT) = OUT(\Psi)$. Since it can be shown that MTT_{sp} is closed under left composition with top-down tree translations we get that $yOUT(MTT_{\mathrm{sp}}) = yOUT(T)$, i.e., $\mathrm{MTT}_{\mathrm{sp}}$s and top-down tree transducers generate the same class of string languages. If we consider $\mathrm{MTT}_{\mathrm{sp}}$s with monadic output alphabet, then the class of path languages generated by them taking regular tree languages as input is also equal to $yT(REGT)$ (cf. the proof of Lemma 7.6 of [EM98]). Thus, the classes of path and string languages generated by $\mathrm{MTT}_{\mathrm{sp}}$s are equal.

We now move to our second main result. First we define the operation $\mathrm{rub}_{b_1,\ldots,b_n}$ which inserts the symbols $b_1,\ldots,b_n$ ("*rub*bish") anywhere in the strings of the language to which it is applied. Let A be an alphabet, $L \subseteq A^*$ a language, and $b_1,\ldots,b_n$ new symbols not in A. Then $\mathrm{rub}_{b_1,\ldots,b_n}(L)$ denotes the language

$$\{w_1a_1w_2a_2\cdots w_ka_kw_{k+1} \mid a_1\cdots a_k \in L, k \geq 1, w_1,\ldots,w_{k+1} \in \{b_1,\ldots,b_n\}^*\}.$$

The following theorem shows that if an MTT M generates $\mathrm{rub}_0(L)$ (where $\mathrm{rub}_0 = \mathrm{rub}_{b_1,\ldots,b_n}$ for $n = 1$ and $b_1 = 0$) then, due to the nondeterminism inherent in rub_0, M cannot make use of its copying facility.

Theorem 6. *Let $\mathcal{L}$ be a class of tree languages which is closed under finite state relabelings and under intersection with regular tree languages, and let $L \subseteq A^*$. If $\mathrm{rub}_0(L) \in yMTT(\mathcal{L})$ then $L \in yMTT_{\mathrm{sp}}(\mathcal{L})$.*

Proof. Let $M = (Q, \Sigma, \Delta, q_0, R)$ be an MTT and $K \in \mathcal{L}$ such that $y\tau_M(K) = \mathrm{rub}_0(L)$ and $\Delta^{(0)} = A \cup \{0\}$. By Lemma 6.6 of [EM98] we may assume that M is nondeleting, i.e., for every $q \in Q^{(m)}$ and $j \in [m]$, y_j appears at least once in the right-hand side of each q-rule. Consider a string of the form

$$a_1 0^{n_1} a_2 0^{n_2} \cdots a_l 0^{n_l} a_{l+1}$$

with $l \geq 0$, $a_1, \ldots, a_{l+1} \in A$, and all $n_1, \ldots, n_l \geq 0$ pairwise different. We call such a string δ-string. Clearly, it is sufficient to consider only δ-strings in order to generate a tree language R with $yR = L$ (if we can construct an $\mathrm{MTT}_{\mathrm{sp}}$ which generates as yield language at least all δ-strings in $\mathrm{rub}_0(L)$, then by deletion of 0s we obtain an $\mathrm{MTT}_{\mathrm{sp}}$ which generates L as yield language). Consider the right-hand side of a rule of M in which some parameter y_j occurs more than once. If, during the derivation of a tree which has as yield a δ-string, this rule was applied, then the tree which is substituted for y_j in this derivation contains at most one symbol in A. Because otherwise, due to copying, the resulting string would not be a δ-string. Hence, when deriving a δ-string, a rule which contains multiple occurrences of a parameter y_j is only applicable if the yield of the tree being substituted for y_j contains at most one symbol in A. Based on this fact we can construct the $\mathrm{MTT}_{\mathrm{sp}}$ M' which generates L. The information whether the yield of the tree which will be substituted for a certain parameter contains none, one, or more than one occurrences of a symbol in A is determined by relabeling the input tree. Then this information is kept in the states of M'. More precisely, we will define a finite state relabeling N which relabels $\sigma \in \Sigma^{(k)}$ in the tree $\sigma(s_1, \ldots, s_k)$ by $(\sigma, (\phi_1, \ldots, \phi_k))$, where for every $i \in [k]$ and $q \in Q$,

$$\phi_i(q) = \begin{cases} e & \text{if } yM_q(s_i) \text{ contains no symbol in } A \\ a & \text{if } yM_q(s_i) = waw' \text{ with } w, w' \in (Y \cup \{0\})^* \\ dd & \text{otherwise,} \end{cases}$$

where $a \in A$ and d is an arbitrary symbol in A. Before we define N, let us define an auxiliary notion. For $w \in (\Delta^{(0)} \cup Y)^*$ let $\mathrm{oc}(w)$ be defined as follows. If $w \in (Y \cup \{0\})^*$, then $\mathrm{oc}(w) = e$; if $w = w_1 a w_2$ with $a \in A$ and $w_1, w_2 \in (Y \cup \{0\})^*$, then $\mathrm{oc}(w) = a$; and otherwise $\mathrm{oc}(w) = dd$.

Let $N = (Q_N, \Sigma, \Gamma, Q_N, R_N)$ be the finite state relabeling with

- $Q_N = \{\phi \mid \phi : Q \to (\{e, dd\} \cup A)\}$,
- $\Gamma = \{(\sigma, (\phi_1, \ldots, \phi_k))^{(k)} \mid \sigma \in \Sigma^{(k)}, k \geq 0, \phi_1, \ldots, \phi_k \in Q_N\}$, and

– R_N containing for every $\phi_1, \ldots, \phi_k \in Q_N$ and $\sigma \in \Sigma^{(k)}$ with $k \geq 0$ the rule

$$\sigma(\langle \phi_1, x_1 \rangle, \ldots, \langle \phi_k, x_k \rangle) \to \langle \phi, (\sigma, (\phi_1, \ldots, \phi_k))(x_1, \ldots, x_k) \rangle,$$

where for every $q \in Q$, $\phi(q) = \mathrm{oc}(y(\mathrm{rhs}_M(q, \sigma)\Theta))$ and Θ denotes the second order substitution (where b is an arbitrary binary symbol)

$$[\![\langle q', x_i \rangle \leftarrow \mathrm{comb}_b(\phi_i(q') y_1 \cdots y_m) \mid \langle q', x_i \rangle \in \langle Q, X_k \rangle^{(m)}, m \geq 0]\!].$$

It should be clear that N realizes the relabeling as described above.

We now define $M' = (Q', \Gamma, \Delta', q_0', R')$ to be the MTT with

– $Q' = \{(q, \varphi) \mid q \in Q^{(m)}, m \geq 0, \varphi : [m] \to (\{e, dd\} \cup A)\}$, where the rank of (q, φ) with $q \in Q^{(m)}$ is $|\{j \in [m] \mid \varphi(j) = dd\}|$,
– $\Delta' = (\Delta - \{0\}) \cup \{b^{(2)}, \mathrm{dummy}^{(2)}, e^{(0)}\}$, where b, dummy, and e are not in Δ,
– $q_0' = (q_0, \varnothing)$, and
– R' consisting of the following rules. For every $(q, \varphi) \in {Q'}^{(n)}$ and $(\sigma, (\phi_1, \ldots, \phi_k)) \in \Gamma^{(k)}$ with $n, k \geq 0$ and $q \in Q^{(m)}$ let

$$\langle (q, \varphi), (\sigma, (\phi_1, \ldots, \phi_k))(x_1, \ldots, x_k) \rangle (y_1, \ldots, y_n) \to \zeta$$

be in R', where $\zeta = \mathrm{comb}_{\mathrm{dummy}}(y_1 \cdots y_n)$ if there is a $j \in [m]$ such that $\varphi(j) = dd$ and y_j occurs more than once in $t = \mathrm{rhs}_M(q, \sigma)$ and otherwise ζ is obtained from t by the following replacements:

1. Replace each subtree $\langle q', x_i \rangle (t_1, \ldots, t_l)$ with $\langle q', x_i \rangle \in \langle Q, X_k \rangle^{(l)}$, $l \geq 0$, and $t_1, \ldots, t_l \in T_{\langle Q, X_k \rangle \cup \Delta}(Y_m)$, by the tree $\langle (q', \varphi'), x_i \rangle (t_{j_1}, \ldots, t_{j_{l'}})$, where $\{j_1, \ldots, j_{l'}\} = {\varphi'}^{-1}(dd)$ with $j_1 < \cdots < j_{l'}$ and for every $j \in [l]$, $\varphi'(j) = \mathrm{oc}(y(t_j \Theta \Psi))$ with Θ defined as above, and

$$\Psi = [y_\nu \leftarrow \varphi(\nu) \mid \nu \in [m]].$$

2. For $j \in [m]$, replace y_j by $\varphi(j)$ if $\varphi(j) \neq dd$, and otherwise replace it by y_ν with $\nu = |\{\mu \mid \mu < j \text{ and } \varphi(\mu) = dd\}| + 1$.
3. Replace each occurrence of 0 by e.

Obviously M' is sp. If we now consider the yields of all trees in $\tau_{M'}(\tau_N(K))$ which do not contain a dummy symbol, then we obtain L. By Theorem 7.4(1) of [EV85] $R = \tau_{M'}^{-1}(T_{\Delta' - \{\mathrm{dummy}\}})$ is a regular tree language. Hence $K' = \tau_N(K) \cap R$ is in $\mathcal{L}$ and $L = \tau_{M'}(K')$ is in $yMTT_{\mathrm{sp}}(\mathcal{L})$. □

Note that Theorems 5 and 6 can be applied to $\mathcal{L} = REGT$. Due to the next lemma they can also be applied to $\mathcal{L} = MTT^n(REGT)$ for $n \geq 1$.

Lemma 7. *Let $\mathcal{L}$ be a class of tree languages. If $\mathcal{L}$ is closed under finite state relabelings, then so is $MTT(\mathcal{L})$.*

Proof. Let $M = (Q, \Sigma, \Delta, q_0, R)$ be an MTT and let $N = (Q_N, \Delta, \Gamma, F, R_N)$ be a finite state relabeling. We now sketch how to construct a finite state relabeling N' and an MTT M' such that for every $s \in T_\Sigma$, $\tau_{M'}(\tau_{N'}(s)) = \tau_N(\tau_M(s))$. The idea is similar to the proof of Theorem 6. The relabeling N' replaces the symbol σ in $\sigma(s_1, \ldots, s_k) \in T_\Sigma$ by $(\sigma, (\phi_1, \ldots, \phi_k))$, where each ϕ_i associates with every $q \in Q^{(m)}$ a mapping of type $Q_N^m \to Q_N$ such that for $p_1, \ldots, p_m \in Q_N$, $\phi_i(q)(p_1, \ldots, p_m) = p$ if $M_q(s_i) \Rightarrow^*_{\hat{N}} \langle p, \tilde{s}\rangle$, where $\hat{N}$ is the extension of N to $\Delta \cup Y_M$ by rules $y_j \to \langle p_j, y_j\rangle$. Thus, if we know in which states $p_1, \ldots, p_m$ the relabeling N arrives after processing the trees which will be substituted for the parameters $y_1, \ldots, y_m$, respectively, then $\phi_i(q)(p_1, \ldots, p_k)$ is the state in which N arrives after processing the part of the output tree of M that corresponds to $M_q(s_i)$. The information on $p_1, \ldots, p_k$ is encoded into the states of M'; i.e., each state of M' is of the form (q, φ), where $q \in Q^{(m)}$, $\varphi : [m] \to Q_N$, and $\varphi(j)$ is the state in Q_N in which N arrives after processing the tree which is substituted for y_j in a derivation by M. Together we have sufficient information to "run" N on the right-hand side of M to obtain the corresponding rules of M'. □

In the next lemma we will show that the n-fold application of rub_0 can be simulated by a single application of $\mathrm{rub}_{0,1}$; i.e., if we know that $\mathrm{rub}_{0,1}(L) \in yMTT(\mathcal{L})$, then this means that also $\mathrm{rub}_0^n(L)$ for any $n \geq 2$ is in $yMTT(\mathcal{L})$. Note that $\mathrm{rub}_0^n(L) = \mathrm{rub}_{b_1,\ldots,b_n}(L)$, where $b_1, \ldots, b_n$ are new symbols not in L.

Lemma 8. *Let $\mathcal{L}$ be a class of tree languages which is closed under finite state relabelings. If $\mathrm{rub}_{0,1}(L) \in yMTT(\mathcal{L})$ then for every $n \geq 2$, $\mathrm{rub}_{b_1,\ldots,b_n}(L) \in yMTT(\mathcal{L})$.*

Proof. It is straightforward to construct an MTT M_{yield} which translates every input tree into its yield, represented as a monadic tree (e.g., $\sigma(a, b)$ is translated into $a(\hat{b})$). In fact in Example 1(6, yield) of [BE98] it is shown that this tree translation can be defined in monadic second order logic (MSO). By Theorem 7.1 of [EM98] the MSO definable tree translations are precisely those realized by finite copying macro tree transducers. We will now define a top-down tree transducer M_n which translates a monadic tree over the ranked alphabet $\Sigma = \{0^{(1)}, 1^{(1)}, \hat{0}^{(0)}, \hat{1}^{(0)}\}$ into a tree with yield in $\{b_1, \ldots, b_n\}^*$. This is done as follows. We use a Huffman code to represent each b_i by a string over $\{0, 1\}$; more precisely, the string 0^i1 represents b_{i+1} for every $0 \leq i \leq n-1$. M_n has states $1, \ldots, n$ and, starting in state 1, it arrives in state i after processing $i-1$ consecutive 0s. In state i, M_n outputs b_i (in the yield) if it processes a 1 and moves back to state 1.

Let $n \geq 2$ and define $M_n = ([n], \Sigma, \Gamma, 1, R)$ to be the top-down tree transducer with $\Gamma = \{\gamma^{(2)}, b_1^{(0)}, \ldots, b_n^{(0)}\}$ and R as follows.

$$
\begin{array}{lll}
\langle \nu, 1(x_1)\rangle & \to \gamma(b_\nu, \langle 1, x_1\rangle) & \text{for } \nu \in [n] \\
\langle \nu, 0(x_1)\rangle & \to \langle \nu + 1, x_1\rangle & \text{for } \nu \in [n-1] \\
\langle n, 0(x_1)\rangle & \to \gamma(b_n, \langle 1, x_1\rangle) & \\
\langle \nu, \hat{1}\rangle & \to e & \text{for } \nu \in [n] \\
\langle \nu, \hat{0}\rangle & \to e & \text{for } \nu \in [n]
\end{array}
$$

Clearly, $y\tau_{M_n}(T_\Sigma) = \{b_1, \ldots, b_n\}^*$ and hence if $yL = \{0,1\}^*$ then

$$y\tau_{M_n}(\tau_{M_{\text{yield}}}(L)) = \{b_1, \ldots, b_n\}^*.$$

Let Δ be a ranked alphabet. If we change M_n to have as input ranked alphabet $\Sigma' = \Sigma \cup \{\delta^{(1)} \mid \delta \in \Delta^{(0)}\} \cup \{\hat{\delta}^{(0)} \mid \delta \in \Delta^{(0)}\}$, as output alphabet $\Gamma' = \Gamma \cup \Delta^{(0)}$, and for every $\nu \in [n]$ the additional rules $\langle \nu, \delta(x_1)\rangle \to \gamma(\delta, \langle 1, x_1\rangle)$ and $\langle \nu, \hat{\delta}\rangle \to \delta$, then for every tree language K over Δ with $yK = \text{rub}_{0,1}(L)$, $y\tau_{M_n}(\tau_{M_{\text{yield}}}(K)) = \text{rub}_{b_1,\ldots,b_n}(L)$.

We can now compose $\tau_{M_{\text{yield}}}$ with τ_{M_n} to obtain again a finite copying MTT which realizes $\tau_{M_{\text{yield}}} \circ \tau_{M_n}$. This follows from the fact that MSO definable translations are closed under composition (cf. Proposition 2 of [BE98]) and that M_n is finite copying (it is even linear, i.e., 1-copying).

In Corollary 7.9 of [EM98] it is shown that finite copying MTTs with regular look-ahead have the same string generating power as finite copying top-down tree transducers with regular look-ahead. Hence, there is a finite copying top-down tree transducer with regular look-ahead M'' such that $y\tau_{M''}(K) = \text{rub}_{b_1,\ldots,b_n}(L)$ if $yK = \text{rub}_{0,1}(L)$. Since regular look-ahead can be simulated by a relabeling (see Proposition 18.1 in [GS97]) we get that $\text{rub}_{b_1,\ldots,b_n}(L) \in yT(DQRELAB(MTT(\mathcal{L})))$ and, by Lemma 7 and the closure of MTT under right composition with T (Theorem 4.12 of [EV85]), this means that $\text{rub}_{b_1,\ldots,b_n}(L)$ is in $yMTT(\mathcal{L})$. □

The proof of Lemma 8 in fact shows that $yMTT(\mathcal{L})$ is closed under deterministic generalized sequential machine (GSM) mappings. For the case of nondeterministic MTTs it is shown in Theorem 6.3 of [DE98] that the class of string languages generated by them is closed under nondeterministic GSM mappings.

We are now ready to prove that there is a string language which can be generated by a nondeterministic top-down tree transducer with monadic input but not by the composition closure of MTTs.

Theorem 9. $yN\text{-}T_{\text{mon}}(REGT) - \bigcup_{n\geq 0} yMTT^n(REGT) \neq \varnothing$.

Proof. Let $n \geq 1$. Since $MTT^n(REGT)$ is closed (i) under intersection with $REGT$ (follows trivially from the fact that $REGT$ is preserved by the inverse of MTT^n, cf. Theorem 7.4(1) in [EV85]) and (ii) under finite state relabelings (Lemma 7), we can apply Theorem 6 to $\mathcal{L} = MTT^n(REGT)$. We obtain that $\text{rub}_{b_1,\ldots,b_n}(L) \in yMTT^n(REGT)$ implies $\text{rub}_{b_1,\ldots,b_{n-1}}(L) \in yMTT_{\text{sp}}(MTT^{n-1}(REGT))$. By Theorem 5 the latter class equals $yT(MTT^{n-1}(REGT))$ and since $MTT \circ T = MTT$ (Theorem 4.12 of [EV85]), it equals $yMTT^{n-1}(REGT)$. Hence, by induction, $L \in yT(REGT)$.

Let us now consider the concrete language $L_{\text{exp}} = \{a^{2^n} \mid n \geq 0\}$. By the above we know that if $\text{rub}_{b_1,\ldots,b_n}(L) \in yMTT^n(REGT)$, then $L \in yT(REGT)$. Hence for $L = \text{rub}_b(L_{\text{exp}})$ we get that $\text{rub}_{b_1,\ldots,b_n}(L) \in yMTT^n(REGT)$ implies $\text{rub}_b(L_{\text{exp}}) \in yT(REGT)$. But by Corollary 3.2.16 of [ERS80] it is known that $\text{rub}_b(L_{\text{exp}})$ is *not* in $yT(REGT)$ (the proof uses a bridge theorem which would imply that L_{exp} can be generated by a finite copying top-down tree transducer;

but the languages generated by such transducers have the "Parikh property" and hence cannot be of exponential growth).

Altogether we get that $\mathrm{rub}_{b_1,\ldots,b_n,b}(L_{\exp})$ is not in $yMTT^n(REGT)$. By Lemma 8 this means that $\mathrm{rub}_{0,1}(L_{\exp}) \notin yMTT^n(REGT)$. It is easy to show that $\mathrm{rub}_{0,1}(L_{\exp})$ can be generated by a nondeterministic top-down tree transducer with monadic input; in fact, in Corollary 3.2.16 of [ERS80] it is shown that this language can be generated by an ET0L system. The class of languages generated by ET0L systems is precisely the class of string languages generated by nondeterministic top-down tree transducers with monadic input [Eng76]. □

Note that the last statement in the proof of Theorem 9 implies that $ET0L - \bigcup_{n\geq 0} yMTT^n(REGT) \neq \varnothing$, where $ET0L$ is the class of languages generated by ET0L systems. It is known that the IO-hierarchy $\bigcup_{n\geq 0} yYIELD^n(REGT)$ is inside $\bigcup_{n\geq 0} yMTT^n(REGT)$ (this follows, e.g., from Corollary 4.13 of [EV85]). From Theorem 9 we obtain the following corollary.

Corollary 10. $\mathrm{rub}_{0,1}(L_{\exp})$ *is not in the IO-hierarchy.*

5 Conclusions and Further Research Topics

In this paper we have proved that macro tree transducers which are simple in the parameters generate the same class of string languages as top-down tree transducers. Furthermore we have shown that there is a string language which can be generated by a nondeterministic top-down tree transducer with a regular monadic input language but not by the composition closure of MTT.

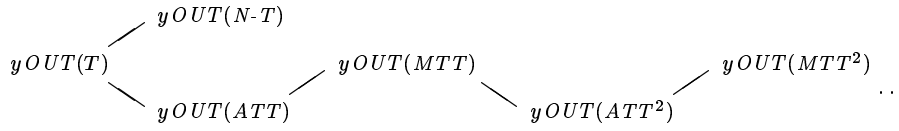


Fig. 2. Inclusion diagram for classes of string languages generated by tree transducers

Let us now consider another type of tree transducer: the attributed tree transducer (ATT) [Fül81]. Since the class ATT of translations realized by ATTs is a proper subclass of MTT it follows that $\mathrm{rub}_{0,1}(L_{\exp})$ is not in the class $yOUT(ATT)$ of string languages generated by ATTs. Since nondeterministic top-down tree transducers with monadic input equal cooperating regular tree grammars [FM98] and attributed tree transducers have the same term generating power as context-free hypergraph grammars, it follows that there is a tree language which can be generated by a cooperating regular tree grammar but not by a context-free hypergraph grammar. This remained open in [Man98a].

It is known that the class of string languages generated by top-down tree transducers is properly contained in that generated by ATTs (see, e.g., [Eng86]). Together with Theorem 9 this means that the two leftmost inclusions in Fig. 2 are proper (inclusions are edges going from left to right). However, it is open whether the other inclusions in Fig. 2 are proper. For instance, we do not know

whether there is a language which can be generated by an MTT but not by an ATT. Note that for the corresponding classes of tree languages we know the answer: the language $\{\gamma^{2^n}(\alpha) \mid n \geq 0\}$ of monadic trees of exponential height can be generated by an MTT but not by an ATT (cf. Example 6.1 in [Man98b]).

Acknowledgement I wish to thank Joost Engelfriet for helpful discussions.

References

[BE98] R. Bloem and J. Engelfriet. A comparison of tree transductions defined by monadic second order logic and by attribute grammars. Technical Report 98-02, Leiden University, 1998.

[CF82] B. Courcelle and P. Franchi-Zannettacci. Attribute grammars and recursive program schemes. *Theoret. Comput. Sci.*, 17:163–191 and 235–257, 1982.

[DE98] Frank Drewes and Joost Engelfriet. Decidability of finiteness of ranges of tree transductions. *Inform. and Comput.*, 145:1–50, 1998.

[EM98] J. Engelfriet and S. Maneth. Macro tree transducers, attribute grammars, and MSO definable tree translations. Technical Report 98-09, Leiden University, 1998.

[Eng76] J. Engelfriet. Surface tree languages and parallel derivation trees. *Theoret. Comput. Sci.*, 2:9–27, 1976.

[Eng80] J. Engelfriet. Some open questions and recent results on tree transducers and tree languages. In R.V. Book, editor, *Formal language theory; perspectives and open problems*. New York, Academic Press, 1980.

[Eng86] J. Engelfriet. The complexity of languages generated by attribute grammars. *SIAM J. Comput.*, 15(1):70–86, 1986.

[ERS80] J. Engelfriet, G. Rozenberg, and G. Slutzki. Tree transducers, L systems, and two-way machines. *J. of Comp. Syst. Sci.*, 20:150–202, 1980.

[EV85] J. Engelfriet and H. Vogler. Macro tree transducers. *J. of Comp. Syst. Sci.*, 31:71–146, 1985.

[Fis68] M.J. Fischer. *Grammars with macro-like productions*. PhD thesis, Harvard University, Massachusetts, 1968.

[FM98] Z. Fülöp and S. Maneth. A characterization of ET0L tree languages by cooperating regular tree grammars. (To appear in "Grammatical Models of Multi-Agent Systems", Gordon and Breach, London), 1998.

[Fül81] Z. Fülöp. On attributed tree transducers. *Acta Cybernetica*, 5:261–279, 1981.

[FV98] Z. Fülöp and H. Vogler. *Syntax-Directed Semantics – Formal Models based on Tree Transducers*. EATCS Monographs on Theoretical Computer Science (W. Brauer, G. Rozenberg, A. Salomaa, eds.). Springer-Verlag, 1998.

[GS97] F. Gécseg and M. Steinby. Tree automata. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages, Volume 3*, chapter 1. Springer-Verlag, 1997.

[Küh98] A. Kühnemann. A pumping lemma for output languages of macro tree transducers. Technical Report TUD/FI95/08, Technical University Dresden, 1998. (also in *Proc. CAAP'96, LNCS 1059*, pages 44-58. Springer-Verlag, 1997.).

[Man98a] S. Maneth. Cooperating distributed hyperedge replacement grammars. In A. Kelemenová, editor, *Proc. MFCS'98 Satellite Workshop on Grammar Systems*, pages 149–164. Silesian University, 1998. (To appear in *Grammars*).

[Man98b] S. Maneth. The generating power of total deterministic tree transducers. *Inform. and Comput.*, 147:111–144, 1998.

Matching Specifications for Message Sequence Charts

Anca Muscholl

Institut für Informatik, Universität Stuttgart,
Breitwiesenstr. 20-22, 70565 Stuttgart, Germany

Abstract. Message sequence charts (MSC) are widely used in the early design of communication protocols. They allow describing the communication skeleton of a system. We consider a basic verification task for hierarchical MSCs, the matching problem via MSC templates. We characterize the complexity of checking properties which are expressible by and-or templates, resp. by LTL formulas. Both problems are shown to be PSPACE-complete.

1 Introduction

Message sequence charts (MSC) are an ITU-standardized notation which is widely used for the early design of communication protocols. They describe in a graphical way how message passing is supposed to be performed between concurrent processes. Although MSCs do not contain the full information that is needed for implementing the described protocols, they can be used for various analysis purposes. For example, one can use MSCs to detect mistakes in the design, like *race conditions* [1] or *non-local choice* [2]. Some tools for performing basic verification tasks have been developed, [1, 3].

The problem considered in this paper is to verify certain properties of MSC graphs (hierarchical MSCs) by means of template (MSC) graphs. Matching a template graph with a system graph means that a specific *set of executions in the template* is required to occur in an execution of the system, such that the causal order of events is preserved. The occurrence of executions is meant as an embedding, i.e. it allows gaps. (Actually, determining the existence of an exact mapping is easily seen to be undecidable, [7]). Two semantics have been already considered for template graphs, or-graphs resp. and-graphs, [7].

Here we focuss on specifications (templates) given as and-or MSC graphs, which can be seen as alternating transition systems with an associated causal order related to Mazurkiewicz traces, [5]. Like alternating automata on ω-words [10], it is more convenient to translate specifications into and-or MSC graphs than just or-graphs. The matching problem for and-or template graphs is shown to be PSPACE-complete. However, our proof shows somewhat surprisingly that and-or templates are not more expressive than or-templates (or and-templates). And-or MSC templates just provide a more succinct representation of specifications. They express properties given by shuffle ideals (of finite sets) and liveness

W. Thomas (Ed.): FOSSACS'99, LNCS 1578, pp. 273-287, 1999.

conditions. We also consider the problem of specifying templates by LTL formulas, that is by matching the causal orders of sequences which satisfy a given formula. For this semantics we show that deciding the existence of a matching is PSPACE-complete, too. The paper is organized as follows: Sect. 2 gives basic notions and definitions, in Sect. 3 we consider the and-or matching problem, and Sect. 4 deals with the matching problem for LTL specifications.

2 Preliminaries

Definition 1 (MSC). *A message sequence chart* $M = (\mathcal{E}, <, \mathcal{P}, L, T)$ *is given by a poset* $(\mathcal{E}, <)$ *of events, a set* $\mathcal{P}$ *of processes, a mapping* $L : \mathcal{E} \to \mathcal{P}$ *that associates each event with a process, and a mapping* $T : \mathcal{E} \to \{\mathsf{s}, \mathsf{r}\}$ *that describes the type of each event (*send *or* receive*).*

The partial order $<$ is called the *visual order* of events and it is obtained from the syntactical representation of the chart (e.g. represented according to the standard syntax ITU-Z 120). The visual order is induced by an acyclic relation $<_c \cup (\bigcup_{P \in \mathcal{P}} <_P)$ that is explained in the following. First, there is a one-to-one correspondence between send events, $\mathcal{E} \cap T^{-1}(\mathsf{s})$, and receive events, $\mathcal{E} \cap T^{-1}(\mathsf{r})$. Let $\mathcal{M}$ denote the graph of this correspondence, i.e. the set of messages. Then we have $e <_c f$ for every message $(e, f) \in \mathcal{M}$ (*message ordering*). Secondly, for every process $P \in \mathcal{P}$ the set $\mathcal{E} \cap L^{-1}(P)$ is totally ordered by $<_P$ (*process line ordering*).

In general, the visual order provides more ordering than intended by the designer. Therefore every chart has an associated causal structure providing the intended ordering [1]. Causal structures are related to *pomsets* [9], *event structures* [8], and *Mazurkiewicz traces* [4]. A causal structure is obtained from a chart by means of a given semantics, which depends on the system architecture. Formally, the causal structure of a chart $M = (\mathcal{E}, <, \mathcal{P}, L, T)$ is given as $\mathrm{tr}(M) = (\mathcal{E}, \prec, \mathcal{P}, L, T)$, where the only difference between M and $\mathrm{tr}(M)$ is the poset $(\mathcal{E}, \prec)$, with $\prec$ denoting the *causal order*. The partial order $\prec$ is generated by a so-called *precedence relation* $\prec\!\!\!\cdot$, which depends on the implementation. The meaning is that for any two events e and f, we have $e \prec\!\!\!\cdot f$ if and only if event e must terminate before event f starts.

As the semantics used throughout the paper, we define the precedence relation for an architecture with fifo queues. This means that every one-directional communication between two processes is done through a fifo channel. For this architecture we first impose the following constraint on the visual order: For any messages $(e, f), (e', f') \in \mathcal{M}$ with $e <_P e'$ and $L(f) = L(f') = P'$ for some $P, P' \in \mathcal{P}$ we require that $f <_{P'} f'$. Let $e, f \in \mathcal{E}$ be two events. Then $e \prec\!\!\!\cdot f$ for the *fifo semantics* if one of the following holds:

1. A send preceded by some event on the same process:
$$T(f) = \mathsf{s} \ \wedge \ e <_P f \ \text{ for some process } P\,.$$
2. Message ordering: $e <_c f$.

3. Messages ordered by the fifo queue:

$$T(e) = T(f) = \mathsf{r} \ \wedge \ e <_P f \ \text{ for some process } P \ \wedge$$
$$\exists e', f' \ (e' <_c e \ \wedge \ f' <_c f \ \wedge \ e' <_{P'} f' \ \text{ for some process } P') \,.$$

Slightly abusing the notation, we identify $\mathrm{tr}(M)$ with the set of all linearizations of the causal order of $\mathrm{tr}(M)$. We also write $\mathrm{tr}(\alpha)$ for a sequence $\alpha \in \mathcal{E}^\infty$ and mean by that the set of all linearizations of the causal order associated with α.

Given two charts $M_i = (\mathcal{E}_i, <_i, \mathcal{P}, L_i, T_i)$ over the same set of processes we define their product as the chart $M_1 M_2 = (\mathcal{E}_1 \dot\cup \mathcal{E}_2, <, \mathcal{P}, L_1 \dot\cup L_2, T_1 \dot\cup T_2)$, where $< = <_1 \cup <_2 \cup \bigcup_{P \in \mathcal{P}} (\mathcal{E}_1 \cap L^{-1}(P)) \times (\mathcal{E}_2 \cap L^{-1}(P))$.

Definition 2 (MSC graph). *An MSC graph $N = (\mathcal{S}, \tau, s_0, c, \mathcal{P})$ is given as a finite, directed graph $(\mathcal{S}, \tau, s_0)$ with state set $\mathcal{S}$, transition relation $\tau \subseteq \mathcal{S} \times \mathcal{S}$, source state $s_0 \in \mathcal{S}$, together with a mapping c associating each state s with a finite chart $c(s)$ over the process set $\mathcal{P}$.*

Let $\xi = s_1, s_2, \ldots$ be a (possibly infinite) path in N, i.e. $(s_i, s_{i+1}) \in \tau$ for every i. The chart defined by ξ is given as $c(\xi) = c(s_1)c(s_2)\ldots$. We denote by $tr(\xi)$ the causal structure associated with $c(\xi)$, resp. by $\prec_\xi$ the associated causal order.

A maximal path of N is a path in the graph which starts with the source state and is either infinite or it ends in a sink state.

In order to simplify the presentation we assume that in an MSC graph N every state $s \in \mathcal{S}$ is associated with a single event. This is no restriction, due to the following observation: If $\alpha = \alpha_1 \cdots \alpha_k$ denotes any topological sorting of the visual order of a chart M, then $\mathrm{tr}(\alpha) = \mathrm{tr}(M)$. Let $M = c(s)$ for a state $s \in \mathcal{S}$. We can replace s by a sequence of states $s = s_0 \stackrel{\alpha_1}{\to} s_1 \stackrel{\alpha_2}{\to} \cdots s_k$ and add transitions from s_k to all successors of s.

Definition 3 (Matching). *Under a given semantics, a template M with the causal structure $tr(M) = (\mathcal{E}_M, \prec_M, L_M, T_M, \mathcal{P}_M)$ matches an MSC N with the causal structure $tr(N) = (\mathcal{E}_N, \prec_N, L_N, T_N, \mathcal{P}_N)$ if and only if $\mathcal{P}_M \subseteq \mathcal{P}_N$ and there exists an injective mapping (called* embedding*) $h : \mathcal{E}_M \to \mathcal{E}_N$ such that*

- *for each $e \in \mathcal{E}_M$, we have $L_N(h(e)) = L_M(e)$ and $T_N(h(e)) = T_M(e)$ (preserving processes and types), and*
- *if $e_1 \prec_M e_2$ then $h(e_1) \prec_N h(e_2)$ (preserving the causal order).*

A path ξ_1 in a graph M matches a path ξ_2 in a graph N if the chart $c(\xi_1)$ matches the chart $c(\xi_2)$.

Under the fifo semantics it suffices to consider only the type and the location of events. Let $\mathcal{P} = \{P_1, \ldots, P_m\}$ be the set of processes and let $E = \{\mathsf{s}_{ij}, \mathsf{r}_{ij} \mid 1 \le i \ne j \le m\}$. The set E consists of *abstract events* and is related to any set of events $\mathcal{E}$ by means of a mapping $\mathrm{ev} : \mathcal{E} \to E$. Consider a message $(e, f) \in \mathcal{M}$ with

$L(e) = P_i$, $L(f) = P_j$. Then we let $\mathrm{ev}(e) = \mathsf{s}_{ij}$ and $\mathrm{ev}(f) = \mathsf{r}_{ij}$. Let χ be a path in an MSC graph, then $\mathrm{msg}(\chi) \subseteq E$ denotes the set $\{\mathrm{ev}(e) \mid e \text{ occurs in } c(\chi)\}$. The mapping ev extends to a homomorphism $\mathrm{ev} : \mathcal{E}^\infty \to E^\infty$. Assume that α is a topological sorting of $c(\chi)$. Let $\mathrm{ev}(\chi) \in E^\infty = E^* \cup E^\omega$ denote the (finite or infinite) sequence of abstract events $\mathrm{ev}(\alpha)$.

For the remaining of the paper we will refer to the elements of E simply as *events*. All notions introduced so far, e.g. precedence rules, causal order, matchings, can be transferred to the events in E.

3 And-Or MSC Graphs

An and-or MSC graph $M = (\mathcal{S}, \tau, s_0, c, \mathcal{S}_\wedge, \mathcal{S}_\vee, \mathcal{P})$ is given as an MSC graph where the set of states is partitioned in two sets, the set of and-states $\mathcal{S}_\wedge$ and the set of or-states $\mathcal{S}_\vee$.

Definition 4 (And-or MSC graphs). *Let $M = (\mathcal{S}, \tau, s_0, c, \mathcal{S}_\wedge, \mathcal{S}_\vee, \mathcal{P})$ be an and-or MSC graph. A* run *R of M is a tree $R = (X, \to, x_0, \ell, ev)$ with root x_0 and nodes labelled by states, $\ell : X \to \mathcal{S}$, such that:*

1. *$x \to y$ in R implies that $(\ell(x), \ell(y)) \in \tau$.*
2. *Every node $x \in X$ with $\ell(x) = s \in \mathcal{S}_\vee$ has one successor in R, if $\{s' \mid \tau(s, s')\} \neq \emptyset$, otherwise it has no successor.*
3. *Every node $x \in X$ with $\ell(x) = s \in \mathcal{S}_\wedge$ has exactly k successors in R, all labelled differently, where $k = |\{s' \mid \tau(s, s')\}|$.*

Moreover, $ev : X \to E$ labels each node x by the event $ev(e)$, where $e = c(\ell(x))$.

A run R is called maximal *if all leaves are labelled by states without any successors in M and the root is labelled by the source state.*

An and-or MSC graph is an example for alternating transition systems. It can be used for specifying scenarios between a component of a system and the environment. Here, the moves of the environment are modelled as usually as universal moves, i.e. the component is required to meet its specification no matter how the environment behaves. Consider for example the and-or graph in Fig. 1, where state N_1 is an and-state and N_2, N_3 are or-states. For simplicity, the states are labelled by messages, not by single events. The MSC graph expresses that there are infinitely many connections requested by P_2 (from P_1), and all of them are either successful or P_2 gets into an infinite loop requesting a connection. Moreover, each time when process P_2 requests a connection, it expects (several) data transmissions from P_3.

Definition 5 (And-or matching). *Given a template M as an and-or MSC graph and a system N as an MSC graph. We say that the template matches the system if there is some maximal run R of M and a maximal path χ of N such that every path in R matches χ.*

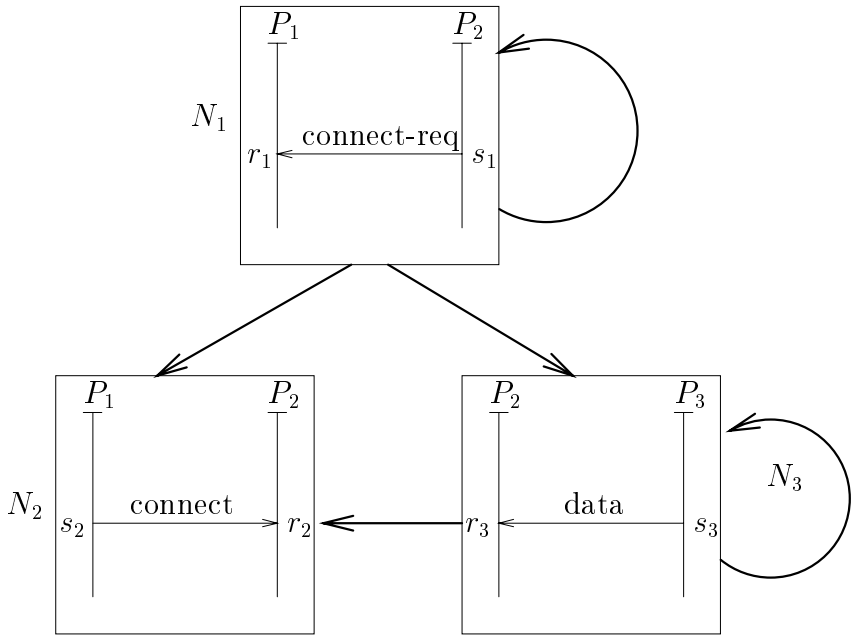


Fig. 1. An and-or MSC graph.

Consider again the and-or template graph M of the above example. In every maximal run of M the and-node N_1 occurs infinitely often. Therefore, a maximal run of M matches an MSC N only if N contains infinitely many events of the same type as s_1, r_1. This holds also for s_2, r_2, since every maximal run of M includes paths $N_1^k N_2$ for every k, and we have $r_1 \prec s_2 \prec r_2$. The situation differs for node N_3. We might have a maximal run where N_3 occurs on every path just once, before N_2. But we do not have any events e in N_1, f in N_3 such that $e \prec f$. Hence, s_3 for example can be mapped for each occurrence of the node N_3 in the run of M to the same event in the MSC N. (The specification given by M is in some sense incomplete, it does not correspond to the intuition given above). Moreover, the combination between states occurring infinitely often and dependency between events makes matching for and-or templates to a quite complex task.

3.1 The Complexity of And-Or Matching

Our main result is that matching and-or template graphs is PSPACE-complete. Thus, and-or matching appears to be computationally more difficult than matching or-templates, resp. and-templates. The latter problems have been shown to be NP-complete, [7].

The next proposition gives the lower bound for the and-or matching problem.

Proposition 1. *The problem whether an and-or template graph matches a system graph is PSPACE-hard.*

Proof. We give a reduction from TQBF, i.e. the question whether a quantified Boolean formula is true or not. Suppose that $Q_0 x_0 \cdots Q_l x_l \; F(x_0, \ldots, x_l)$ is in prenex normal form, with $Q_i \in \{\exists, \forall\}$ and F a quantifier-free formula in 3-CNF. That is, $F = C_1 \wedge \cdots \wedge C_m$, with C_j denoting disjunctions of at most three literals. For simplifying notations, the graphs M, N described below are such that states are labelled by charts (instead of events). Recall that this can be easily translated to a labelling by events.

Let the set of processes be $\mathcal{P} = \{P_i, P_i', R_i, R_i' \mid 0 \leq i \leq l\}$. Let also p_i (resp. p_i') denote a message from P_i to P_i' (resp. from P_i' to P_i). Analogously, let n_i resp. n_i' denote a message from R_i to R_i', resp. from R_i' to R_i. The system graph N (see also Fig. 2) is the single MSC $p_1' p_1 n_1' n_1 \cdots p_l' p_l n_l' n_l$. Note that the events e with $L(e) \in \{P_i, P_i'\}$ are totally ordered in the causal order. Similarly, all events e with $L(e) \in \{R_i, R_i'\}$ are totally ordered.

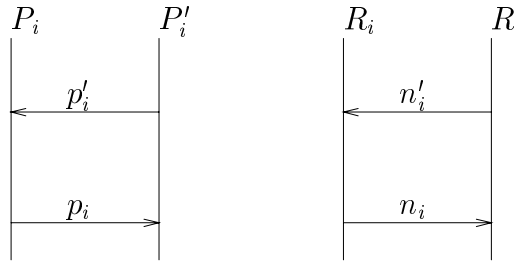


Fig. 2. The system graph.

The template graph $M = (\mathcal{S}, \tau, s_0, c, \mathcal{S}_\vee, \mathcal{S}_\wedge)$ is given by the vertex set

$$\mathcal{S} = \{s_i, s_i^+, s_i^- \mid 0 \leq i \leq l\} \cup \{t_i, t_{i,1}, t_{i,2}, t_{i,3} \mid 1 \leq i \leq m\} \cup \{t\}\,,$$

and the transition relation $\tau \subseteq \mathcal{S} \times \mathcal{S}$:

$$\tau = \{(s_i, s_i^+), (s_i, s_i^-), (s_j^+, s_{j+1}), (s_j^-, s_{j+1}) \mid 0 \leq i, j \leq l, j \neq l\} \cup$$
$$\{(s_l^+, t), (s_l^-, t)\} \;\cup\; \{(t, t_i) \mid 1 \leq i \leq m\} \cup \{(t_i, t_{i,1}), (t_i, t_{i,2}), (t_i, t_{i,3}) \mid 1 \leq i \leq m\}$$

For $s \in \mathcal{S}$ we define $s \in \mathcal{S}_\wedge$ if and only if either $s = s_i$ and $Q_i = \forall$, or $s = t$. That is, the nodes s_i, $0 \leq i < l$, are of type corresponding to the quantifier Q_i. Moreover, the node t is an and-node corresponding to the conjunction of the clauses. All remaining nodes are or-nodes. The states s_i^+, s_i^- are labelled by single messages (see also Fig. 3):

$$c(s) = \begin{cases} p_i & \text{for } s = s_i^+, 0 \leq i \leq l \\ n_i & \text{for } s = s_i^-, 0 \leq i \leq l \end{cases}$$

Let $C_j = x_{j,1} \vee x_{j,2} \vee x_{j,3}$ be a clause. If $x_{j,1} = x_k$ is a positive literal then let $c(t_{j,1}) = n'_k$. For a negative literal $x_{j,1} = \bar{x}_k$ we let $c(t_{j,1}) = p'_k$. The definition of $c(t_{j,2}), c(t_{j,3})$ is similar, depending on $x_{j,2}$ resp. $x_{j,3}$. For example, in Fig. 3 we represented the clause $x_1 \vee \bar{x}_2 \vee x_3$. All remaining states are labelled by $\emptyset$.

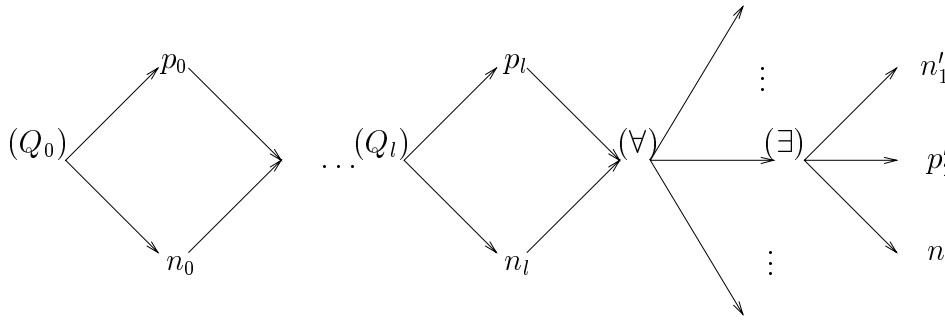


Fig. 3. The template graph.

There is a natural bijection between the assignments $\sigma : \{x_1, \ldots, x_n\} \to \{\mathbf{t}, \mathbf{f}\}$ for F and the paths from s_0 to t in M. Formally, if s_i^+ belongs to a path then x_i is assigned the value $\mathbf{t}$, whereas if s_i^- belongs to a path, then x_i is assigned the value $\mathbf{f}$. It is not difficult to verify that a run R of M matches the system chart N if and only if every path in R from s_0 to t defines an accepting assignment for F. Hereby, we choose for every clause a leaf corresponding to a true literal. The (easy) proof is left to the reader.

3.2 Matching and-or templates in PSPACE

Showing the lower bound for the and-or matching problem was quite straightforward, which is not the case with the upper bound. Actually, it is a priori not clear why it would be possible to restrict the and-or matching problem to a finite instance of matching. The aim of the next propositions is to give a decomposition of maximal runs in an and-or template graph in two parts. The first part is a polynomially bounded tree, all paths of which have to be matched against a finite path in the system. The remaining part is the so-called recurrent part, for which only (abstract) events have to be recorded, all of which have to occur in a loop of the system graph.

Let $x = x_1, \ldots, x_k = y$ be a path in a run $R = (X, \to, x_0, \ell, \mathrm{ev})$. We write $x \prec y$ whenever the event $\mathrm{ev}(x)$ precedes causally the event $\mathrm{ev}(y)$ in the sequence $\mathrm{ev}(x_1) \cdots \mathrm{ev}(x_k)$. Let $y{\downarrow} = \{x \in X \mid x \prec y\}$.

Definition 6. *Let $R = (X, \to, x_0, \ell, ev)$ be a run in an and-or MSC graph M. Let $X_e = \{x \in X \mid ev(x) = e\}$. We define a set of events $E_{fin}(R) \subseteq E$ as*

$$E_{fin}(R) = \{e \in E \mid \max\{|x\downarrow| \mid x \in X_e\} < \infty\}.$$

By $X_{fin}(R) \subseteq X$ we denote the subset of vertices $X_{fin}(R) = \{x \downarrow | \ ev(x) \in E_{fin}(R)\}$.

Intuitively, $E_{\rm fin}(R) \subseteq E$ contains all events which require an exact matching for the corresponding nodes x (resp. paths from $x\downarrow$), i.e. for nodes x with $\mathrm{ev}(x) \in E_{\rm fin}(R)$. Equivalently, for nodes x with events from $E \setminus E_{\rm fin}(R)$ the only information needed is $\mathrm{ev}(x) \in E$. Consider again the and-or template of Fig. 1. We have $E_{\rm fin}(R) \subseteq \{\mathsf{s}_{32}, \mathsf{r}_{32}\}$ for every maximal run R of M. Put it another way, $s_{21} \notin E_{\rm fin}(R)$, since N_1 has to repeat infinitely often on some path of the run. Moreover, $\mathsf{s}_{12} \notin E_{\rm fin}(R)$ since every maximal run includes paths $N_1^k N_2$ for every k and $r_1 \prec s_2$. The situation differs for node N_3. We might have a maximal run where N_3 occurs on every path just once, before N_2. Since there is no dependence from N_1 to N_3 all events e with $\mathrm{ev}(e) = \mathsf{s}_{32}$ can be mapped to the same event on the system path (similarly for $\mathrm{ev}(e) = \mathsf{r}_{32}$). Thus, node N_2 represents the bounded part, which has to be matched exactly against the system.

Notations: Let $M = (\mathcal{S}, \tau, s_0, c, \mathcal{S}_\wedge, \mathcal{S}_\vee)$ be an and-or graph and consider a run $R = (X, \to, x_0, \ell, \mathrm{ev})$. Let $\rho = (x_1, x_2, \dots)$ be a sequence of vertices from X, then $\ell(\rho)$ denotes the sequence of states $(\ell(x_1), \ell(x_2), \dots) \in \mathcal{S}^\infty$. For $x \in X$ we denote by R_x the subtree of R with root x and by $\pi(x)$ the path $(x_0, \dots, x_k)$ from x_0 to $x = x_k$. By $\pi_{\rm fin}(x)$ we denote the subsequence $(x_{i_1}, \dots, x_{i_l})$ of $\pi(x)$ containing exactly the vertices from $X_{\rm fin}(R)$. (I.e. we have $x_j \in X_{\rm fin}(R)$ if and only if $j = i_m$ for some m.) More generally, for a (finite or infinite) path ρ in R we denote by $\pi_{\rm fin}(\rho)$ the subsequence of vertices of ρ belonging to $X_{\rm fin}(R)$.

Remark 1. Note that for any path ρ in R the subsequence $\pi_{\rm fin}(\rho)$ is finite. Moreover, there exists a sequence of events $\alpha \in (E \setminus E_{\rm fin}(R))^\infty$ such that $\mathrm{ev}(\rho)$ and $\mathrm{ev}(\pi_{\rm fin}(\rho))\alpha$ define the same causal structure, i.e. $\mathrm{tr}(\mathrm{ev}(\rho)) = \mathrm{tr}(\mathrm{ev}(\pi_{\rm fin}(\rho))\alpha)$.

Lemma 1. *Let M be an and-or template graph and consider a run R' of M. Let $e \in E_{fin}(R')$. Then there exists a run $R = (X, \to, x_0, \ell, ev)$ of M satisfying the following conditions:*

1. *Every path of R is a subpath of some path in R'.*
2. *Let $x, y, z \in X$ be such that $x \in X_e$ and $y, z \in x\downarrow$. Then $\ell(y) = \ell(z)$ implies $y = z$.*
3. *We have $E_{fin}(R') \subseteq E_{fin}(R)$. Moreover, if R' is maximal, then R is maximal, too.*

The previous lemma says that it suffices to consider maximal runs R where for every path ρ the subsequence $\pi_{\rm fin}(\rho)$ has length at most $|\mathcal{S}|$, with $\mathcal{S}$ denoting the set of states of M.

Lemma 2. *Let M be an MSC and let χ be a loop in an MSC graph such that $msg(M) \subseteq msg(\chi)$. Then M matches χ^ω.*

Lemma 3. *Let R be a run in an and-or template graph M and let N be a system graph. Then R matches N if and only if R matches some path in N of the form $\chi_0\chi_1^\omega$, where χ_0 is a finite path and χ_1 is a (possibly empty) loop.*

Lemma 4. *Let $R = (X, \to, x_0, \ell, ev)$ be a run in an and-or template graph M and let $\chi = \chi_0\chi_1^\omega$ be a path in a system graph N such that R matches χ. Then we have*

1. *$(E \setminus E_{fin}(R)) \subseteq msg(\chi_1)$*
2. *Every path ρ in R is such that $\pi_{fin}(\rho)$ matches $\chi_0\chi_1^{|\mathcal{S}|}$.*

The proposition below says that it is sufficient to consider maximal runs R such that any two vertices $x, y \in X_{\text{fin}}(R)$ which are such that the subpaths $\pi_{\text{fin}}(x)$ and $\pi_{\text{fin}}(y)$ have the same sequence of state labels also have equal subtrees. We assume in the following w.l.o.g. that the source state of the template has no incoming edges, thus the root x_0 of a run R belongs to $X_{\text{fin}}(R)$.

Proposition 2. *Let M be an and-or template graph and consider a run R of M. Then a run $R' = (X, \to, x_0, \ell, ev)$ of M exists such that:*

1. *For all vertices $x, y \in X_{fin}(R')$ with $\ell(\pi_{fin}(x)) = \ell(\pi_{fin}(y))$ we have $R'_x = R'_y$.*
2. *For every path ρ' in R', the causal structure $tr(\pi_{fin}(\rho'))$ is equal to $tr(\pi_{fin}(\rho))$, for some path ρ in R.*
3. *If R satisfies Lemma 1, then R' satisfies Lemma 1, too. Moreover, if R is maximal, then R' is also maximal and we have finally $E_{fin}(R) \subseteq E_{fin}(R')$.*

Proof. Consider a maximal run R satisfying Lem. 1. We assume that R does not satisfy the first condition of the statement and we consider two vertices x, y from $X_{\text{fin}}(R)$ with $\ell(\pi_{\text{fin}}(x)) = \ell(\pi_{\text{fin}}(y))$, but $R_x \neq R_y$. Clearly, x, y are incomparable w.r.t. the successor relation in R, since they are labelled by the same state. We claim that the run R' obtained from R by replacing R_y by a copy of R_x satisfies Conds. (2) and (3) of the statement. To see this, note that for every maximal path ρ' in R' containing y there is a corresponding maximal path ρ in R containing x such that $\ell(\pi_{\text{fin}}(\rho)) = \ell(\pi_{\text{fin}}(\rho'))$ (here π_{fin} is meant w.r.t. R.) Moreover, the nodes in $\rho \setminus \pi_{\text{fin}}(\rho)$ (resp. in $\rho' \setminus \pi_{\text{fin}}(\rho')$) are labelled by events from $E \setminus E_{\text{fin}}(R)$.

For subtrees R' of R we define $m(R')$ as

$$m(R') = \max\{|\pi_{\text{fin}}(\rho)| \mid \rho \text{ is a path in } R'\}\,.$$

Note that $m(R')$ is defined w.r.t. R, i.e. $\pi_{\text{fin}}(\rho)$ is given by R. We show below by induction on $m(R')$ how to obtain a run $\hat{R}$ satisfying the requirement of the proposition with regard to $X_{\text{fin}}(R)$. This means that any two vertices x, y of $\hat{R}$ with $\ell(\pi_{\text{fin}}(x)) = \ell(\pi_{\text{fin}}(y))$ will have equal subtrees. However, we will obtain $E_{\text{fin}}(R) \subseteq E_{\text{fin}}(\hat{R})$. If $E_{\text{fin}}(R) = E_{\text{fin}}(\hat{R})$ then we are done, since in this case

$\pi_{\text{fin}}(\rho)$ as given by R is the same as $\pi_{\text{fin}}(\rho)$ as given by $\hat{R}$. Otherwise we repeat the construction with $\hat{R}$. After at most $|E|$ steps we obtain the desired run.

For defining $\hat{R}$ we consider all vertices $x \in X_{\text{fin}}(R)$ at depth 1, i.e. $\pi_{\text{fin}}(x) = (x_0, x)$ has length 1. Suppose that x is labelled by $s \in \mathcal{S}$ and let us choose some fixed vertex x_s of this kind, for each possible s. Consider the subtree R_{x_s} rooted at x_s. Clearly, $m(R_{x_s}) < m(R)$. By induction we may assume that any two vertices y, z in R_{x_s} with $\ell(\pi_{\text{fin}}(y)) = \ell(\pi_{\text{fin}}(z))$ have equal subtrees. We can replace all subtrees R_x for x with $\pi_{\text{fin}}(x) = (x_0, x)$, $\ell(x) = s$, by R_{x_s} and obtain the desired run $\hat{R}$.

The next proposition gives the characterization for deciding the existence of a matching from an and-or graph into an MSC graph.

Proposition 3. *Let M be an and-or template graph and let N be a system graph. Let $\mathcal{S}$ denote the set of states of M. Then M matches N if and only if there exist*

1. *A set of events $G \subseteq E$ of M.*
2. *A path χ in N to a strongly connected component C of N with $G \subseteq msg(C)$.*
3. *A tree $T_0 = (X, \to, x_0, \ell, ev)$ labelled by $\ell : X \to \mathcal{S}$, $ev : X \to E$, such that for any two distinct nodes $x, y \in X$ which are either siblings or comparable in T_0, $\ell(x) \neq \ell(y)$. Moreover, every path of T_0 matches the path χ in N.*
4. *A maximal run R of M such that for every maximal path ρ of R we have $tr(ev(\rho)) = tr(ev(\xi)\alpha)$, where ξ is a maximal path in T_0 and $\alpha \in G^\infty$.*

Proof. By Lem. 3, 4 and Prop. 2 we can assume that we have a maximal run $R = (X, \to, x_0, \ell, \text{ev})$ of M which satisfies the first condition of Prop. 2 and matches a path $\chi\chi_1^\omega$ in N. Let C denote the strongly connected component represented by χ_1. Let $G = E \setminus E_{\text{fin}}(R)$, then $G \subseteq \text{msg}(\chi_1)$ by Lem. 4. By assumption, the root x_0 of R belongs to $X_{\text{fin}}(R)$. Define first a tree $T_0' = (X_{\text{fin}}(R), \to, \ell, x_0)$ by letting $x \to y$ in T_0' whenever there is a path $x \to x_1 \to \cdots \to x_k \to y$ in R with $x_i \notin X_{\text{fin}}(R)$ for every i. The tree T_0 is now defined by identifying two vertices x, y in T_0' whenever $\ell(\pi(x)) = \ell(\pi(y))$. By Cond. (1) of Prop. 2 this step is well-defined. By Rem. 1 every maximal path ρ in R has the same causal structure as $\text{ev}(\pi_{\text{fin}}(\rho))\alpha$, for some $\alpha \in G^\infty$. Moreover, $\pi_{\text{fin}}(\rho)$ is a maximal path in T_0.

Remark 2. The length of the path χ in the previous proposition can be bounded by a polynomial in $|\mathcal{S}|$ and $|\mathcal{S}'|$, where $\mathcal{S}'$ is the set of states of N.

The next lemma implies that we can check the last condition of Prop. 3 in PSPACE.

Lemma 5. *Let $M = (\mathcal{S}, \tau, s_0, c, \mathcal{S}_\wedge, \mathcal{S}_\vee, \mathcal{P})$ be an and-or template graph. Suppose that we are given a state $r \in \mathcal{S}$, a set $L \subseteq \mathcal{S}$ of sink states in M and a set of events $I(s) \subseteq E$ for each $s \in L$, as well as a set $G \subseteq E$. Then we can check in PSPACE whether a maximal run $R = (X, \to, x_0, \ell, ev)$ of M exists satisfying the following conditions:*

1. *State r labels the root x_0 of R.*
2. *For every state $s \in L$ there is at least one leaf labelled by s. Moreover, every non-leaf node x is labelled by $ev(x) \in G$.*
3. *For every path $y \xrightarrow{+} x$ in R to a leaf x with $\ell(x) = s$ we have either $ev(y) \in I(s)$ or $y = x_0$.*

Theorem 1. *There is a PSPACE algorithm for deciding whether an and-or template graph M matches a system graph N.*

Proof. We have to check in PSPACE the existence of T_0, R, G, χ, C as in Prop. 3. Clearly, $G \subseteq E$ and C can be guessed and stored. By Rem. 2 the path χ can also be stored. The problem arises with T_0, since the size of T_0 might be exponential (however, the depth and the degree of T_0 are linear in $|\mathcal{S}|$, hence we can store paths of T_0.) The main idea is to guess the tree T_0 implicitly, in a DFS traversal where we store together with the current path also the (ordered sequence of) siblings of the intermediate nodes. Using Lem. 5 it is sufficient to verify the existence of a suitable run R piecewise, along with the DFS traversal.

The (nondeterministic) PSPACE algorithm works as follows (see also Fig. 4). Assume that the current path in T_0 is $s_0, \ldots, s_k$, $k < |\mathcal{S}|$, and that we also stored the (ordered) sequence L_i of siblings of s_i in T_0, $1 \leq i < k$. Moreover, we guessed sets of events $I(s) \subseteq E$ for all $s \in L_{i+1}$ and we verified the existence of maximal runs satisfying Lem. 5 with $r = s_i$, $L = L_{i+1}$, G. If $k < |\mathcal{S}|$ then we can either proceed with DFS and guess $s_{k+1}, L_{k+1}, \ldots$, or s_k is a leaf in T_0 and we apply Lem. 5 with $L = \emptyset$. Furthermore we verify that $(s_0, \ldots, s_k)$ is downward closed w.r.t. the causal order in the run built so far by using Lem. 5. For this, we use the sets $I(s_i)$. Using Lem. 5 we checked that a suitable run with root s_{i-1} exists s.t. every node on the path from s_{i-1} to s_i is labelled by an event from $I(s_i)$. Thus, it suffices to check for each $i < j \leq k$ and each $e' \in I(s_i)$ that $e' \prec\!\!\!\cdot\, \mathrm{ev}(s_j)$ is not satisfied. Finally, we verify that the path $(s_0, \ldots, s_k)$ matches χ and then we backtrack in the DFS traversal.

3.3 Specifying Properties by And-Or Templates

Consider a template given as an and-or MSC graph M. We want to determine the set of event sequences $L(M) \subseteq E^\infty$ which are described by M. That is, we let $\alpha \in L(M)$ if $\alpha = \mathrm{ev}(\chi)$ for some path χ in a system graph such that there is a matching of some maximal run of M into the path χ. Prop. 3 gives us the basic description of the set $L(M)$ as a finite union over languages $L(M, T_0, G) \subseteq E^\infty$, where T_0 is a tree labelled by states of M and $G \subseteq E$ is a set of events of M. Let $\alpha \in L(M, T_0, G)$ if

- every event $e \in G$ occurs infinitely often in α, and
- every path of T_0 matches α.

For any language $F \subseteq E^*$ we denote by $F \sqcup\!\sqcup E^*$ the shuffle ideal generated by F, i.e. $F \sqcup\!\sqcup E^* = \{v_0 u_1 v_1 \cdots u_n v_n \mid u_1 \cdots u_n \in F,\ v_i \in E^*\}$. For subsets $G \subseteq E$

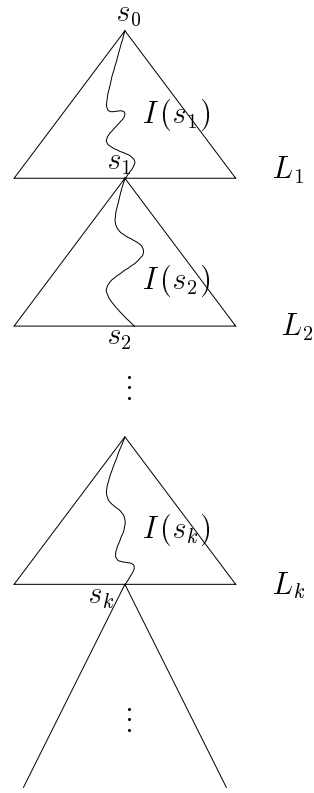


Fig. 4. Guessing the maximal run R, resp. the path $(s_0, \ldots, s_k)$ in T_0.

we denote by $\inf(G)$ the set of sequences $\alpha \in E^\infty$ where every event $e \in G$ occurs infinitely often. For $\alpha \in E^\infty$ we denote below by $\mathrm{tr}(\alpha) \subseteq E^\infty$ the set of linearizations of the causal order of α. Let $\mathrm{tr}(F) = \cup_{\alpha \in F} \mathrm{tr}(\alpha)$.

Proposition 4. *Let M be an and-or MSC template graph over the event set E. The language $L(M)$ associated with the template M has the form*

$$\bigcup_{F,G} (F \sqcup\!\sqcup E^*) \mathit{inf}(G)\,,$$

where $F \subseteq E^*$ is a finite set, $tr(F) = F$ and $G \subseteq E$.

Proof. Any language $L(M, T_0, G)$ as described above is a (finite) intersection of languages of the form $(\mathrm{tr}(\alpha) \sqcup\!\sqcup E^*)\inf(G)$, where α is such that $\mathrm{tr}(\alpha) \subseteq \mathrm{tr}(\beta)$ for some path χ in T_0 with $\beta = \mathrm{ev}(\chi)$. Let $\alpha, \beta \in E^*$, $G, H \subseteq E$. Then we have

$$(\mathrm{tr}(\alpha) \sqcup\!\sqcup E^*)\inf(G) \cap (\mathrm{tr}(\beta) \sqcup\!\sqcup E^*)\inf(H) = (\Gamma \sqcup\!\sqcup E^*)\inf(G \cup H)\,,$$

where Γ contains all sequences γ of minimal length such that for some $\alpha' \in \mathrm{tr}(\alpha)$, $\beta' \in \mathrm{tr}(\beta)$, both α', β' are subwords of γ. Moreover, it is not hard to check that Γ can be chosen s.t. $\mathrm{tr}(\Gamma) = \Gamma$.

4 Matching LTL Properties

In this section we consider properties (templates) specified by linear temporal logic. Our LTL formulas define finite or infinite strings over the alphabet $E = \{\mathsf{s}_{ij}, \mathsf{r}_{ij} \mid 1 \leq i \neq j \leq m\}$ of events. They are built over atomic propositions P_e, $e \in E$, using the temporal operators X (nexttime) and U (until), and the Boolean connectives $\wedge, \vee, \neg$.

Definition 7 (LTL Matching Problem). *Given a template M as an LTL formula ϕ and a system N as an MSC graph. We say that the template M matches the system N if there is some sequence of events $\alpha \in E^\omega$ such that $\alpha \models \phi$ and α matches some maximal path in N.*

Theorem 2. *The LTL matching problem is PSPACE-complete.*

We will show the above theorem in a more general setting using automata. Recall that every state of the system graph N is labelled by a single event. Thus we can easily associate to N a nondeterministic Büchi automaton $\mathcal{A}_N$ of the same size such that $L(\mathcal{A}_N)$ consists of all maximal executions of N. On the other hand it is well-known how to associate to every LTL formula ϕ an alternating Büchi automaton with $O(|\phi|)$ states accepting exactly the sequences satisfying ϕ, see e.g. [10]. Moreover, for every alternating Büchi automaton with n states there is an equivalent nondeterministic Büchi automaton with $2^{O(n)}$ states which can be built off-line using only $O(n)$ space, see [6]. We are thus led to a language-theoretical formulation of the LTL matching problem:

The matching problem for alternating automata: Given an alternating Büchi automaton $\mathcal{A}_M$ and a nondeterministic Büchi automaton $\mathcal{A}_N$ over E. Then we ask whether some sequences $\alpha \in L(\mathcal{A}_M)$, $\beta \in L(\mathcal{A}_N)$ exist such that α matches β.

Proposition 5. *The matching problem for alternating automata is PSPACE-complete.*

Proof. Due to the PSPACE-hardness of the satisfiability problem for LTL it suffices to show the upper bound. Let $\mathcal{A}_M$ denote an alternating Büchi automaton and let $\mathcal{B}_M = (Q, E, q_0, \delta, F)$ denote an equivalent nondeterministic Büchi automaton as given by [6]. Let $\mathcal{A}_N = (Q', E, q_0', \delta', F')$ denote the nondeterministic Büchi automaton representing the system. By an immediate modification and extension of Lem. 3 we note that M matches N if and only if final states $f \in F$, $f' \in F'$ exist such that

- for some paths $\pi = (q_0, \ldots, q_k = f)$ in $\mathcal{B}_M$, resp. $\pi' = (q_0', \ldots, q_l' = f')$ in $\mathcal{A}_N$, the execution of π matches the execution of π', i.e. $\mathrm{ev}(\pi)$ matches $\mathrm{ev}(\pi')$, and

– some loops ρ around f, resp. ρ' around f' exist satisfying $\mathrm{msg}(\rho) \subseteq \mathrm{msg}(\rho')$.

Clearly, we can choose π of length $k \leq |Q|$. But since $|Q| \in 2^{O(n)}$ we cannot guess π directly. On the other hand, we have to match π against π' and this means that we have to consider permutations, due to the causal order of the events. To overcome these problems we consider below the shape of π and π' in more details and exploit the fact that matching allows gaps.

We denote by $C_1, C_2, \ldots, C_m$ the maximal strongly connected components of the subgraph of $\mathcal{A}_N$ induced by π'. Thus, $\pi' = \pi'_1 \cdots \pi'_m$, such that w.l.o.g. π'_i induces C_i. The components C_i are naturally ordered, and we write $C_i < C_j$ for $i < j$.

Consider some mapping μ matching π into π'. We decompose π into maximal segments π_i, $\pi = \pi_1 \cdots \pi_r$, such that for every segment π_i all events of $\mathrm{ev}(\pi_i)$ are matched by μ into the same component π'_j, for some j. Let $C(i) \in \{1, \ldots, m\}$ denote the component C_j with $\mu(e) \in \pi'_j$ for all events e in $\mathrm{ev}(\pi_i)$. Note that for every $1 \leq i < j \leq r$, $C(i) > C(j)$ implies that $e \not\prec e'$ for all events e, e' with e occurring in $\mathrm{ev}(\pi_i)$, resp. e' occurring in $\mathrm{ev}(\pi_j)$. Otherwise, by the definition of matchings, we would require that $\mu(e) \prec \mu(e')$, contradicting $C(i) > C(j)$. The second basic observation is that every segment π_i matches *any* path ρ_i in $C(i)$ which visits every state of $C(i)$ sufficiently often. Therefore, we claim that a path π in $\mathcal{B}_M$ matches some path π' in $\mathcal{A}_N$ if and only if

– some maximal strongly connected components $C_1, \ldots, C_m$ exist in $\mathcal{A}_N$ with $q'_0 \in C_1$, as well as transitions from C_i to C_{i+1} for all i,
– π can be decomposed as $\pi = \pi_1 \cdots \pi_r$ such that for every i some component $C(i) \in \{C_1, \ldots, C_m\}$ exists such that $\mathrm{msg}(\pi_i) \subseteq \mathrm{msg}(C(i))$,
– for every $1 \leq i < j \leq r$ with $C(i) > C(j)$ we have $e \not\prec e'$ for all events e, e' occurring in $\mathrm{ev}(\pi_i)$, resp. in $\mathrm{ev}(\pi_j)$.

We already showed one direction of the claim. For the converse, assume that $C_i, \pi_j, C(k)$ exist as above. Let ρ_k denote the subpath of π consisting of all segments π_i with $C(i) = C_k$. Since $\mathrm{msg}(\rho_k) \subseteq \mathrm{msg}(C_k)$ and C_k is strongly connected, we can determine a path ρ'_k looping in C_k such that ρ_k matches ρ'_k (via some mapping μ). Moreover, ρ'_k can be chosen such that $\rho' = \rho'_1 \cdots \rho'_m$ is a path in $\mathcal{A}_N$. Finally the last condition above yields that π matches ρ'. This is seen by noting that for any events e, e' occurring in $\mathrm{ev}(\rho_i)$, resp. $\mathrm{ev}(\rho_j)$, $e \prec e'$ implies $i < j$, hence also $\mu(e) \prec \mu(e')$.

The PSPACE algorithm on input $\mathcal{A}_M, \mathcal{A}_N$ works as follows. First we guess $m \leq |Q'|$ strongly connected components $C_1, \ldots, C_m$ of $\mathcal{A}_N$, such that $q'_0 \in C_1$ and there exist edges from C_i to C_{i+1}, for all i. We guess the path π in $\mathcal{B}_M$ and its decomposition $\pi = \pi_1 \cdots \pi_r$ on-line. The only information which is (temporarily) stored concerns event types, which are required in order to check that $e \not\prec e'$ for all events e, e' occurring in $\mathrm{ev}(\rho_i)$, resp. $\mathrm{ev}(\rho_j)$, satisfying $i < j$ and $C(i) > C(j)$. For this step, we have to record at most m subsets of E. More precisely, assume that $E_1, \ldots, E_m \subseteq E$ are initially empty. Along with guessing π_i and some $C(i) = C_k$ we store $\mathrm{msg}(\pi_i) \subseteq E$ and check that for no events $e \in E_j$, $e' \in \mathrm{msg}(\pi_i)$ we have $e \prec e'$, whenever $C_k < C_j$ (i.e. $k < j$). Then we add $\mathrm{msg}(\pi_i)$ to E_k. Moreover, we verify that $\mathrm{msg}(\pi_i) \subseteq \mathrm{msg}(C(i))$.

5 Conclusion

In this paper we characterized the complexity of matching template MSCs which are given as and-or graphs, resp. by LTL formulas. Under these semantics the matching problem becomes PSPACE-complete. However, our proofs show that the increased complexity (compared with previously investigated template semantics, e.g. or-graphs) is due rather to a more succinct representation than to more expressiveness. This leads to the question whether using negations resp. requiring that certain events occur only finitely often might increase the expressiveness.

Acknowledgments. The referees are kindly acknowledged for the careful reading and the suggestions for improving the presentation of the paper.

References

1. R. Alur, G. Holzmann, and D. Peled. An analyzer for message sequence charts. *Software Concepts and Tools*, 17(2):70–77, 1996.
2. H. Ben-Abdallah and S. Leue. Syntactic detection of process divergence and non-local choice in message sequence charts. In E. Brinksma, editor, *Proceedings of the Tools and Algorithms for the Construction and Analysis of Systems, Third International Workshop, TACAS'97*, number 1217 in Lecture Notes in Computer Science, pages 259–274, Enschede, The Netherlands, 1997. Springer.
3. H. Ben-Abdallah and S. Leue. Mesa: Support for scenario-based design of concurrent systems. In B. Steffen, editor, *Proceedings of the Tools and Algorithms for the Construction and Analysis of Systems, 4th International Conference, TACAS'98*, number 1384 in Lecture Notes in Computer Science, pages 118–135, Lisbon, Portugal, 1998. Springer.
4. V. Diekert and G. Rozenberg, editors. *The Book of Traces.* World Scientific, Singapore, 1995.
5. A. Mazurkiewicz. Concurrent program schemes and their interpretations. DAIMI Rep. PB 78, Aarhus University, Aarhus, 1977.
6. S. Miyano and T. Hayashi. Alternating finite automata on ω-words. *Theoretical Computer Science*, 32:321–330, 1984.
7. A. Muscholl, D. Peled, and Z. Su. Deciding properties of message sequence charts. In M. Nivat, editor, *Proceedings of the 1st International Conference on Foundations of Software Science and Computation Structures (FoSSaCS'98), Lisbon, Portugal, 1998*, number 1378 in Lecture Notes in Computer Science, pages 226–242, Berlin-Heidelberg-New York, 1998. Springer.
8. M. Nielsen, G. Plotkin, and G. Winskel. Petri nets, event structures and domains, part 1. *Theoretical Computer Science*, 13:85–108, 1981.
9. V. R. Pratt. Modelling concurrency with partial orders. *International Journal of Parallel Programming*, 15(1):33–71, 1986.
10. M. Y. Vardi. An automata-theoretic approach to linear-temporal logic. In F. Moller and G. Birtwistle, editors, *Logics for concurrency*, number 1043 in Lecture Notes in Computer Science, pages 238–266, Berlin-Heidelberg-New York, 1996. Springer.

Probabilistic Temporal Logics via the Modal Mu-Calculus

Murali Narasimha[1] Rance Cleaveland[2] Purush Iyer[1]

[1] Dept of Computer Science, NC State University, Raleigh, NC 27695-7534.
[2] Dept of Computer Science, SUNY at Stony Brook, Stony Brook, NY 11794-4400.

Abstract. This paper presents a mu-calculus-based modal logic for describing properties of probabilistic labeled transition systems (PLTSs) and develops a model-checking algorithm for determining whether or not states in finite-state PLTSs satisfy formulas in the logic. The logic is based on the distinction between (probabilistic) "systems" and (non-probabilistic) "observations": using the modal mu-calculus, one may specify sets of observations, and the semantics of our logic then enable statements to be made about the measures of such sets at various system states. The logic may be used to encode a variety of probabilistic modal and temporal logics; in addition, the model-checking problem for it may be reduced to the calculation of solutions to systems of non-linear equations.

1 Introduction

Classical temporal-logic model checking [CES86, McM93] provides a basis for automatically checking the correctness of finite-state systems such as hardware designs and communication protocols. In this framework, systems are modeled as *transition systems*, and requirements are posed as formulas in temporal logic. A model checker then accepts two inputs, a transition system and a temporal formula, and returns "true" if the system satisfies the formula and "false" otherwise.

In traditional model checking, system models include information about the possible choices of execution steps in any given state. The corresponding temporal logics then combine a language for describing properties of system "runs" with quantifiers for indicating when all/some of the runs of a system have a given property [Koz83, EH86]. When system models include *probabilistic* information regarding their operational behavior, however, one frequently wishes to determine not just whether or not all/some system behaviors have a given property, but "how many" of them do. Many important questions of design and performance in distributed systems and communication protocols, such as "hot-spot" detection or reliability information, can be addressed more appropriately in such a probabilistic framework. Several examples of applying probabilistic model-checking to practical situations have been reported by Hansson [Han94]. Such motivations have led to the study of numerous probabilistic variants of temporal logic and model checking [ASB+95, BdA95, CY88, Han94, HK97, LS91, PZ93, Var85].

W. Thomas (Ed.): FOSSACS'99, LNCS 1578, pp. 288-305, 1999.

The goal of this paper is to develop a uniform framework for temporal logics for probabilistic systems. To this end, we show how the unifying classical temporal logic, the *modal mu-calculus* [Koz83, EL86], may be altered by adding probabilistic quantifiers constraining the "probability" with which probabilistic systems satisfy mu-calculus formulas. We then show how a variety of existing probabilistic logics may be represented in our framework and present a model-checking algorithm.

2 Probabilistic Transition Systems and the Logic GPL

This section introduces the model of probabilistic computation used in this paper and defines the syntax and semantics of our logic, Generalized Probabilistic Logic.

2.1 Reactive Probabilistic Labeled Transition Systems

We use the *reactive probabilistic labeled transition systems* (PLTS for short) of [vGSST90, LS91] as models of probabilistic computation. These are defined with respect to fixed sets Act and $Prop$ of atomic *actions* and *propositions*, respectively. The former set records the interactions the system may engage in with its environment, while the latter provides information about the states the system may enter.

Definition 1. A PLTS L is a tuple (S, δ, P, I), where

- $(s, s', s_1 \in) S$ is a countable set of states;
- $\delta \subseteq S \times Act \times S$ is the transition relation;
- $P : \delta \rightarrow (0, 1]$, the transition probability distribution, satisfies:

$$\sum_{(s,a,s') \in \delta} P(s, a, s') \in \{0, 1\}$$

  for all $s \in S$, $a \in Act$; and
- $I : S \rightarrow 2^{Prop}$ is the interpretation function.

Intuitively, a PLTS records the operational behavior of a system, with S representing the possible system states and δ the execution steps enabled in different system states; each such step is labeled with an action, and the intention is that when the environment of the system enables the action, the system may engage in a transition labeled by the action. When this is the case, $P(s, a, s')$ represents the probability with which the transition (s, a, s') is selected as opposed to other transitions labeled by a emanating from state s. Note that the conditions on P ensure that if $(s, a, s') \in \delta$ for some s', then $\sum_{(s,a,s') \in \delta} P(s, a, s') = 1$. In what follows we write $s \xrightarrow{a} s'$ if $(s, a, s') \in \delta$.

In this paper we wish to view a (state in a) PTLS as an "experiment" in the probabilistic sense, with an "outcome", or "observation", representing a resolution of all the possible probabilistic choices of transitions the system might

experience as it executes. More specifically, given a state in the PLTS we can unroll the PLTS into an infinite tree rooted at this state. An observation would then be obtained from this tree by resolving all probabilistic choices, i.e. by removing all but one edge for any given action from each node in the tree. Figure 1 presents a sample PLTS, its unrolling from a given state, and an associated observation.

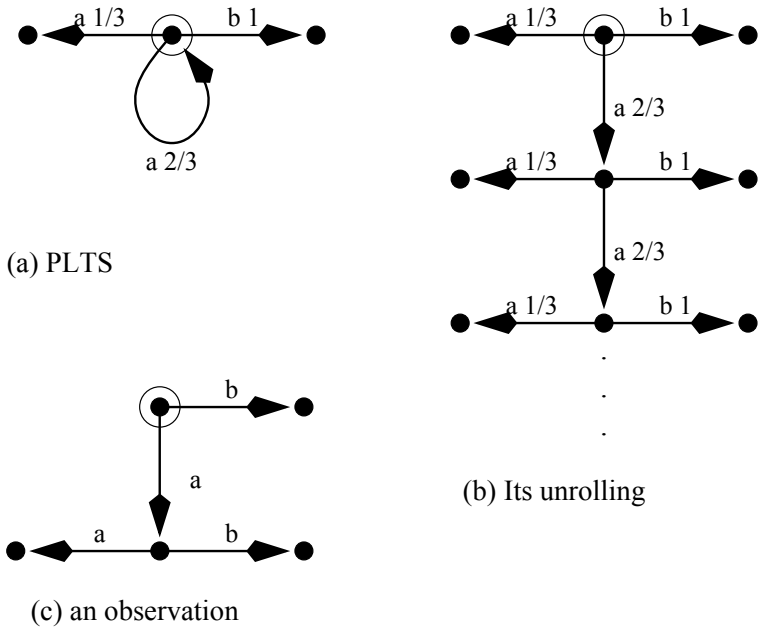


Fig. 1. A PLTS, its unrolling from a state, and an observation.

2.2 Syntax of GPL

Generalized Probabilistic Logic (GPL) is parameterized with respect to a set $(X, Y \in) Var$ of propositional variables, a set $(a, b \in) Act$ of actions, and a set $(A \in) Prop$ of atomic propositions. The syntax of GPL may then be given using the following BNF-like grammar, where $0 \leq p \leq 1$.

$$\phi ::= A \mid \neg A \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid Æ_{>p}\psi \mid Æ_{\geq p}\psi$$
$$\psi ::= \phi \mid X \mid \psi_1 \wedge \psi_2 \mid \psi_1 \vee \psi_2 \mid \langle a \rangle \psi \mid [a]\psi \mid \mu X.\psi \mid \nu X.\psi$$

The operators μ and ν bind variables in the usual sense, and one may define the standard notions of free and bound variables. Also, we refer to an occurrence of a bound variable X in a formula as a μ-occurrence if the closest enclosing binding operator for X is μ and as a ν-occurrence otherwise. GPL formulas are

required to satisfy the following additional restrictions: they must contain no free variables, and no sub-formula of the form $\mu X.\psi$ ($\nu X.\psi$) may contain a free ν-occurrence (μ-occurrence) of a variable.[3] In what follows we refer to formulas generated from nonterminal ϕ etc. as *state formulas* and those generated from ψ as *fuzzy formulas*; the formulas of GPL are the state formulas. We use $(\phi, \phi' \in)\Phi$ to represent the set of all state formulas and $(\psi, \psi' \in)\Psi$ for the set of all fuzzy formulas. In the remainder of the paper we write $\gamma[\gamma'/X]$ to denote the the simultaneous substitution of γ' for all free occurrences of X in γ. We also note that although the logic limits the application of $\neg$ to atomic propositions, this does not restrict the expressiveness of the logic, as we indicate later.

The next subsection defines the formal semantics of GPL, but the intuitive meanings of the operators may be understood as follows. Fuzzy formulas are to be interpreted as specifying sets of *observations* of PLTSs, which are themselves non-probabilistic trees as discussed above. An observation is in the set corresponding to the fuzzy formula if the root node of the observation satisfies the formula interpreted as a traditional mu-calculus formula: so $\langle a \rangle \psi$ holds of an observation if the root has an a-transition leading to the root of an an observation satisfying ψ, while it satisfies $[a]\psi$ if every a-transition leads to such an observation. Conjunction and disjunction have their usual interpretation. $\mu X.\psi$ and $\nu X.\psi$ are fixpoint operators describing the "least" and "greatest" solutions, respectively, to the "equation" $X = \psi$. It will turn out that any state in a given PLTS defines a probability space over observations and that our syntactic restrictions ensure that the sets of observations defined by any fuzzy formula are *measurable* in a precise sense. State formulas will then be interpreted with respect to states in PLTSs, with a state satisfying a formula of the form $Æ_{\geq p}\psi$ if the measure of observations corresponding to the state is at least p.

2.3 Semantics of GPL

This subsection formalizes the notions described informally above. We first define observations of a PLTS and show how the observations from a given state in a PLTS form a probability space. We then use these probability spaces to interpret GPL formulas. In what follows we fix sets Act and $Prop$.

PLTSs and Measure Spaces of Observations To define the observation trees of a PLTS we introduce *partial computations*, which will form the nodes of the trees.

Definition 2. Let $L = (S, \delta, P, I)$ be a PLTS. Then a sequence of the form $s_0 \xrightarrow{a_1} s_1 \cdots \xrightarrow{a_n} s_n$ is a *partial computation* of L if $n \geq 0$ and for all $0 \leq i < n$, $s_i \xrightarrow{a_{i+1}} s_{i+1}$.

Note that any $s \in S$ is a partial computation. If $\sigma = s_0 \xrightarrow{a_1} s_1 \cdots \xrightarrow{a_n} s_n$ is a partial computation then we define $\mathsf{fst}(\sigma)$ to be s_0 and $\mathsf{lst}(\sigma)$ to be s_n. We also

[3] In other words, formulas must be alternation-free in the sense of [EL86].

use $(\sigma, \sigma' \in)\mathcal{C}_L$ to refer to the set of all partial computations of L and take $\mathcal{C}_L(s) = \{\sigma \in \mathcal{C}_L \mid \mathsf{fst}(\sigma) = s\}$ for $s \in S$. We define the following notations for partial computations.

Definition 3. Let $\sigma = s_0 \xrightarrow{a_1} s_1 \cdots \xrightarrow{a_n} s_n$ and $\sigma' = s'_0 \xrightarrow{a'_1} s'_1 \cdots \xrightarrow{a_{n'}} s'_{n'}$ be partial computations of PLTS $L = (S, \delta, P, I)$, and let $a \in Act$.

1. If $s_n \xrightarrow{a} s'_0$ then $\sigma \xrightarrow{a} \sigma'$ is the partial computation $s_0 \xrightarrow{a_1} s_1 \cdots \xrightarrow{a_n} s_n \xrightarrow{a} s'_0 \xrightarrow{a'_1} s'_2 \cdots \xrightarrow{a_{n'}} s'_{n'}$.
2. σ' is a *prefix* of σ if $\sigma' = s_0 \xrightarrow{a_1} s_1 \cdots \xrightarrow{a_i} s_i$ for some $i \leq n$.

We also introduce the following terminology for sets of partial computations.

Definition 4. Let $L = (S, \delta, P, I)$ be a PLTS, and let $C \subseteq \mathcal{C}_L$ be a set of computations.

1. C is *prefix-closed* if, for every $\sigma \in C$ and σ' a prefix of σ, $\sigma' \in C$.
2. C is *deterministic* if for every $\sigma, \sigma' \in C$ with $\sigma = s_0 \xrightarrow{a_1} s_1 \cdots \xrightarrow{a_n} s_n \xrightarrow{a} s \cdots$ and $\sigma' = s_0 \xrightarrow{a_1} s_1 \cdots \xrightarrow{a_n} s_n \xrightarrow{a'} s' \cdots$, either $a \neq a'$ or $s = s'$.

The term prefix-closed is standard, but the notion of determinacy of sets of partial computations deserves some comment. Intuitively, if two computations in a deterministic set of partial computations share a common prefix, then the first difference they can exhibit must involve transitions labeled by different actions; they cannot involve different transitions with the same action label.

We can now define the deterministic trees, or *d-trees*, of a PLTS L as follows.

Definition 5. Let $L = (S, \delta, P, I)$ be a PLTS. Then $\emptyset \neq T \subseteq \mathcal{C}_L$ is a *d-tree* if the following hold.

1. There exists an $s \in S$ such that $T \subseteq \mathcal{C}_L(s)$.
2. T is prefix-closed.
3. T is deterministic.

If C is a d-tree then we use $\mathsf{root}(C)$ to refer to the s such that $C \subseteq \mathcal{C}_L(s)$ and $\mathsf{edges}(C)$ to refer to the relation $\{(\sigma, a, \sigma') \mid \sigma, \sigma' \in C \wedge \exists s' \in S. \sigma' = \sigma \xrightarrow{a} s\}$.

We use $\mathcal{T}_L$ to refer to all the d-trees of L and set $\mathcal{T}_L(s) = \{T \in \mathcal{T}_L \mid \mathsf{root}(T) = s\}$. We call T' a *prefix* of T if $T' \subseteq T$. We write $T \xrightarrow{a} T'$ if $\{\mathsf{root}(T) \xrightarrow{a} \sigma' \mid \sigma' \in T'\} \subseteq T$; intuitively, T' is then the subtree of T pointed to by an a-labeled edge. A d-tree T is *finite* if $|T| < \infty$. Finally, we say that a d-tree is *maximal* if there exists no d-tree T' with $T \subset T'$ and use $\mathcal{M}_L$ and $\mathcal{M}_L(s)$ to refer to the set of all maximal d-trees of L and all maximal d-trees of L rooted at s, respectively.

We wish to view the maximal deterministic d-trees of a PLTS as the "outcomes" of the PLTS and to talk about the likelihoods of different sets of outcomes. In order to do this, we define a probability space over maximal d-trees rooted at a given state of L. The construction of this space is very similar in spirit to the standard sequence space construction for Markov chains [KSK66]:

we define a collection of "basic cylindrical sets" of maximal trees and use them to build a probability space over sets of maximal trees. The technical details appear below; in what follows, fix $L = (S, \delta, P, I)$.

A *basic cylindrical subset* of $\mathcal{M}_L(s)$ contains all trees sharing a given finite prefix.

Definition 6. Let $s \in S$, and let $T \in \mathcal{T}_L(s)$ be finite. Then $B_T \subseteq \mathcal{M}_L(s)$ is defined as: $B_T = \{\, T' \in \mathcal{M}_L \mid T \subseteq T' \}$.

We can also define the *measure* of a basic cylindrical set as follows.

Definition 7. Let $T \in \mathcal{T}_L(s)$ be finite, and let B_T be the associated basic cylindrical set. Then the *measure*, $\mathsf{m}(B_T)$, of B_T is given by:

$$\mathsf{m}(B_T) = \Pi_{(\sigma,a,\sigma') \in \mathsf{edges}(T)} P(\mathsf{lst}(\sigma), a, \mathsf{lst}(\sigma')).$$

Intuitively, $\mathsf{m}(B_T)$ represents the proportion of all maximal d-trees emanating from the root of B_T that have B_T as a prefix.

For any given state s in L we can form the associated collection of basic cylindrical sets $\mathcal{B}_s^-$ consisting of sets of the form B_T for finite T with $\mathsf{root}(T) = s$. We can then define a probability space $(\mathcal{M}_L(s), \mathcal{B}_s, \mathsf{m}_s)$ as follows.

Definition 8. Let $s \in S$. Then $\mathcal{B}_s$ is the smallest field of sets containing $\mathcal{B}_s^-$ and closed with respect to denumerable unions and complementation. $\mathsf{m}_s : \mathcal{B}_s \to [0,1]$ is then defined inductively as follows.

$$\begin{aligned} \mathsf{m}_s(B_T) &= \mathsf{m}(B_T) \\ \mathsf{m}_s(\bigcup_{i \in I} B_i) &= \sum_{i \in I} \mathsf{m}_s(B_i) \quad \text{for pairwise disjoint } B_i \\ \mathsf{m}_s(B^c) &= 1 - \mathsf{m}_s(B) \end{aligned}$$

It is easy to show that for any s, m_s is a probability measure over $\mathcal{B}_s$. Consequently, $(\mathcal{M}_L(s), \mathcal{B}_s, \mathsf{m}_s)$ is indeed a probability space. We refer to a set $M \subseteq \mathcal{M}_L(s)$ as *measurable* if $M \in \mathcal{B}_s$.

Semantics of Fuzzy Formulas In the remainder of this section we define the semantics of GPL formulas with respect to a fixed PLTS $L = (S, \delta, P, I)$ by giving mutually recursive definitions of a relation $\models_L \subseteq S \times \Phi$ and a function $\Theta_L : \Psi \to 2^{\mathcal{M}_L}$. The former indicates when a state satisfies a state formula, while the latter returns the set of maximal d-trees satisfying a given fuzzy formula. In this subsection we present Θ_L; the next subsection then considers $\models_L$. In what follows we fix $L = (S, \delta, P, I)$.

Our intention in defining $\Theta_L(\psi)$ is that it return trees that, interpreted as (non-probabilistic) labeled transition systems, satisfy ψ interpreted as a mu-calculus formula. To this end, we augment Θ_L with an extra environment parameter $e : Var \to 2^{\mathcal{M}_L}$ that is used to interpret free variables. The formal definition of Θ_L is the following.

Definition 9. The function Θ_L is defined inductively as follows.

- $\Theta_L(\phi)e = \cup_{s \models_L \phi} \mathcal{M}_L(s)$
- $\Theta_L(X)e = e(X)$
- $\Theta_L(\langle a \rangle \psi)e = \{T \mid \exists T' : T \xrightarrow{a} T' \wedge T' \in \Theta_L(\psi)e\}$
- $\Theta_L([a]\psi)e = \{T \mid (T \xrightarrow{a} T') \Rightarrow T' \in \Theta_L(\psi)e\}$
- $\Theta_L(\psi_1 \wedge \psi_2)e = \Theta_L(\psi_1)e \cap \Theta_L(\psi_2)e$
- $\Theta_L(\psi_1 \vee \psi_2)e = \Theta_L(\psi_1)e \cup \Theta_L(\psi_2)e$
- $\Theta_L(\mu X.\psi)e = \cup_{i=0}^{\infty} M_i$, where $M_0 = \emptyset$ and $M_{i+1} = \Theta_L(\psi)e[X \mapsto M_i]$.
- $\Theta_L(\nu X.\psi)e = \cap_{i=0}^{\infty} N_i$, where $N_0 = \mathcal{M}_L$ and $N_{i+1} = \Theta_L(\psi)e[X \mapsto N_i]$.

When ψ has no free variables, $\Theta(\psi)e = \Theta(\psi)e'$ for any environments e, e'. In this case we drop the environment e and write $\Theta_L(\psi)$.

Some comments about this definition are in order. Firstly, it is straightforward to show that the semantics of all the operators except μ and ν are those that would be obtained by interpreting maximal deterministic trees as labeled transition systems and fuzzy formulas as mu-calculus formulas in the usual style [Koz83]. Secondly, because d-trees are deterministic it follows that if $T \in \Theta_L(\langle a \rangle \psi)$ then $T \in \Theta_L([a]\psi)$. Finally, the definitions we have given for μ and ν differ from the more general accounts that rely on the Tarski-Knaster fixpoint theorem. However, because of the "alternation-free" restriction we impose on our logic and the fact that d-trees are deterministic, the meanings of $\mu X.\psi$ and $\nu X.\psi$ are still least and greatest fixpoints in the usual sense.

We close this section by remarking on an important property of Θ_L. For a given $s \in S$ let $\Theta_{L,s}(\psi) = \Theta_L(\psi) \cap \mathcal{M}_L(s)$ be the maximal d-trees from s "satisfying" ψ. We have the following.

Theorem 10. *For any $s \in S$ and $\psi \in \Psi$, $\Theta_{L,s}(\psi)$ is measurable*[4].

Semantics of State Formulas We now define the semantics of state formulas by defining the relation $\models_L$.

Definition 11. Let $L = (S, \delta, P, I)$ be a PLTS. Then $\models_L$ is defined inductively as follows.

- $s \models^e_L A$ iff $A \in I(s)$.
- $s \models^e_L \neg A$ iff $A \notin I(s)$.
- $s \models^e_L \phi_1 \wedge \phi_2$ iff $s \models \phi_1$ and $s \models \phi_2$.
- $s \models^e_L \phi_1 \vee \phi_2$ iff $s \models \phi_1$ or $s \models \phi_2$.
- $s \models^e_L Æ_{>p} \psi$ iff $\mathsf{m}_s(\Theta_{L,s}(\psi)e) > p$.
- $s \models^e_L Æ_{\geq p} \psi$ iff $\mathsf{m}_s(\Theta_{L,s}(\psi)e) \geq p$.

An atomic proposition is satisfied by a state if the proposition is a member of the propositional labeling of the state. Conjunction and disjunction are interpreted in the usual manner, while a state satisfies a formula $Æ_{>p}\psi$ iff the measure of the observations of ψ rooted at s exceeds p, and similarly for $Æ_{\geq p}\psi$.

[4] The question of whether the observations of non-alternation free formula are measurable is still open

Properties of the Semantics We close this section by remarking on some of the properties of GPL. The first shows that the modal operators for fuzzy formulas enjoy certain distributivity laws with respect to the propositional operators.

Lemma 12. *For a PLTS L, fuzzy formulas ψ_1 and ψ_2 and $a \in Act$, we have:*

1. $\Theta_L(\langle a\rangle(\psi_1 \vee \psi_2)) = \Theta_L(\langle a\rangle\psi_1 \vee \langle a\rangle\psi_2)$
2. $\Theta_L([a](\psi_1 \vee \psi_2)) = \Theta_L([a]\psi_1 \vee [a]\psi_2)$
3. $\Theta_L(\langle a\rangle(\psi_1 \wedge \psi_2)) = \Theta_L(\langle a\rangle\psi_1 \wedge \langle a\rangle\psi_2)$
4. $\Theta_L([a](\psi_1 \wedge \psi_2)) = \Theta_L([a]\psi_1 \wedge [a]\psi_2)$
5. $\Theta_L([a]\psi_1 \wedge \langle a\rangle\psi_2) = \Theta_L(\langle a\rangle(\psi_1 \wedge \psi_2))$

That $[a]$ distributes over $\vee$ and $\langle a\rangle$ over $\wedge$ is due to the determinacy of d-trees.

Based on Theorem 10 and the definition of Θ_L, the next lemma also holds.

Lemma 13. *Let $s \in S$, $a \in Act$ and $\psi, \psi_1, \psi_2 \in \Psi$. Then we have the following.*

$$\mathsf{m}_s(\Theta_L(\psi_1 \vee \psi_2)) = \mathsf{m}_s(\Theta_L(\psi_1)) + \mathsf{m}_s(\Theta_L(\psi_2)) - \mathsf{m}_s(\Theta_L(\psi_1 \wedge \psi_2)) \quad (1)$$

$$\mathsf{m}_s(\Theta_L(\langle a\rangle\psi)) = \sum_{(s,a,s')\in\delta} P(s,a,s') * \mathsf{m}_{s'}(\Theta_L(\psi)) \quad (2)$$

$$\mathsf{m}_s(\Theta_L([a]\psi)) = \begin{cases} \mathsf{m}_s(\Theta_L(\langle a\rangle\psi)) & \text{if } (s,a,s') \in \delta \text{ for some } s' \\ 1 & \text{otherwise} \end{cases} \quad (3)$$

Finally, although our logic only allows a restricted form of negation, we do have the following.

Lemma 14. *Let $L = (S, \delta, P, I)$ be a PLTS with $s \in S$, and let ψ and ϕ be fuzzy and state formulas, respectively. Then there exist formulas $\mathsf{neg}(\psi)$ and $\mathsf{neg}(\phi)$ such that:*

$$\Theta_{L,s}(\mathsf{neg}(\psi)) = \mathcal{M}_L(s) - \Theta_{L,s}(\psi) \quad \text{and} \quad s \models_L \mathsf{neg}(\phi) \Leftrightarrow s \not\models_L \phi.$$

Proof. Follows from the duality of $\wedge/\vee$, $[a]/\langle a\rangle$, ν/μ, and $Æ_{>p}/Æ_{\geq 1-p}$.

3 Expressiveness of GPL

In this section we illustrate the expressive power of GPL by showing how three quite different probabilistic logics may be encoded within it.

3.1 Encoding Probabilistic Modal Logic

Probabilistic Modal Logic (PML) [LS91] is a probabilistic version of Hennessy-Milner logic [HM85] that has been shown to characterize probabilistic bisimulation equivalence over PLTSs. The formulas of the logic are generated by the following grammar:

$$\phi ::= A \mid \phi_1 \wedge \phi_2 \mid \neg\phi \mid \langle a\rangle_p \phi$$

where $0 \leq p \leq 1$, $A \in Prop$ and $a \in Act$. Formulas are interpreted with respect to states in a given PLTS $L = (S, \delta, P, I)$ via a relation $\models_L^{PML} \subseteq S \times \phi$. The definition appears below; the cases for $\neg$ and $\wedge$ have been omitted.

$$\begin{array}{ll} s \models_L^{PML} A & \text{iff } A \in I(s) \\ s \models_L^{PML} \langle a \rangle_p \phi & \text{iff } \sum_{\{s' \mid (s,a,s') \in \delta \wedge s' \models_L^{PML} \phi\}} P(s,a,s') \geq p \end{array}$$

Note that a state s satisfies $\langle a \rangle_p \phi$ provided that the probability of taking an a-transition to a state satisfying ϕ is at least p. This observation suggests the following encoding function E_{PML} for translating PML formulas into GPL formulas.

$$E_{PML}(\phi) = \begin{cases} \phi & \text{if } \phi \in Prop \\ E_{PML}(\phi_1) \wedge E_{PML}(\phi_2) & \text{if } \phi = \phi_1 \wedge \phi_2 \\ \mathsf{neg}(E_{PML}(\phi')) & \text{if } \phi = \neg \phi' \\ Æ_{\geq p} \langle a \rangle E_{PML}(\phi') & \text{if } \phi = \langle a \rangle_p \phi' \end{cases}$$

In essence, the translation effectively replaces all occurrences of $\langle a \rangle_p$ by $Æ_{\geq p}$. We have the following.

Theorem 15. *Let ϕ be a PML formula and s be a state of PLTS L. Then $s \models_L^{PML} \phi$ iff $s \models E_{PML}(\phi)$.*

3.2 Encoding pCTL*

pCTL* [ASB+95] represents a probabilistic variant of the temporal logic CTL* [EH86]. The latter logic is interpreted with respect to Kripke structures; the former is interpreted with respect to structures referred to in [ASB+95] as *Markov processes* (MP), which may be viewed as probabilistic Kripke structures. It turns out that MPs form a subclass of PLTSs. This section will show that pCTL* has a uniform encoding in GPL.

A Markov process may be seen as a PLTS having only one action and in which every state has at least one outgoing transition.

Definition 16. Let $Act = \{a\}$. Then a Markov process (MP) is a PLTS (S, δ, P, I) such that for any $s \in S$, $\sum_{\{s' \mid (s,a,s') \in \delta\}} P(s,a,s') = 1$.

It is straightforward to see that the d-trees of a MP are in fact isomorphic to sequences of states from the MP: a sequence $\pi = s_0 s_1 \ldots$ coincides with the d-tree $\{\sigma_0, \sigma_1, \ldots\}$, where $\sigma_0 = s_0$ and $\sigma_{i+1} = \sigma_i \xrightarrow{a} s_{i+1}$. It then turns out that the measure space of d-trees for a state in a MP coincides with the standard sequence space construction for Markov chains [KSK66]. Consequently, in the following we will use the function m_s to refer to the measure of both sets of sequences and sets of d-trees. We also use the following notations on infinite sequences $\pi = s_0 s_1 \ldots$: $\pi[i] = s_i$, and $\pi^i = s_i s_{i+1} \ldots$.

Interpreting GPL over Markov Processes As every state in a MP has an outgoing transition, the semantics of the GPL constructs $\langle a \rangle \psi$ and $[a]\psi$ coincide. That is, when M is a MP following from Definition 9 implies that $\Theta_M(\langle a \rangle \psi) = \Theta_M([a]\psi)$.

In the rest of this subsection we will show that pCTL* can be encoded in GPL. What makes the encoding possible are that:

- The logic GPL is a two-level logic, much like CTL* and pCTL*. Consequently, probabilistic quantifiers in pCTL* formulae can be translated to probabilistic quantifier of GPL formulae.
- The semantics of fuzzy formulae are sets of sequences, when the model is a Markov chain, and thus, fuzzy formulae play the role of linear-time μ-calculus formulae. Given that alternation-free linear time modal μ-calculus is as expressive as linear time temporal logic (LTL) [Sti92], the LTL portion of pCTL* (i.e., the path formulae of pCTL*) can be embedded into fuzzy formulae.

This encoding contrasts with the encoding of CTL* into modal μ-calculus [EL86], where alternation is needed in the translation; the reason being that, unlike GPL, modal μ-calculus does not have path quantifiers.

pCTL* Let $(A \in) Prop$ be a set of atomic propositions. The grammar below summarizes the syntax of pCTL*, which has two levels—*state* formulas (ϕ) and *path* formulas (ψ). State formulas specify properties that hold in states of a MP while path formulae specify properties of execution sequences.

$$\phi ::= A \mid \neg\phi \mid \phi_1 \vee \phi_2 \mid Pr_{<p}\psi \mid Pr_{>p}\psi$$
$$\psi ::= \phi \mid \neg\psi \mid \psi_1 \vee \psi_2 \mid \mathsf{X}\psi \mid \psi_1 \mathsf{U} \psi_2$$

Here $Pr_{>p}$ and $Pr_{<p}$ are probabilistic quantifiers, while X denotes the next-state and U the until operator, respectively

The semantics of pCTL* formulas is given with respect to a MP $M = (S, \delta, P, I)$ via a relation $\models_M$ relating states in M to state formulas, and paths (infinite state sequences) in M to path formulas. The interpretations of $\neg$ and $\vee$ are standard, and we omit them; what follows defines the meanings of the other operators.

$$\begin{array}{lll}
s \models_M^{pCTL^*} A & \text{iff } A \in I(s) \\
s \models_M^{pCTL^*} Pr_{<p}\psi & \text{iff } \mathsf{m}_s(\{\pi \mid \pi \models_M^{pCTL^*} \psi\}) < p \\
s \models_M^{pCTL^*} Pr_{>p}\psi & \text{iff } \mathsf{m}_s(\{\pi \mid \pi \models_M^{pCTL^*} \psi\}) > p \\
\pi \models_M^{pCTL^*} \phi & \text{iff } \pi[0] \models_M^{pCTL^*} \phi \\
\pi \models_M^{pCTL^*} \mathsf{X}\psi & \text{iff } \pi^1 \models_M^{pCTL^*} \psi \\
\pi \models_M^{pCTL^*} \psi_1 \mathsf{U} \psi_2 & \text{iff } \exists k \geq 0 : \pi^k \models_M^{pCTL^*} \psi_2 \wedge \forall j : 0 \leq j < k : \pi^k \models_M^{pCTL^*} \psi_1
\end{array}$$

Our encoding of pCTL* in GPL translates state formulas into state formulas and path formulas into fuzzy ones. Our approach relies on the following recursive

characterization of U: $\pi \models_M \psi_1 \,\mathsf{U}\, \psi_2$ iff $\pi \models_M \psi_2 \vee (\psi_1 \wedge \mathsf{X}(\psi_1 \,\mathsf{U}\, \psi_2))$. The encoding may now be given as a function E_{pCTL^*} as follows, where γ is either a state formula or a path formula.

$$E_{pCTL^*}(\gamma) = \begin{cases} \gamma & \text{if } \gamma \in Prop \\ \mathsf{neg}(E_{pCTL^*}(\gamma')) & \text{if } \gamma = \neg\gamma' \\ E_{pCTL^*}(\gamma_1) \vee E_{pCTL^*}(\gamma_2) & \text{if } \gamma = \gamma_1 \vee \gamma_2 \\ Æ_{\geq p}\mathsf{neg}(E_{pCTL^*}(\psi)) & \text{if } \gamma = Pr_{<p}\psi \\ Æ_{>p}E_{pCTL^*}(\psi) & \text{if } \gamma = Pr_{>p}\psi \\ \langle a\rangle E_{pCTL^*}(\psi) & \text{if } \gamma = \mathsf{X}\psi \\ \mu X.(E_{pCTL^*}(\psi_2) \vee (E_{pCTL^*}(\psi_1) \wedge \langle a\rangle X)) & \text{if } \gamma = \psi_1 \,\mathsf{U}\, \psi_2 \end{cases}$$

We now have the following.

Theorem 17. *Let M be a MP, let s be a state in M, and let π be a path in M. Then:*

1. *For any pCTL* state formula ϕ, $s \models_M^{pCTL^*} \phi$ iff $s \models_M E_{pCTL^*}(\phi)$.*
2. *For any pCTL* path formula ψ, $\pi \models_M^{pCTL^*} \psi$ iff $\pi \in \Theta_M(E_{pCTL^*}(\psi))$*

3.3 Reconstructing the Logic of Huth and Kwiatkowska

Huth and Kwiatkowska develop a notion of *quantitative model checking* [HK97] in which one calculates the likelihood with which a system state satisfies a formula. The basis for their approach lies in a semantics for the modal mu-calculus that assigns "probabilities", rather than truth values, to assertions about states in a PLTS. In this section we briefly review their approach, offer a criticism of it, and show how GPL provides a principled means of remedying the criticism.

The syntax of their logic coincides with the semantics of our fuzzy formulas with the following exceptions: (1) they allow negation (although in such a way that negations can be eliminated in the usual manner); (2) the only atomic propositions are tt ("true") and ff ("false"); (3) no use of the probabilistic quantifiers $Æ_{\geq p}$ and $Æ_{>p}$ is allowed. They then present three semantics for the logic that differ only in their interpretation of conjunction. Each interprets formulas as functions mapping states to numbers in $[0,1]$; formally, given PLTS L, $[\![\psi]\!]_L : S \to [0,1]$ represents the interpretation of formula ψ. What follows presents the relevant portions of these semantics.

$$\begin{aligned} [\![\mathsf{tt}]\!]_L(s) &= 1 \\ [\![\langle a\rangle\psi]\!]_L(s) &= \sum_{s' \in \delta(s,a)} P(s,a,s') \cdot [\![\psi]\!]_L(s') \\ [\![\psi_1 \wedge \psi_2]\!]_L(s) &= f([\![\psi_1]\!]_L(s), [\![\psi_2]\!]_L(s)) \end{aligned}$$

The meanings of the other boolean and modal operators may be obtained using dualities (e.g. $[\![[a]\psi]\!]_L(s) = 1 - ([\![\langle a\rangle\neg\psi]\!])$, while the meanings of fixed points may be obtained using the usual Tarski-Knaster construction. The semantics of $\wedge$ contains a parameter f; [HK97] provides three different instantiations of f.

1. $f(x,y) = \min(x,y)$
2. $f(x,y) = x \cdot y$
3. $f(x,y) = \max(x+y-1, 0)$

Each unfortunately has its drawbacks. The first two fail to validate some expected logical equivalences; for example it not the case that tt is equivalent to $\psi \vee \neg\psi$. The authors refer to the third as a "fuzzy" interpretation and indicate that it is intended only to provide a "lower approximation" on probabilities; "real" probabilities are therefore not calculated.

GPL permits a similar interpretation to be attached to the mu-calculus, but in such a way that exact probabilities may be assigned to formulas. Consider the function $[\![\psi]\!]_L^{GPL}$ given by:

$$[\![\psi]\!]_L^{GPL}(s) = \mathsf{m}_s(\Theta_L(\psi)).$$

One can show that this interpretation preserves much of the semantics of Huth and Kwiatkowska; in particular, Lemmas 13 and 14 show that this definition attaches the same interpretations to the modalities. It is also the case that expected logical equivalences hold, and that this interpretation yields a probability with a precise, measure-theoretic interpretation. Finally, it should be easy to observe that our logic coincides with probabilistic bisimulation [LS91] – a property not true of Huth and Kwiatkowska's interpretation.

4 Model Checking

This section now describes a procedure for determining whether or not a given state in a finite-state PLTS satisfies a GPL formula. We present the algorithm in two stages. The first shows how to calculate the measure of observations that are rooted at a given PLTS state and satisfy a fuzzy formula; the second then shows how this routine may be used to implement full GPL model checking. We assume that the formulas to be considered have no *unguarded* occurrences of bound variables. That is, in every sub-formula of the form $\sigma X.\psi$, where σ is either μ or ν, each occurrence of X in ψ falls within the scope of a $\langle a \rangle$ or a $[a]$ operator. Any mu-calculus formula may be transformed into one satisfying this restriction. In the remainder of this section we fix a specific PLTS $L = (S, \delta, P, I)$.

4.1 Computing the Measure of Fuzzy Formulas

This subsection sketches a procedure *modchk-fuzzy* whose task is to compute $\mathsf{m}_{s_0}(\Theta_L(\psi))$ for a given fuzzy formula ψ and a state s_0 of the PLTS. The algorithm consists of the following steps.

1. From L, s_0 and ψ, construct a dependency graph.
2. From the graph, extract a system of (non-linear) *measure* equations.
3. Calculate a specific solution to these equations; one of the results will be $\mathsf{m}_{s_0}(\Theta_L(\psi))$.

The remainder of this subsection describes each of these steps in more detail, with intuitive explanations for why the constructions work.

A graph construction. The first step in *modchk-fuzzy* involves constructing a graph that describes the relationship between the quantity $\mathsf{m}_{s_0}(\Theta_L(\psi))$ that we wish to compute and quantities of the form $\mathsf{m}_s(\Theta_L(\psi'))$, where s is a derivative of s_0 and ψ' a formula derived appropriately from ψ. This graph will have vertices of the form (s, F), where $s \in S$ and F is a set of fuzzy formulas. The edges from (s, F) then provide "local" information regarding $\mathsf{m}_s(\Theta_L(\wedge F))$.

In order to define the graph formally we need the following notions.

Definition 18. For a closed fuzzy formula ψ define the closure, written as $Cl(\psi)$, as the smallest set of formulae satisfying the following rules:

- $\psi \in Cl(\psi)$
- if $\psi' = \psi_1 \wedge \psi_2$ or $\psi_1 \vee \psi_2$ then $\psi_1, \psi_2 \in Cl(\psi)$
- if $\psi' = \langle a \rangle \psi''$ or $[a]\psi''$ for some $a \in Act$, then $\psi'' \in Cl(\psi)$
- if $\psi' = \sigma X.\psi''$ then $\psi''[\sigma X.\psi''/X] \in Cl(\psi)$ (σ is either μ or ν)

One may easily show that $Cl(\psi)$ contains no more elements than ψ contains sub-formulas.

The node set N in the graph is the set $S \times 2^{Cl(\psi)}$; that is,nodes have form (s, F), where $s \in S$ and $F \subseteq Cl(\psi)$. We further introduce the following classification on nodes.

- (s, F) is a *true node* if $F = \emptyset$ or if every element of F has form $[a]\psi'$ and for every such a, s is incapable of an a-transition.
- (s, F) is a *false node* if there exists a state formula $\phi \in F$ with $s \not\models_L \phi$ or if there exists a formula of the form $\langle a \rangle \psi'$ and s is incapable of an a-transition.
- (s, F) is an *and-node* if there exists a formula $\psi_1 \wedge \psi_2 \in F$.
- (s, F) is an *action-node* if every formula in F has form $\langle a \rangle \psi'$ or $[a]\psi'$.
- (s, F) is a *μ-node* if there exists a formula $\psi' \in F$ containing a top-level fixpoint sub-formula of form $\mu X.\psi''$; it is a *ν-node* otherwise.

Note that these categories overlap one another.

The edges in the graph are labeled by elements drawn from the set $Act \cup \{\epsilon^+, \epsilon^-\}$ (where it is assumed that $\epsilon^+, \epsilon^- \notin Act$). The edge set $E \subseteq N \times (Act \cup \{\epsilon^+, \epsilon^-\}) \times N$ is defined as follows.

1. If $n = (s, F)$ is a true node or a false node,[5] then n is a sink node;
2. else if (s, F) contains state formulas then $((s, F), \epsilon^+, (s, F')) \in E$, where F' is F with all state formulas deleted;
3. else if (s, F) contains a fixpoint formula $\psi' = \sigma X.\psi''$ (where σ is μ or ν) then $((s, F), \epsilon^+, (s, F - \{\psi'\} \cup \{\psi''[\psi'/X]\})) \in E$;
4. else if $\psi = \psi_1 \wedge \psi_2 \in F$ then $((s, F), \epsilon^+, (s, F - \{\psi\} \cup \{\psi_1, \psi_2\})) \in E$;
5. else if (s, F) is not an and-node and $\psi = \psi_1 \vee \psi_2 \in F$ then $((s, F), \epsilon^+, (s, F - \{\psi\} \cup \{\psi_1\}) \in E$, $((s, F), \epsilon^+, (s, F - \{\psi\} \cup \{\psi_2\}) \in E$, and $((s, F), \epsilon^-, (s, F - \{\psi\} \cup \{\psi_1, \psi_2\})) \in E$;

[5] Determining whether a node is false may require determining if $s \models_L \phi$ for some state formula. This can be done by (recursively) invoking the model-checking procedure described in the next section.

6. else if (s, F) is an action node, let $F_a = \{\psi' \mid \langle a\rangle\psi' \in F \text{ or } [a]\psi' \in F\}$. Then for any $a \in Act$ with $F_a \neq \emptyset$ and $s' \in S$ such that $(s, a, s') \in \delta$, $((s, F), a, (s', F_a)) \in E$.

Intuitively, an edge $((s, F), \ell, (s', F'))$ indicates a "local relationship" between $\mathsf{m}_s(\Theta_L(\wedge F))$ and $\mathsf{m}_{s'}(\Theta_L(\wedge F'))$. To see this, first note that if (s, F) is a true node (false node) then $\mathsf{m}_s(\Theta_L(\wedge F)) = 1(0)$. Now suppose that (s, F) is an or-node to which case 5 applies. This means that $F = F' \cup \{\psi_1 \vee \psi_2\}$, and the semantics of the logic entails that $\wedge F$ and $(\wedge F' \wedge \psi_1) \vee (\wedge F' \wedge \psi_2)$ are logically equivalent. From Lemma 13 we may therefore conclude the following.

$$\mathsf{m}_s(\Theta_L(\wedge F)) = \mathsf{m}_s(\Theta_L(\wedge F' \wedge \psi_1)) + \mathsf{m}_s(\Theta_L(\wedge F' \wedge \psi_2)) - \mathsf{m}_s(\Theta_L(\wedge(F' \cup \{\psi_1, \psi_2\})))$$

This observation is encoded in the ϵ^+ and ϵ^- edges emanating from (s, F). Similar observations hold for the other nodes, with the exception of action nodes, which we discuss in more detail below.

Generating equations from the graph. We now explain how to generate a system of equations from the graph described above. The system will contain one variable, X_n, for each node n in the graph and one equation containing this variable as its left-hand side. The right-hand side of the equation for X_n is generated as follows, based on the edges emanating from n.

1. If n is a true node then the equation for X_n is $X_n = 1$; if n is a false node, the equation for X_n is $X_n = 0$.
2. If there is an edge of the form (n, ϵ^+, n') then the equation for X_n is
$$X_n = \sum_{(n,\epsilon^+,n') \in E} X_{n'} - \sum_{((n,\epsilon^-,n') \in E} X_{n'}.$$
3. If $n = (s, F)$ is an action node, let $A_n = \{a \mid (n, a, n') \in E\}$. Then the equation for X_n is
$$X_n = \prod_{a \in A_n} \sum_{(n,a,(s',F')) \in E} (P(s, a, s') \cdot X_{(s',F')}).$$

Intuitively, these equations are intended to reflect relationships among the measures associated with each vertex. The right-hand side of in the equation associated with an action node reflects this intuition. A small example illustrates why. Suppose that action node (s, F) is such that $F = \{\langle a\rangle\psi_1, \langle b\rangle\psi_2\}$. Since this is not a false node, it follows that s has both a- and b-transitions. The question is, what is the measure of observations rooted at s and satisfying $\wedge F$? Each such observation would select one a-transition and one b-transition from s, with the target of the a-transition then being the root of an observation satisfying ψ_1 and similarly for the target of the b-transition. For a given combination of single a- and b-transitions with target states s_a and s_b, the measure of observations using these transitions and satisfying $\wedge F$ is $P(s, a, s_a) \cdot \mathsf{m}_{s_a}(\Theta_L(\psi_1)) \cdot P(s, b, s_b) \cdot \mathsf{m}_{s_b}(\Theta_L(\psi_2))$. Using simple symbol pushing, it is then easy to show that the total measure of

observations emanating from s and satisfying $\wedge F$ is characterized by the right-hand side of the equation above.

We now have the following.

Lemma 19. *Let* $\mathbf{E} = \{X_n = E_n\}$ *be the equations generated above, and let* $\mathbf{A}$ *be the "vector"* $\{X_n = \mathsf{m}_s(\Theta_L(\wedge F))\}$*, where* $n = (s, F)$*. Then* A *is a solution to* $\mathbf{E}$*.*

Solving the equations. The previous lemma indicates that the equations we generate are "faithful" to the measures we wish to calculate in the sense that they are indeed a solution to the equations. However, in general there will be many such solutions, and the question then arises as to how we determine which solution indeed corresponds to the measures we want. The procedure *modchk-fuzzy* does so as follows.

1. Compute the strongly connected components of the graph from which the equations are constructed and topologically sort them.
2. Propagate solutions as far as possible: If a solution has been computed for a variable, replace all occurrences of the variable in the right-hand sides by the variable.
3. Beginning at the end of the strongly connected component list, process each component C as follows.
   (a) If C contains a μ-node, assign each variable corresponding to a node in C the value 0; otherwise, assign each variable the value 1.
   (b) Repeatedly calculate new values for the variables of C by evaluating each right-hand side using the old values. Stop when values don't change (or fall within a tolerance ϵ that is a parameter to the algorithm).
   (c) Propagate these values.

In general, this algorithm requires the specification of an "error tolerance" ϵ because the quantities being manipulated are real numbers. So the algorithm is approximation-based. However, all the functions being used are continuous, and hence the iteration process described above converges. We now have the following.

Lemma 20. *Let* $s \in S$ *and* ψ *be a fuzzy formula. Then the quantity calculated for* $X_{(s,\{\psi\})}$ *converges to* $\mathsf{m}_s(\Theta_L(\psi))$*.*

4.2 Model Checking and GPL

The procedure *modchk-fuzzy* may now be used to build a model-checker for GPL. This model checker engages in a case analysis on the formula ϕ and performs the obvious operations if the formula is not of the form $Æ_{\geq p}\psi$ or $Æ_{>p}\psi$. In these latter two cases, *modchk-fuzzy* is called to calculate $\mathsf{m}_s(\Theta_L(\psi))$, and the answer compared to p appropriately. As *modchk-fuzzy* is an approximation-based numerical algorithm, the usual numerical issues must be confronted in performing these comparisons. In particular, if the computed answer is close enough to p to fall within the margin of error, then only indeterminate answers can be given.

4.3 Discussion about complexity

The algorithm just described relies on the use of numerical approximation techniques. However, in certain cases exact solutions can be calculated. For example, if the PLTS is in fact a MP then the equation system generated is linear. In addition, results of [CY88] suggest that this linear system can be converted into one that has a unique solution. In this case, the equations can be solved exactly.

The non-linearity of the equations we consider, for model-checking PLTS, is a direct consequence of the program model (which allows different kinds of actions, i.e., when $|Act| > 1$) and our semantics (where observations are deterministic trees). Consequently, non-linearity in the measure equations is a fact that any solution technique, we adopt, will have to contend with. Furthermore, since there can be no direct technique for solving arbitrary polynomial equations (due to a result of Galois) we will have to depend upon iterative techniques. A characteristic of iterative techniques, shared by our work, is that the complexity depends upon the precision of answers demanded. We have been investigating the use of symbol algebra tools, such as Maple, in implementing our model-checking procedure and hope to report our experiences in the near future.

5 Concluding Remarks

We have presented a uniform framework for defining temporal logics on reactive probabilistic transition systems. Our approach is based on using the modal mu-calculus to define measurable sets of observations of such systems. We have shown that our logic is expressive enough to encode two different existing temporal logics, and we have also demonstrated that it may be used to rectify an infelicity in a third. A model-checking procedure for the logic was also presented.

As for future work, we believe that we can improve on the algorithm presented here by using results similar to those in [CY88] to transform our equation systems into ones having unique solutions. If this is the case, then we can use traditional solution techniques for nonlinear equations to compute measures in a numerically robust manner. We would also like to implement these algorithms. Another important issue for future work is that of applying our logic to more general transition systems (for example, the transition systems of [Seg95]) and establishing its relation to probabilistic automata[Paz71]. Such an extension would allow translation of pCTL* interpreted over probabilistic non-deterministic systems into our framework, much like the translation we have shown in this paper, and provide an efficient model-checking procedure for the same. It would also be useful to investigate the adaptation of our techniques to models of distributed computation in which resources may probabilistically fail, such as the one presented in [PSC$^+$98].

The work being presented here also has applications to edge-profile-driven data flow-analysis [Ram96, BGS98], where the likelihood with which program properties hold is calculated; such calculation can then be used to perform profile-driven optimization [BL92]. Recent work [Ste91, Sch98] on reducing traditional data flow analysis problems to a model-checking problem can be extended

to reduce profile driven DFA to probabilistic model-checking, and we propose to investigate this further.

Acknowledgments: Murali Narasimha would like to thank S. Arun-Kumar and E. Kaltofen for several helpful discussions on this topic.

References

[ASB+95] A. Aziz, V. Singhal, F. Balarin, R.K. Brayton, and A.L. Sangiovanni-Vincentelli. It usually works: The temporal logic of stochastic systems. In *Proc of Computer Aided Verification '95*. Springer-Verlag, July 1995.

[BdA95] A. Bianco and L. de Alfaro. Model-checking of probabilistic and nondeterministic systems. In *Proc. Foundations of software technology and theoretical computer science*, Lecture notes in Computer science, vol 1026, pages 499–513. Springer-Verlag, December 1995.

[BGS98] Rastislav Bodik, Rajiv Gupta, and Mary Lou Soffa. Complete removal of redundant computations. *ACM SIGPLAN Notices*, 33(5):1–14, May 1998.

[BL92] Thomas Ball and James R. Larus. Optimally profiling and tracing prograns. In Ravi Sethi, editor, *Proceedings of the 19th Annual Symposium on Principles of Programming Languages*, pages 59–70, Albuquerque, NM, January 1992. ACM Press.

[CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[Cle90] Rance Cleaveland. Tableau-based model checking in the propositional mu-calculus. *Acta Informatica*, 27(8):725–747, 1990.

[CY88] C. Courcoubetis and M. Yannakakis. Verifying temporal properties of finite-state probabilistic programs. In *Proc. 1988 IEEE Symp. on the Foundations of Comp. Sci.*, 1988.

[EH86] E. A. Emerson and J. Y. Halpern. "sometimes" and "not never" revisited: On branching time versus linear time temporal logic. *JACM*, 33(1):151–178, 1986.

[EL86] E. Allen Emerson and Chin-Laung Lei. Efficient model-checking in fragments of the propositional mu-calculus. In *Proceedings 1st Annual IEEE Symp. on Logic in Computer Science, LICS'86, Cambridge, MA, USA, 16–18 June 1986*, pages 267–278. IEEE Computer Society Press, Los Alamitos, CA, 1986.

[Han94] H. Hansson. *Time and Probability in Formal Design of Distributed Systems*. Elsevier, 1994.

[HK97] Michael Huth and Marta Kwiatkowska. Quantitative analysis and model checking. In *Proceedings, Twelth Annual IEEE Symposium on Logic in Computer Science*, pages 111–122, Warsaw, Poland, 29 June–2 July 1997. IEEE Computer Society Press.

[HM85] M. Hennessy and R. Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the Association of Computing Machinery*, 32(1):137–161, 1985.

[Koz83] D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27(1):333–354, 1983.

[KSK66] J. G. Kemeny, J. L. Snell, and A. W. Knapp. *Denumerable Markov Chains*. Van Nostrand, New Jersey, 1966.

[LS91] K. G. Larsen and A. Skou. Bisimulation through probabilistic testing. *Information and Computation*, 94, 1991.

[McM93] McMillan, K. L. *Symbolic Model Checking.* Kluwer Academic Publishers, Norwell Massachusetts, 1993.

[Paz71] Azaria Paz. *Introduction to Probablistic Automata.* Academic Press, New York, 1971.

[PSC+98] A. Philippou, O. Sokolksy, R. Cleaveland, I. Lee, and S. Smolka. Probabilistic resource failure in real-time process algebra. 1998. To appear in Proceedings of *CONCUR.*

[PZ93] Amir Pnueli and Lenore D. Zuck. Probabilistic verification. *Information and Computation*, 103(1):1–29, March 1993.

[Ram96] G. Ramalingam. Data flow frequency analysis. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 267–277, Philadelphia, Pennsylvania, 21–24 May 1996.

[Sch98] David A. Schmidt. Data flow analysis is model checking of abstract interpretation. In *Conference Record of POPL '98: The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 38–48, San Diego, California, 19–21 January 1998.

[Seg95] R. Segala. A compositional trace-based semantics for probabilistic automata. In *CONCUR95*, pages 324–338, 1995.

[SEJ93] A. P. Sistla, A. Emerson, and C. Jutla. Model checking in fragments of the mu-calculus. In *Proceedings of the International Conference on Computer Aided Verification*, volume Vol 697 of *LNCS*, pages 385–396. Springer-Verlag, June 1993.

[Ste91] Bernhard Steffen. Data flow analysis as model checking. *Lecture Notes in Computer Science*, vol. 526, 1991.

[Sti92] C. Stirling. *Modal and Temporal Logics*, volume 2 of *Handbook of Logic in Computer Science*, pages 478–551. Oxford Science Press, 1992.

[Var85] M. Y. Vardi. Automatic verification of probabilistic concurrent finite-state programs. In *IEEE Symposium on Foundations of Computer Science*, pages 327–338, 1985.

[vGSST90] Rob van Glabbeek, Scott A. Smolka, Bernhard Steffen, and Chris M. N. Tofts. Reactive, generative, and stratified models of probabilistic processes. In *Proc. of Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 130–141. IEEE Computer Society Press, June 1990.

A π-calculus Process Semantics of Concurrent Idealised ALGOL

Christine Röckl[1] and Davide Sangiorgi[2]

[1] Technische Universität München, D-80290 München, roeckl@in.tum.de
[2] INRIA–Sophia Antipolis, F-06902 Sophia Antipolis, davide.sangiorgi@inria.fr

Abstract. We study the use of the π-calculus for semantical descriptions of languages such as *Concurrent Idealised ALGOL* (CIA), combining imperative, functional and concurrent features. We first present an operational semantics for CIA, given by SOS rules and a contextual form of behavioural equivalence; then a π-calculus semantics. As behavioural equivalence on π-calculus processes we choose the standard (weak early) bisimilarity. We compare the two semantics, demonstrating that there is a close operational correspondence between them and that the π-calculus semantics is sound. This allows for applying the π-calculus theory in proving behavioural properties of CIA phrases. We discuss laws and examples which have served as benchmarks to various semantics, and a more complex example involving procedures of higher order.

1 Introduction

Reynolds formalised Idealised ALGOL (IA) as a *simple imperative language* enriched with a procedural mechanism provided by a *typed call-by-name λ-calculus* [Rey81]. IA combines in an elegant way imperative and functional features, and since its introduction has been the object of extensive study (cf. [OT97]). *Concurrent Idealised ALGOL* (CIA) was introduced by Brookes as an extension of IA with shared variable parallelism [Bro96]. CIA allows parallel composition of commands and features an **await** operator for imposing atomicity. Brookes [Bro96] has presented an elegant denotational model for CIA, extending a Kripke-style Possible Worlds semantics. From a semantical point of view, CIA is a challenging language, since it combines imperative, functional and concurrent features, and possesses an atomicity construct.

In this paper we study semantics of CIA given by a translation into the π-calculus. The main reasons for using the π-calculus are the following. It offers a well-developed theory that we wish to exploit, through the translation, to reason on CIA terms. We also intend to profit from the π-calculus being, syntactically, a first-order language, i.e., values only consist of names (in typed versions, there may also be basic values such as integers and booleans). In contrast, CIA is higher-order, thus values may be arbitrary terms. In higher-order languages, defining satisfactory notions of behavioural equivalences—not to mention proof techniques for them—may be hard. Proofs of process equivalences are complicated by universal quantifications over terms. Further, it is in general hard to establish that a notion of bisimilarity is a congruence. (For higher-order languages,

W. Thomas (Ed.): FOSSACS'99, LNCS 1578, pp. 306-321, 1999.

this is usually proved using Howe's technique [How96]; attempts to extend this technique to languages with local state, however, have been unsuccessful so far; see discussions in [FHJ95].) A further advantage of the π-calculus semantics is that, as states are represented by processes, no *snapback effects* (reversibility of state changes, cf. [AM96,AM97,OT97]) can occur; models representing states by functions—usually denotational models do so—suffer from snapback effects, which are usually removed by means of logical relations [OT97].

Our study is also motivated by the question of how appropriate the π-calculus is for giving semantics to languages such as CIA. Previous work gives evidence that the π-calculus can model references, functions and various forms of (non atomic) parallelism [Wal95,Jon93,KS98,Mil92], but so far only limited forms of combinations of these have been considered. In the case of imperative languages, little effort has been spent in comparing π-calculus to operational semantics, and in using π-calculus translations for proving properties of the source languages. Denotational approaches indicate a strong similarity between local names in the π-calculus and local references in imperative languages; note that the mathematical techniques employed in modelling the π-calculus [Sta96,FMS96] were originally developed for the semantic description of local references. Yet names and references behave rather differently: receiving from a channel is destructive—it consumes a value—whereas reading from a reference is not; a reference has a unique location, whereas a channel may be used by several processes for both reading and writing; etc. Hence it is unclear if and how interesting properties of imperative languages can be proved via a translation into the π-calculus.

Section 2 briefly introduces an SOS-style operational semantics for CIA along with a contextual form of behavioural equivalence. Then a π-calculus semantics is presented, together with soundness results for the encoding (Sections 3 and 4). The main part of this paper is devoted to the discussion of concrete examples (Section 5). We prove laws and examples from [MS88,Bro96,MT90a,MT90b], as well as a more complex example involving procedures of higher order, namely the equivalence between two CIA descriptions of two-places buffers (n-place buffers could be dealt with similarly). Then we show that our semantics is not fully abstract (Section 6). We present equivalent CIA phrases, the translations of which are not bisimilar. We show how to handle these examples using types, especially I/O-types. It is unclear whether the type systems we propose already yield full abstraction (we conjecture they do not). Yet introducing more and more sophisticated types deteriorates the applicability to concrete proofs. However, our experiments have led us to the conclusion that in most cases I/O types suffice.

2 Concurrent Idealised ALGOL

Syntax, typing and notations for CIA closely follow [Rey81,Bro96]. *Data types* consist of integers and booleans; *phrase types* are constructible from variables, expressions and commands using arrow type (for simplicity we omit tupling):

$$\begin{array}{lll} \tau & ::= & \mathit{int} \mid \mathit{bool} & \text{Data Types} \\ \sigma & ::= & \mathbf{var}[\tau] \mid \mathbf{exp}[\tau] \mid \mathbf{comm} \mid (\sigma \to \sigma') & \text{Phrase Types} \end{array}$$

Data and variable types are lifted up to expression types via the rules

$$\frac{\Gamma \vdash v : \tau}{\Gamma \vdash v : \mathbf{exp}[\tau]} \qquad \text{and} \qquad \frac{\Gamma \vdash \iota : \mathbf{var}[\tau]}{\Gamma \vdash !\iota : \mathbf{exp}[\tau]}.$$

Variables can be declared on data types only, whereas *procedure definition*, *recursion* and *conditional* are uniformly applicable to all phrase types. An *environment* Γ is a partial function from *identifiers* to types, with *domain* $\mathrm{dom}(\Gamma)$.
The syntax is defined according to [Bro96]. However, for defining behavioural equivalences we find it convenient to have explicit constructs for input (on variables) and output (of expressions); alternatively, we could have allowed the observer direct access to the variables (we shall come back to this in Section 7). Further we allow for the use of conditionals in the body of **await** statements. The body of an **await** statement therefore consists of assignments, sequential composition and conditionals. Syntax and typing rules are presented in Table 1 at the end of this paper.
We define an SOS-style operational semantics of CIA, using small-step transition rules (as opposed to a big-step or natural semantics) in order to capture the nondeterministic behaviour resulting from the interaction of phrases via shared variables. The rules are quite standard, with the exception of those needed for modelling the atomicity required by **await**. Let P and P' be phrases of variable, expression or command type which do not contain free identifiers; σ and σ' are assignments closing up on all free variables of P and P'. We call a pair $\langle P, \sigma\rangle$ a *configuration*, and, if P is a command, we call it a *command configuration*. In the sublanguage without **await** (CIA-{**await**}), the SOS rules are of the form

$$\langle P,\sigma\rangle \xrightarrow{\mathbf{out}\langle v\rangle} \langle P',\sigma'\rangle, \quad \langle P,\sigma\rangle \xrightarrow{\mathbf{in}\langle v\rangle} \langle P',\sigma'\rangle, \quad \langle P,\sigma\rangle \xrightarrow{\tau} \langle P',\sigma'\rangle, \quad \langle P,\sigma\rangle \xrightarrow{\surd} \sigma',$$

where $\mathbf{out}\langle v\rangle$ is the output of value v, $\mathbf{in}\langle v\rangle$ the input of value v; τ is an invisible (internal) action; and the tick $\surd$ denotes termination. If P is an expression, the tick carries the value resulting from its evaluation.
The command **await** guarantees for an atomic execution of a sequential composition of assignments and conditionals once its guard has been evaluated to true (an evaluation to false results in a repetition after some period of busy-waiting). During the evaluation of the guard and the execution of the body of an **await** statement, any other computation has to be stopped. We achieve this by introducing *locked configurations* $\langle P,\sigma\rangle_\ell$. The tag ℓ represents a lock. Whenever an **await** statement is executed, the configuration is marked with the lock ℓ, and all but the **await** component are prevented from running (this component is marked itself so to be distinguishable from its context). The lock is released either if the guarding boolean expression has been evaluated to false, or otherwise after the command has been completed. The rules for locked configurations are of the form $\langle P,\sigma\rangle_\ell \xrightarrow{\tau} \langle P',\sigma'\rangle_\ell$; further there are rules for introducing and eliminating the lock from the configurations. Relation $\stackrel{\epsilon}{\Longrightarrow}$ is the reflexive and transitive closure of $\xrightarrow{\tau}$, and $\stackrel{\mu}{\Longrightarrow}$ is given by $\stackrel{\epsilon}{\Longrightarrow}\xrightarrow{\mu}\stackrel{\epsilon}{\Longrightarrow}$ (arbitrarily many invisible steps before and after the μ transition).
Behavioural equality is defined in two steps: We first apply the (standard) definition of bisimilarity in value-passing process calculi to CIA command configu-

rations (Definition 1); then, by closing it under all (closing) contexts, we obtain an *observational congruence* applicable to all phrase types (Definition 2).

Definition 1 (Configuration bisimulation). A binary relation $\mathcal{R}$ upon command configurations is a *configuration bisimulation* if it is symmetric, and $E_1 \mathcal{R} E_2$ implies,

1. if $E_1 \xrightarrow{\surd} \sigma_1$ then there is σ_2 s.t. $E_2 \stackrel{\surd}{\Longrightarrow} \sigma_2$,
2. if $E_1 \xrightarrow{\tau} E_1'$ then there is E_2' s.t. $E_2 \stackrel{\epsilon}{\Longrightarrow} E_2'$ and $E_1' \mathcal{R} E_2'$,
3. if $E_1 \xrightarrow{\mu} E_1'$ and μ is an output or an input, then there is E_2' s.t. $E_2 \stackrel{\mu}{\Longrightarrow} E_2'$ and $E_1' \mathcal{R} E_2'$.

We write $E_1 \approx E_2$ if there is a configuration bisimulation $\mathcal{R}$ with $E_1 \mathcal{R} E_2$.

We say that a context Con is *closed wrt.* a phrase P if $\emptyset \vdash Con[P] : \mathbf{comm}$ (i.e., $Con[P]$ does not contain free identifiers nor variables).

Definition 2 (Observational congruence). Let P_1, P_2 be arbitrary phrases. Then P_1 and P_2 are *observationally congruent*, written $P_1 \approx_{oc} P_2$, if for every context Con which is closed wrt. P_1 and P_2, $\langle Con[P_1], \emptyset \rangle \approx \langle Con[P_2], \emptyset \rangle$.

Observational congruence is the notion of behavioural equality on CIA phrases we are interested in. It is however hard to prove equalities following its definition, due to the universal quantification over the contexts.
We conclude the section with a useful fact about locked configurations. The behaviour of an **await** statement is deterministic, both due to the absence of parallel composition within its body and the incapability of expressions to change a given assignment.

Lemma 1. $\langle C, \sigma \rangle_\ell \xrightarrow{\tau} \langle C', \sigma' \rangle_\zeta$ *with* $\zeta \in \{\ell, \epsilon\}$ *implies* $\langle C, \sigma \rangle_\ell \approx \langle C', \sigma' \rangle_\zeta$.

Corollary 1. *For every configuration* $\langle C, \sigma \rangle_\ell$ *the following holds: Either it diverges (i.e., there is an infinite computation of silent steps starting from* $\langle C, \sigma \rangle_\ell$*) or there is another configuration* $\langle C', \sigma' \rangle$ *such that* $\langle C, \sigma \rangle_\ell \stackrel{\epsilon}{\Longrightarrow} \langle C', \sigma' \rangle$ *and* $\langle C, \sigma \rangle_\ell \approx \langle C', \sigma' \rangle$.

3 The π-calculus

We translate CIA into a π-calculus language supplied with a simple type system. This type system provides integer, boolean, product and channel types; we omit the typing rules which are quite standard, assuming that all processes and expressions we write are well-typed. *Channels* are used to transmit values; they are ranged over by $a, b, \ldots$; *variables* are ranged over by $x, y, \ldots$. Together, channels and variables constitute the *names*, $p, q, \ldots$. *Integer* and *boolean constants* are denoted by $n, m, \ldots$. Channels and constants are the *values*, ranged over by v. $\otimes$ denotes basic operators like addition, subtraction, complement, etc.

$$
\begin{array}{lll}
e & ::= v \mid x \mid \otimes e \mid e \otimes e & \text{Expressions} \\
\pi & ::= \overline{p}\langle \tilde{e} \rangle \mid p(\tilde{y}) \mid \tau & \text{Prefix} \\
R & ::= 0 \mid \pi.R \mid R{+}R \mid R|R \mid (\nu\, p)R \mid [x = n]R \mid [x \neq n]R \mid !p(\tilde{y}).P & \text{Processes.}
\end{array}
$$

A process is *closed* if it does not contain free variables. Otherwise it is *open*. For the semantics of the π-calculus we adopt a labelled transition system. In contrast to reduction semantics [Mil91], this allows us to use labelled forms of bisimulation and to use the associated proof techniques [MS92]. Process transitions (in the *early style*) are of the form $P \xrightarrow{\mu} P'$, where μ is given by

$$\mu ::= (\nu\, \tilde{b})\overline{a}\langle\tilde{v}\rangle \mid a\langle\tilde{v}\rangle \mid \tau.$$

$(\nu\, \tilde{b})\overline{a}\langle\tilde{v}\rangle$ denotes the output of the values $\tilde{v}$ on the name a, where $\tilde{b}$ are those channels among the names of $\tilde{v}$ which are private to the sender process; $a\langle\tilde{v}\rangle$ is the input of values $\tilde{v}$ over the channel a; finally, τ represents an internal action. We use the standard SOS rules of the π-calculus. As in typed π-calculi (such as in [Wal95]), there are rules for evaluating an expression to a value, so to be able to infer transitions like $\overline{a}\langle 2+3\rangle.P \xrightarrow{\overline{a}\langle 5\rangle} P$. Weak transitions can be obtained by adding arbitrarily many silent steps before and after a strong transition. We write $\stackrel{\epsilon}{\Longrightarrow}$ for the reflexive and transitive closure of $\xrightarrow{\tau}$, adopting the standard convention that $\hat{\tau} \stackrel{\text{def}}{=} \epsilon$, and $\hat{\mu} \stackrel{\text{def}}{=} \mu$ for all visible labels μ.
Bisimilarity is defined in the usual way (cf. for instance [MPW89]):

Definition 3 (Early bisimulation). A binary relation $\mathcal{R}$ upon closed processes is a *(weak early) bisimulation* if it is symmetric, and $R\mathcal{R}S$ implies

$$\text{if } R \xrightarrow{\mu} R' \text{ then there is } S' \text{ s.t. } S \stackrel{\hat{\mu}}{\Longrightarrow} S' \text{ and } R'\mathcal{R}S'.$$

Two processes R and S are *(weakly early) bisimilar*, written $R \approx_\pi S$, if there is a (weak early) bisimulation $\mathcal{R}$ with $R\mathcal{R}S$.

The definition extends to open processes by closing over all substitutions. In the case of channel variables, however, one can often establish syntactic conditions to avoid the substitution of all channels for a variable, but simply substitute *one* fresh channel for the variable instead [San95a]. This also holds for those processes which we obtain by translating CIA, in Section 4 (we shall not discuss this further in this extended abstract). Also, even though early bisimilarity is not preserved by arbitrary summation, it is preserved by guarded summation, which suffices in our case. The bisimulation proof technique can be made more powerful by combining it with up-to techniques, like "up to expansion" and "up to injective substitutions" [Mil89,MS92,San95b] (expansion is an asymmetric variant of bisimulation taking into account the number of internal steps performed by the processes [AKH92]).

4 Interpreting CIA in the π-calculus

The π-calculus interpretation of CIA is given by the rules in Tables 2 and 3 at the end of this paper. The storage is modelled by *registers* of the form (in the π-calculus, recursive process definitions are derivable from replication [Mil91])

$$\mathbf{Reg}_\iota[v] \stackrel{\text{def}}{=} \overline{\mathbf{get}}_\iota\langle v\rangle.\mathbf{Reg}_\iota[v] + \mathbf{put}_\iota(w).\mathbf{Reg}_\iota[w].$$

Processes in the scope of $\mathrm{fn}_\iota \stackrel{\text{def}}{=} \{\mathbf{get}_\iota, \mathbf{put}_\iota\}$ are allowed to read and modify the content of $\mathbf{Reg}_\iota$. Configurations $\langle C, \sigma\rangle$, with $\Gamma(\sigma) = \{\iota_i\}_i$, translate to

$$[\![\langle C, \sigma\rangle]\!]_p \stackrel{\text{def}}{=} (\nu\, \mathrm{fn}_{\iota_1}, \ldots, \mathrm{fn}_{\iota_n})(\prod_i \mathbf{Reg}_{\iota_i}[\sigma(\iota_i)] \mid [\![C]\!]_p).$$

CIA-{await}

A π-calculus interpretation of CIA necessitates certain care even in the absence of **await**, due to the language combining imperative features with higher order. We translate all phrases P into parameterised processes $[\![P]\!]_p$; the fresh name p is used to signal the termination of the execution of $[\![P]\!]_p$. The sequential composition of two commands, for instance, is written as

$$[\![C_1; C_2]\!]_p \stackrel{\text{def}}{=} (\nu\, q)([\![C_1]\!]_q \mid q.[\![C_2]\!]_p).$$

First only $[\![C_1]\!]_q$ is able to execute because of name q guarding $[\![C_2]\!]_p$. As soon as $[\![C_1]\!]_q$ terminates, it signals so on $\overline{q}$ thus releasing $[\![C_2]\!]_p$. Yet another example of sequentiality are declarations,

$$[\![\mathbf{new}\ [\tau]\ \iota := E\ \mathbf{in}\ C]\!]_p \stackrel{\text{def}}{=} (\nu\, q)([\![E]\!]_q \mid q(x).(\nu\, \mathrm{fn}_\iota)(\mathbf{Reg}_\iota[x] \mid [\![C]\!]_p)).$$

Here parameter q does not only guard $[\![C]\!]_p$, but is also used to transmit the result of the evaluation of $[\![E]\!]_q$ to register $\mathbf{Reg}_\iota$. Suppose E is a value v, then

$$\begin{aligned}[\![\mathbf{new}\ [\tau]\ \iota := v\ \mathbf{in}\ C]\!]_p &\stackrel{\text{def}}{=} (\nu\, q)(\overline{q}\langle v\rangle.0 \mid q(x).(\nu\, \mathrm{fn}_\iota)(\mathbf{Reg}_\iota[x] \mid [\![C]\!]_p)) \\ &\approx_\pi (\nu\, \mathrm{fn}_\iota)(\mathbf{Reg}_\iota[v] \mid [\![P]\!]_p),\end{aligned}$$

where $\approx_\pi$ is an application of some simple π-calculus laws (precisely the law $(\nu\, q)(\overline{q}\langle v\rangle.R \mid q(x).S) \approx_\pi (\nu\, q)(R \mid S\{v/x\})$ and the garbage-collection law $(\nu\, q)R \approx_\pi R$ if q is not free in R). Identifiers are modelled by processes sending along a specified channel which is used to invoke a copy of the argument they represent. Both procedural arguments and recursion are translated using replication, so fresh copies are available at every call (recall that CIA is a call-by-name language). For instance, if P is a free identifier, called x_P in the π-calculus translation, then

$$\begin{aligned}&[\![\mathbf{new}\ [int]\ \iota := 1\ \mathbf{in}\ P(!\iota)]\!]_p \\ &\quad\approx_\pi \underbrace{(\nu\, \mathrm{fn}_\iota)(\mathbf{Reg}_\iota[1])}_{\text{Declaration}} \mid \underbrace{(\nu\, q)(\overline{x}_P\langle q\rangle.0 \mid q(v))}_{\text{Invoking a copy of procedure } P}.\ \underbrace{(\nu\, x)(\overline{v}\langle x, p\rangle)}_{\text{Communicating argument and termination signal}}.\ \underbrace{!x(r).\mathbf{get}_\iota(z).\overline{r}\langle z\rangle.0))}_{\text{Procedural argument}}).\end{aligned}$$

There is a close *operational correspondence* between configurations $\langle P, \sigma\rangle$ and their encodings $[\![\langle P, \sigma\rangle]\!]_p$. For the proof that the interpretation is sound, command configurations $\langle C, \sigma\rangle$ are of particular interest (recall that $\approx$ is defined exactly upon these). Using $\succeq_\pi$ for the expansion relation (cf. Section 3), we give

some of the correspondences (the others are similar):

- $\langle C,\sigma\rangle \xrightarrow{\surd} \sigma'$ implies $[\![\langle C,\sigma\rangle]\!]_p \succeq_\pi (\overline{p}.0)^{\sigma'}$;
- $[\![\langle C,\sigma\rangle]\!]_p \xrightarrow{\overline{p}} R$ implies $R \succeq_\pi 0$ and $\langle C,\sigma\rangle \xrightarrow{\surd} \sigma'$;
- $\langle C,\sigma\rangle \xrightarrow{\mathbf{out}\langle v\rangle} \langle C',\sigma'\rangle$ implies $[\![\langle C,\sigma\rangle]\!]_p \overset{\overline{\mathbf{out}}\langle v\rangle}{\Longrightarrow} [\![\langle C',\sigma'\rangle]\!]_p$;
- $[\![\langle C,\sigma\rangle]\!]_p \xrightarrow{\tau} R$ implies either $R \succeq_\pi [\![\langle C',\sigma'\rangle]\!]_p$ such that $\langle C,\sigma\rangle \overset{\epsilon}{\Longrightarrow} \langle C',\sigma'\rangle$, or $[\![\langle C,\sigma\rangle]\!]_p \approx_\pi R \xrightarrow{\overline{\mathbf{out}}\langle v\rangle} [\![\langle C',\sigma'\rangle]\!]_p$ such that $\langle C,\sigma\rangle \xrightarrow{\mathbf{out}\langle v\rangle} \langle C',\sigma'\rangle$.

The operational correspondence relates every possible transition of a configuration and of its encoding. A similar operational correspondence result holds for weak transitions. Exploiting the congruence properties of $\approx_\pi$, the compositionality of the encoding, and the operational correspondence results, we can prove that the encoding is sound. In the proof we also make use of an auxiliary encoding $\mathcal{C}'$ which yields an even closer operational correspondence with CIA, and is obtained from $\mathcal{C}$ by removing some "administrative" silent steps.
Let $\approx_{oc}^{-}$ be the observational congruence on CIA-{**await**} defined analogously to $\approx_{oc}$ on full CIA.

Theorem 1 (Soundness). $[\![P_1]\!]_p \approx_\pi [\![P_2]\!]_p$ *implies* $P_1 \approx_{oc}^{-} P_2$ *for arbitrary CIA-*{**await**} *phrases* P_1 *and* P_2.

The converse (completeness) holds in the case of closed commands, but does not extend to arbitrary phrases, as we shall discuss in Section 6.

Full CIA

The encoding $[\![.]\!]^\ell$ of phrases in full CIA follows the same compositional scheme as for CIA-{**await**}, for instance

$$[\![C_1;C_2]\!]^\ell_p \stackrel{\text{def}}{=} (\nu\, q)([\![C_1]\!]^\ell_q \mid q.[\![C_2]\!]^\ell_p).$$

What is different wrt. the encoding $[\![.]\!]$ is the use of a lock to impose mutual exclusion on input, output, reading from and writing to a variable, and on **await**. Before any of these commands can be executed, the lock has to be acquired; it is released upon their termination. The lock is implemented by a process $\ell.0$. At any time at most one copy of the lock is available to the whole program. Acquiring the lock and continuing as R is modelled by $\overline{\ell}.R$ (the input "requires" the lock); releasing the lock and continuing as R is translated by $\ell.0 \mid R$ (a new copy of the lock is released). Reading from a variable, for instance, now becomes:

$$[\![!V]\!]^\ell_p \stackrel{\text{def}}{=} (\nu\, q)([\![V]\!]^\ell_q \mid q(gt,pt).\underbrace{\overline{\ell}}_{\text{Take lock}}.gt(x).(\underbrace{\ell.0}_{\text{Release lock}} \mid \overline{p}\langle x\rangle.0))$$

The command **await** is translated following a busy-wait strategy (cf. Table 3). In fact, its encoding is similar to that of the **while** loop (modulo the lock, cf. Table 2), only that a and p change their roles in the bodies of the conditionals. Our previous example translates to

$$[\![\mathbf{new}\,[int]\,\iota := 1\ \mathbf{in}\ P(!\iota)]\!]^\ell_p \approx_\pi (\nu\,\mathrm{fn}_\iota)(\mathbf{Reg}_\iota[1] \mid (\nu\, q)(\overline{x}_P\langle q\rangle.0 \mid q(v).(\nu\, x)(\overline{v}\langle x,p\rangle.!x(r).\overline{\ell}.\mathbf{get}_\iota(z).(\ell.0 \mid \overline{r}\langle z\rangle.0)))).$$

The differing compilation rules are given in Table 3. The results of operational correspondence and soundness are similar to those in CIA-{**await**} except that now the π-calculus terms contain a lock ℓ. So, instead of $[\![P]\!]_p$ we now work with processes of the form $(\nu\,\ell)(\ell.0 \mid [\![P]\!]_p^\ell)$. (In the operational correspondence, the configurations themselves are not locked, as Corollary 1 allows us to abstract from those being locked.)

Theorem 2 (Soundness). $(\nu\,\ell)(\ell.0 \mid [\![P_1]\!]_p^\ell) \approx_\pi (\nu\,\ell)(\ell.0 \mid [\![P_2]\!]_p^\ell)$ *implies* $P_1 \approx_{oc} P_2$ *for arbitrary CIA phrases* P_1 *and* P_2.

The following result relates the two translations, $[\![.]\!]$ and $[\![.]\!]^\ell$, which allows us to use the simpler encoding in the absence of **await**.

Theorem 3. $[\![P_1]\!]_p \approx_\pi [\![P_2]\!]_p$ *implies* $(\nu\,\ell)(\ell.0 \mid [\![P_1]\!]_p^\ell) \approx_\pi (\nu\,\ell)(\ell.0 \mid [\![P_2]\!]_p^\ell)$, *and thus* $P_1 \approx_{oc} P_2$, *for arbitrary CIA-*{**await**} *phrases* P_1 *and* P_2.

5 Examples of reasoning

Considering benchmark laws and examples from [MS88,Bro96,MT90a,MT90b], we demonstrate that the π-calculus semantics yields simple proofs of these well-known equalities. Further we show by a more complex example how to tackle procedures of higher order.

• Basic properties of CIA operators, such as associativity of sequential composition, or associativity and commutativity of parallel composition, are straightforward consequences of analogous π-calculus laws (like associativity and commutativity of parallel composition in the π-calculus).

• Suppose that ι does not occur free in P', and consider the following laws:

$$\begin{array}{ll}
\text{(L1)} & \mathbf{new}\,[\tau]\,\iota := v\,\mathbf{in}\,P' \;=\; P' \\
\text{(L2)} & \mathbf{new}\,[\tau]\,\iota := v\,\mathbf{in}\,(P;P') \;=\; (\mathbf{new}\,[\tau]\,\iota := v\,\mathbf{in}\,P);P' \\
\text{(L3)} & \mathbf{new}\,[\tau]\,\iota := v\,\mathbf{in}\,(P';P) \;=\; P';(\mathbf{new}\,[\tau]\,\iota := v\,\mathbf{in}\,P) \\
\text{(L4)} & \mathbf{new}\,[\tau]\,\iota := v\,\mathbf{in}\,(P \parallel P') \;=\; (\mathbf{new}\,[\tau]\,\iota := v\,\mathbf{in}\,P) \parallel P'.
\end{array}$$

The π-calculus proofs of these laws are all similar, and purely algebraic. As an example, we present the proof of L2; recall from Section 4 that $\mathrm{fn}_\iota \stackrel{\mathrm{def}}{=} \{\mathbf{get}_\iota, \mathbf{put}_\iota\}$:

$$\begin{array}{lll}
[\![\mathbf{new}\,[\tau]\,\iota := v\,\mathbf{in}\,(P;P')]\!]_p & \approx_\pi (\nu\,\mathrm{fn}_\iota)(\mathbf{Reg}_\iota[v] \mid (\nu\,q)([\![P]\!]_q \mid q.[\![P']\!]_p)) & (1) \\
& \approx_\pi (\nu\,q)((\nu\,\mathrm{fn}_\iota)(\mathbf{Reg}_\iota[v] \mid ([\![P]\!]_q \mid q.[\![P']\!]_p))) & (2) \\
& \approx_\pi (\nu\,q)((\nu\,\mathrm{fn}_\iota)(\mathbf{Reg}_\iota[v] \mid [\![P]\!]_q) \mid q.[\![P']\!]_p) & (3) \\
& \approx_\pi [\![(\mathbf{new}\,[\tau]\,\iota := v\,\mathbf{in}\,P);P')]\!]_p. &
\end{array}$$

Line (1) contains the encoding with v already written to $\mathbf{Reg}_\iota$; in Section 4 we have shown that this process is bisimilar to the original encoding. In (2) the restriction on q is moved to an outer level, and in (3) the restriction on fn_ι is removed from $[\![P']\!]_p$.

• The proof of the law $(\lambda(x:\theta).\,P)P' \;=\; P\{P'/x\}$ (validity of β-reduction) is an extension of the proof of the validity of β-reduction in the π-calculus encoding of the call-by-name λ-calculus [Mil92]; it uses distributivity properties of private replications, and structural induction (in this induction, there are more cases to

consider wrt. the proof of the call-by-name λ-calculus, but the structure of the proof is similar).

• The law $\mathbf{new}\ [int]\,\iota := 1\ \mathbf{in}\ P(!\iota) \;=\; P(1)$ (where P is a free identifier of appropriate type) is proved algebraically:

$$\begin{aligned}
&[\![\mathbf{new}\ [int]\,\iota := 1\ \mathbf{in}\ P(!\iota)]\!]_p \\
&\quad \approx_\pi (\nu\,\mathrm{fn}_\iota)(\mathbf{Reg}_\iota[1] \mid (\nu\,q)(\overline{x}_P\langle q\rangle.0 \mid q(v).(\nu\,x)(\overline{v}\langle x,p\rangle.!x(r).\mathbf{get}_\iota(z).\overline{r}\langle z\rangle.0))) \\
&\quad \approx_\pi (\nu\,q)(\overline{x}_P\langle q\rangle.0 \mid q(v).(\nu\,x)(\overline{v}\langle x,p\rangle.(\nu\,\mathrm{fn}_\iota)(\mathbf{Reg}_\iota[1] \mid !x(r).\mathbf{get}_\iota(z).\overline{r}\langle z\rangle.0))) \\
&\quad \approx_\pi (\nu\,q)(\overline{x}_P\langle q\rangle.0 \mid q(v).(\nu\,x)(\overline{v}\langle x,p\rangle.!x(r).\overline{r}\langle 1\rangle.0)) \\
&\quad = \; [\![P(1)]\!]_p .
\end{aligned}$$

• Suppose again that P is a free identifier of appropriate type. Proving the law

$$\mathbf{new}\ [int]\,\iota := 0\ \mathbf{in}\ P(\iota :=!\iota + 1) \;=\; P(\mathbf{skip})$$

essentially consists in showing that for arbitrary non-negative integer values v,

$$(\nu\,\mathrm{fn}_\iota)(\mathbf{Reg}_\iota[v] \mid !x(r).[\![\iota :=!\iota + 1]\!]_r) \approx_\pi !x(r).[\![\mathbf{skip}]\!]_r ,$$

where x denotes the formal parameter of P (owing to P being a free identifier, name x is provided by the observer).

• A simple π-calculus bisimulation relation can be used to prove that iteration is expressible via recursion, i.e., if x is not free in B and C then

$$\mathbf{while}\ B\ \mathbf{do}\ C \;=\; \mathbf{rec}\,x.\,\mathbf{if}\ B\ \mathbf{then}\ (C;\,x)\ \mathbf{else}\ \mathbf{skip}.$$

• In our last, more substantial, example we show that two implementations of a *two-place buffer* are equivalent (the example can be generalised to n-place buffers). For simplicity we assume that all buffers store integer values. The example involves both procedures of higher order and the **await** statement. Procedure **B** below defines a one-place buffer; x_p represents the clients, x_n a value stored by a client, and x_r is a client location, where a value retrieved from the buffer is to be stored. We use sugared notation for the declarations and conditionals.

$$\begin{aligned}
\mathbf{B} \stackrel{\mathrm{def}}{=}\ & \lambda(x_p : \theta_c).\,\mathbf{new}\ [bool]\,fl := \mathrm{ff},\, ct := 0\ \mathbf{in} \\
& \quad (x_p(\lambda(x_n : int).\,\mathbf{await}\ (!fl = \mathrm{ff})\ \mathbf{then}\ (ct := x_n;\, fl := \mathrm{tt}))) \qquad /*\ \mathbf{put}\ */ \\
& \quad\ (\lambda(x_r : \mathbf{var}[int]).\,\mathbf{await}\ (!fl = \mathrm{tt})\ \mathbf{then}\ (x_r :=!ct;\, fl := \mathrm{ff})). \qquad /*\ \mathbf{get}\ */
\end{aligned}$$

Analogously one can define buffers with two, or even more, places. Buffer $\mathbf{B}_1$ below, e.g., is a two-place buffer. It possesses local variables ct_1 and ct_2, for storing values, and a counter ib to indicate how many values are currently stored.

$$\begin{aligned}
\mathbf{B}_1 \stackrel{\mathrm{def}}{=}\ & \lambda(x_p : \theta_c).\,\mathbf{new}\ [int]\,ib := 0,\, ct_1 := 0,\, ct_2 := 0\ \mathbf{in} \\
& \quad (x_p(\lambda(x_n : int).\,\mathbf{await}\ (!ib \le 1)\ \mathbf{then} \\
& \qquad\qquad (\mathbf{if}\ (!ib = 0)\ \mathbf{then}\ ct_1 := x_n\ \mathbf{else}\ ct_2 := x_n); \\
& \qquad\qquad ib :=!ib + 1)) \\
& \qquad (\lambda(x_r : \mathbf{var}[int]).\,\mathbf{await}\ (!ib \ge 1)\ \mathbf{then} \\
& \qquad\qquad x_r :=!ct_1; \\
& \qquad\qquad \mathbf{if}\ (!ib = 2)\ \mathbf{then}\ ct_1 :=!ct_2; \\
& \qquad\qquad ib :=!ib - 1)).
\end{aligned}$$

n-place buffers defined like $\mathbf{B}_1$ are single monolithic terms. Yet we can also define n-place buffers in a modular way, by connecting n one-place buffers. In this case,

however, it is necessary to distinguish the first $n-1$ buffers from the last, which acts as a barrier buffer. For the barrier buffer, we take the term $\mathbf{B}$ from above; the head buffers $\mathbf{HB}$ are defined as follows:

$$\begin{array}{l}\mathbf{HB} \stackrel{\text{def}}{=} \lambda(x_p : \theta_c).\, \lambda(x_{pt} : int \to \mathbf{comm}).\, \lambda(x_{gt} : \mathbf{var}[int] \to \mathbf{comm}).\\ \quad \mathbf{new}\ [bool]\, \mathit{fl}_h := \mathrm{ff},\ [int]\, ct_h := 0\ \mathbf{in}\\ \quad\quad (x_p(\lambda(x_n : int).\, \mathbf{await}\ (\neg!\mathit{fl}_h)\ \mathbf{then}\ (ct_h := x_n;\, \mathit{fl}_h := \mathrm{tt})))\\ \quad\quad\quad x_{gt}\\ \quad\quad \parallel \mathbf{rec}\, x.\, (\mathbf{if}\ (!\mathit{fl}_h)\ \mathbf{then}\ (x_{pt}(!ct_h);\, \mathit{fl}_h := \mathrm{ff};\, x)\ \mathbf{else}\ x)\end{array}$$

Again, x_p represents the clients; x_{pt} and x_{gt} represent the **put** and **get** procedures of the server buffer which $\mathbf{HB}$ is connected to. The arguments for the clients x_p are a **put** procedure defined in $\mathbf{HB}$ itself, and the **get** procedure of the server buffer. The boolean variable fl_h indicates whether $\mathbf{HB}$ is full (in which case a value is currently stored in ct_h). Whenever $\mathbf{HB}$ is full, it attempts to transmit its content to the server buffer, using the **put** procedure of the server. We can then define a 2-place buffer by

$$\mathbf{B}_2 \stackrel{\text{def}}{=} \lambda(x_p : \theta_c).\, \mathbf{B}\,(\mathbf{HB}\ x_p).$$

For proving that $\mathbf{B}_1$ and $\mathbf{B}_2$ are observationally congruent, i.e., $\mathbf{B}_1 \approx_{oc} \mathbf{B}_2$, we translate them into the π-calculus so to be able to exploit the proof techniques developed for it. First, however, applying the previously validated CIA law for β-reduction (F1) we infer $\mathbf{B}_2 \approx_{oc} \mathbf{B}'_2$, where

$$\begin{array}{l}\mathbf{B}'_2 \stackrel{\text{def}}{=} \lambda(x_p : \theta_c).\, \mathbf{new}\ [bool]\, \mathit{fl} := \mathrm{ff},\, \mathit{fl}_h := \mathrm{ff},\ [int]\, ct := 0,\ ct_h := 0\ \mathbf{in}\\ \quad\quad (x_p(\lambda(x_n : int).\, \mathbf{await}\ (\neg!\mathit{fl}_h)\ \mathbf{then}\ (ct_h := x_n;\, \mathit{fl}_h := \mathrm{tt})))\\ \quad\quad\quad (\lambda(x_r : \mathbf{var}[int]).\, \mathbf{await}\ (!\mathit{fl})\ \mathbf{then}\ (x_r :=!ct;\, \mathit{fl} := \mathrm{ff}))\\ \quad\quad \parallel \mathbf{rec}\, x.\, (\mathbf{if}\ (!\mathit{fl}_h)\ \mathbf{then}\ ((\mathbf{await}\ (\neg!\mathit{fl})\ \mathbf{then}\ (ct :=!ct_h;\, \mathit{fl} := \mathrm{tt}));\, \mathit{fl}_h := \mathrm{ff});\, x).\end{array}$$

It remains to show that $\mathbf{B}_1 \approx_{oc} \mathbf{B}'_2$. Let $\mathbf{B}_1^{\mathrm{body}}$ and $\mathbf{B}_2^{\mathrm{body}}$ be the bodies of the procedures $\mathbf{B}_1$ and $\mathbf{B}'_2$; they are obtained by stripping off the leading λ. It suffices to prove, by "bisimulation up to expansion" (cf. Section 3), that the encodings of $\mathbf{B}_1^{\mathrm{body}}$ and $\mathbf{B}_2^{\mathrm{body}}$ are bisimilar. Due to the presence of **await** we have to use locks, hence the encoding $[\![.]\!]^{\ell}$, and, as required by Theorem 2, close the encoding processes under the lock ℓ. Roughly, the bisimulation up to expansion $\mathcal{R}$ which we use for the proof is of the following form (we omit those processes resulting from calls from clients that have not been served immediately):

$$\begin{array}{ll}\mathcal{R} \stackrel{\text{def}}{=} \{((\nu\,\ell)(\ell.0 \mid [\![\mathbf{B}_1^{\mathrm{body}}]\!]^{\ell}_p),\ (\nu\,\ell)(\ell.0 \mid [\![\mathbf{B}_2^{\mathrm{body}}]\!]^{\ell}_p)), & \text{empty buffers}\\ \quad ((\nu\,\ell)(\ell.0 \mid [\![\mathbf{B}_1^{\mathrm{body}}(v)]\!]^{\ell}_p),\ (\nu\,\ell)(\ell.0 \mid [\![\mathbf{B}_2^{\mathrm{body}}(v)]\!]^{\ell}_p)), & \text{one value stored}\\ \quad ((\nu\,\ell)(\ell.0 \mid [\![\mathbf{B}_1^{\mathrm{body}}(v,w)]\!]^{\ell}_p),\ (\nu\,\ell)(\ell.0 \mid [\![\mathbf{B}_2^{\mathrm{body}}(v,w)]\!]^{\ell}_p)) & \text{two values stored}\\ \quad \mid v, w : int\}, & \end{array}$$

where (informally) $\mathbf{B}_1^{\mathrm{body}}(v)$ (resp. $\mathbf{B}_1^{\mathrm{body}}(v,w)$) is like $\mathbf{B}_1^{\mathrm{body}}$ but with a value v (resp. values v, w) stored in it; similarly for $\mathbf{B}_2^{\mathrm{body}}(v)$ (resp. $\mathbf{B}_2^{\mathrm{body}}(v,w)$). Consider for instance the first pair of the relation; here the buffers are empty, i.e., $!ib = 0$ in $\mathbf{B}_1^{\mathrm{body}}$ and $!\mathit{fl} =!\mathit{fl}_h = \mathrm{ff}$ in $\mathbf{B}_2^{\mathrm{body}}$. In that state the values of ct_1, ct_2, ct and ct_h do not matter, as they cannot be read. With corresponding

sequences of transitions, s, the buffers accept a value v from their client and, after storing it, signal the termination of that activity, thus

$$(\nu\,\ell)(\ell.0 \mid [\![\mathbf{B}_1^{\mathrm{body}}]\!]_p^\ell) \stackrel{s}{\Longrightarrow} (\nu\,\ell)(\ell.0 \mid [\![\mathbf{B}_1^{\mathrm{body}}(v)]\!]_p^\ell)$$
$$(\nu\,\ell)(\ell.0 \mid [\![\mathbf{B}_2^{\mathrm{body}}]\!]_p^\ell) \stackrel{s}{\Longrightarrow} (\nu\,\ell)(\ell.0 \mid [\![\mathbf{B}_2^{\mathrm{body}}(v)]\!]_p^\ell).$$

Precisely s is a sequence of visible actions consisting of: the client requesting that a value be stored ($x_{pt}\langle r\rangle$, where r will be used to signal the termination, see below); the buffers asking for a value (action $(\nu\,q)\overline{x}_n\langle q\rangle$, where x_n is a previously agreed channel to be used for invoking **get**, and q is a newly created one); the client providing a value (action $q\langle v\rangle$); and, finally, the buffer signalling that v has been stored (action $\overline{r}$). During this execution, the buffers hold the lock; it is released at the same time the client is informed of the termination.
Now $!ib = 1$ in $\mathbf{B}_1^{\mathrm{body}}$ and $!fl = \mathrm{tt}$ in $\mathbf{B}_2^{\mathrm{body}}$; value v is assigned to ct_1 and ct, respectively. We can assume this, despite $\mathbf{B}_2^{\mathrm{body}}$ first storing v in ct_h, as

$$(\nu\,\ell)(\ell.0 \mid [\![\mathbf{B}_2^{\mathrm{body}}(ct_h := v)]\!]_p^\ell) \succeq_\pi (\nu\,\ell)(\ell.0 \mid [\![\mathbf{B}_2^{\mathrm{body}}(ct := v)]\!]_p^\ell)$$

($\succeq_\pi$ denotes expansion as introduced in Section 3). Note that this application of the "up to" techniques is vital to the proof of the example (otherwise the relation would yield an extremely large number of pairs).
We do not know how to prove this or the previous examples directly in the operational semantics of ALGOL without going through a universal quantification over contexts (recall the problems with reasoning directly within the ALGOL semantics, discussed in the Introduction).

6 Refinements

For certain open CIA phrases, the ordinary π-calculus (weak early) bisimilarity turns out to be too discriminating, i.e., there exist observationally congruent CIA phrases whose translations into the π-calculus yield processes which are not bisimilar. Refining types, however, makes behavioural equivalences coarser (more process equivalences can be established), simply because the number of well-typed observers decreases.
In CIA, reading from a global variable does not influence the overall behaviour of a term as long as the value is not used in future interactions. This is not captured by the usual π-calculus bisimilarity, where all visible actions are treated identically. As a consequence, the equality (where κ is an integer variable)

$$\mathbf{new}\,[int]\,\iota := 0\ \mathbf{in}\ (\iota :=!\kappa \parallel \mathbf{output}\,5) = \mathbf{output}\,5, \tag{1}$$

which is operationally true in CIA, does not yield bisimilar π-calculus encodings; only the translation of the left-hand term may perform a $\mathbf{get}_\kappa$ transition.
To overcome this problem, we have adopted two measures. If A and B are open CIA phrases with free variables $\{x_i\}_i$, then instead of requiring that $[\![A]\!]$ and $[\![B]\!]$ be bisimilar, we demand bisimilarity between $\prod \mathbf{Reg}_i[\sigma(x_i)] \mid [\![A]\!]$ and $\prod \mathbf{Reg}_i[\sigma(x_i)] \mid [\![B]\!]$ (notice that in contrast to Section 4 the registers are not made local by a restriction), where σ is a function mapping all x_i's to some fixed initial value, e.g., 0 and "false". (Using some fixed initial value is possible because, intuitively, both program and observer have unlimited access to registers.) To ensure that—apart from input and output—communication between

program and observer is only possible via these registers, we use a type system distinguishing between the capabilities of using a channel in input and output (I/O types, cf. [PS93,BS98]). So, if ι is a free register, we can assign an external observer only the input capability on $\mathbf{get}_\iota$ and the output capability on $\mathbf{put}_\iota$. The corresponding equivalence on π-calculus processes, for which soundness theorems similar to Theorems 1 and 2 hold, is closer to the observational congruence in CIA than the ordinary bisimilarity; it allows us to prove (1), as well as, e.g.,

$$\mathbf{while}\ \mathrm{tt}\ \mathbf{do}\ (\iota := 0\,;\ \iota := 1) = \mathbf{while}\ \mathrm{tt}\ \mathbf{do}\ (\iota := 1\,;\ \iota := 0).$$

Again, this equality is valid in CIA but not in the π-calculus applying its ordinary bisimilarity.
Yet full abstraction is not gained by introducing I/O types. Consider the following example, where P is a free identifier:

$$\begin{array}{l}\mathbf{new}\ [int]\,\iota := 0\ \mathbf{in}\\ \quad P(\iota);\\ \quad \mathbf{if}\ (!\iota = 0)\ \mathbf{then\ skip}\\ \qquad\qquad \mathbf{else\ diverge}\end{array} \quad = \quad \begin{array}{l}\mathbf{new}\ [int]\,\iota := 0\ \mathbf{in}\\ \quad P(\iota);\\ \quad \mathbf{if}\ (!\iota = 0)\ \mathbf{then}\ (\mathbf{if}\ (!\iota = 1)\ \mathbf{then\ diverge\ else\ skip})\\ \qquad\qquad \mathbf{else\ diverge}.\end{array}$$

This example hinges on the unlimited access the observer has on fn_ι, in the π-calculus, once ι has been exported by calling P: Suppose the phrases have been signalled the termination of P, and ι is assigned a 0. One would naturally conclude that both phrases should terminate. Yet, the access the observer has gained on ι at the time P was called, does not cease with the termination of the procedure (recall that in the π-calculus encoding, P is a free identifier). Hence, the observer can write on ι even after having signalled the termination of P. Now, suppose the variable has already positively been tested for 0. In this case the left-hand phrase is bound to terminate, whereas the right-hand one may still diverge (if the observer sets ι to 1 before the second test).
For validating this example, a refined typing would be necessary, which allows one to express linearity (the observer could use certain names only once) and sequentiality (the observer could use a given name only as long he/she does not use another given name) constraints on the use of names. Such a type system could also be used to force the observer to respect the atomicity of **await** statements (before accessing a register, the observer should grab the lock; and release it afterwards). This would allow us to validate equivalences like

$$\mathbf{await}\ \mathrm{tt}\ \mathbf{then}\ (\kappa := !\kappa + 1;\ \kappa := !\kappa + 1) = \mathbf{await}\ \mathrm{tt}\ \mathbf{then}\ (\kappa := !\kappa + 2).$$

We see no technical difficulties in adopting such a type system, as we have done with the I/O types. Indeed, type systems for the π-calculus of this kind already exist [Hon96,KPT96,Kob97]; bisimilarity-based equivalences for them, as well as related algebraic properties, can be given by developing those for I/O types. However, even this further type refinement might not yield completeness of the interpretation. Moreover, our experiments have led us to the conviction that the I/O types are usually sufficient for reasoning, and that further typing would just make concrete proofs too complex.

7 Further results and discussion

The approach presented in this paper is applicable to other languages with state. We have, e.g., modelled a variation of CIA by using call-by-value, instead of call-

by-name, and by extending variables to higher order (this implies that not only values but also references and commands are stored in the registers); some of these modifications have been made following the languages in [MT90a,MT90b]. During the execution of an **await** statement, only one thread of computation is active (cf. Section 2 and [Bro96]), yielding a purely sequential behaviour. The degree of parallelism in the presence of an active (i.e., currently running) **await** statement can be increased by, e.g., a simultaneous execution of phrases which do not access variables affected by the **await** statement. This can be modelled, in the SOS semantics, by locks carrying along information about the concerned variables; in the π-calculus semantics, multiple locks can be introduced. The necessary information on the access to variables can be gained by some simple preliminary static analysis. Of course, such an increase in parallelism changes the overall semantics; nevertheless there are behavioural correspondences between the more sequential and the more parallel version: First, if two phrases are bisimilar in the more parallel version, then they are also bisimilar in the sequential one (cutting off branches from the transition systems). Second, a phrase may yield a divergent computation (transition trace) in the sequential semantics if and only if it does so in the parallel one (transitions occurring interleaved in the parallel semantics are *causally independent*, so they can be interchanged resulting in a computation of the sequential semantics). We have proved both these results by reasoning on the π-calculus translations.

We have considered as closed only such programs that do not possess open identifiers *nor* variables, using explicit input and output constructs An alternative approach is to provide the observer with direct access to the global variables:

$$\begin{aligned} \langle P, \sigma \rangle &\xrightarrow{\mathbf{read}_\iota(v)} \langle P, \sigma \rangle && \text{if } \Gamma(\iota) = \mathbf{var}[\tau] \text{ and } \sigma(\iota) = v, \\ \langle P, \sigma \rangle &\xrightarrow{\mathbf{write}_\iota(v)} \langle P, \sigma\{\iota \leftarrow v\} \rangle && \text{if } \Gamma(\iota) = \mathbf{var}[\tau]. \end{aligned}$$

To obtain operational correspondence and soundness (cf. Section 4), the translation into the π-calculus would have to take into account I/O types (cf. Section 6). Semantics given to IA and CIA by O'Hearn and Tennent [OT95], Pitts [Pit96] and Brookes [Bro96], make use of relational parametricity. Comparing proofs conducted in these theories, with bisimulation-based proofs carried out in the π-calculus might clarify the relationship between these two notions.

Acknowledgements
We thank J. Esparza, P. W. O'Hearn and U. S. Reddy for helpful discussions.
This work was partly supported by Teilprojekt A3 SAM of SFB 342 "Werkzeuge und Methoden für die Nutzung paralleler Rechnerarchitekturen", and by the PROCOPE project 9723064.

References

[AKH92] S. Arun-Kumar and M. Hennessy. An efficiency preorder for processes. *Acta Informatica*, 29:737–760, 1992.

[AM96] S. Abramsky and G. McCusker. Linearity, sharing and state: a fully abstract game semantics for idealized algol with active expressions. *Electronic Notes in Theoretical Computer Science*, 3, 1996.

[AM97] S. Abramsky and G. McCusker. Full abstraction for idealized algol with passive expressions. Submitted for Publication, 1997.

[Bro96] S. Brookes. The essence of parallel algol. In *Proc. LICS'96*. IEEE, 1996. App. in vol. 2 of [OT97].

[BS98] M. Boreale and D. Sangiorgi. Bisimulation in name-passing calculi without matching. In *thirteen LICS Conf.* IEEE Computer Society Press, 1998.

[FHJ95] W. Ferreira, M. Hennessy, and A. Jeffrey. A theory of weak bisimulation for core CML. Technical report, School of Cognitive and Computing Sciences, University of Sussex, 1995.

[FMS96] M. Fiore, E. Moggi, and D. Sangiorgi. A fully-abstract model for the π-calculus. In *11th LICS*. IEEE Computer Society Press, 1996.

[Hon96] K. Honda. Composing processes. In *Proc. 23rd POPL*. ACM Press, 1996.

[How96] D. J. Howe. Proving congruence of bisimulation in functional programming languages. *Information and Computation*, 124(2):103–112, 1996.

[Jon93] C. B. Jones. A π-calculus semantics for an object-based design notation. In E. Best, editor, *Proc. CONCUR '93*, volume 715 of *LNCS*, pages 158–172. Springer Verlag, 1993.

[Kob97] N. Kobayashi. A partially deadlock-free typed process calculus. In *Proc. 12th LICS Conf.* IEEE Computer Society Press., 1997.

[KPT96] N. Kobayashi, B. C. Pierce, and D. N. Turner. Linearity and the π-calculus. In *Proc. 23rd POPL*. ACM Press, 1996.

[KS98] J. Kleist and D. Sangiorgi. Imperative objects and mobile processes. In *Proc. IFIP Working Conference on Programming Concepts and Methods (PROCOMET'98)*. North-Holland, 1998.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[Mil91] R. Milner. The polyadic π-calculus: a tutorial. Technical Report ECS–LFCS–91–180, LFCS, 1991.

[Mil92] R. Milner. Functions as processes. *J. of Math. Struct. in Computer Science*, 17:119–141, 1992.

[MPW89] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. Report ECS-LFCS-89-85,86, Dept. of Computer Science, University of Edinburgh, 1989. Two volumes, also appeared in *Information and Computation 100:1–77*,1992.

[MS88] A. R. Meyer and K. Sieber. Towards fully abstract semantics for local variables: preliminary report. In *Proc. 15th POPL*, 1988. Also appeared in vol. 2 of [OT97].

[MS92] R. Milner and D. Sangiorgi. The problem of weak bisimulation up-to. In *Proc. CONCUR'92*, volume 630 of *LNCS*. Springer, 1992.

[MT90a] I. A. Mason and C. L. Talcott. Equivalence in functional languages with effects. Technical report, University of Stanford, 1990.

[MT90b] I. A. Mason and C. L. Talcott. References, local variables and operational reasoning. Technical report, University of Stanford, 1990.

[OT95] P. W. O'Hearn and R. D. Tennent. Parametricity and local variables. *J. ACM*, 42(3):658–709, 1995. Also appeared in [OT97].

[OT97] P. W. O'Hearn and R. D. Tennent, editors. *ALGOL-like Languages*. Progress in Theoretical Computer Science. Birkhäuser, 1997. Two volumes.

[Pit96] A. M. Pitts. Reasoning about local variables with operationally-based logical relations. In *Proc. LICS'96*. IEEE, 1996. Also appeared in vol. 2 of [OT97].

[PS93] B. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes. In *Proc. LICS'93*. IEEE, 1993. Also appeared in Mathematical Structures in Computer Science 6:5(1996) pp. 409–453.

[Rey81] J. C. Reynolds. The essence of ALGOL. In *Algorithmic Languages*, pages 345–372. North-Holland, 1981. Also appeared in vol. 1 of [OT97].

[San95a] D. Sangiorgi. Lazy functions and mobile processes. Technical Report RR-2515, INRIA-Sophia Antipolis, 1995. To appear in "Festschrift volume in honor of Robin Milner's 60th birthday", MIT Press.

[San95b] Davide Sangiorgi. On the proof method for bisimulation (extended abstract). In *Proc. MFCS'95*, volume 969 of *LNCS*, pages 479–488. Springer Verlag, 1995. Full version available electronically.

[Sta96] Ian Stark. A fully abstract domain model for the π-calculus. In *Proc. LICS'96*, pages 36–42. IEEE Computer Society Press, 1996.

[Wal95] D. Walker. Objects in the π-calculus. *Information and Computation*, 116:253–271, 1995.

$$\Gamma \vdash v : \tau$$

$$\Gamma \vdash \iota : \mathbf{var}[\tau] \quad \text{when } \Gamma(\iota) = \mathbf{var}[\tau]$$

$$\frac{\Gamma \vdash E_1 : \mathbf{exp}[\tau] \quad \Gamma \vdash E_2 : \mathbf{exp}[\tau]}{\Gamma \vdash E_1 \otimes E_2 : \mathbf{exp}[\tau]} \otimes : \tau \times \tau \to \tau$$

$$\Gamma \vdash \mathbf{skip} : \mathbf{comm}$$

$$\frac{\Gamma \vdash E : \mathbf{exp}[\tau]}{\Gamma \vdash \mathbf{output}\, E : \mathbf{comm}}$$

$$\frac{\Gamma \vdash \iota : \mathbf{var}[\tau]}{\Gamma \vdash \mathbf{input}\, \iota : \mathbf{comm}}$$

$$\frac{\Gamma \vdash V : \mathbf{var}[\tau] \quad \Gamma \vdash E : \mathbf{exp}[\tau]}{\Gamma \vdash V := E : \mathbf{comm}}$$

$$\frac{\Gamma \vdash C_1 : \mathbf{comm} \quad \Gamma \vdash C_2 : \mathbf{comm}}{\Gamma \vdash C_1 ; C_2 : \mathbf{comm}}$$

$$\frac{\Gamma \vdash C_1 : \mathbf{comm} \quad \Gamma \vdash C_2 : \mathbf{comm}}{\Gamma \vdash C_1 \parallel C_2 : \mathbf{comm}}$$

$$\frac{\Gamma \vdash B : \mathbf{exp}[bool] \quad \Gamma \vdash P_1 : \theta \quad \Gamma \vdash P_2 : \theta}{\Gamma \vdash \mathbf{if}\, B\, \mathbf{then}\, P_1\, \mathbf{else}\, P_2 : \theta}$$

$$\frac{\Gamma \vdash B : \mathbf{exp}[bool] \quad \Gamma \vdash C : \mathbf{comm}}{\Gamma \vdash \mathbf{while}\, B\, \mathbf{do}\, C : \mathbf{comm}}$$

$$\frac{\Gamma \vdash B : \mathbf{exp}[bool] \quad \Gamma \vdash C : \mathbf{comm}}{\Gamma \vdash \mathbf{await}\, B\, \mathbf{then}\, C : \mathbf{comm}}\ C \text{ seq. comp. of ass., cond.}$$

$$\frac{\Gamma \vdash E : \mathbf{exp}[\tau] \quad \Gamma, \iota : \mathbf{var}[\tau] \vdash C : \mathbf{comm}}{\Gamma \vdash \mathbf{new}\, [\tau]\iota := E\, \mathbf{in}\, C : \mathbf{comm}}$$

$$\Gamma \vdash x : \theta \quad \text{when } \Gamma(x) = \theta$$

$$\frac{\Gamma, x : \theta \vdash P : \theta}{\Gamma \vdash \mathbf{rec}\, x.P : \theta}$$

$$\frac{\Gamma, x : \theta \vdash P : \theta'}{\Gamma \vdash \lambda(x : \theta).P : (\theta \to \theta')}$$

$$\frac{\Gamma \vdash P_1 : (\theta \to \theta') \quad \Gamma \vdash P_2 : \theta}{\Gamma \vdash P_1(P_2) : \theta'}$$

Table 1: Syntax and typing of CIA

$$
\begin{aligned}
[\![\iota]\!]_p &\stackrel{\text{def}}{=} \overline{p}\langle \mathbf{get}_\iota, \mathbf{put}_\iota\rangle.0 \\
[\![v]\!]_p &\stackrel{\text{def}}{=} \overline{p}\langle v\rangle.0 \\
[\![!V]\!]_p &\stackrel{\text{def}}{=} (\nu\, q)([\![V]\!]_q \mid q(gt, pt).gt(x).\overline{p}\langle x\rangle.0) \\
[\![E_1 \otimes E_2]\!]_p &\stackrel{\text{def}}{=} (\nu\, q, r)([\![E_1]\!]_q \mid [\![E_2]\!]_r \mid q(x).r(y).\overline{p}\langle x \otimes y\rangle.0) \\
[\![\mathbf{skip}]\!]_p &\stackrel{\text{def}}{=} \overline{p}.0 \\
[\![\mathbf{output}\, E]\!]_p &\stackrel{\text{def}}{=} (\nu\, q)([\![E]\!]_q \mid q(x).\overline{\mathbf{out}}\langle x\rangle.\overline{p}.0) \\
[\![\mathbf{input}\, \iota]\!]_p &\stackrel{\text{def}}{=} \mathbf{in}(x).\overline{\mathbf{put}}_\iota\langle x\rangle.\overline{p}.0 \\
[\![V := E]\!]_p &\stackrel{\text{def}}{=} (\nu\, q, r)([\![V]\!]_q \mid [\![E]\!]_r \mid q(gt, pt).r(x).\overline{pt}\langle x\rangle.\overline{p}.0) \\
[\![C_1;\, C_2]\!]_p &\stackrel{\text{def}}{=} (\nu\, q)([\![C_1]\!]_q \mid q.[\![C_2]\!]_p) \\
[\![C_1 \parallel C_2]\!]_p &\stackrel{\text{def}}{=} (\nu\, q, r)([\![C_1]\!]_q \mid [\![C_2]\!]_r \mid q.r.\overline{p}.0) \\
[\![\mathbf{if}\, B\, \mathbf{then}\, P_1\, \mathbf{else}\, P_2]\!]_p &\stackrel{\text{def}}{=} (\nu\, q)([\![B]\!]_q \mid q(x).([x = \mathrm{tt}]\,[\![P_1]\!]_p \mid [x = \mathrm{ff}]\,[\![P_2]\!]_p)) \\
[\![\mathbf{while}\, B\, \mathbf{do}\, C]\!]_p &\stackrel{\text{def}}{=} (\nu\, a)(!a.(\nu\, q)([\![B]\!]_q \mid q(x).([x = \mathrm{tt}]\,(\nu\, r)([\![C]\!]_r \mid r.\overline{a}.0) \mid \\
&\qquad\qquad [x = \mathrm{ff}]\,\overline{p}.0)) \mid \overline{a}.0) \\
[\![\mathbf{new}\, [\tau]\, \iota := E\, \mathbf{in}\, C]\!]_p &\stackrel{\text{def}}{=} (\nu\, q)([\![E]\!]_q \mid q(x).(\nu\, \mathrm{fn}_\iota)(\mathbf{Reg}_\iota[x] \mid [\![C]\!]_p)) \\
[\![x]\!]_p &\stackrel{\text{def}}{=} \overline{x}\langle p\rangle.0 \\
[\![\mathbf{rec}\, x.\, P]\!]_p &\stackrel{\text{def}}{=} (\nu\, x)(!x(r).[\![P]\!]_r \mid \overline{x}\langle p\rangle.0) \\
[\![\lambda(x : \theta).\, P]\!]_p &\stackrel{\text{def}}{=} (\nu\, v)(\overline{p}\langle v\rangle.v(x, q).[\![P]\!]_q) \\
[\![P_1 P_2]\!]_p &\stackrel{\text{def}}{=} (\nu\, q)([\![P_1]\!]_q \mid q(v).(\nu\, x)(\overline{v}\langle x, p\rangle.!x(r).[\![P_2]\!]_r))
\end{aligned}
$$

Table 2: Encoding CIA-{**await**} in the π-calculus

$$
\begin{aligned}
[\![!V]\!]^\ell_p &\stackrel{\text{def}}{=} (\nu\, q)([\![V]\!]^\ell_q \mid q(gt, pt).\overline{\ell}.gt(x).(\ell.0 \mid \overline{p}\langle x\rangle.0)) \\
[\![\mathbf{output}\, E]\!]^\ell_p &\stackrel{\text{def}}{=} (\nu\, q)([\![E]\!]^\ell_q \mid q(x).\overline{\ell}.\overline{\mathbf{out}}\langle x\rangle.(\ell.0 \mid \overline{p}.0)) \\
[\![\mathbf{input}\, \iota]\!]^\ell_p &\stackrel{\text{def}}{=} \overline{\ell}.\mathbf{in}(x).(\ell.0 \mid \overline{\ell}.\overline{\mathbf{put}}_\iota\langle x\rangle.(\ell.0 \mid \overline{p}.0)) \\
[\![V := E]\!]^\ell_p &\stackrel{\text{def}}{=} (\nu\, q, r)([\![V]\!]^\ell_q \mid [\![E]\!]^\ell_r \mid q(gt, pt).r(x).\overline{\ell}.\overline{pt}\langle x\rangle.(\ell.0 \mid \overline{p}.0)) \\
[\![\mathbf{await}\, B\, \mathbf{then}\, C]\!]^\ell_p &\stackrel{\text{def}}{=} (\nu\, a)(!a.(\nu\, q)(\overline{\ell}.[\![B]\!]_q \mid q(x).([x = \mathrm{tt}]\,(\nu\, r)([\![C]\!]_r \mid r.(\ell.0 \mid \overline{p}.0)) \mid \\
&\qquad\qquad [x = \mathrm{ff}]\,(\ell.0 \mid \overline{a}.0))) \mid \overline{a}.0)
\end{aligned}
$$

Table 3: Encoding Full CIA in the π-calculus—Modifications to Table 2

Author Index

Abadi, M. 1
Aceto, L. 41
Arruabarrena, R. 56

Baldan, P. 73
Barthe, G. 90
Benke, M. 104
Bodei, C. 120
Boer, F. S. de 135
Bogaert, B. 150
Boreale, M. 165

Cardelli, L. 212
Cleaveland, R. 288
Corradini, A. 73

Dantsin, E. 180
Degano, P. 120
De Nicola, R. 165

Esparza, J. 14

Ghani, N. 197
Gordon, A. D. 212

Huhn, M. 227

Ingólfsdóttir, A. 41
Iyer, P. 288

Knoop, J. 14

Lenisa, M. 243
Lucio, P. 56

Maneth, S. 258
Montanari, U. 73
Muscholl, A. 273

Narasimha, M. 288
Navarro, M. 56
Niebert, P. 227
Nielson, F. 120
Nielson, H. R. 120

Paiva, V. de 197
Pugliese, R. 165

Ritter, E. 197
Röckl, C. 306

Sangiorgi, D. 31, 306
Seynhaeve, F. 150

Tison, S. 150

Voronkov, A. 180

Wallner, F. 227