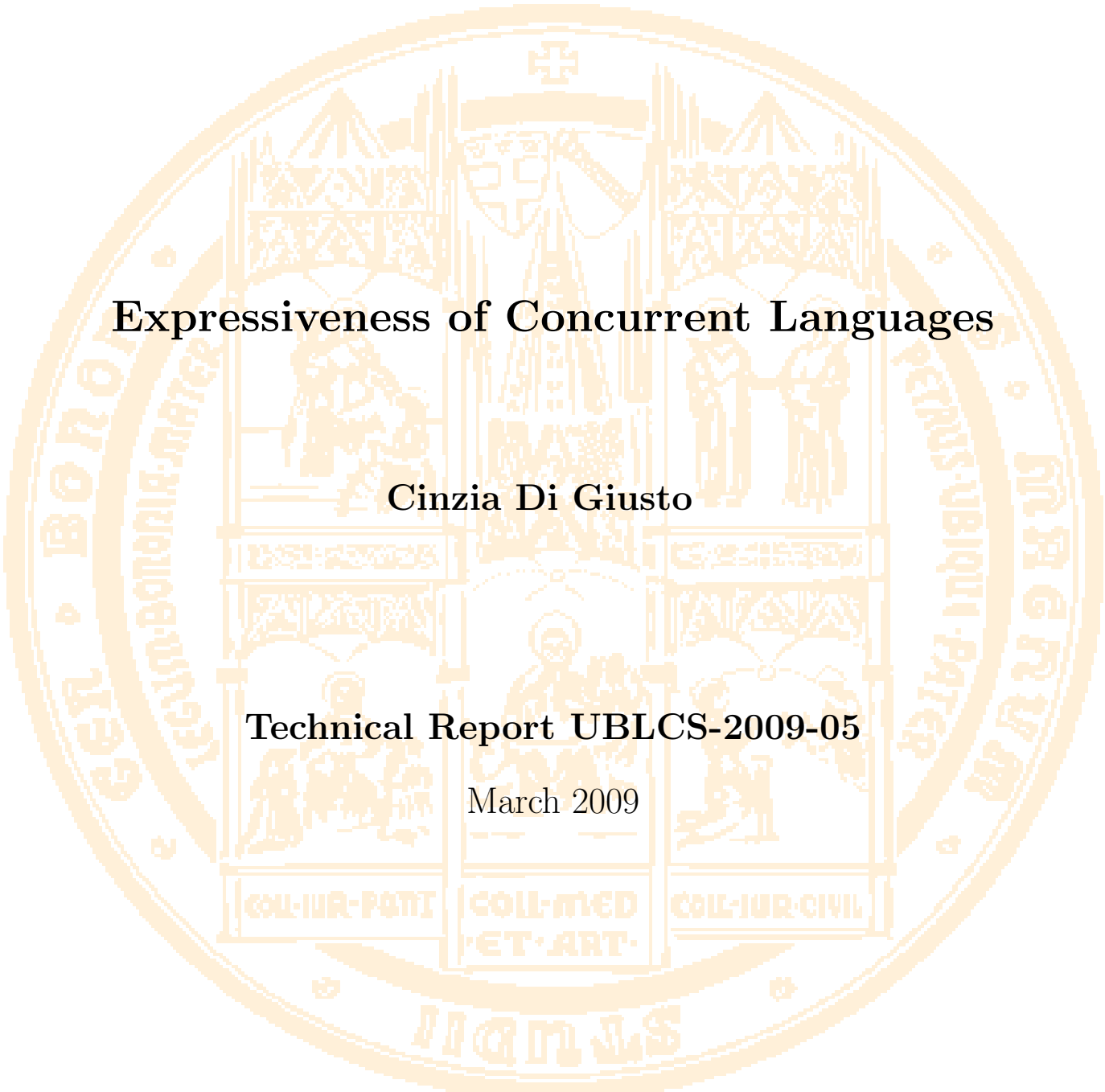

Expressiveness of Concurrent Languages

Cinzia Di Giusto

Technical Report UBLCS-2009-05

March 2009

Department of Computer Science

University of Bologna

Mura Anteo Zamboni 7
40127 Bologna (Italy)

The University of Bologna Department of Computer Science Research Technical Reports are available in PDF and gzipped PostScript formats via anonymous FTP from the area `ftp.cs.unibo.it:/pub/TR/UBLCS` or via WWW at URL `http://www.cs.unibo.it/`. Plain-text abstracts organized by year are available in the directory `ABSTRACTS`.

Recent Titles from the UBLCS Technical Report Series

2008-11 *Programming service oriented applications*, Guidi, C., Lucchi, R., June 2008.

2008-12 *A Foundational Theory of Contracts for Multi-party Service Composition*, Bravetti, M., Zavattaro, G., June 2008.

2008-13 *A Theory of Contracts for Strong Service Compliance*, Bravetti, M., Zavattaro, G., June 2008.

2008-14 *A Uniform Approach for Expressing and Axiomatizing Maximal Progress and Different Kinds of Time in Process Algebra*, Bravetti, M., Gorrieri, R., June 2008.

2008-15 *On the Expressive Power of Process Interruption and Compensation*, Bravetti, M., Zavattaro, G., June 2008.

2008-16 *Stochastic Semantics in the Presence of Structural Congruence: Reduction Semantics for Stochastic Pi-Calculus*, Bravetti, M., July 2008.

2008-17 *Measures of conflict and power in strategic settings*, Rossi, G., October 2008.

2008-18 *Lebesgue's Dominated Convergence Theorem in Bishop's Style*, Sacerdoti Coen, C., Zoli, E., November 2008.

2009-01 *A Note on Basic Implication*, Guidi, F., January 2009.

2009-02 *Algorithms for network design and routing problems (Ph.D. Thesis)*, Bartolini, E., February 2009.

2009-03 *Design and Performance Evaluation of Network on-Chip Communication Protocols and Architectures (Ph.D. Thesis)*, Concer, N., February 2009.

2009-04 *Kernel Methods for Tree Structured Data (Ph.D. Thesis)*, Da San Martino, G., February 2009.

2009-05 *Expressiveness of Concurrent Languages (Ph.D. Thesis)*, di Giusto, C., February 2009.

2009-06 *EXAM-S: an Analysis tool for Multi-Domain Policy Sets (Ph.D. Thesis)*, Ferrini, R., February 2009.

2009-07 *Self-Organizing Mechanisms for Task Allocation in a Knowledge-Based Economy (Ph.D. Thesis)*, Marcozzi, A., February 2009.

2009-08 *3-Dimensional Protein Reconstruction from Contact Maps: Complexity and Experimental Results (Ph.D. Thesis)*, Medri, F., February 2009.

2009-09 *A core calculus for the analysis and implementation of biologically inspired languages (Ph.D. Thesis)*, Versari, C., February 2009.

2009-10 *Probabilistic Data Integration*, Magnani, M., Montesi, D., March 2009.

Dottorato di Ricerca in Informatica
Università di Bologna e Padova

Expressiveness of Concurrent Languages

Cinzia Di Giusto

March 2009

Coordinatore:	Tutore:
Simone Martini	Maurizio Gabbrielli

Progress in theoretical computer science brings understanding in place of confusion and confidence in place of fear.

C.A.R. Hoare
Preface of Milner's book Communication and Concurrency

Abstract

The aim of this thesis is to go through different approaches for proving expressiveness properties in several concurrent languages. We analyse four different calculi exploiting for each one a different technique.

We begin with the analysis of a synchronous language, we explore the expressiveness of a fragment of $CCS_!$ (a variant of Milner's CCS where replication is considered instead of recursion) w.r.t. the existence of faithful encodings (i.e. encodings that respect the behaviour of the encoded model without introducing unnecessary computations) of models of computability strictly less expressive than Turing Machines. Namely, grammars of types 1,2 and 3 in the Chomsky Hierarchy.

We then move to asynchronous languages and we study full abstraction for two Linda-like languages. Linda can be considered as the asynchronous version of CCS plus a shared memory (a multiset of elements) that is used for storing messages. After having defined a denotational semantics based on traces, we obtain fully abstract semantics for both languages by using suitable abstractions in order to identify different traces which do not correspond to different behaviours.

Since the ability of one of the two variants considered of recognising multiple occurrences of messages in the store (which accounts for an increase of expressiveness) reflects in a less complex abstraction, we then study other languages where multiplicity plays a fundamental role. We consider the language CHR (Constraint Handling Rules) a language which uses multi-headed (guarded) rules. We prove that multiple heads augment the expressive power of the language. Indeed we show that

if we restrict to rules where the head contains at most n atoms we could generate a hierarchy of languages with increasing expressiveness (i.e. the CHR language allowing at most n atoms in the heads is more expressive than the language allowing at most m atoms, with $m < n$).

Finally we analyse a language similar but simpler than CHR. The κ-calculus is a formalism for modelling molecular biology where molecules are terms with internal state and sites, bonds are represented by shared names labelling sites, and reactions are represented by rewriting rules. Depending on the shape of the rewriting rules, several dialects of the calculus can be obtained. We analyse the expressive power of some of these dialects by focusing on decidability and undecidability for problems like reachability and coverability.

Acknowledgements

I am greatly indebted to my advisor Maurizio Gabbrielli. During the last four years he has always been present, giving suggestions, listening to me, finding counter examples, reading and correcting my drafts. Thanks for all this and above all for having been a friend.

I would like to thank Catuscia Palamidessi for having hosted me in her team *Comete* at École Polytechnique in Paris. I have greatly benefit from her and her team experience and suggestions. In particular I owe much to Frank Valencia, for the discussions, the criticisms, for the huge amount of things he taught me and also for the laughs and for the friendship.

Many thanks to Catuscia Palamidessi and Frank de Boer for having accepted to evaluate this dissertation. Thanks for your comments.

I would like to thank Jorge Pérez, who proofread this work and whose suggestions were fundamental. Thanks for your priceless help.

I am also indebted to many people with whom I discussed the work in this dissertation: Nadia Busi, Giorgio Delzanno, Roberto Gorrieri, Cosimo Laneve, Simone Martini, Maria Chiara Meo, Mogens Nielsen, Fabio Panzieri, Davide Sangiorgi and Gianluigi Zavattaro.

Finally I am most grateful to all friends I met here in Bologna and during my PhD: Antonio, Claudio, Cristian, Giulio, Marco, Matteo, Micaela, Stefano and Valentina.

I dedicate this thesis to Enrico, thanks for all the love, the support, the patience and for the things to come.

Cinzia Di Giusto

Bologna, March 2009

Contents

List of Tables

List of Figures

Chapter 1

Introduction

It was a dark and stormy night ...

Charles M. Schulz
Snoopy

This dissertation investigates and examines the application of some of the techniques used for assessing the expressiveness of concurrent languages. We will go through the different techniques by analysing several languages and providing some novel results; in some cases we will apply techniques commonly used in the process algebra field to other areas like logic programming. Here we generally introduce the thesis by describing the domain of interest.

1.1 Concurrency

Concurrency theory is concerned with the modelling of dynamic systems that consist of the composition of several parts that interact among each other. Nowadays many systems fit in this definition: Internet, mobile computing, parallel computing (grids), and if we consider systems that cannot be classified under the standard

computer world we can refer to biological systems: interactions between molecules or, abstracting even more, to complex interactions between societies (animal or even human interactions). More precisely a concurrent model can be represented by a group of independent entities called *processes* which collaborate and combine by exchanging messages.

In the past twenty years, since the importance and ubiquity of these systems have been proven real, scientists have tried to give a mathematical foundation to concurrent computations. These systems differentiate from the sequential ones not only in the domain of application but in their very conceptual foundations: concurrent systems usually deal with infinite computations – therefore the system cannot be described as a function with inputs and outputs – and non determinism plays a fundamental role in their behaviour.

Starting from the '80s, several models for concurrent computations have been proposed. The first approach was to consider extensions of the λ-calculus with some form of parallelism – consider for example the works on PCF by Plotkin [109]. Then in 1980 Milner [91] proposed a new approach represented by the Calculus of Communicating Systems (CCS). CCS is generally accepted as the first proposal of process algebras. In some sense, CCS resembles the λ-calculus: processes are treated in the same way the λ-calculus deals with functions. The aim of the two languages is clearly different: CCS constructors focus on how to express interaction and communication between processes whereas the λ-calculus describes functions and their behaviour.[1]

Independently in the same years two other languages were proposed: Hoare's Communicating Sequential Processes (CSP) [71], and Bergstra and Klop's Algebra of Communicating Processes (ACP) [11]. Both languages, similarly to CCS, allow the description of systems in terms of processes that operate independently, and interact with each other through message-passing communication.

Taking inspiration from these basic calculi a huge variety of languages has been proposed. Take as an example CCS: it can be extended in several ways. Consider,

[1] For a refined version of [91] see [93].

for instance, the type of messages exchanged. In CCS only pure synchronisation is allowed: processes synchronise on some given channels but the configuration of connection cannot change dynamically. Milner et al. then proposed the π-calculus [95] as a generalisation of CCS in precisely this sense: This is achieved by giving the possibility of passing names of connections along the channels. Taking again inspiration from the λ-calculus where terms can be passed around, Sangiorgi in [113] generalises the notion of mobility of the π-calculus, letting the process to exchange not only names but also other processes. In this way a series of languages can be obtained called *Higher Order π-calculi* (see also [84, 115]).

Once the relevance of a language has been proved, one can also focus on certain features and change them to serve specific purposes. Take for example protocols of communication: the π-calculus (and CCS) uses synchronous communication which is not always convenient for describing certain settings. Consequently Honda and Tokoro in [73] (and independently Boudol in [15]) proposed an asynchronous version of the π-calculus. Or one can focus on mobility of processes and instead of Higher Order calculi one can think of a sort of bounded place in which computation can occur as in Mobile ambients [28] by Cardelli and Gordon or as in the Join calculus [57] by Fournet and Gonthier.

Hence, from these few examples one can see that many variants of the landmark languages have been introduced. There are many reasons that support this phenomenon (see also [98]): for example new variants may simplify the presentation of the calculus (a constructor can be easier to understand or implement) or a specific language can be tailored to a specific domain of application. Therefore there are two problems that need to be addressed. First, every time a new language is proposed two legitimate questions need to be answered: Why is this language useful? What does the new language add w.r.t. other proposals? These raise the issue of *expressiveness*. Second, we have to endow these languages with suitable reasoning techniques that permits to analyse the terms of the language. The thesis focuses mainly on expressiveness; however, we find it necessary to mention both issues. This is the content of the next two sections.

1.2 Expressiveness

When considering sequential languages the expressiveness problem has been settled already and falls under the realm of computability theory. Due to the peculiarities of concurrent models, computability theory cannot always be used as it falls short to account for more specific behaviours: e.g. two languages can be proven to be Turing powerful and nevertheless because of the action of distribute and concurrent operations one can exhibit a behaviour that is not observable in the second language. This distinctive aspect will be clarified later.

Expressiveness studies for concurrent languages can be classified depending on the reference for comparison. This way, while absolute expressiveness describes the possible behaviours of a process algebra without referring back to other calculi, relative expressiveness aims to compare two different process algebras. See, e.g. Parrow [105] for a recent survey on the issue.

In this thesis we are mainly interested in the notion of relative expressiveness. Suppose we have two calculi $\mathcal{L}_1$ and $\mathcal{L}_2$, we want to be able to determine if $\mathcal{L}_1$ is as expressive as $\mathcal{L}_2$ or if the two calculi could be distinguished in some way. When we say that one language is as expressive as another we presuppose the existence of some encoding/mapping $[\![\cdot]\!] : \mathcal{L}_2 \to \mathcal{L}_1$ that translates the terms of $\mathcal{L}_2$ into terms of $\mathcal{L}_1$. Notice that the languages considered are usually Turing powerful therefore some encoding will always exist; however we are interested only in certain kinds of mappings: the ones that satisfy some given and *reasonable* criteria.

This is also known as *language encoding*.

1.2.1 Language encoding

Language encoding or embedding was first proposed by Shapiro in [120] and by de Boer and Palamidessi in [38].

The idea is that a language $\mathcal{L}$ is more expressive than a language $\mathcal{L}'$ or, equivalently, $\mathcal{L}'$ can be encoded in $\mathcal{L}$, if each program written in $\mathcal{L}'$ can be translated into an $\mathcal{L}$ program in such a way that: (1) the intended observable behaviour of

the original program is preserved (more precisely, it can be reconstructed from the observables of the translated program) and (2) the translation process satisfies some additional property (conditions) which indicate how easy this process is and how reasonable the decoding of the observables is. For example, typically one requires the translation to be compositional w.r.t. (some of) the operators of the language. As discussed in [38], these additional properties are needed in order to use the notion of encoding as a tool for language comparison: i.e. for showing that under certain hypothesis a language cannot be encoded in another. In fact, since the languages that we are considering are often Turing complete, they would always admit an encoding, provided that the observables for the target language are powerful enough. Indeed, if we "flatten" the notion of observables: e.g. by associating to every program P the same trivial empty observables, then every program falls in the same equivalence class.

The technique is ubiquitous in process calculi. As an example in [100], Palamidessi shows that if we fix some criteria for the encoding (a uniform fully distributed translation) then it is not possible to encode the π-calculus into the asynchronous π-calculus up to any *reasonable* notion of equivalence. This is achieved by showing the incapability of the asynchronous π-calculus to break the symmetry in certain configurations. This is known as the leader election problem (see [125] for a survey on the subject).

Gorla in [67] and more recently in [68], while comparing communication primitives (synchronism, communication media, pattern matching) discusses the possible criteria for defining when an encoding could be considered reasonable. In particular the author requires an encoding to be (1) compositional, (2) name invariant, (3) divergence preserving, (4) operationally correspondent and (5) success sensitive.

An example where the same approach has been followed but the area of application is different is [85] by Laneve and Vitale. They show that the κ-calculus, a language for modelling molecular biology, is more expressive than a restricted version of the calculus, called $\texttt{nano}\kappa$, which is obtained by restricting to "binary reactants" only (that is, by allowing at most two process terms in the left hand side

of rules, while n terms are allowed in κ). This result is obtained by showing that, under some specific assumptions, a particular (self-assembling) protocol cannot be expressed in nanoκ.

1.2.2 Decidability vs Undecidability

Until now we have discussed techniques that aim to prove directly that either two languages are equivalent by providing an encoding and showing its correctness with bisimulation or that two languages could be told apart by showing that under some hypothesis such an encoding does not exist. In order to show that an encoding between two languages $\mathcal{L}_1$ and $\mathcal{L}_2$ does not exist one could also proceed indirectly by showing that a certain property is decidable in $\mathcal{L}_1$ but not in $\mathcal{L}_2$.

This is particular relevant in the context of concurrent languages. Indeed the presence of non-determinism permits to provide easily unfaithful encodings of Turing equivalent models, i.e. encodings which add unnecessary computations, a sort of *garbage*. The presence of this garbage gives the possibility of deciding properties like reachability of a given configuration, which are usually undecidable in Turing equivalent models.

In literature we find several examples of this technique: in [19] Busi et al. show that termination is decidable in a CCS-like language while in [26] Busi and Zavattaro show that reachability is decidable in a fragment of the Mobile Ambients calculus.

Decidability proofs can often be carried out by reducing to problems on Petri Nets, a well known and extensively studied formalism. One of the first studies in this sense is [48] by Dufourd et al. which gives a classification of decidable properties on classes of Petri Nets. In particular, the technique described in [55] (derived from studies on Petri Nets) has been extensively used for comparing the expressive power of reactive systems. Finkel and Schnoebelen show that the presence of a well-quasi-ordering on transition systems permits to prove the existence of a terminating computation. For example in [3] Abdulla et al. use well structured transition systems (transition systems equipped with a well quasi ordering) to compare the expressive power of Multiset Rewriting Systems w.r.t. decidability of properties like

coverability and reachability. Similarly in [25] Busi and Zavattaro, by exploiting the technique of well structured transition systems, prove that altough the language analysed is Turing powerful, recognising terminating computations is decidable.

1.3 Reasoning Techniques

The second problem that needs to be tackled is how elements or programs of the calculus can be compared. This is important if one considers that usually one is interested in making sure an implementation is faithful to some (formal) specification. The correctness of such a comparison is formalised by a behavioural (or observational) equivalence. Basically, one has two ways of defining observational (or behavioural) equivalences: either by using a proper *bisimilarity* or a *testing semantics* (see [83] for a survey on equivalence checking).

Bisimilarity [117, 118] is generally accepted as the finest behavioural equivalence one would like to impose on processes. Bisimulation was originally introduced by Milner and Park [93, 103] for CCS-like process calculi to describe the operational behaviour of processes. Informally speaking, two process are bisimilar if any action by one of them can be mimicked (or matched) by an equal action from the other in such a way that the derivatives are still bisimilar. As an example on how this technique has been applied consider [110]: here Pous uses some proof techniques based on weak bisimulation (a bisimulation where internal actions are not considered) for reasoning about distributed implementations of process algebras with mobility. Moreover it could happen that the concept needs to be adapted to different languages; this way several formalisations have been introduced. For example in [116] Sangiorgi investigates other forms of bisimulation e.g. *open bisimulation* that takes into account names instantiation in the π-calculus. Or in [114] Sangiorgi discusses some suitable definitions of bisimulation for the Higher Order π-calculus.

The second technique we mention is testing semantics. Testing semantics was originally proposed for the λ-calculus in [97]. In [40] the concept was adapted to the concurrent language CCS. In testing semantics, the leading intuition is that of

experiment: two programs are equivalent if they pass the same tests, where a test is an observer, usually a distinguished process, and a way of observing it. This tool is useful for proving safety properties. For example it has been used in [1] for checking secrecy properties of cryptographic protocols.

1.4 Summary of the thesis

In what follows we describe the organisation of this dissertation: Chapter 2 quickly introduces some of the necessary terminology.

The core of the thesis is then divided into four chapters. Each chapter is devoted to a different technique. Since the languages analysed are different every language is introduced in the corresponding chapter.

In Chapter 3, we begin with the analysis of a synchronous language: we explore the expressiveness of a fragment of $\text{CCS}_!$, a variant of CCS where replication is considered instead of recursion. We study the existence of faithful encodings (i.e. encodings that respect the behaviour of the encoded model without introducing unnecessary computations) of models of computability strictly less expressive than Turing Machines. Namely, grammars of types 1,2 and 3 in the Chomsky Hierarchy. We provide faithful encodings of type 3 grammars (Regular Languages). We then show that it is impossible to provide a faithful encoding of type 2 grammars (Context Free Languages). We show that $\text{CCS}_!$ processes in our fragment can generate languages which are not type 2. We finally show that the languages generated by processes in the fragment considered are type 1 (Context Sensitive Languages). [2]

We then move to asynchronous languages and in Chapter 4 we study full abstraction for two Linda-like languages. Linda can be considered as the asynchronous version of CCS plus a shared memory (a multiset of elements) that is used for storing messages. The first variant provides primitives for adding and removing messages from a shared memory, local choice, parallel composition and recursion. The second one adds the possibility of checking for the absence of a message in the store. After

[2] Part of this work also appeared in [6].

having defined a trace-based denotational semantics, we obtain a fully abstract semantics for both languages by using suitable abstractions. These are used to identify different traces which do not correspond to different operational behaviours. The ability of the second variant considered of recognising multiple occurrences of messages in the store (which accounts for an increase of expressiveness) reflects in a less complex abstraction: i.e. different traces correspond to different behaviours and simpler saturations is needed for accomplish full abstraction.[3]

We thus study other languages where multiplicity plays a fundamental role. In particular, we focus on Multiset Rewriting Systems. In Chapter 5, we consider the language CHR (Constraint Handling Rules) which is a general purpose, committed-choice declarative language which, differently from other similar languages, uses multi-headed (guarded) rules. CHR inherits many features from logic programming languages. Hence, in this chapter, we apply techniques typical from process algebra to a language that has some concurrent characteristics but that has not been used for describing concurrent systems.

We prove that multiple heads augment the expressive power of the language. In fact, we first show that restricting to single head rules affects the Turing completeness of CHR, provided that the underlying constraint theory (where constructing elements of the language are defined) does not contain function symbols. Next we show that, also when considering generic constraint theories, under some rather reasonable assumptions it is not possible to encode CHR (with multi-headed rules) into a single-headed CHR language while preserving the semantics of programs. Finally, we show that also the number of atoms in the heads of rules matters, as the CHR language allowing at most n atoms in the heads is more expressive than the language allowing at most m atoms, for an m, $m < n$.[4]

Finally, in Chapter 6 we analyse a language similar but simpler than CHR. The κ-calculus is a formalism for molecular biology where molecules are terms with internal states and sites, bonds are represented by shared names labelling sites, and reactions

[3]Part of this work has been presented in [44].

[4]Part of this work is reported in [66].

are represented by rewriting rules. Depending on the shape of the rewriting rules, several dialects of the calculus can be obtained. We analyse the expressive power of some of these dialects by focusing on the thin boundary between decidability and undecidability for problems like reachability and coverability.[5]

Chapter 7 draws some conclusions and discuss some possible future developments.

1.5 Contributions

Most of the material presented in this dissertation has been previously reported in the following works appeared in Proceedings of International Conferences:

- J. Aranda, C. Di Giusto, M. Nielsen and F. Valencia. *CCS with Replication in the Chomsky Hierarchy: The Expressive Power of Divergence.* In APLAS '07, volume 4807 of LNCS, pages 383-398. Springer, 2007.

- C. Di Giusto, M. Gabbrielli, M. C. Meo. *Expressiveness of multiple heads in CHR.* In SOFSEM '09, volume 5404 of LNCS, pages 205–216. Springer, 2009.

- G. Delzanno, C. Di Giusto, M. Gabbrielli, C. Laneve, G. Zavattaro *On the Qualitative Analysis of Calculi for Formal Molecular Biology* Under submission.

- C. Di Giusto, M. Gabbrielli. *Full abstraction for Linda.* in ESOP '08, volume 4960 of LNCS, pages 78–92. Springer, 2008.

[5]This work has also appeared in [41].

Chapter 2

Background

If you do not know history, it is as if you were born yesterday. If you were born yesterday, then any leader can tell you anything.

Howard Zinn
Speech on "War and Social Justice", 2008

Here we introduce the necessary background for the following chapters.

2.1 RAM

In the following chapters, whenever we want to show the Turing completeness of a language we will encode RAMs (Random Access Machines) or Minsky machines into it. We recall here some basic notions on this Turing equivalent formalism. A RAM [96] $M(v_0, v_1)$ is a two-counter machine which consists of two registers R_1 and R_2 holding arbitrary large natural numbers and initialised with the values v_0 and v_1, and a program, i.e. a finite sequence of numbered instructions which modify the two registers. There are three types of instructions $j : Inst()$ where j is the number of the instruction:

- $j : Succ(R_i)$: adds 1 to the content of register R_i and goes to instruction $j+1$;
- $j : DecJump(R_i, l)$: if the content of the register R_i is not zero, then decreases it by 1 and goes to instruction $j+1$, otherwise jumps to instruction l;
- $j : Halt$: stops computation and returns the value in register R_1.

where $1 \leq i \leq 2$, $j, l \leq n$ and n is the maximum number of instructions of the program.

An internal state of the machine is given by a tuple (p_i, r_1, r_2) where the program counter p_i indicates the next instruction and r_1, r_2 are the current contents of the two registers. Given a program, its computation proceeds by executing the instructions as indicated by the program counter. The execution stops when the program counter reaches the $Halt$ instruction. In some encodings we will not use the $Halt$ instruction to signal termination instead we allow jumps to instruction numbers greater than the maximum number of instructions of the program. In Chapter 3 we generalise Minsky machines to RAMs with n registers.

2.2 Fixpoint theory

The second chapter makes use of some concepts from the fixpoint theory. In the following we introduce the basic needed notions (a more complete introduction can be find in [49] or in [56]):

The theorems we will need are build on the notions of poset, upper and lower bound:

Definition 2.1 (Poset) *A* partially ordered set *(poset)* $(S, \leq)$ *is a set equipped with a binary relation* $\leq$ *which is reflexive, anti-symmetric and transitive.*

Definition 2.2 (Upper and Lower bound) *Given a poset* $(S, \leq)$ *an element (or a point)* $x \in S$ *is an* upper bound *of a subset* X *of* S *if* $\forall y \in X \; y \leq x$. x *is the* least upper bound *(or join) of* X*, denoted by* $lub_S X$*, if* x *is an upper bound of* X *and if* $\forall y \in X \; y \leq z$ *with* $z \in S$*, then* $x \leq z$.

Dually $x \in S$ *is a* lower bound *of a subset* X *of* S *if* $\forall y \in X\ x \leq y$. x *is the* greatest lower bound *(or meet) of* X*, denoted by* $glb_S X$*, if* x *is a lower bound of* X *and if* $\forall y \in X\ z \leq y$ *with* $z \in S$*, then* $z \leq x$.

If we specialise the notion of poset we obtain complete partial orders (cpo) and lattices:

Definition 2.3 (Complete partial order) *A* directed *set* D *is a set where every finite subset* E *of* D *has an upper bound in* E.

A complete partial order *is a poset* $(S, \leq)$ *such that for every directed set* D *of* S *there exist a* $lub_S D$.

Definition 2.4 (Complete lattice) *A* complete lattice *is a poset* $(S, \leq)$ *such that for every subset* X *of* S *there exist a* $lub_S X$ *and a* $glb_S X$*. In particular* $\bot = lub_S \emptyset = glb_S S$ *and* $\top = glb_S \emptyset = lub_S S$

On complete lattices we can define fixpoints:

Definition 2.5 (Fixpoint) *Given a poset* $(S, \leq)$ *and a function* $f : S \to S$*,* $x \in S$ *is a* fixpoint *of* f *iff* $x = f(x)$. x *is a* prefixpoint *iff* $x \leq f(x)$ *and* x *is a* postfixpoint *iff* $f(x) \leq x$

Finally we give the definition of monotonicity and continuity for functions over posets:

Definition 2.6 (Monotone function) *Let* $(S, \leq)$ *and* $(T, \preceq)$ *be posets. A function* $f : S \to T$ *is monotone iff:*

$$\forall x, y \in S \text{ such that } x \leq y \text{ then } f(x) \preceq f(y)$$

Definition 2.7 (Continuous function) *Let* $(S, \leq)$ *and* $(T, \preceq)$ *be posets. A function* $f : S \to T$ *is continuous iff:*

$$\forall D \subseteq S \text{ directed, } f(lub_S D) = lub_T\{f(d) \mid d \in D\}$$

We now give two results which characterise fixpoints w.r.t. some characteristics of the function f.

Theorem 2.1 (Knaster-Tarski) *A monotonic function f on a complete lattice $(L, \leq)$ has a least fixpoint and a greatest fixpoint. Moreover the least fixpoint is the meet of all its prefixpoints and the greatest fixpoint is the join of all its postfixpoints.*

The ordinal powers of a monotonic function $f : C \to C$ on a complete partial order $(C, \leq)$ are defined as

$$f \uparrow^n = \begin{cases} \bot_C & n = 0 \\ f(f \uparrow^{n-1}) & \text{otherwise} \end{cases}$$

This second result is usually attributed to Kleene.

Theorem 2.2 (Kleene) *If f is a continuous function on a complete partial order $(C, \leq)$ and $x_0 \in C$ is a prefixpoint of f then $lub\{f^n(x_0) \mid n \in \mathbb{N}\}$ is the least fixpoint of f greater than x_0. In particular $f \uparrow \omega$ is the least prefixpoint and the least fixpoint of f.*

2.3 Petri net

Place/Transition Petri nets (or P/T nets) are an interesting infinite state model for the representation and analysis of parallel processes. We recall here the basic notation, for a full description of this computational model see [111].

Definition 2.8 *A* P/T net *is a tuple $N = (S, T, F, m_0)$, where S and T are the finite sets of places and transitions, such that $S \cap T = \emptyset$, and F is the transition function associating to each transition two finite multisets of places called the* pre-set *and the* post-set *of the transition.*

A finite multiset over the set S of places is called a marking*, and m_0 is the initial marking. Given a marking m and a place p, the number of instances of p in m is denoted with $m(p)$.*

The marking m of a P/T net can be modified by means of transitions firing: a transition with pre-set t' and post-set t'' can fire if its pre-set is included in m, and upon transition firing the new marking of the net becomes $m \setminus t' \cup t''$ where $\setminus$ and $\cup$ are the difference and union operators for multisets, respectively.

We are interested in this formal model since several problems, such as reachability or coverability, are decidable (for a survey on this topic see [51]).

Definition 2.9 (Boundedness or coverability) *A P/T net is* bounded *if its set of reachable markings is finite.*

Definition 2.10 (Reachability) *The* reachability *problem for P/T nets consists of deciding if given a P/T net $N = (S, T, F, m_0)$ and a marking m of N, if m can be reached from m_0.*

Thus we have the following decidable properties:

Theorem 2.3 *Given a P/T net N, the following properties are decidable:*

- *Coverability*
- *Reachability*
- *Termination*

P/T nets can be extended in several ways, obtaining a hierarchy of models where several properties remain decidable, for a survey on this topic see [48].

Chapter 3

Below Turing Expressiveness - The Role of Non-Determinism

In re mathematica ars proponendi pluris facienda est quam solvendi.

Georg Cantor

In this chapter we study the existence of faithful encodings into $CCS_!$ of models of computability *strictly less* expressive than Turing Machines. Namely, grammars of Types 1 (Context Sensitive Languages), 2 (Context Free Languages) and 3 (Regular Languages) in the Chomsky Hierarchy. We provide faithful encodings of Type 3 grammars. We show that it is impossible to provide a faithful encoding of Type 2 grammars and that termination-preserving $CCS_!$ processes can generate languages which are not Type 2. We finally show that the languages generated by termination-preserving $CCS_!$ processes are Type 1 .

3.1 Introduction

As already mention in the Chapter 1, when speaking of concurrency, Milner's CCS [93], a calculus for the modelling and analysis of synchronous communication, is a standard representative of process calculi.

Infinite behaviour is ubiquitous in concurrent systems. Hence, it ought to be represented by process terms. In the context of CCS we can find at least two representations of them: *Recursive definitions* and *Replication*. Recursive process definitions take the form $A(y_1, \ldots, y_n)$ each assumed to have a unique, possibly recursive, *parametric process definition* $A(x_1, \ldots, x_n) \stackrel{\texttt{def}}{=} P$. The intuition is that $A(y_1, \ldots, y_n)$ behaves as P with each y_i replacing x_i. Replication takes the form $!P$ and it means $P \mid P \mid \cdots$; an unbounded number of copies of the process P in parallel. An interesting result is that in the π-calculus, itself a generalisation of CCS, parametric recursive definitions can be encoded using replication up to weak bisimilarity. This is rather surprising since the syntax of $!P$ and its description are so simple. In fact, in [19] it is stated that in CCS recursive expressions are more expressive than replication. More precisely, it is shown that it is impossible to provide a weak-bisimulation preserving encoding from CCS with recursion, into the CCS variant in which infinite behaviour is specified only with replication. From now on we shall use CCS to denote CCS with recursion and $\text{CCS}_!$ to the CCS variant with replication.

Now, a remarkable expressiveness result in [20] states that, in spite of its being less expressive than CCS in the sense mentioned above, $\text{CCS}_!$ is Turing powerful. This is done by encoding (Deterministic) Random Access Machines (RAM) in $\text{CCS}_!$. Nevertheless, the encoding is not *faithful* (or deterministic) in the sense that, unlike the encoding of RAMs in CCS, it may introduce computations which do not correspond to the expected behaviour of the modeled machine. Such computations are forced to be *infinite* and thus regarded as non-halting computations which are therefore ignored. Only the finite computations correspond to those of the encoded RAM.

A crucial observation from [20] is that to be able to force wrong computation to be infinite, the $\text{CCS}_!$ encoding of a given RAM can, during evolution, move from a state which may terminate (i.e. weakly terminating state) into one that cannot terminate (i.e., strongly non-terminating state). In other words, the encoding does not *preserve (weak) termination* during evolution. It is worth pointing that since

RAMs are deterministic machines, their faithful encoding in CCS given in [19] does preserve weak termination during evolution. A legitimate question is therefore: What can be encoded with termination-preserving $CCS_!$ processes?

We shall investigate the expressiveness of $CCS_!$ processes which indeed preserve (weak) termination during evolution. This way we disallow the technique used in [20] to unfaithfully encode RAMs.

A sequence of actions s (over a finite set of actions) performed by a process P specifies a sequence of interactions with P's environment. For example, $s = a^n.\bar{b}^n$ can be used to specify that if P is input n a's by environment then P can output n b's to the environment. We therefore find it natural to study the expressiveness of processes w.r.t. sequences (or patterns) of interactions (languages) they can describe. In particular we shall study the expressiveness of $CCS_!$ w.r.t. the existence of termination-preserving encodings of grammars of Types 1 (Context Sensitive grammars), 2 (Context Free grammars) and 3 (Regular grammars) in the Chomsky Hierarchy whose expressiveness corresponds to (non-deterministic) Linear-bounded, Pushdown and Finite-State Automata, respectively. As elaborated later in the related work, similar characterisations are stated in the Caucal hierarchy of transition systems for other process algebras [18].

It is worth noticing that by using the non termination-preserving encoding of RAM's in [19] we can encode Type 0 grammars (which correspond to Turing Machines) in $CCS_!$.

Now, in principle the mere fact that a computation model fails to generate some particular language may not give us a definite answer about its computation power. For a trivial example, consider a model similar to Turing Machines except that the machines always print the symbol a on the first cell of the output tape. The model is essentially Turing powerful but fails to generate b. Nevertheless, our restriction to termination-preserving processes is a natural one, much like restricting non-deterministic models to deterministic ones, meant to rule out unfaithful encodings of the kind used in [20]. As matter of fact, Type 0 grammars can be encoded by using the termination-preserving encoding of RAMs in CCS [19].

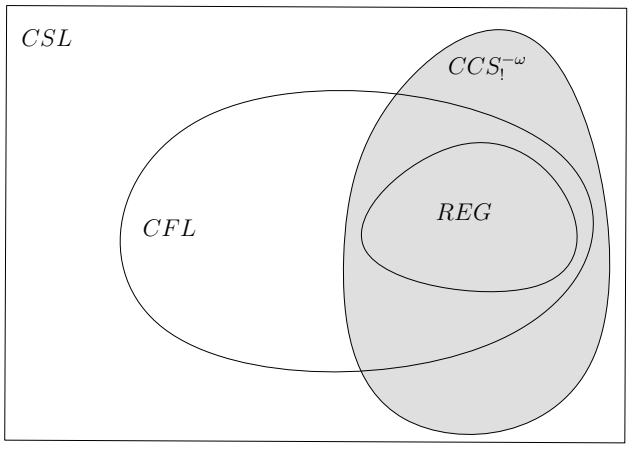

Figure 3.1: Termination-Preserving $\text{CCS}_!$ Processes ($\text{CCS}_!^{-\omega}$) in the Chomsky Hierarchy.

For simplicity, let us use $\text{CCS}_!^{-\omega}$ to denote the set of $\text{CCS}_!$ processes which preserve weak termination during evolution as described above. We first provide a language preserving encoding of Regular grammars into $\text{CCS}_!^{-\omega}$. We also prove that $\text{CCS}_!^{-\omega}$ processes can generate languages which cannot be generated by any Regular grammar. Our main contribution is to show that it is *impossible* to provide language preserving encodings from Context-Free grammars into $\text{CCS}_!^{-\omega}$. Conversely, we also show that $\text{CCS}_!^{-\omega}$ can generate languages which cannot be generated by any Context-free grammar. We conclude our classification by stating that all languages generated by $\text{CCS}_!^{-\omega}$ processes are context sensitive. The results are summarised in Fig. 3.1.

This chapter is organised as follows. Section 3.2 introduces the CCS calculi under consideration. We then discuss in Sections 3.3, 3.4, 3.5 how unfaithful encodings are used in [20] to provide an encoding of RAM's. We prove the above-mentioned results in Section 3.6. Finally, some concluding remarks are given in Section 3.7. [1]

[1] A preliminary version of this work appeared in [6].

3.2 The Calculi

In what follows we shall briefly recall the CCS constructs and its semantics as well as the CCS$_!$ calculus.

Finite CCS. In CCS, processes can perform actions or synchronise on them. These actions can be either offering port *names* for communication, or the so-called *silent* action τ. We presuppose a countable set $\mathcal{N}$ of port *names*, ranged over by $a, b, x, y \ldots$ and their primed versions. We then introduce a set of *co-names* $\overline{\mathcal{N}} = \{\overline{a} \mid a \in \mathcal{N}\}$ disjoint from $\mathcal{N}$. The set of *labels*, ranged over by l and l', is $\mathcal{L} = \mathcal{N} \cup \overline{\mathcal{N}}$. The set of *actions* Act, ranged over by α and β, extends $\mathcal{L}$ with a new symbol τ. Actions a and $\overline{a}$ are thought of as *complementary*, so we decree that $\overline{\overline{a}} = a$. We also decree that $\overline{\tau} = \tau$.

The processes specifying finite behaviour are given by:

$$P, Q \ldots := \mathbf{0} \mid \alpha.P \mid (\nu a)P \mid P \mid Q \tag{3.1}$$

Intuitively $\mathbf{0}$ represents the process that does nothing. The process $\alpha.P$ performs an action α then behaves as P. The restriction $(\nu a)P$ behaves as P except that it can offer neither a nor $\bar{a}$ to its environment. The names a and $\bar{a}$ in P are said to be *bound* in $(\nu a)P$. The *bound names* of P, $bn(P)$, are those with a bound occurrence in P, and the *free names* of P, $fn(P)$, are those with a not bound occurrence in P. The set of names of P, $n(P)$, is then given by $fn(P) \cup bn(P)$. Finally, $P \mid Q$ represents parallelism; either P or Q may perform an action, or they can also synchronise when performing complementary actions.

Notation 3.1 *We shall write the summation $P+Q$ as an abbreviation of the process $(\nu\ u)\ (\overline{u} \mid u.P \mid u.Q)$. We also use $(\nu a_1 \ldots a_n)P$ as a short hand for $(\nu a_1) \ldots (\nu a_n)P$. We often omit the "$\mathbf{0}$" in $\alpha.\mathbf{0}$.*

The above description is made precise by the operational semantics in Table 3.1. A transition $P \xrightarrow{\alpha} Q$ says that P can perform α and evolve into Q.

In the literature there are at least two alternatives to extend the above syntax to express infinite behaviour. We describe them next.

$$\text{ACT}\ \frac{}{\alpha.P \xrightarrow{\alpha} P} \qquad \text{RES}\ \frac{P \xrightarrow{\alpha} P'}{(\nu\ a)\ P \xrightarrow{\alpha} (\nu\ a)\ P'}\ \text{if}\ \alpha \notin \{a, \overline{a}\}$$

$$\text{PAR}_1\ \frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q} \qquad \text{PAR}_2\ \frac{Q \xrightarrow{\alpha} Q'}{P \mid Q \xrightarrow{\alpha} P \mid Q'}$$

$$\text{COM}\ \frac{P \xrightarrow{l} P' \quad Q \xrightarrow{\bar{l}} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}$$

Table 3.1: An operational semantics for finite processes.

3.2.1 Parametric Definitions: CCS and CCS_p

A typical way of specifying infinite behaviour is by using parametric definitions [94]. In this case we extend the syntax of finite processes (Equation 3.1) as follows:

$$P, Q, \ldots := \ldots \mid A(y_1, \ldots, y_n) \tag{3.2}$$

Here $A(y_1, \ldots, y_n)$ is an *identifier* (also *call*, or *invocation*) of arity n. We assume that every such an identifier has a unique, possibly recursive, *definition*

$$A(x_1, \ldots, x_n) \stackrel{\texttt{def}}{=} P_A$$

where the x_i's are pairwise distinct, and the intuition is that $A(y_1, \ldots, y_n)$ behaves as its *body* P_A with each y_i replacing the *formal parameter* x_i. For each $A(x_1, \ldots, x_n) \stackrel{\texttt{def}}{=} P_A$, we require $fn(P_A) \subseteq \{x_1, \ldots, x_n\}$.

Following [65], we should use CCS_p to denote the calculus with parametric definitions with the above syntactic restrictions.

Remark 3.2 *As shown in [65], however, CCS_p is equivalent w.r.t. strong bisimilarity to the standard CCS. We shall then take the liberty of using the terms CCS and CCS_p to denote the calculus with parametric definitions as done in [94].*

The rules for CCS_p are those in Table 3.1 plus the rule:

$$\text{CALL}\ \frac{P_A[y_1, \ldots, y_n/x_1, \ldots, x_n] \xrightarrow{\alpha} P'}{A(y_1, \ldots, y_n) \xrightarrow{\alpha} P'}\ \text{if}\ A(x_1, \ldots, x_n) \stackrel{\texttt{def}}{=} P_A \tag{3.3}$$

As usual $P[y_1 \ldots y_n/x_1 \ldots x_n]$ results from replacing every free occurrence of x_i with y_i renaming bound names in P wherever needed to avoid capture.

3.2.2 Replication: CCS$_!$

One simple way of expressing infinite is by using replication. Although, mostly found in calculus for mobility such as the π-calculus and mobile ambients, it is also studied in the context of CCS in [19, 65].

For replication the syntax of finite processes (Equation 3.1) is extended as follows:

$$P, Q, \ldots := \ldots \mid\ !P \tag{3.4}$$

Intuitively the process $!P$ behaves as $P \mid P \mid \ldots \mid P \mid !P$; unboundedly many P's in parallel. We call CCS$_!$ the calculus that results from the above syntax The operational rules for CCS$_!$ are those in Table 3.1 plus the following rule:

$$\text{REP}\ \frac{P \mid !P \xrightarrow{\alpha} P'}{!P \xrightarrow{\alpha} P'} \tag{3.5}$$

3.3 The Role of Strong Non-Termination

In this section we shall single out the fundamental non-deterministic strategy for the Turing-expressiveness of CCS$_!$. First we need a little notation.

Notation 3.3 *Define $\stackrel{s}{\Longrightarrow}$, with $s = \alpha_1 \ldots \alpha_n \in \mathcal{L}^*$, as*

$$(\xrightarrow{\tau})^* \xrightarrow{\alpha_1} (\xrightarrow{\tau})^* \ldots (\xrightarrow{\tau})^* \xrightarrow{\alpha_n} (\xrightarrow{\tau})^*.$$

For the empty sequence $s = \epsilon$, $\stackrel{s}{\Longrightarrow}$ is defined as $(\xrightarrow{\tau})^*$.

We shall say that a process generates a sequence of non-silent actions s if it can perform the actions of s in a finite maximal sequence of transitions. More precisely:

Definition 3.1 (Sequence and language generation) *The process P* generates *a sequence $s \in \mathcal{L}^*$ if and only if there exists Q such that $P \stackrel{s}{\Longrightarrow} Q$ and $Q \not\xrightarrow{\alpha}$ for any $\alpha \in Act$. We define the language of (or generated by) a process P, $L(P)$, as the set of all sequences P generates.*

The above definition basically states that a sequence is generated when no reduction rules can be applied. It is inspired by language generation of the model of computations we are comparing our processes with. Namely, formal grammars where a sequence is generated when no rewriting rules can be applied.

As we shall see below (strong) non-termination plays a fundamental role in the expressiveness of CCS$_!$. We borrow the following terminology from rewriting systems:

Definition 3.2 (Termination) *We say that a process P is* (weakly) terminating *(or that it can terminate) if and only if there exists a sequence s such that P generates s. We say that P is* (strongly) non-terminating*, or that it* cannot terminate *if and only if P cannot generate any sequence.*

Busi et al. in [20] show the Turing-expressiveness of CCS$_!$, by providing a CCS$_!$ encoding $[\![\cdot]\!]$ of RAMs [96]. The encoding is said to be *unfaithful* (or non-deterministic) in the following sense: Given M, during evolution $[\![M]\!]$ may make a transition, by performing a τ action, from a weakly terminating state (process) into a state which do not correspond to any configuration of M. Nevertheless such states are strongly non-terminating processes. Therefore, they may be thought of as being configurations which cannot lead to a halting configuration. Consequently, the encoding $[\![M]\!]$ does not *preserve (weak) termination* during evolution.

Remark 3.4 *The work [20] considers also guarded-summation for CCS$_!$. The results about the encodability of RAM's our work builds on can straightforwardly be adapted to our guarded-summation free CCS$_!$ fragment. (See Section 3.4)*

Now rather than giving the full encoding of RAMs in CCS$_!$, let us use a much simpler example which uses the same technique in [20]. Below we encode a typical context sensitive language in CCS$_!$.

Example 3.1 Consider the following processes:

$$\begin{aligned} P &= (\nu\ k_1, k_2, k_3, u_b, u_c)\ (\ \overline{k_1} \mid \overline{k_2} \mid Q_a \mid Q_b \mid Q_c) \\ Q_a &= !k_1.a.(\overline{k_1} \mid \overline{k_3} \mid \overline{u_b} \mid \overline{u_c}) \\ Q_b &= k_1.!k_3.k_2.u_b.b.\overline{k_2} \\ Q_c &= k_2.(!u_c.c \mid u_b.DIV) \end{aligned}$$

where $DIV = !\tau$. It can be verified that $L(P) = \{a^n b^n c^n\}$. Intuitively, in the process P above, Q_a performs (a sequence of actions) a^n for an arbitrary number n (and also produces n u_b's). Then Q_b performs b^m for an arbitrary number $m \leq n$ and each time it produces b it consumes a u_b. Finally, Q_c performs c^n and diverges if $m < n$ by checking if there are u_b's that were not consumed. □

The Power of Non-Termination. Let us underline the role of strong non-termination in Example 3.1. Consider a run

$$P \stackrel{a^n b^m}{\Longrightarrow} \ldots$$

Observe that the name u_b is used in Q_c to test if $m < n$, by checking whether some u_b were left after generating b^m. If $m < n$, the non-terminating process DIV is triggered and the extended run takes the form

$$P \stackrel{a^n b^m c^n}{\Longrightarrow} \stackrel{\tau}{\longrightarrow} \stackrel{\tau}{\longrightarrow} \ldots$$

Hence the sequence $a^n b^m c^n$ arising from this run (with $m < n$) is therefore not included in $L(P)$.

The tau move. It is crucial to observe that there is a τ transition arising from the moment in which $\overline{k_2}$ chooses to synchronise with Q_c to start performing the c actions. One can verify that if $m < n$ then the process just before that τ transition is weakly terminating while the one just after is strongly non-terminating.

Formally the class of termination-preserving processes is defined as follows.

Definition 3.3 (Termination Preservation) *A process P is said to be weakly termination-preserving if and only if whenever $P \stackrel{s}{\Longrightarrow} Q \stackrel{\tau}{\longrightarrow} R$:*

- *if Q is weakly terminating then R is weakly terminating.*

We use $CCS_!^{-\omega}$ to denote the set of $CCS_!$ processes that are termination-preserving.

One may wonder why only τ actions are not allowed in Definition 3.3 when moving from a weakly terminating state into a strongly non-terminating one. The next proposition answers to this.

Proposition 3.1 *For every $P, P', \alpha \neq \tau$ if $P \xrightarrow{\alpha} P'$ and P is weakly terminating then P' must be weakly terminating.*

Proof: As a mean of contradiction let P' be a strongly non-terminating process such that $P \xrightarrow{\alpha} P'$ where $\alpha \neq \tau$. Let γ be an arbitrary maximal sequence of transitions from P. Since $P \xrightarrow{\alpha} P'$, the action α will be performed in γ as a visible action or in a synchronisation with its complementary action $\bar{\alpha}$. In the synchronisation case, one can verify that there exists another maximal sequence γ' identical to γ except that in γ', α and $\bar{\alpha}$ appear as visible actions instead of their corresponding synchronisation. Therefore, there exists a sequence $P \stackrel{t_1}{\Longrightarrow} Q \xrightarrow{\alpha} R \stackrel{t_2}{\Longrightarrow} \not\rightarrow$ (Fig. 3.2). From $P \stackrel{t_1}{\Longrightarrow} Q \xrightarrow{\alpha} R$ and $P \xrightarrow{\alpha} P'$, we can show that $P \xrightarrow{\alpha} P' \stackrel{t_1}{\Longrightarrow} R \stackrel{t_2}{\Longrightarrow} \not\rightarrow$ (Fig. 3.3) thus contradicting the assumption that P' is a strongly non-terminating process. □

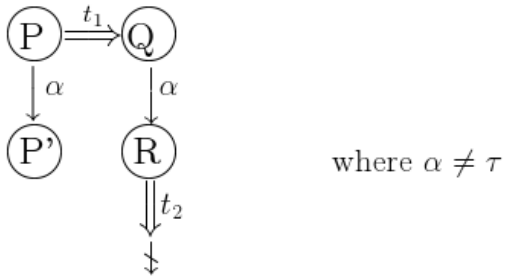

Figure 3.2: Alternative evolutions of P involving α

We conclude this section with a proposition which relates preservation of termination and the language of a process.

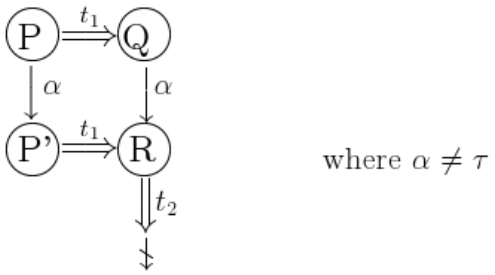

Figure 3.3: Confluence from P to R

Proposition 3.2 *Suppose that P is terminating-preserving and that $L(P) \neq \emptyset$. For every Q, if $P \stackrel{s}{\Longrightarrow} Q$ then $\exists s'$ such that $s.s' \in L(P)$.*

Proof: Let Q an arbitrary process such that $P \stackrel{s}{\Longrightarrow} Q$. Since $L(P) \neq \emptyset$ then P is weakly terminating. From Definition 3.3 and Proposition 3.1 it follows that Q is weakly terminating. Hence there exists a sequence s' such that $P \stackrel{s}{\Longrightarrow} Q \stackrel{s'}{\Longrightarrow} R \not\rightarrow$ and thus from Definition 3.1 we have $s.s' \in L(P)$ as wanted. □

3.4 CCS$_!$ without choice

In this section we show that the encoding proposed by Busi, Gabbrielli and Zavattaro in [20] of RAMs (Minsky machines) into CCS$_!$ with guarded summation can be adapted to the summation free fragment. We extend the syntax of CCS$_!$ (Equation 3.4) by considering the operator +:

$$P, Q, \cdots := \ldots \mid P + Q \tag{3.6}$$

with the following operational rule:

$$\text{SUM } \frac{P \stackrel{\alpha}{\longrightarrow} P'}{P + Q \stackrel{\alpha}{\longrightarrow} P'} \tag{3.7}$$

Hence the encoding of the instructions and of a register r_j storing the value c_j

is:

$$\begin{aligned}
[\![(i : Succ(r_j)]\!] := \quad & !p_i.(\overline{inc_j} \mid inc.\overline{p_{i+1}}) \\
[\![(i : DecJump(r_j, s)]\!] := \quad & !p_i.(dec_j \mid (dec.\overline{p_{i+1}} + zero.\overline{ps})) \\
[\![(r_j : c_j)]\!] := \quad & \overline{nr_j} \mid \\
& !nr_j.(\nu\ m, i, d, u)\ (outm \mid !m.(inc_j.\bar{i} + dec_j.\bar{d}) \mid \\
& !i.(\overline{m} \mid \overline{inc} \mid \overline{u} \mid d.u.(\overline{m} \mid \overline{dec})) \mid \\
& d.(\overline{zero} \mid u.DIV \mid \overline{nr_j}) \mid \\
& \textstyle\prod_{c_j}(\overline{u} \mid d.u.(\overline{m} \mid \overline{dec})))
\end{aligned}$$

where DIV is a process able to activate an infinite observable computation, for instance $\overline{w'} \mid !w'.\overline{w'}$.

Along the computation, some "garbage process" can appear:

$$G_j : (\nu\ m, i, d, u)\ (!m.(inc_j.\bar{i} + dec_j.\bar{d}) \mid !i.(\overline{m} \mid \overline{inc} \mid \overline{u} \mid d.u.\overline{dec}) \mid u.DIV)$$

Definition 3.4 *Let R be a RAM with program instructions $(1 : I_1), \ldots, (m : I_m)$ and registers $r_1, \ldots, r_n$. Given the configuration $(i, c_1, \ldots, c_n)$ of R, we define*

$$\begin{aligned}
[\![(i, c_1, \ldots, c_n)]\!]_R = & (\nu\ p_1 \ldots p_m, nr_1, inc_1, dec_1 \ldots nr_n, inc_n, dec_n, inc, dec, zero) \\
& (\overline{p_1} \mid [\![(1 : I_1)]\!] \mid \ldots \mid [\![(m : I_m)]\!] \mid \prod_{i \in TI} p_i.\overline{w} \mid \\
& [\![r_1 = c_1]\!] \mid \ldots \mid [\![r_n = c_n]\!] \prod_{k_1} G_1 \mid \ldots \mid \prod_{k_n} G_n)
\end{aligned}$$

where the modelling of program instructions $[\![(i : I_i)]\!]$, the modelling of registers $[\![r_j = c_j]\!]$, the set of terminating indexes TI, and the garbage $G_1, \ldots, G_n$ have been defined above, and $k_1 \ldots k_n$ are natural numbers.

The following theorem, whose proof is in [21] states the correctness of the encoding.

Theorem 3.5 *Let R be a RAM with program $(1 : I_1), \ldots, (m : I_m)$ and state $(i, c_1, \ldots, c_n)$ and let the process P be in $[\![(i, c_1, \ldots, c_n]\!]_R$. Then $(i, c_1, \ldots, c_n)$ terminates if and only if P converges. Moreover P converges if and only if $P \approx \tau.P + \overline{w}$.*

This proves that convergence and weak bisimulation are undecidable in $\text{CCS}_!$.

If we consider the $\text{CCS}_!$ fragment without choice it is still possible to adapt the previous encoding to our language:

$$
\begin{array}{ll}
[\![(i : Succ(r_j)]\!]_m := & [\![(i : Succ(r_j)]\!] \\
[\![(i : DecJump(r_j, s)]\!]_m := & !p_i.(dec_j \mid (\langle\!\langle dec.\overline{p_{i+1}} + zero.\overline{ps}\rangle\!\rangle)) \\
\langle\!\langle dec.\overline{p_{i+1}} + zero.\overline{ps}\rangle\!\rangle := & (\nu\ lv_d, lv_z)\ (dec.(\overline{p_{i+1}} \mid \overline{lv_z} \mid lv_d.DIV) \mid \\
 & zero.(\overline{ps} \mid \overline{lv_d} \mid lv_z.DIV)) \\
[\![(r_j : c_j)]\!]_m := & \overline{nr_j} \mid \\
 & !nr_j.(\nu\ m, i, d, u)\ (\overline{m} \mid !m.\langle\!\langle (inc_j.\bar{i} + dec_j.\bar{d})\rangle\!\rangle \mid \\
 & !i.(\overline{m} \mid \overline{inc} \mid \overline{u} \mid d.u.(\overline{m} \mid \overline{dec})) \mid \\
 & d.(\overline{zero} \mid u.DIV \mid \overline{nr_j}) \mid \\
 & \textstyle\prod_{c_j}(\overline{u} \mid d.u.(\overline{m} \mid \overline{dec}))) \\
\langle\!\langle inc_j.\bar{i} + dec_j.\bar{d}\rangle\!\rangle := & (\nu\ lv_d, lv_i)\ (inc_j.(\bar{i} \mid \overline{lv_d} \mid lv_i.DIV) \mid \\
 & dec_j.(\bar{d} \mid \overline{lv_i} \mid lv_d.DIV))
\end{array}
$$

Where DIV is a process able to activate an infinite observable computation, for instance $\overline{w'} \mid !w'.\overline{w'}$.

The translation of choice in both $\langle\!\langle dec.\overline{p_{i+1}} + zero.\overline{ps}\rangle\!\rangle$ and $\langle\!\langle inc_j.\bar{i} + dec_j.\bar{d}\rangle\!\rangle$ introduces more computations which do not follow the expected behaviour of the modeled RAM . However these computations are also infinite. Intuitively, once the choice has been done, e.g. *inc* (*zero*) or *dec* can still participate in the computation as they are in parallel, in this case the local variables lv_d, $lv_i(lv_z)$ trigger divergence, ensuring that the computation cannot terminate.

So more formally we can define a RAM machine in the following way:

Definition 3.5 *Let R be a RAM with program instructions $(1 : I_1), \ldots, (m : I_m)$ and registers $r_1, \ldots, r_n$. Given the configuration $(i, 0, \ldots, 0)$ of R, we define the following encoding in $\text{CCS}_!$ without choice:*

$$
\begin{array}{rl}
[\![(i, c_1 \ldots c_n)]\!]_{R_m} = & (\nu\ p_1 \ldots p_m, nr_1, inc_1, dec_1 \ldots nr_n, inc_n, dec_n, inc, dec, zero) \\
 & (\overline{p_1} \mid [\![(1 : I_1)]\!]_m \mid \ldots \mid [\![(m : I_m)]\!]_m \mid \prod_{i \in TI} p_i.\overline{w} \mid \\
 & [\![r_1 = 0]\!]_m \mid \ldots \mid [\![r_n = 0]\!]_m
\end{array}
$$

where the modelling of program instructions $[\![(i : I_i)]\!]_m$*, the modelling of registers* $[\![r_j = c_j]\!]_m$*, the set of terminating indexes TI have been defined above.*

As the computations move forward garbage processes $G_1, \ldots, G_n$ and the residual terms from $\langle\!\langle dec.\overline{p_{i+1}} + zero.\overline{ps}\rangle\!\rangle$ and $\langle\!\langle inc_j.\bar{i} + dec_j.\bar{d}\rangle\!\rangle$ can appear. For the sake of simplicity, we omit the garbage processes and residual terms describe above in the configuration, but they could be included similarly as in Definition 3.4. So similarly as before (the proof is a straightaforward adaptation of the one in [21])the following theorem states the correctness of the encoding $[\![\cdot]\!]_{R_m}$.

Theorem 3.6 *Let* R *be a RAM with program* $(1 : I_1), \ldots, (m : I_m)$ *and registers* $r_1, \ldots, r_n$*. Given the initial configuration* $(1, 0, \ldots, 0)$ *of* R *and let the process* P *be in* $[\![(i, 0, \ldots, 0)]\!]_{R_m}$*. Then* $(i, 0, \ldots, 0)$ *terminates if and only if* P *converges. Moreover* P *converges if and only if* $P \approx \tau.P + \overline{w}$*.*

This proves that convergence and weak bisimulation are undecidable in $\text{CCS}_!$ without choice. From now on the term $\text{CCS}_!$ will refer to the $\text{CCS}_!$ variant without choice.

3.5 Undecidability results for $\text{CCS}_!^{-\omega}$

To prove the undecidability of whether a given $\text{CCS}_!$ process preserves termination, we reduce to the halting problem of RAMs.

Lemma 3.1 *Let* P *be a* $CCS_!$ *process, if* $L(P) = \emptyset$ *then* P *is termination-preserving.*

Proof: Let first observe that by definition $L(P) = \emptyset$ iff P is not weakly terminating. As a mean of contradiction, let $L(P) = \emptyset$ and P be non-terminating preserving, since P is not terminating-preserving then $P \stackrel{s}{\Longrightarrow} Q \stackrel{\tau}{\longrightarrow} R$ such that Q is weakly terminating and R is non-weakly terminating. Therefore Q recognises at least one sequence and consequently also P. As P recognises at least one sequence, $L(P) \neq \emptyset$, a contradiction. □

Notice that the notion of weakly termination and convergence are not the same, as the first one takes into account the visible actions whereas the second one not. For example $!a$ is not weakly terminating but it is convergent, and $(\nu\ a)\ (\overline{a}\ |!a.\overline{a}\ |\ b.a)$ is not convergent but it is weakly terminating, in fact $L((\nu\ a)\ (\overline{a}\ |!a.\overline{a}\ |\ b.a)) = \{b\}$. However when considering RAMs the two notion are equivalent, indeed we can prove the following:

Lemma 3.2 *$[\![M]\!]_{R_m}$ is weakly terminating iff $[\![M]\!]_{R_m}$ is convergent.*

Proof: If $[\![M]\!]_{R_m}$ is convergent, then there exists a correct terminating run of the encoding, made of τ actions which at the end it executes $\overline{w}$, after which there is no further observable actions. If there would be a further observable action it would be w' and it would generate an infinite observable computation. Therefore if $[\![M]\!]_{R_m}$ is convergent then $L([\![M]\!]_{R_m}) = \{w\}$ hence $[\![M]\!]_{R_m}$ is weakly terminating.

Conversely if $[\![M]\!]_{R_m}$ is weakly terminating, there is at least a sequence generated from $[\![M]\!]_{R_m}$, since there are only two observable names in $[\![M]\!]_{R_m}$: $\{w, w'\}$, this sequence can be:

1. A sequence with just τ actions, i.e. the string is ϵ: In this case $[\![M]\!]_{R_m}$ would be convergent but $[\![M]\!]_{R_m}$ would exhibit w as well. Thus a sequence with only τ actions cannot exist.

2. A sequence with w': if w' appears in the sequence then the reached process is of the form $Q\ |!w'.\overline{w'}$, therefore the process would be non-weakly terminating. Hence a sequence with w' cannot exist.

3. A sequence with w: From item 1 and 2, the sequence can be made of only $w's$. But there is only a w in the sequence as the computation only allows to access one register. Once the instruction associated to w has been executed ($p_i.w$, where i is the register index), there is no further activation of registers.

Therefore if $[\![M]\!]_{R_m}$ is weakly terminating then $L([\![M]\!]_{R_m}) = \{w\}$, therefore there is a finite sequence of τ actions from $[\![M]\!]_{R_m}$ ended at w, otherwise an infinite

sequence of τ actions would mean an infinite execution of the RAM M associated. Thus it would be impossible to reach w along that computation. So $[\![M]\!]_{R_m}$ is convergent.

Hence we conclude that $[\![M]\!]_{R_m}$ is weakly terminating iff $[\![M]\!]_{R_m}$ is convergent. □

The previous lemma is necessary to prove the following undecidability result:

Theorem 3.7 *The property of determine whether a CCS$_!$ process P is terminating-preserving is undecidable.*

Proof: Let us consider the encoding $[\![\cdot]\!]_{R_m}$ from RAM into CCS$_!$[2], it suffices to prove that a RAM M halts iff $(\nu\ a)\ (\overline{a} \mid a.[\![M]\!]_{R_m} \mid a.DIV)$ is not terminating preserving.

First, we prove that $(\nu\ a)\ (\overline{a} \mid a.[\![M]\!]_{R_m} \mid a.DIV)$ is not terminating preserving iff $[\![M]\!]_{R_m}$ is weakly terminating.

If $[\![M]\!]_{R_m}$ is non-weakly terminating (hence $L([\![M]\!]_{R_m}) = \emptyset$) then $(\nu\ a)\ (\overline{a} \mid a.[\![M]\!]_{R_m} \mid a.DIV)$ is non-weakly terminating ($L((\nu\ a)\ (\overline{a} \mid a.[\![M]\!]_{R_m} \mid a.DIV)) = \emptyset$) and therefore by lemma 3.1 $(\nu\ a)\ (\overline{a} \mid a.[\![M]\!]_{R_m} \mid a.DIV)$ is terminating-preserving.

On the other hand, if $[\![M]\!]_{R_m}$ is weakly terminating, then $(\nu\ a)\ (\overline{a} \mid a.[\![M]\!]_{R_m} \mid a.DIV)$ is weakly terminating but $(\nu\ a)\ (\overline{a} \mid a.[\![M]\!]_{R_m} \mid a.DIV) \stackrel{\tau}{\Longrightarrow} (\nu\ a)\ \ (\overline{a} \mid a.[\![M]\!]_{R_m} \mid a.DIV) \stackrel{\tau}{\longrightarrow} (\nu\ a)\ (a.[\![M]\!]_{R_m} \mid DIV)$ where $(\nu\ a)\ \ (a.[\![M]\!]_{R_m} \mid DIV)$ is non-weakly terminating therefore $(\nu\ a)\ (\overline{a} \mid a.[\![M]\!]_{R_m} \mid a.DIV)$ is not terminating preserving .

As $[\![M]\!]_{R_m}$ is weakly terminating iff $[\![M]\!]_{R_m}$ is convergent by lemma 3.2 and $(\nu\ a)\ (\overline{a} \mid a.[\![M]\!]_{R_m} \mid a.DIV)$ is not terminating preserving iff $[\![M]\!]_{R_m}$ is weakly terminating, therefore $[\![M]\!]_{R_m}$ is convergent iff $(\nu\ a)\ (\overline{a} \mid a.[\![M]\!]_{R_m} \mid a.DIV)$ is not terminating preserving. Finally as M halts iff $[\![M]\!]_{R_m}$ is convergent, we conclude that M halts iff $(\nu\ a)\ (\overline{a} \mid a.[\![M]\!]_{R_m} \mid a.DIV)$ is not terminating preserving. □

[2]Recall this is the CCS$_!$ variant without choice

3.6 $CCS_!$ and Chomsky Hierarchy

In this section we study the expressiveness of termination-preserving $CCS_!$ processes in the Chomsky hierarchy. Recall that, in a strictly decreasing expressive order, Types 0, 1, 2 and 3 in the Chomsky hierarchy correspond, respectively, to unrestricted-grammars (Turing Machines), Context Sensitive Grammars (Non-Deterministic Linear Bounded Automata), Context Free Grammars (Non-Deterministic PushDown Automata), and Regular Grammars (Finite State Automata).

We assume that the reader is familiar with the notions and notations of formal grammars. A grammar is a quadruple $G = (\Sigma, N, S, P)$ where Σ are the terminal symbols, N the non-terminals, S the initial symbol, P the set of production rules. The language of (or generated by) a formal grammar G, denoted as $L(G)$, is defined as all those strings in Σ^* that can be generated by starting with the start symbol S and then applying the production rules in P until no more non-terminal symbols are present.

3.6.1 Encoding Regular Languages

Regular Languages (REG) are those generated by grammars whose production rules can only be of the form $A \to a$ or $A \to a.B$. They can be alternatively characterised as those recognised by regular expressions which are given by the following syntax:

$$e = \emptyset \mid \epsilon \mid a \mid e_1 + e_2 \mid e_1.e_2 \mid e^*$$

where a is a terminal symbol.

Definition 3.6 *Given a regular expression e, we define $[\![e]\!]$ as the $CCS_!$ process $(\nu\ m)\ ([\![e]\!]_m \mid m)$ where $[\![e]\!]_m$, with $m \notin fn([\![e]\!])$, is inductively defined as in Figure 3.4.*

Remark 3.8 *The conditionals on language emptiness in Definition 3.6 are needed to make sure that the encoding of regular expressions always produce termination-preserving processes. To see this consider the case $a + \emptyset$. Notice that while $[\![a]\!] = a$*

$$
\begin{array}{ll}
[\![\emptyset]\!]_m = & DIV \\
[\![\epsilon]\!]_m = & \overline{m} \\
[\![a]\!]_m = & a.\overline{m} \\
[\![e_1 + e_2]\!]_m = & \begin{cases} [\![e_1]\!]_m & \text{if } L(e_2) = \emptyset \\ [\![e_2]\!]_m & \text{if } L(e_1) = \emptyset \\ [\![e_1]\!]_m + [\![e_2]\!]_m & \text{otherwise} \end{cases} \\
[\![e_1.e_2]\!]_m = & (\nu\ m_1)\ ([\![e_1]\!]_{m_1} \mid m_1.[\![e_2]\!]_m) \text{ with } m_1 \notin fn(e_1) \\
[\![e^*]\!]_m = & \begin{cases} \overline{m} & \text{if } L(e) = \emptyset \\ (\nu\ m')\ (\overline{m'} \mid !m'.[\![e]\!]_{m'} \mid m'.\overline{m}) \text{ with } m' \notin fn(e) & \text{otherwise} \end{cases}
\end{array}
$$

where $DIV = !\tau$.

Figure 3.4: Encoding of regular expressions

and $[\![\emptyset]\!] = DIV$ *are termination-preserving,* $a{+}DIV$ *is not. Hence* $[\![e_1{+}e_2]\!]$ *cannot be defined as* $[\![e_1]\!]{+}[\![e_2]\!]$. *Since the emptiness problem is decidable for regular expressions, it is clear that given* e, $[\![e]\!]$ *can be effectively constructed.*

The following proposition states the correctness of the encoding.

Proposition 3.3 *Let* $[\![e]\!]$ *be as in Definition 3.6. We have* $L(e) = L([\![e]\!])$ *and furthermore* $[\![e]\!]$ *is termination-preserving.*

Proof: The proof will proceed by induction on the structure of regular expressions. If $e = a$ or $e = \epsilon$ then the thesis follows straightforwardly.

Hence suppose $e = e_1 + e_2$ where both $L(e_1)$ and $L(e_2)$ are not-empty then we have that $L(e_1 + e_2) = L(e_1) \cup L(e_2)$, by inductive hypothesis we have that this is equivalent to $L([\![e_1]\!]) \cup L([\![e_2]\!])$ and both $[\![e_1]\!], [\![e_2]\!]$ are termination preserving. However it is easy to see that $L([\![e_1]\!]) \cup L([\![e_2]\!]) = L([\![e_1]\!] + [\![e_2]\!])$ and by definition 3.6 we can conclude that this is equal to $L([\![e_1 + e_2]\!])$. Moreover since the two processes $[\![e_1]\!], [\![e_2]\!]$ are termination preserving and since they are not sharing any channel

apart from the coordination channel m, the parallel composition of the two cannot introduce divergent computation, therefore $[\![e_1 + e_2]\!]$ is termination-preserving.

A similar proof can be given for $e = e_1.e_2$, indeed $L(e_1.e_2) = \{s.t \mid s \in L(e_1)$ and $t \in L(e_2)\}$ which by inductive hypothesis is equal to $\{s.t \mid s \in L([\![e_1]\!])$ and $t \in L([\![e_2]\!])\}$. From definition 3.6 it can be observed that this is the same as $L([\![e_1.e_2]\!])$. Similarly as before if both $L(e_1)$ and $L(e_2)$ are not-empty since we are not introducing divergent computation the process $[\![e_1.e_2]\!]$ is termination-preserving. Otherwise if one of the languages is empty the whole language reduces to the empty set and as proved in 3.1 the language is termination preserving.

Finally if $e = e_1^*$ (and $L(e_1)$ is not empty) then $L(e_1^*) = \bigcup_{i \geq 0} L(e_1^i)$ and

$$e_1^i = \begin{cases} \epsilon & \text{if } i = 0 \\ e_1.e_1^{i-1} & \text{otherwise} \end{cases}$$

By inductive hypothesis we have that $L(e_1) = L([\![e_1]\!])$. We will first prove that $L(e_1^*) \subseteq L([\![e_1^*]\!]$. Let $s \in L(e_1^*)$ by definition s is either ϵ or $s \in L(e_1^i) = L(\underbrace{e_1.\ldots.e_1}_{\text{i times}})$. In the first case from definition 3.6 (by synchronising at the first step $\overline{m'}$ with $m'.\overline{m}$) we have that $\epsilon \in L([\![e_1^*]\!])$. Otherwise taking the other synchronisation the process can generate i times the characters $L([\![e_1]\!])$ thus concluding that $s \in L([\![e_1^*]\!])$. Conversely if $s \in L([\![e_1^*]\!])$, s can either be ϵ or a concatenation of sequences generated by $[\![e_1]\!]$. The first case is obvious. In the second case by inductive hypothesis we have that $s \in L(\underbrace{[\![e_1]\!].\ldots.[\![e_1]\!]}_{\text{i times}})$ which we have already shown being equal to $L(\underbrace{e_1.\ldots.e_1}_{\text{i times}})$ and thus concluding $s \in L(e_1^*)$. Moreover the process is termination-preserving since divergence is never introduced. □

From the standard encoding from Type 3 grammars to regular expressions and the above proposition we obtain the following result.

Theorem 3.9 *For every Type 3 grammar G, we can construct a termination-preserving CCS$_!$ process P_G such that $L(G) = L(P_G)$.*

Proof: Follows immediately from Proposition 3.3 □

The converse of the theorem above does not hold; Type 3 grammars are strictly less expressive.

Theorem 3.10 *There exists a termination-preserving CCS$_!$ process P such that $L(P)$ is not Type 3.*

Proof: The above statement can be shown by providing a process which generates the typical $a^n b^n$ context-free language. Namely, let us take

$$P = (\nu\ k, u)\ (\overline{k}\ |!(k.a.(\overline{k} \mid \overline{u})) \mid k.!(u.b)).$$

One can easily verify that P is termination-preserving and that $L(P) = a^n b^n$. □

3.6.2 Impossibility Result: Context Free Languages

Context-Free Languages (CFL) are those generated by Type 2 grammars: grammars where every production is of the form $A \rightarrow \gamma$ where A is a non-terminal symbol and γ is a string consisting of terminals and/or non-terminals.

We have already seen that termination-preserving CCS$_!$ process can encode a typical CFL language such as $a^n b^n$. Nevertheless, we shall show that they cannot in general encode Type 2 grammars.

The nesting of restriction processes plays a key role in the following results CCS$_!$.

Definition 3.7 *The maximal number of nesting of restrictions $|P|_\nu$ can be inductively given as follows:*

$$|(\nu\ x)\ P|_\nu = 1 + |P|_\nu \qquad |P \mid Q|_\nu = max(|P|_\nu, |Q|_\nu)$$
$$|\alpha.P|_\nu = |!P|_\nu = |P|_\nu \qquad |\mathbf{0}|_\nu = 0$$

A very distinctive property of CCS$_!$ is that the maximal nesting of restrictions is invariant during evolution.

Proposition 3.4 *Let P and Q be CCS$_!$ processes. If $P \stackrel{s}{\Longrightarrow} Q$ then $|P|_\nu = |Q|_\nu$.*

Proof: The proposition can be proved by induction on the reductions steps of the operational semantics:

- ACT $\dfrac{}{\alpha.P \xrightarrow{\alpha} P}$: from definition 3.7 $|\alpha.P|_\nu = |P|_\nu$.
- RES $\dfrac{P \xrightarrow{\alpha} P'}{(\nu\ a)\ P \xrightarrow{\alpha} (\nu\ a)\ P'}$ if $\alpha \notin \{a, \overline{a}\}$: by inductive hypothesis we have that $|P|_\nu = |P'|_\nu$ hence by definition 3.7 $|(\nu\ a)\ P|_\nu = |(\nu\ a)\ P'|_\nu$.
- $\text{PAR}_1 \dfrac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q}$: by inductive hypothesis we have that $|P|_\nu = |P'|_\nu$ hence by definition 3.7 $|P \mid Q|_\nu = |P' \mid Q|_\nu$. (Similarly one can prove rule PAR_2)
- COM $\dfrac{P \xrightarrow{l} P' \quad Q \xrightarrow{\bar{l}} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}$: by inductive hypothesis we have that $|P|_\nu = |P'|_\nu$ $|Q|_\nu = |Q'|_\nu$and hence by definition 3.7 $|P \mid Q|_\nu = |P' \mid Q'|_\nu$.
- REP $\dfrac{P \mid !P \xrightarrow{\alpha} P'}{!P \xrightarrow{\alpha} P'}$: by inductive hypothesis we have that $|P \mid !P|_\nu = |P'|_\nu$ hence by definition 3.7 $|P \mid !P|_\nu = max(|P|_\nu, |!P|_\nu)$ but $|P|_\nu = |!P|_\nu$ thus concluding that $|!P|_\nu = |P'|_\nu$.

□

Remark 3.11 *In CCS because of the* unfolding *of recursive definitions the nesting of restrictions can increase unboundedly during evolution*[3]*. E.g., consider* $A(a)$ *where* $A(x) \overset{\texttt{def}}{=} (\nu\ y)\ (x.\bar{y}.R \mid y.A(x))$ *(see Section 3.2.1) which has the following sequence of transitions*

$$A(a) \overset{aaa\ldots}{\Longrightarrow} (\nu y)(R \mid (\nu y)(R \mid (\nu y)(R \mid \ldots)))$$

[3]Also in the π-calculus [119], an extension of $\text{CCS}_!$ where names are communicated, the nesting of restrictions can increase during evolution due to its name-extrusion capability.

Another distinctive property of CCS$_!$ is that if a CCS$_!$ process can perform a given action β, it can always do it by performing a number of actions bounded by a value that depends only on the size of the process. In fact, as stated below, for a significant class of processes, the bound can be given solely in terms of the maximal number of nesting of restrictions.

Now, the above statement may seem incorrect since as mentioned earlier CCS$_!$ is Turing expressive. One may think that β above could represent a termination signal in a TM encoding, then it would seem that its presence in a computation cannot be determined by something bounded by the syntax of the encoding. Nevertheless, recall that the Turing encoding in [20] may wrongly signal β (i.e., even when the encoded machine does not terminate) but it will diverge afterwards.

The following section is devoted to some lemmas needed for proving our impossibility results for CCS$_!$ processes.

3.6.3 Trios-Processes.

For technical reasons we shall work with a family of CCS$_!$ processes, namely *trios-processes*. These processes can only have prefixes of the form $\alpha.\beta.\gamma$. The notion of trios was introduced for the π-calculus by Parrow in [104]. We shall adapt trios and use them as a technical tool for our purposes.

We shall say that a CCS$_!$ process T is a *trios-process* iff all prefixes in T are *trios*; i.e., they all have the form $\alpha.\beta.\gamma$ and satisfy the following: If $\alpha \neq \tau$ then α is a *name* bound in T, and similarly if $\gamma \neq \tau$ then γ is a *co-name* bound in T. For instance $(\nu l)(\tau.\tau.\bar{l} \mid l.a.\tau)$ is a trios-process. We will view a trio $l.\beta.\bar{l}$ as linkable node with incoming link l from another trio, outgoing link $\bar{l}$ to another trio, and contents β.

Interestingly, the family of trios-processes can capture the behaviour of arbitrary CCS$_!$ processes via the following encoding:

Definition 3.8 *Given a CCS$_!$ process P, $[\![P]\!]$ is the trios-process $(\nu\ l)\ (\tau.\tau.\bar{l} \mid [\![P]\!]_l)$ where $[\![P]\!]_l$, with $l \notin n(P)$, is inductively defined as follows:*

$$
\begin{array}{ll}
[\![0]\!]_l = & 0 \\
[\![\alpha.P]\!]_l = & (\nu\ l')\ (l.\alpha.\overline{l'}\ \mid\ [\![P]\!]_{l'})\ \textit{where}\ l' \notin n(P) \\
[\![P \mid Q]\!]_l = & (\nu\ l', l'')\ (l.\overline{l'}.\overline{l''}\ \mid\ [\![P]\!]_{l'} \mid [\![Q]\!]_{l''})\ \textit{where}\ l', l'' \notin n(P) \cup n(Q) \\
[\![!P]\!]_l = & (\nu\ l')\ (!l.\overline{l'}.\bar{l}\ \mid\ ![\![P]\!]_{l'})\ \textit{where}\ l' \notin n(P) \\
[\![(\nu\ x)\ P]\!]_l = & (\nu\ x)\ [\![P]\!]_l
\end{array}
$$

Notice that the trios-process $[\![\alpha.P]\!]_l$ encodes a process $\alpha.P$ much like a linked list. Intuitively, the trio $l.\alpha.\overline{l'}$ has an outgoing link l to its continuation $[\![P]\!]'_l$ and incoming link l from some previous trio. The other cases can be explained analogously. Clearly the encoding introduces additional actions but they are all silent—i.e., they are synchronisations on the bound names l, l' and l''.

Unfortunately the above encoding is not invariant w.r.t. language equivalence because the replicated trio in $[\![!P]\!]_l$ introduces divergence. E.g, $L((\nu x)!x) = \{\epsilon\}$ but $L([\![(\nu x)!x]\!]) = \emptyset$. It has, however, a pleasant invariant property: *weak bisimilarity.*

Definition 3.9 (Weak Bisimilarity) *A* (weak) simulation *is a binary relation $\mathcal{R}$ satisfying the following: $(P, Q) \in \mathcal{R}$ implies that:*

- *if $P \stackrel{s}{\Longrightarrow} P'$ where $s \in \mathcal{L}^*$ then $\exists Q' : Q \stackrel{s}{\Longrightarrow} Q' \wedge (P', Q') \in \mathcal{R}$.*

The relation $\mathcal{R}$ is a bisimulation *iff both $\mathcal{R}$ and its converse $\mathcal{R}^{-1}$ are simulations. We say that P and Q are* (weak) bisimilar*, written $P \approx Q$ iff $(P, Q) \in \mathcal{R}$ for some bisimulation $\mathcal{R}$.*

Proposition 3.5 *For every CCS$_!$ process P, $P \approx [\![P]\!]$ where $[\![P]\!]$ is the trios-process constructed from P as in Definition 3.8.*

Proof: Follows immediately from Definition 3.8. □

Another property of trios is that if a trios-process T can perform an action α, i.e., $T \stackrel{s.\alpha}{\Longrightarrow}$, then $T \stackrel{s'.\alpha}{\Longrightarrow}$ where s' is a sequence of actions whose length bound can be given solely in terms of $|T|_\nu$.

Proposition 3.6 *Let T be a trios-process such that $T \stackrel{s\cdot\beta}{\Longrightarrow}$. There exists a sequence s', whose length is bounded by a value depending only on $|T|_\nu$, such that $T \stackrel{s'\cdot\beta}{\Longrightarrow}$.*

Proof: Our approach is to consider a minimal sequence of visible actions $t = \beta_1 \ldots \beta_m$ performed by T leading to β (i.e., $P \stackrel{t}{\Longrightarrow}$ and $\beta_m = \beta$) and analyse the *causal dependencies* among the (*occurrences* of) the actions in this t. Intuitively, β_j depends on β_i if T, while performing t, could not had performed β_j without performing β_i first. For example in

$$T = (\nu l)(\nu l')(\nu l'')(\tau.a.\bar{l} \mid \tau.b.\overline{l'} \mid l.l'.\overline{l''} \mid l''.c.\tau)$$

$\beta = c$, $t = abc$, we see that c depends on a and b, but b does not depend on a since T could had performed b before a.

We then consider the unique directed acyclic graph G_t arising from the transitive reduction[4] of the partial ordered induced by the dependencies in t. Because t is minimal, β is the only sink of G_t.

We write $\beta_i \rightsquigarrow_t \beta_j$ (β_j depends directly on β_i) iff G_t has an arc from β_i to β_j. The crucial observation from our restrictions over trios is that if $\beta_i \rightsquigarrow_t \beta_j$ then (the trios corresponding to the occurrences of) β_i and β_j must occur in the scope of a restriction process R_{ij} in T (or in some evolution of T while generating t). Take e.g, $T = \tau.a.\tau \mid (\nu\ l)\ (\tau.b.\bar{l} \mid l.c.\tau)$ with $t = a.b.c$ and $b \rightsquigarrow c$. Notice that the trios corresponding to the actions b and c appear within the scope of the restriction in T.

To give an upper bound on the number of nodes of G_t (i.e., the length of t), we give an upper bound on its length and maximal in-degree. Take a path $\beta_{i_1} \rightsquigarrow_t \beta_{i_2} \ldots \rightsquigarrow_t \beta_{i_u}$ of size u in G_t. With the help of the above observation, we consider sequences of restriction processes $R_{i_1 i_2} R_{i_2 i_3} \ldots R_{i_{u-1} i_u}$ such that for every $k < u$ the actions β_{i_k} and $\beta_{i_{k+1}}$ (i.e., the trios where they occur) must be under the scope of $R_{i_k i_{k+1}}$. Note that any two different restriction processes with a common trio under their scope (e.g. $R_{i_1 i_2}$ and $R_{i_2 i_3}$) must be nested, i.e., one must be under the scope of the other. This induces tree-like nesting among the elements of the sequence

[4]The transitive reduction of a binary relation r on X is the smallest relation r' on X such that the transitive closure of r' is the same as the transitive closure of r.

of restrictions. E.g., for the restrictions corresponding to $\beta_{i_1} \rightsquigarrow_t \beta_{i_2} \rightsquigarrow_t \beta_{i_3} \rightsquigarrow_t \beta_{i_4}$ we could have a tree-like situation with $R_{i_1 i_2}$ and $R_{i_3 i_4}$ being under the scope of $R_{i_2 i_3}$ and thus inducing a nesting of at least two. We show that for a sequence of restriction processes, the number m of nesting of them satisfies $u \leq 2^m$. Since the nesting of restrictions remains invariant during evolution (Proposition 3.4) then $u \leq 2^{|T|_\nu}$. Similarly, we give an upper bound $2^{|T|_\nu}$ on the indegree of each node β_j of G_t (by considering sequences $R_{i_1 j}, \ldots, R_{i_m j}$ such that $\beta_{i_k} \rightsquigarrow \beta_j$, i.e having common trio corresponding to β_j under their scope). We then conclude that the number of nodes in G_t is bounded by $2^{|T|_\nu \times 2^{|T|_\nu}}$. □

Main Impossibility Result. We can now prove our main impossibility result.

Theorem 3.12 *There exists a Type 2 grammar G such that for every termination-preserving CCS$_!$ process P, $L(G) \neq L(P)$.*

Proof: It suffices to show that no process in CCS$_!^{-\omega}$ can generate the CFL $a^n b^n c$. Suppose, as a mean of contradiction, that P is a CCS$_!^{-\omega}$ process such that $L(P) = a^n b^n c$.

Pick a sequence $\rho = P \stackrel{a^n}{\Longrightarrow} Q \stackrel{b^n c}{\Longrightarrow} T \not\rightarrow$ for a sufficiently large n. From Proposition 3.5 we know that for some R, $[\![P]\!] \stackrel{a^n}{\Longrightarrow} R \stackrel{b^n c}{\Longrightarrow}$ and $R \approx Q$. Notice that R may not be a trios-process as it could contain prefixes of the form $\beta.\gamma$ and γ. However, such prefixes into $\tau.\beta.\gamma$ and $\tau.\tau.\gamma$, we obtain a trios-process R' such that $R \approx R'$ and $|R|_\nu = |R'|_\nu$. We then have $R' \stackrel{b^n c}{\Longrightarrow}$ and, by Proposition 3.6, $R' \stackrel{s'.c}{\Longrightarrow}$ for some s' whose length is bounded by a constant k that depends only on $|R'|_\nu$. Therefore, $R \stackrel{s'.c}{\Longrightarrow}$ and since $R \approx Q$, $Q \stackrel{s'.c}{\Longrightarrow} D$ for some D. With the help of Proposition 3.4 and from Definition 3.8 it is easy to see that $|R'|_\nu = |R|_\nu = |[\![P]\!]|_\nu \leq 1 + |P| + |P|_\nu$ where $|P|$ is the size of P. Consequently the length of s' must be independent of n, and hence for any $s'' \in \mathcal{L}^*$, $a^n s' c s'' \notin L(P)$. Nevertheless $P \stackrel{a^n}{\Longrightarrow} Q \stackrel{s'.c}{\Longrightarrow} D$ and therefore from Proposition 3.2 there must be at least one string $w = a^n s' c w' \in L(P)$; a contradiction. □

It turns out that the converse of Theorem 3.12 also holds: termination-preserving CCS$_!$ processes can generate non CFL's.

Theorem 3.13 *There exists a termination-preserving CCS$_!$ process P such that $L(P)$ is not a CFL.*

Proof: Take

$$P = (\nu\ k, u)\ (\overline{k}\ |!k.a.(\overline{k} \mid \overline{u})) \mid k.!u.(b \mid c))$$

One can verify that P is termination-preserving. Furthermore, $L(P) \cap a^*b^*c^* = a^nb^nc^n$, hence $L(P)$ is not a CFL since CFL's are closed under intersection with regular languages. □

Now, notice that if we allow the use of CCS$_!$ processes which are not termination-preserving, we can generate a^nb^nc straightforwardly by using a process similar to that of Example 3.1.

Example 3.2 Consider the process P below:

$$\begin{aligned} P &= (\nu\ k_1, k_2, k_3, u_b)\ (\ \overline{k_1} \mid \overline{k_2} \mid Q_a \mid Q_b \mid Q_c) \\ Q_a &= !k_1.a.(\overline{k_1} \mid \overline{k_3} \mid \overline{u_b}) \\ Q_b &= k_1.!k_3.k_2.u_b.b.\overline{k_2} \\ Q_c &= k_2.(c \mid u_b.DIV) \end{aligned}$$

where $DIV = !\tau$. One can verify that $L(P) = \{a^nb^nc\}$. □

Termination-Preserving CCS. Type 0 grammars can be encoded by using the termination-preserving encoding of RAMs in CCS given in [19]. However, the fact that preservation of termination is not as restrictive for CCS as it is for CCS$_!$ can also be illustrated by giving a simple termination-preserving encoding of Context-Free grammars.

Theorem 3.14 *For every type 2 grammar G, there exists a termination-preserving CCS process P_G, such that $L(P_G) = L(G)$.*

Proof: For simplicity we restrict ourselves to Type 2 grammars in Chomsky normal form. All production rules are of the form $A \rightarrow B.C$ or $A \rightarrow a$. We can encode the productions rules of the form $A \rightarrow B.C$ as the recursive definition $A(d) \stackrel{\text{def}}{=}$

$(\nu\ d')\ (B(d') \mid d'.C(d))$ and the terminal production $A \rightarrow a$ as the definition $A(d) \stackrel{\text{def}}{=} a.\overline{d}$. Rules with the same head can be dealt using the summation $P + Q$. One can verify that, given a Type 2 grammar G, the suggested encoding generates the same language as G.

Notice, however, that there can be a grammar G with a non-empty language exhibiting derivations which do not lead to a sequence of terminal (e.g., $A \rightarrow B.C$, $A \rightarrow a$, $B \rightarrow b$, $C \rightarrow D.C$,$D \rightarrow d$). The suggested encoding does not give us a termination-preserving process. However one can show that there exists another grammar G', with $L(G) = L(G')$ whose derivations can always lead to a final sequence of terminals . The suggested encoding applied to G' instead, give us a termination-preserving process. □

3.6.4 Inside Context Sensitive Languages (CSL)

Context-Sensitive Languages (CSL) are those generated by Type 1 grammars. We conjecture that every language generated by a termination-preserving CCS$_!$ process is context sensitive.

The next proposition reveals a key property of any given weakly termination-preserving CCS$_!$ process P which can be informally described as follows. Suppose that P generates a sequence s of size n. By using a technique similar to the proof of Theorem 3.12 and Proposition 3.6, we conjecture that we can prove that there must be a trace of P that generates s with a total number of τ actions bounded by kn where k is a constant associated to the size of P. More precisely,

Conjecture 3.1 *Let P be a termination-preserving CCS$_!$ process. There exists a constant k such that for every $s = \alpha_1 \ldots \alpha_n \in L(P)$ then there must be a sequence*

$$P(\stackrel{\tau}{\longrightarrow})^{m_0} \stackrel{\alpha_1}{\longrightarrow} (\stackrel{\tau}{\longrightarrow})^{m_1} \ldots (\stackrel{\tau}{\longrightarrow})^{m_{n-1}} \stackrel{\alpha_n}{\longrightarrow} (\stackrel{\tau}{\longrightarrow})^{m_n} \not\rightarrow$$

with $\Sigma_{i=0}^{n} m_i \leq kn$.

Now recall that context-sensitive grammars are equivalent to linear bounded non-deterministic Turing machines. That is a non-deterministic Turing machine

with a tape with only kn cells, where n is the size of the input and k is a constant associated with the machine. Given P, we can define a non-deterministic machine which simulates the runs of P using the semantics of $\text{CCS}_!$ and which uses as many cells as the total number of performed actions, silent or visible, multiplied by a constant associated to P. Therefore, with the help of Conjecture 3.1, we obtain the following result.

Theorem 3.15 *If P is a termination-preserving $CCS_!$ process then $L(P)$ is a context sensitive language.*

Notice that from the above theorem and Theorem 3.12 it follows that the languages generated by termination-preserving $\text{CCS}_!$ processes form a proper subset of context sensitive languages.

3.7 Conclusions and Related Works

We have studied one possible interpretation of the non-determinism which is intrinsic in the language. We disallow the ability of synchronising simultaneously on the same channel, ability that is necessary for encoding Turing Machines. Indeed we prove that the calculus obtained in this way, $\text{CCS}_!^{-\omega}$, can faithfully encode Regular languages but it is impossible to provide a faithful encoding of Context Free Languages. Finally we conjecture that the languages generated by termination-preserving $\text{CCS}_!$ processes are Context Sensitive.

The closest related work is that in [19, 20] already discussed in the introduction. Furthermore in [19] the authors also provide a discrimination result between $\text{CCS}_!$ and CCS by showing that the divergence problem (i.e., given P, whether P has an infinite sequence of τ moves) is decidable for the former calculus but not for the latter.

In [65] Giambiagi et al. study replication and recursion in CCS focusing on the role of name scoping. In particular they show that $\text{CCS}_!$ is equivalent to CCS with recursion with static scoping. The standard CCS in [93] is shown to have dynamic

scoping. A survey on the expressiveness of replication vs recursion is given in [101] where several decidability results about variants of π, CCS and Ambient calculi can be found. None of these works study replication with respect to computability models less expressive than Turing Machines.

In [99] Nielsen et al. showed a separation result between replication and recursion in the context of temporal concurrent constraint programming (tccp) calculi. They show that the calculus with replication is no more expressive than finite-state automata while that with recursion is Turing Powerful. The semantics of tccp is rather different from that of CCS. In particular, unlike in CCS, processes interact via the shared-memory communication model and communication is asynchronous.

In the context of calculi for security protocols, the work in [75] uses a process calculus to analyse the class of ping-pong protocols introduced by Dolev and Yao. Huttel and Srba show that all nontrivial properties, in particular reachability, become undecidable for a very simple recursive variant of the calculus. The authors then show that the variant with replication renders reachability decidable. The calculi considered are also different from CCS. For example no restriction is considered and communication is asynchronous.

There is extensive work in process algebras and rewriting transition systems providing expressiveness hierarchies similar to that of Chomsky as well as results closely related to those of formal grammars. For example works involving characterisation of regular expression w.r.t. bisimilarity include [82, 92] and more recently [7]. An excellent description is provided in [18]. These works do not deal with replication nor the restriction operator which are fundamental to our study.

As for future work, we plan to provide a proof for Conjecture 3.1 or to find a counterexample. Moreover a somewhat complementary study to the one carried in this chapter would be to investigate what extension to $\text{CCS}_!$ is needed for providing faithful encoding of RAMs. Clearly the extension with recursion does the job but there may be simpler process constructions from process algebra which also do the job.

Chapter 4

Full Abstraction Techniques for Asynchrounous Languages

To repeat abstractly, universally, and distinctly in concepts the whole inner nature of the world, and thus to deposit it as a reflected image in permanent concepts always ready for the faculty of reason, this and nothing else is philosophy.

Arthur Schopenhauer
The World as Will and Representation

This chapter, differently from the previous one, studies an asynchronous language. The language is analysed w.r.t. the existence of fully abstract models. Full abstraction was introduced in the '60 (see [122] for an introduction on the subject) the idea is to deal with a denotational semantics of the language under consideration and then reduce the equivalence of terms to the equality of their semantics values. As an example of how this technique has been used see [70] where Hennessy and Plotkin exhibit a fully abstract model for a parallel programming language or see [109] where Plotkin proved that a fully abstract model for PCF does not exist.

Here we investigate full abstraction of a trace semantics for two Linda-like languages. The first language provides primitives for adding and removing messages

from a shared memory, local choice, parallel composition and recursion. The second one adds the possibility of checking for the absence of a message in the store. After having defined a denotational semantics based on traces, we obtain fully abstract semantics for both languages by using suitable abstractions in order to identify different traces which do not correspond to different operational behaviours.

4.1 Introduction

One of the fundamental purposes of a semantics is to provide a rigorous mean for proving the correctness of programs w.r.t. some behavioural specification. Several different tools (operational, denotational, algebraic and logic) can be used to this aim and ideally one would like to have a compositional and fully abstract semantics.

Compositionality is of course an important feature since it is the foundation for managing large systems complexity when considering program verification, analysis and (modular) design.

Full abstraction is also a desirable feature since it allows to simplify and "economise" as much as possible a semantics while preserving its correctness. However, in general this is a rather difficult target to achieve. To be more precise and to set the ground for the content of this chapter, following [17, 37, 74] we can summarise the terms of the problem as follows. Given a language L, define a semantics that associates to each process (or program) P in L a set of observable properties $\mathcal{O}(P)$. This is usually done in operational terms by using a transition system and a suitable definition of $\mathcal{O}(P)$ which identifies computational aspects relevant for a specific class of applications. In case such semantics is compositional, i.e. if we can reconstruct $\mathcal{O}(P \; op \; Q)$ from $\mathcal{O}(P)$ and $\mathcal{O}(Q)$ for any operator op of the language L, we have a satisfactory semantics, since the observational equivalence on processes induced by $\mathcal{O}(P)$ is preserved by contexts. More precisely, we have that $\mathcal{O}(P) = \mathcal{O}(Q)$ iff, for any context $C[\bullet]$, $\mathcal{O}(C[P]) = \mathcal{O}(C[Q])$ holds.

However often this is not the case and in order to obtain a compositional semantics some richer semantic structures than those used in $\mathcal{O}(P)$ need to be con-

sidered. For example, as we will see in Section 4.4, typically pairs representing the input/output behaviour of a process are not sufficient to obtain compositionality and one has to use traces. It can happen that these richer semantic structures "add too much" in the sense that the semantics $[\![\cdot]\!]$ based on them allows to distinguish processes which have the same behaviour w.r.t. $\mathcal{O}(P)$, under any possible context. In this case suitable abstractions must be used in $[\![\cdot]\!]$ in order to obtain a fully abstract result which, in general, can be stated as follows: $[\![P]\!] = [\![Q]\!]$ iff, for any context $C[\bullet]$, $\mathcal{O}(C[P]) = \mathcal{O}(C[Q])$ holds.

In this chapter we investigate the full abstraction problem, as described above, for two variants of Linda. Linda [63] is a programming paradigm which allows interprocess communication through a shared data space, also called tuple space, where processes can post and retrieve messages (also called tuples). The shared memory paradigm offers some advantages since it decouples communication between processes: communication is in fact asynchronous and processes do not need to be aware of each other identity or location. Indeed, the Linda paradigm has received also a commercial interest, mainly due to the applications which use the Java Spaces from Sun Microsystems [58] and TSpaces from IBM [76] models, both based on Linda (a more detailed comparison of Linda implementations can be found in [127]). Distributed Linda-like languages have also been investigated. Notably, Klaim [39] is an implemented language based on the Linda paradigm where the central store is replaced by several distributed local stores and processes mobility among different locations is supported.

Fully abstract semantics based on traces for input/output observables have been studied many years ago for several concurrent languages, as we shall discuss in Section 4.6. However, to the best of our knowledge no one has yet addressed this problem for a Linda-like language.

Many different formalisations and variants of Linda have been defined. Here we use essentially the process-algebraic formalisation of Linda introduced in [23, 24] Busi et al. and we consider two very basic Linda dialects. The first one, which we call Linda-core, apart from the usual operators in process algebra (choice, parallel

composition, recursion) contains the two Linda primitives *in* and *out* which allow to remove and add messages to the store, respectively. For Linda-core we define a compositional, fixpoint trace semantics which is correct but not fully abstract when considering the input/output pairs. Hence we introduce a suitable abstraction on traces and show that this allows us to obtain a fully abstract semantics. The second dialect (Linda-inp) enriches the syntax of Linda-core by allowing also a construct (*inp*) which allows to check the absence of information in the store. We prove that in this case a much simpler abstraction on traces is sufficient to obtain a full abstraction result. This accounts for the augmented expressive power of the language with *inp*, which can be formally proven by using the techniques in [23, 131]. Unfortunately, due to the saturation operator, the fully abstract semantics are not compositional. This is unavoidable in our trace model, since the properties that we need to abstract depend on sets of traces (rather than on single ones). Of course this does not mean that in general a compositional fully abstract semantics based on traces does not exist. However, in case it existed, it would use traces substantially more complicated than ours.

The remainder of the chapter is organised as follows. Section 4.2 introduces the Linda languages under consideration while Section 4.3 defines their denotational semantics. We then provide the fully abstract semantics for the core language in Section 4.4. Section 4.5 contains the main theorem on the full abstraction for the language extended with the *inp* primitive. Finally, Section 4.6 concludes by discussing some related works. [1]

4.2 Preliminaries

In this section, following the process algebraic view of Linda proposed in [23] we recall the syntax of the Linda languages that we consider and their operational semantics.

[1] A preliminary version of this chapter appeared in [44].

4.2.1 Linda-core

As previously mentioned, Linda is a paradigm which provides a simple model to describe communication between processes. The central notion in Linda is the one of *tuple space*. A tuple space is a shared data space (i.e. a common store) where all the *tuples* representing the information to be exchanged are stored. Here we shall abstract from the specific nature of tuples assuming that these are elementary messages. Communication is represented by the concurrent and asynchronous activity of several processes which add or remove messages from the common store. I.e. the sender dispatches a message through a non-blocking operation which adds the tuple in the tuple space. Then the message has an independent existence until a receiver retrieves and removes it from the shared space. Such kind of communication is called *generative* (see [63]).

Processes of the language Linda-core, denoted by $P, Q, \ldots$, are then given by the following grammar:

$$P ::= \mathbf{0} \mid out(a).P \mid in(a).P \mid P \mid P \mid P + P \mid recX.P \tag{4.1}$$

where we assume that $a \in Msg$ and Msg denotes the set of all possible messages (or tuples), ranged over by $a, b, \ldots$.

Intuitively $\mathbf{0}$ represents the process that does nothing. The process $out(a).P$ adds the message a to the store and then behaves as P. The message a which has been added to the store will be visible to other processes only after the completion of the $out(a)$ action, however note that other interpretations are possible for this primitive (see [22]). If a is present in the tuple space, $in(a).P$ removes the message and then behaves as P. Otherwise if a is not present, the process $in(a).P$ is suspended until a becomes available in the store. The parallel construct $P \mid Q$ is interpreted in terms of interleaving. The process $P+Q$ can non-deterministically choose to behave either as P or as Q (hence we have a form of local choice). Finally we have the recursion operator where we assume that guarded recursion is used.

The operational semantics of Linda-core is described by means of a transition system $T = (Conf, \rightarrow)$. Configurations $Conf$ are pairs of the form $\langle P, \mathcal{M} \rangle$ where P

R1 $\langle out(a).P, \mathcal{M}\rangle \rightarrow \langle P, \mathcal{M} \oplus \{a\}\rangle$

R2 $\langle in(a).P, \mathcal{M} \oplus \{a\}\rangle \rightarrow \langle P, \mathcal{M}\rangle$

R3 $\dfrac{\langle P, \mathcal{M}\rangle \rightarrow \langle P', \mathcal{M}'\rangle}{\langle P \mid Q, \mathcal{M}\rangle \rightarrow \langle P' \mid Q, \mathcal{M}'\rangle}$ and $\dfrac{\langle Q, \mathcal{M}\rangle \rightarrow \langle Q', \mathcal{M}'\rangle}{\langle P \mid Q, \mathcal{M}\rangle \rightarrow \langle P \mid Q', \mathcal{M}'\rangle}$

R4 $\langle P + Q, \mathcal{M}\rangle \rightarrow \langle P, \mathcal{M}\rangle$ and $\langle P + Q, \mathcal{M}\rangle \rightarrow \langle Q, \mathcal{M}\rangle$

R5 $\dfrac{\langle P[recX.P/X], \mathcal{M}\rangle \rightarrow \langle P', \mathcal{M}'\rangle}{\langle recX.P, \mathcal{M}\rangle \rightarrow \langle P', \mathcal{M}'\rangle}$

Figure 4.1: An operational semantics for Linda-core.

is a process and $\mathcal{M}$ is a multiset containing tuples, also called tuple space or store. The transition relation $\rightarrow \subseteq \mathit{Conf} \times \mathit{Conf}$ is the least relation satisfying the rules in Figure 4.1, which should be self-explaining, provided we introduce the following notation.

Notation 4.1 *To describe updates in the store we will use $\oplus$ and $\ominus$ to denote multisets union and difference, respectively. So $\mathcal{M} \oplus \{a\}$ means that a message (a tuple) 'a' has been added to the store while $\mathcal{M} \ominus \{a\}$ indicates that a copy of 'a' has been removed.*

A transition $\langle P, \mathcal{M}\rangle \rightarrow \langle Q, \mathcal{M}'\rangle$ then means that the process P reduces to Q, possibly by producing some changes in the store which evolves from $\mathcal{M}$ to $\mathcal{M}'$. A sequence of configurations is called run or computation. The reflexive transitive closure of $\rightarrow$ is denoted by $\Rightarrow$. By using the transition system described above we can characterise several different notions of observables. The ones we are interested in here consider simply the input/output behaviour of a process in terms of the tuple space. The input is therefore the initial tuple space, while the output is the final store produced by a process which cannot further procede in the computation (denoted by $\nrightarrow$) either because it is suspended on an *in* operation or because it has consumed all the actions. More precisely we define the observables as follows.

R6 $\langle inp(a)?P : Q, \mathcal{M} \oplus \{a\}\rangle \rightarrow \langle P, \mathcal{M}\rangle$
$\langle inp(a)?P : Q, \mathcal{M}\rangle \rightarrow \langle Q, \mathcal{M}\rangle$ provided $a \notin \mathcal{M}$

Figure 4.2: The rule for inp.

Definition 4.1 (Observables $\mathcal{O}(P)$) *Let P be a Linda process. We define*

$$\mathcal{O}(P) = \{(\mathcal{M}_1, \mathcal{M}_n) \mid \langle P, \mathcal{M}_1\rangle \Rightarrow \langle P_n, \mathcal{M}_n\rangle \not\rightarrow\}. \tag{4.2}$$

4.2.2 Linda-inp

We will now introduce a slightly different variant of Linda, called Linda-inp, obtained by adding a new operator $inp(a)?P : Q$ which allows also to check whether a message is not present in the store. More precisely, the previous construct checks whether the store contains the message a: if the message is present in the store then the process continues with P, otherwise with Q.

Therefore we will add to the grammar in (4.1) the following primitive:

$$P ::= inp(a)?P : P \tag{4.3}$$

The operational semantics for Linda-inp is obtained by (a transition system defined by) adding to the rules of Figure 4.1 the rules contained in Figure 4.2. The observables can be defined as before.

4.3 Denotational semantics

It is easy to see that the operational semantics which associates to a process P its observables $\mathcal{O}(P)$ is not compositional. For example consider the processes $P = out(a).in(a).out(b)$ and $Q = out(b)$. Then $\mathcal{O}(P) = \mathcal{O}(Q)$ holds, however, considering the process $R = in(a).out(ok)$ we have that $(\emptyset, \{ok\}) \in \mathcal{O}(P \mid R) \setminus \mathcal{O}(Q \mid R)$ which means that the observables of a parallel composition cannot be obtained from the observables of the two processes being composed (in parallel). This problem is in general well known, in fact in order to obtain a compositional model more

informative structures than input/output pairs have been used. In particular, models based on traces (or sequences) have been used for many concurrent languages, starting from the early works on dataflow languages [80], imperative ones [17] and concurrent constraint programming [37].

In the following we will define a compositional semantics which correctly models the $\mathcal{O}(P)$ observables and which is based on traces. This semantics is similar to those used for timed Linda in [36] (and therefore to that one of [37]), even though the technical treatment is different. In fact in [36], where maximal parallelism was assumed, the denotational model uses traces of pairs of tuple spaces, representing the input and the output at each step of the computation. Here, due to the interleaving semantics and to local choice, this kind of sequences is not sufficient to obtain a correct model. Essentially the problem is that we have to distinguish the processes $out(a) \mid in(a)$ and $out(a).in(a) + in(a).out(a)$ (because when starting with an empty store the second process can produce an empty store as a result) and this cannot be done by using simply input/output pairs. Hence, here we consider a denotational model which associates to a process a set of sequences of the form $\alpha_1, \ldots, \alpha_n$ where each α_i is an element of the set $\mathcal{A} = \{in(a), out(a), \overline{in}(a), \overline{inp}(a) \mid a \in Msg\}$ (where Msg denotes all the possible messages, as previously mentioned). The first two kinds of actions in $\mathcal{A}$ are obvious as they represent the corresponding operations on the store, while $\overline{in}(a)$ and $\overline{inp}(a)$ are used to express absence of information. We denote with $\mathcal{S}$ the set of all possible sequences defined in this way.

We introduce now two denotational semantics (one for each language we are considering) based on traces which are compositional by construction. Such semantics are the least functions which satisfy the equations in Figure 4.3 for Linda-core and the equations in Figure 4.3 plus that in Figure 4.5 for Linda-inp.

4.3.1 Denotational semantics for Linda-core

More precisely we define denotational semantics for the Linda-core language as a fixpoint:

D1 $[\![\mathbf{0}]\!] = \{\epsilon\}$

D2 $[\![out(a).P]\!] = \{out(a) \cdot s \mid s \in [\![P]\!]\}$

D3 $[\![in(a).P]\!] = \{in(a) \cdot s \mid s \in [\![P]\!]\} \cup \{\overline{in}(a)\}$

D4 $[\![P \mid Q]\!] = [\![P]\!] \mathbin{\tilde{|}} [\![Q]\!]$

where the operator $\mathbin{\tilde{|}}$ is inductively defined as follow:

$$(x \cdot s) \mathbin{\tilde{|}} y = y \mathbin{\tilde{|}} (x \cdot s) = \{(x \cdot t) \mid t \in s \mathbin{\tilde{|}} y\} \ \cup \ \{y \cdot x \cdot s\}$$

$$(x \cdot s) \mathbin{\tilde{|}} (y \cdot t) = (y \cdot t) \mathbin{\tilde{|}} (x \cdot s) =$$

$$\{(x \cdot u) \mid u \in s \mathbin{\tilde{|}} (y \cdot t)\} \cup \{(y \cdot u) \mid u \in (x \cdot s) \mathbin{\tilde{|}} t\}$$

with $x, y \in \mathcal{A}$ and $s, t, u \in \mathcal{S}$.

D5 $[\![P + Q]\!] = [\![P]\!] \cup [\![Q]\!]$

D6 $[\![recX.P]\!] = [\![P[recX.P/X]]\!]$

Figure 4.3: A denotational semantics for Linda-core.

Definition 4.2 *The denotational semantics $[\![\cdot]\!]$ is defined as the least function $[\![\cdot]\!] : Processes \to 2^{\mathcal{S}}$, which satisfy the equations in Figure 4.3. The order on functions is the one induced by set inclusion on the co-domain.*

Well known fixpoint results (see Section 2.2) allow to obtain the semantics as the least fixpoint of the operators defined implicitly by the equations in the Figure 4.3. The equations should be self-explanatory apart from a few details. The denotation of the **0** process is the empty sequence, while the equations D2 and D3 show the expected behaviour for the basic primitives. Note that in equation D3 we have two cases: the first one corresponds to the case in which a is present in the store, thus the computation can proceed (with the sequence s) after the in action. On the other hand, the $\overline{in}(a)$ action represents the absence of a in the store, in which case the computation terminates (the process is suspended). The parallel operator is interpreted in terms of interleaving as usual, while since the choice is local, it can

be modeled by a simple set union. Recursion is treated in the usual way.

The following is devoted to some definitions and lemmata needed for proving that the denotational semantics is correct w.r.t. our notion of observables.

First we need to relate traces with observables, therefore we define the evaluation function of a trace as follows ($\uparrow$ means undefined).

Definition 4.3 *Given a trace $s \in \mathcal{S}$ and a store $\mathcal{M}$, the function $eval_1(s, \mathcal{M})$ is defined by the following cases:*

$$eval_1(\epsilon, \mathcal{M}) = \mathcal{M}$$

$$eval_1(out(x) \cdot t, \mathcal{M}) = eval_1(t, \mathcal{M} \oplus \{x\})$$

$$eval_1(in(x) \cdot t, \mathcal{M}) = \begin{cases} eval_1(t, \mathcal{M} \ominus \{x\}) & \text{if } x \in \mathcal{M} \\ \uparrow & \text{otherwise} \end{cases}$$

$$eval_1(\overline{in}(x) \cdot t, \mathcal{M}) = \begin{cases} \mathcal{M} & \text{if } x \notin \mathcal{M} \text{ and } t = \epsilon \\ \uparrow & \text{otherwise} \end{cases}$$

Lemma 4.1 *Given a Linda-core process P, for every run*

$$\langle P,\ \mathcal{M}_0\rangle \rightarrow \ldots \rightarrow \langle P',\ \mathcal{M}_n\rangle \nrightarrow$$

there exists a trace $s \in [\![P]\!]$ such that $eval_1(s, \mathcal{M}_0) = \mathcal{M}_n$.

Proof: The proof proceeds by induction on the number of steps of the run

$$\langle P, \mathcal{M}_0\rangle \rightarrow \langle P_1, \mathcal{M}_1\rangle \rightarrow \ldots \rightarrow \langle P_n, \mathcal{M}_n\rangle$$

Thus by inductive hypothesis there exists a trace $s_1 \in [\![P_1]\!]$ such that

$$eval_1(s_1, \mathcal{M}_1) = \mathcal{M}_n$$

We now analyse all the possible steps that could have taken place as a first action:

- if $P = out(a).P'$ then $\mathcal{M}_1 = \mathcal{M}_0 \oplus \{a\}$, $s = out(a) \cdot s_1 \in [\![P]\!]$ and $eval_1(out(a) \cdot s_1, \mathcal{M}_0) = eval_1(s_1, \mathcal{M}_1)$

- similarly if $P = in(a).P'$ then $\mathcal{M}_1 = \mathcal{M}_0 \ominus \{a\}$, $s = in(a) \cdot s_1 \in [\![P]\!]$ and $eval_1(in(a) \cdot s_1, \mathcal{M}_0) = eval_1(s_1, \mathcal{M}_1)$
- if $P = Q + R$ then $\mathcal{M}_0 = \mathcal{M}_1$ and $s = s_1 \in [\![P]\!]$
- if $P = Q \mid R$ then without loss of generality suppose that Q is the process that is reducing: $\langle Q, \mathcal{M}_0\rangle \rightarrow \langle Q', \mathcal{M}_1\rangle$ with an action α, hence $P_1 = Q' \mid R$, $s_1 \in [\![Q' \mid R]\!]$ and $s_1 = s_{Q'} \tilde{|} s_R$. Now since $\alpha \cdot s_{Q'} \in [\![Q]\!]$ then $\alpha \cdot s_1 \in [\![Q \mid R]\!]$
- if $P = recX.Q$ then the first action α can take place only if $\langle Q[recX.Q/X, \mathcal{M}_0\rangle \rightarrow \langle Q', \mathcal{M}_1\rangle$. Thus $P_1 = Q'$ and $\alpha \cdot s_1 \in [\![Q[recX.Q/X]]\!] = [\![P]\!]$.

□

Generalising the previous result we can show the following:

Corollary 4.1 *Given a Linda-core process P,*
$\mathcal{O}(P) \subseteq \{(\mathcal{M}_0, eval_1(s, \mathcal{M}_0)) \mid s \in [\![P]\!]$ *and* $eval_1(s, \mathcal{M}_0) \neq \uparrow\}$ *holds.*

Proof: Let $(\mathcal{M}_0, \mathcal{M}_n) \in \mathcal{O}(P)$ hence there exists a run:

$$\langle P, \mathcal{M}_0\rangle \rightarrow \ldots \rightarrow \langle P', \mathcal{M}'\rangle \nrightarrow$$

and for Lemma 4.1 we can associate a trace s in the denotational semantics such that $eval_1(s, \mathcal{M}_0) = \mathcal{M}_n$. □

In order to complete the correctness result (i.e. proving the other inclusion) we will use a fixpoint characterisation of the semantics $[\![\cdot]\!]$ (see Section 2.2 for an introduction on fixpoints). This can be obtained by first considering an interpretation as a mapping $I : Processes \rightarrow 2^{\mathcal{S}}$ which associates to each process a denotation (i.e. a set of sequences). The set $\mathcal{I}$ of all interpretations is easily seen to be a cpo with the ordering induced by $\subseteq$.

A function $\mathcal{F} : \mathcal{I} \rightarrow \mathcal{I}$ is obtained by substituting $[\![\cdot]\!]$ for $\mathcal{F}(I)$ in equations D1-D5 and in the left hand side of equation D6, and by replacing $[\![\cdot]\!]$ for I in the right hand side of equation D6 (see Figure 4.4).

D1 $\mathcal{F}(I)(\mathbf{0}) = \{\epsilon\}$

D2 $\mathcal{F}(I)(out(a).P) = \{out(a) \cdot s \mid s \in \mathcal{F}(I)(P)\}$

D3 $\mathcal{F}(I)(in(a).P) = \{in(a) \cdot s \mid s \in \mathcal{F}(I)(P)\} \cup \{\overline{in}(a)\}$

D4 $\mathcal{F}(I)(P \mid Q) = \mathcal{F}(I)(P) \mathbin{\tilde{|}} \mathcal{F}(I)(Q)$

D5 $\mathcal{F}(I)(P + Q) = \mathcal{F}(I)(P) \cup \mathcal{F}(I)(Q)$

D6 $\mathcal{F}(I)(recX.P) = I(P[recX.P/X])$

Figure 4.4: Definition of $\mathcal{F}$.

We will now inductively define a series of functions $\{\mathcal{F}^n(\perp) \mid n \geq 0\}$, where $\perp$ is the least interpretation, $\mathcal{F}^0(\perp) = \perp$ and $\mathcal{F}^n(\perp) = \mathcal{F}(\mathcal{F}^{n-1}(\perp))$. This characterisation allows us to prove that:

Lemma 4.2 *$\forall n > 0$ if $s \in \mathcal{F}^n$ and $eval_1(s, \mathcal{M}_0) \neq \uparrow$ then $\langle P, \mathcal{M}_0\rangle \rightarrow \ldots \rightarrow \langle P', eval_1(s, \mathcal{M}_0)\rangle \nrightarrow$ is a sequence of length $O(n)$[2] and $(\mathcal{M}_0, eval_1(s, \mathcal{M}_0)) \in \mathcal{O}(P)$.*

Proof: We will proceed by induction on the structure of processes and on induction on the powers of the operator $\mathcal{F}^n(\perp)$:

- $P = \alpha.P_1$: where α is an input or output action, if $s \in \mathcal{F}^n(\perp)(P)$ then

  $\mathcal{F}^n(\perp)(P) =$ (by definition)
  $\{\alpha.s \mid s \in \mathcal{F}(\mathcal{F}^{n-1}(\perp))(P_1)\}$

  By inductive hypothesis there exists a sequence

  $$\langle P_1, \mathcal{M}'\rangle \longrightarrow \ldots \longrightarrow \langle P', eval_1(s, \mathcal{M}')\rangle \nrightarrow$$

  thus $(\mathcal{M}', eval_1(s, \mathcal{M}')) \in \mathcal{O}(P_1)$, dependingly on α we can reconstruct $\mathcal{M}'$ (e.g. if α is an output action then following from rule R1

  $$\langle P, \mathcal{M}_0\rangle \rightarrow \langle P_1, \mathcal{M}_0 \oplus \{a\}\rangle$$

[2]The length of the sequence is n plus a certain number of configurations that are due to the application of rule R4 and that does not correspond to any action in the denotational semantics

therefore $\mathcal{M}_0 \oplus \{a\} = \mathcal{M}'$) and conclude that $(\mathcal{M}_0, eval_1(s, \mathcal{M}_0)) \in \mathcal{O}(P)$.

- $P = P_1 \mid P_2$: if $s \in \mathcal{F}^n(\bot)(P)$ then

$$\begin{aligned} &\mathcal{F}^n(\bot)(P) = && \text{(by definition)} \\ &\mathcal{F}(\mathcal{F}^{n-1}(\bot))(P_1) \tilde{\mid} \mathcal{F}(\mathcal{F}^{n-1}(\bot))(P_2) \end{aligned}$$

  By inductive hypothesis we have two runs $\langle P_1, \mathcal{M}' \rangle \rightarrow \ldots \rightarrow \langle P', eval_1(s_1, \mathcal{M}') \rangle \nrightarrow$ such that $s_1 \in \mathcal{F}(\mathcal{F}^{n-1}(\bot))(P_1)$ and symmetrically $\langle P_2, \mathcal{M}'' \rangle \rightarrow \ldots \rightarrow \langle P', eval_1(s_2, \mathcal{M}'') \rangle \nrightarrow$ such that $s_2 \in \mathcal{F}(\mathcal{F}^{n-1}(\bot))(P_2)$. Hence since there exists a mapping between traces and runs and following from the application of Rule R3, we can reconstruct the trace s and the corresponding run proving that $(\mathcal{M}_0, eval_1(s, \mathcal{M}_0)) \in \mathcal{O}(P)$.

- $P = P_1 + P_2$ if $s \in \mathcal{F}^n(\bot)(P)$ then

$$\begin{aligned} &\mathcal{F}^n(\bot)(P) = && \text{(by definition)} \\ &\mathcal{F}(\mathcal{F}^{n-1}(\bot))(P_1) \cup \mathcal{F}(\mathcal{F}^{n-1}(\bot))(P_2) \end{aligned}$$

  Therefore let us assume that $s \in \mathcal{F}(\mathcal{F}^{n-1}(\bot))(P_1)$ (the other situation is symmetric), by inductive hypothesis $(\mathcal{M}_0, eval_2(s, \mathcal{M}_0)) \in \mathcal{O}(P_1)$ but $\mathcal{O}(P_1) \subseteq \mathcal{O}(P)$ hence concluding the proof.

- $P = recX.P_1$ if $s \in \mathcal{F}^n(\bot)(P)$ then

$$\begin{aligned} &\mathcal{F}^n(\bot)(P) = && \text{(by definition)} \\ &\mathcal{F}(\mathcal{F}^{n-1}(\bot))(P) = && \text{(by definition)} \\ &\mathcal{F}^{n-1}(\bot)(P[recX.P_1/X]) \end{aligned}$$

  Therefore by inductive hypothesis we have that $\langle P[recX.P_1/X], \mathcal{M}_0 \rangle \rightarrow \ldots \rightarrow \langle P', eval_1(s, \mathcal{M}_0) \rangle \nrightarrow$ is a sequence of length $O(n-1)$ and we can easily conclude that $(\mathcal{M}_0, eval_1(s, \mathcal{M}_0)) \in \mathcal{O}(P)$

□

Finally using a well known fixpoint theorem attributed to Kleene (Section 2.2) we can prove that:

D7 $[\![inp(a)?P:Q]\!] = \{in(a)\cdot s \mid s \in [\![P]\!]\} \cup \{\overline{inp}(a)\cdot s \mid s \in [\![Q]\!]\}$

Figure 4.5: The equations for Linda-inp.

Corollary 4.2 *Given a Linda-core process P,*
$\{(\mathcal{M}_0, eval_1(s,\mathcal{M}_0)) \mid s \in [\![P]\!]$ *and* $eval_1(s,\mathcal{M}_0) \neq\uparrow\} \subseteq \mathcal{O}(P)$ *holds.*

Proof: It is easy to see that $\mathcal{F}$ is a continuous function therefore applying the theorem of Kleene for fixpoints it can be shown that the semantics $[\![\cdot]\!]$ is the least fixpoint of $\mathcal{F}$, which can be obtained as the least upper bound of $\{\mathcal{F}^n(\perp) \mid n \geq 0\}$. Indeed given a trace $s \in [\![P]\!]$ for the fixpoint theorem $s \in \mathcal{F}^n$ for some n and then for lemma 4.2 we can conclude $(\mathcal{M}_0, eval_1(s,\mathcal{M}_0)) \in \mathcal{O}(P)$ □

Summarising the previous two results state that:

Theorem 4.2 (Correctness) *Given a Linda-core process P,*
$\mathcal{O}(P) = \{(\mathcal{M}_0, eval_1(s,\mathcal{M}_0)) \mid s \in [\![P]\!]$ *and* $eval_1(s,\mathcal{M}_0) \neq\uparrow\}$ *holds.*

4.3.2 Denotational semantics for Linda-inp

To obtain a denotational semantics for Linda-inp we extend the denotational semantics for Linda-core with the equation that treats the *inp* operator.

Definition 4.4 *The denotational semantics $[\![\cdot]\!]$ is defined as the least function $[\![\cdot]\!]$: Processes $\rightarrow 2^{\mathcal{S}}$, which satisfy the equations in Figure 4.3 plus the one in Figure 4.5. The order on functions is the one induced by set inclusion on the co-domain.*

When considering the Linda-inp language the denotational semantics can be obtained from Figure 4.3 by adding the equation in Figure 4.5. This difference w.r.t. the case of Linda-core is due to the presence of the *inp*, which is described by Equation D7: when a is present both the $inp(a)$ and the $in(a)$ construct are modeled in the same way, but when a is not present we have to distinguish the two cases (by using $\overline{in}(a)$ and $\overline{inp}(a)$) since it would not be correct to use the evaluation given in Definition 4.3 for the $in(a)$.

In order to prove the correctness of the model introduced above we need to add to $eval_1$ the new cases obtaining the evaluation function $eval_2$:

Definition 4.5 *Given a trace $s \in \mathcal{S}$ and a store $\mathcal{M}$, the function $eval_2(s, \mathcal{M})$ is defined by the following cases:*

$$eval_2(\overline{inp}(x) \cdot t, \mathcal{M}) = \begin{cases} eval_2(t, \mathcal{M}) & \text{if } x \notin \mathcal{M} \\ \uparrow & \text{otherwise} \end{cases}$$

$$eval_2(\alpha(x) \cdot t, \mathcal{M}) = eval_1(\alpha(x) \cdot t, \mathcal{M}) \text{ for } \alpha \neq \overline{inp}$$

Moreover we need to adapt Lemma 4.1 and 4.2 adding the cases of the *inp* operator:

Lemma 4.3 *Given a Linda-inp process P, for every run*

$$\langle P, \mathcal{M}_0 \rangle \rightarrow \ldots \rightarrow \langle P', \mathcal{M}_n \rangle \nrightarrow$$

there exists a trace $s \in [\![P]\!]$ such that $eval_2(s, \mathcal{M}_0) = \mathcal{M}_n$.

Proof: The proof proceed in the same way as in Lemma 4.1, we only have to consider the case when $P = inp(a)?Q : R$ by inductive hypothesis there exists a trace $s_1 \in [\![P_1]\!]$ such that $eval_2(s_1, \mathcal{M}_1) = \mathcal{M}_n$. Without loss of generality suppose that $P_1 = Q$ then $\mathcal{M}_1 = \mathcal{M}_0 \ominus \{a\}$, $s = in(a) \cdot s_1 \in [\![P]\!]$ and $eval_2(in(a) \cdot s_1, \mathcal{M}_0) = eval_1(s_1, \mathcal{M}_1)$. □

Generalising the previous result we can show the following:

Corollary 4.3 *Given a Linda-inp process P,*
$\mathcal{O}(P) \subseteq \{(\mathcal{M}_0, eval_1(s, \mathcal{M}_0)) \mid s \in [\![P]\!]$ and $eval_1(s, \mathcal{M}_0) \neq \uparrow\}$ holds.

Proof: Let $(\mathcal{M}_0, \mathcal{M}_n) \in \mathcal{O}(P)$ hence there exists a run:

$$\langle P, \mathcal{M}_0 \rangle \rightarrow \ldots \rightarrow \langle P', \mathcal{M}' \rangle \nrightarrow$$

and for Lemma 4.3 we can associate a trace s in the denotational semantics such that $eval_2(s, \mathcal{M}_0) = \mathcal{M}_n$. □

D7 $\mathcal{F}(I)(inp(a)?P:Q) = \{in(a)\cdot s \mid s \in \mathcal{F}(I)(P)\} \cup \{\overline{inp}(a)\cdot s \mid s \in \mathcal{F}(I)(Q)\}$

Figure 4.6: Definition of $\mathcal{F}$.

In order to complete the correctness result, we will use the fixpoint characterisation introduced before plus the rule in Figure 4.6.

Again this characterisation allows us to prove that:

Lemma 4.4 *$\forall n > 0$ if $s \in \mathcal{F}^n$ and $eval_2(s, \mathcal{M}_0) \neq \uparrow$ then $\langle P, \mathcal{M}_0\rangle \rightarrow \ldots \rightarrow \langle P', eval_2 1(s, \mathcal{M}_0)\rangle \nrightarrow$ is a sequence of length $O(n)$ and $(\mathcal{M}_0, eval_2(s, \mathcal{M}_0)) \in \mathcal{O}(P)$.*

Proof: The proof proceeds as in Lemma 4.2, the only case left is the one of the *inp* operator, so let $P = inp(a)?P_1 : P_2$, if $s \in \mathcal{F}^n(\bot)(P)$ then

$$\mathcal{F}^n(\bot)(P) = \qquad \text{(by definition)}$$
$$\{in(a).s \mid s \in \mathcal{F}(\mathcal{F}^{n-1}(\bot))(P_1)\} \cup \{\overline{inp}(a).s \mid s \in \mathcal{F}(\mathcal{F}^{n-1}(\bot))(P_2)\}$$

By inductive hypothesis there exists either a sequence

$$\langle P_1, \mathcal{M}'\rangle \rightarrow \ldots \rightarrow \langle P', eval_2(s, \mathcal{M}')\rangle \nrightarrow$$

or a sequence $\langle P_2, \mathcal{M}'\rangle \rightarrow \ldots \rightarrow \langle P', eval_2(s, \mathcal{M}')\rangle \nrightarrow$ and $(\mathcal{M}', eval_2(s, \mathcal{M}')) \in \mathcal{O}(P_1)$. Then from R6 we can easily reconstruct $\mathcal{M}'$ and thus conclude that $(\mathcal{M}_0, eval_2(s, \mathcal{M}_0)) \in \mathcal{O}(P)$. □

Finally:

Corollary 4.4 *Given a Linda-inp process P,*
$\{(\mathcal{M}_0, eval_2(s, \mathcal{M}_0)) \mid s \in [\![P]\!]$ and $eval_2(s, \mathcal{M}_0) \neq \uparrow\} \subseteq \mathcal{O}(P)$ holds.

Proof: It is easy to see that $\mathcal{F}$ is a continuous function therefore applying the theorem of Kleene for fixpoints it can be shown that the semantics $[\![\cdot]\!]$ is the least fixpoint of $\mathcal{F}$, which can be obtained as the least upper bound of $\{\mathcal{F}^n(\bot) \mid n \geq 0\}$. Indeed given a trace $s \in [\![P]\!]$ for the fixpoint theorem $s \in \mathcal{F}^n$ for some n and then for lemma 4.4 we can conclude $(\mathcal{M}_0, eval_2(s, \mathcal{M}_0)) \in \mathcal{O}(P)$. □

Summarising, the previous two results state that:

Theorem 4.3 (Correctness) *Given a Linda-inp process P,*
$\mathcal{O}(P) = \{(\mathcal{M}_0, eval_2(s, \mathcal{M}_0)) \mid s \in [\![P]\!] \text{ and } eval_2(s, \mathcal{M}_0) \neq \uparrow\}$ *holds.*

4.4 Full Abstraction for Linda-core

The aim of this section is to obtain a fully abstract semantics for Linda-core. The semantics introduced in the previous section represents a too fine description of the actions that affect the store, since it records all the possible changes while the observables capture only the initial and the final state. It is therefore immediate to find processes which have a different denotation, while having the same input/output behaviour under any possible context.

In order to obtain full abstraction we saturate the denotational semantics by adding all those traces which, intuitively, represent a computation whose input/output behaviour, in any possible context, can be simulated by a trace which is already in the semantics. The formal definition is as follows.

Definition 4.6 (Saturation) *Let $T \subseteq \mathcal{S}$ be a set of traces. We define the saturation of T as the minimal set $Sat(T)$ which satisfies the following rules:*

i) *if $s \in T$ then $s \in Sat(T)$*

ii) *if $s \cdot out(a) \cdot t \cdot in(a) \cdot v \in Sat(T)$ then $s \cdot t \cdot v \in Sat(T)$*

iii) *if $s \cdot out(a) \cdot t \cdot in(a) \cdot v \in Sat(T)$ then $s \cdot out(a) \cdot t \cdot in(a) \cdot out(a) \cdot in(a) \cdot v \in Sat(T)$*

iv) *$s \in Sat(T)$ iff $s \cdot in(a) \cdot out(a) \in Sat(T)$*

v) *if $s \cdot out(a) \cdot t \in Sat(T)$ then $s \cdot t' \in Sat(T)$ where $t' \in \{out(a) \mathbin{\tilde{|}} t\}$*

vi) *all the traces in T of the form $t \cdot \overline{in}(a) \cdot u$ with $u \neq \epsilon$ are removed.*

According to the previous definition in $Sat(T)$ we add all the traces which (i) are derived (inductively) from the traces in T by performing the following operations: (ii) Removing complementary actions $out(a)$ and $in(a)$ which appear, in this order,

in different places of the sequence; it is rather clear that this does not change the operational behaviour described by the original sequence. (iii) Adding a "stuttering step" represented by a sequence $out(a) \cdot in(a)$ of two complementary actions is also allowed, provided that both these actions occur before (in this order) in the sequence. Intuitively, if the $out(a)$ action does not appear before in the sequence we cannot add it, since the presence of a could trigger some new computation; moreover, since the multiplicity of a message is relevant, also in case the sequence contains $out(a)$ and not $in(a)$ we cannot add the sequence $out(a) \cdot in(a)$ because after the added $out(a)$ we would have one more a than in the original sequence, which, again, could trigger new computations. (iv) Stuttering steps of the form $in(a) \cdot out(a)$ can be safely added and removed only at the end of a sequence. (v) As stated in [22] an output prefix $out(a).P$ is observably equivalent to $out(a) \mid P$, note that from this rule follows that the core-language cannot observe the order of appearance of messages. (vi) Finally, $\overline{in}(a)$ represents a process suspended because the message a is not present in the store, hence it is not correct to assume that other actions could take place afterwards. Clearly this is not anymore true (apart from rule (vi)) in presence of a construct which allows to check for absence of information, as we will see in the next section.

The fully abstract semantics is obtained by applying the saturation defined above to the semantics $[\![\cdot]\!]$. In order to prove the full abstraction result we proceed by steps. First we prove that the abstraction introduced by Sat is correct (under any context) w.r.t. $\mathcal{O}(P)$. This result is obtained by first showing that the construction of $Sat([\![P]\!])$ does not add any trace that does not respect the observables of P. This is the content of the following Proposition, whose proof is immediate

Proposition 4.1 *Given a process P,*

$$\mathcal{O}(P) = \{(\mathcal{M}_0, eval_1(s, \mathcal{M}_0)) | s \in Sat([\![P]\!])\}.$$

Proof: For Theorem 4.2, $\mathcal{O}(P) = \{(\mathcal{M}_0, eval_1(s, \mathcal{M}_0)) \mid s \in [\![P]\!]\}$ and since by definition $[\![P]\!] \subseteq Sat([\![P]\!])$, $\mathcal{O}(P) \subseteq \{(\mathcal{M}_0, eval_1(s, \mathcal{M}_0)) \mid s \in Sat([\![P]\!])\}$.

For the other set-inclusion we shall analyse all the possible traces in $Sat([\![P]\!])$. Thus suppose that $s = s_1 \cdot out(x) \cdot s_2 \cdot in(x) \cdot s_3 \in [\![P]\!]$. Then there exists $t = s_1 \cdot s_2 \cdot s_3$ and $u = s_1 \cdot out(x) \cdot s_2 \cdot in(x) \cdot out(x) \cdot in(x) \cdot s_3 \in Sat([\![P]\!])$. It can be easily shown that $eval_1(t, \mathcal{M}_0)$ and $eval_1(u, \mathcal{M}_0)$ are equal to $eval_1(s, \mathcal{M}_0)$ thus we are not adding anything in $\mathcal{O}(P)$. We can proceed similarly for all the other traces obtained applying the rules in definition 4.6. □

Now we are ready to state that the abstract (saturated) semantics is correct under any context w.r.t. the chosen observation criteria. A *context* $C[\bullet]$ is defined as a process with a hole, that is, a process where a subprocess is left unspecified. $C[P]$ is then the process obtained from $C[\bullet]$ by replacing $\bullet$ for the process P.

Theorem 4.4 (Correctness for Linda-core) *Given two Linda-core process A, B, if* $Sat([\![A]\!]) = Sat([\![B]\!])$ *then, for every context* $C[\bullet]$, $\mathcal{O}(C[A]) = \mathcal{O}(C[B])$ *holds.*

Proof: We will first prove $\mathcal{O}(C[A]) \subseteq \mathcal{O}(C[B])$. Let $(\mathcal{M}_0, \mathcal{M}_1) \in \mathcal{O}(C[A])$ then following from Theorem 4.2 there exists $s \in [\![C[A]]\!]$ such that $\mathcal{M}_1 = eval_1(s, \mathcal{M}_0)$. Since the denotational semantics we provide is compositional $s = c \,\tilde{\circ}\, t$ for some suitable $\tilde{\circ}$, where $c \in [\![C[\bullet]]\!]$ and $t \in [\![A]\!]$.

Since $[\![A]\!] \subset Sat([\![A]\!]) = Sat([\![B]\!])$ then $t \in Sat([\![B]\!])$ therefore two cases could arise: (1) $t \in [\![B]\!]$ hence $s \in [\![C[B]]\!]$ and $(\mathcal{M}_0, \mathcal{M}_1) \in \mathcal{O}(C[B])$. (2) $t \notin [\![B]\!]$ then there exists $u \in [\![B]\!]$ such that u is derived from t following the rules in definition 4.6 and $eval_1(t, \mathcal{M}_0) = eval_1(u, \mathcal{M}_0)$. Hence by induction on the structure of c it can be easily proved that $eval_1(c \,\tilde{\circ}\, u, \mathcal{M}_0) = \mathcal{M}_1$ and therefore $(\mathcal{M}_0, \mathcal{M}_1) \in \mathcal{O}(C[B])$.

The other set inclusion $\mathcal{O}(C[B]) \subseteq \mathcal{O}(C[A])$ is symmetrical. □

To obtain full abstraction we need now to prove the converse of the above theorem. This is the central result of this section and is the content of the following.

Theorem 4.5 *Let A, B be two Linda-core processes, if* $Sat([\![A]\!]) \neq Sat([\![B]\!])$ *then there exists a context* $C[\bullet]$ *such that* $\mathcal{O}(C[A]) \neq \mathcal{O}(C[B])$.

Proof: Suppose that there exists $t \in Sat([\![A]\!]) \setminus Sat([\![B]\!])$ and consider a generic $s \in Sat([\![B]\!])$ (thus $t \neq s$). From the definition of Sat it follows that we can choose

s and t as the shortest sequences such that: (i) they do not contain sub-sequences of the form $out(x) \cdot u \cdot in(x) \cdot out(x) \cdot in(x)$, (ii) they do not contain suffixes of the form $in(x) \cdot out(x)$, (iii) every output appears as soon as possible and (iv) between two consecutive inputs the outputs are ordered in lexicographic order.

Then assume that t and s have the following form: $t = r \cdot \alpha(x) \cdot t_1$, $s = r \cdot \beta(y) \cdot s_1$ where the common prefix r can also be empty and $\alpha, \beta \in \mathcal{A}$ with $\alpha \neq \beta$.

The proof is by cases, where we analyse the first couple of different actions α and β. In each case we will construct a context $C[\bullet]$ which allows to distinguish A and B (that is, a context such that $\mathcal{O}(C[A]) \neq \mathcal{O}(C[B])$). In the proof we will use the following notation: if $in(a_1), in(a_2), \ldots, in(a_n)$ are all the input actions which appear, in this order, in the sequence r (which can also contain other *out* actions), then $InComp(r)$ denotes the sequence $out(a_1) \cdot out(a_2) \cdots out(a_n)$: intuitively this sequence is a sort of complement (w.r.t. *in* actions) of r which allows to proceed in the computation when composed in parallel with r. Furthermore, in order to further simplify the notation, in the following we will use these assumptions:

$c_1 = InComp(r)$

c_2 is a sequence consisting of as many $in(x)$ as the $out(x)$ in r

c_3 is a sequence consisting of as many $in(y)$ as the $out(y)$ in r

We have then the following cases:

1. let $\beta(y) \cdot s_1 = \epsilon$, thus $t = r \cdot \alpha(x) \cdot t_1$ and $s = r$. Depending on t we can construct the following distinguishing contexts $C[\bullet]$:

   (a) if $t = r \cdot out(x) \cdot t_1$ then $C[\bullet] = \bullet \mid c_1.c_2.in(x).out(ok)$;

   (b) if $t = r \cdot in(x) \cdot t_1$ noticing that $t_1 \neq out(x)$, the following context can be provided $C[\bullet] = \bullet \mid c_1.out(x).InComp(t_1)$.

   The symmetric case is completely analogous.

2. $\alpha(x) = in(x)$ and $\beta(y) = in(y)$ (with $x \neq y$) then in order to distinguish the two processes we need to make further distinctions (note that by construction $t_1 \neq out(x)$):

   (a) if $eval_1(t_1, \emptyset) \neq \{x\}$ then $C[\bullet] = \bullet \mid c_1.c_3.out(x)$

   (b) if $eval_1(t_1, \emptyset) = \{x\}$ and the actions $out(y)$, $in(y)$ do not appear in t_1 then $C[\bullet] = \bullet \mid c_1.c_3.out(x).InComp(t_2)$

   (c) otherwise since $out(y)$ appears in t_1, it can be provided the following context $C[\bullet] = \bullet \mid c_1.c_3.out(x).in(y).out(y).out(y)$.

3. $\alpha(x) = out(x)$ and $\beta(y) = in(y)$ or vice versa: then it can be easily shown that the context $C[\bullet] = \bullet \mid c_1.c_2.c_3.in(x).out(ok)$ allows to distinguish A and B.

4. $\alpha(x) = out(x)$ and $\beta(y) = out(y)$ (with $x \neq y$). By hypothesis we can choose $t = r \cdot out(x) \ldots in(v) \ldots$ and $s = r \cdot out(y) \ldots in(w) \ldots$ where $in(v)$ and $in(w)$ are the first input actions after $out(x)$ and $out(y)$ respectively. Moreover $out(x)$ does not appear before $in(w)$ in s. Then two cases could arise if $v \neq x$ then the context $C[\bullet] = \bullet \mid c_1.c_4.c_5$ where c_4 and c_5 are sequences of as many $in(v)$ and $in(w)$ as the $out(v)$ and $out(w)$ that precedes the two input actions respectively. Instead if $v = x$ then we can safely assume $in(w)$ does not appear in s and the context $C[\bullet] = \bullet \mid c_1.c_5.InComp(t_1)$ can distinguish the two processes.

5. There are some remaining cases, where the two sequences are different because of a $\overline{in}$ action. However, due to the construction of our semantics, $r \cdot \overline{in}(x) \in Sat(\llbracket\llbracket A \rrbracket\rrbracket)$ iff $r \cdot in(x) \cdot s \in Sat(\llbracket\llbracket A \rrbracket\rrbracket)$. Therefore we can omit to consider the sequence $r \cdot \overline{in}(x)$ and just consider the case of the sequence $r \cdot in(x) \cdot s$, which is included above.

This completes the proof. □

The previous two theorems can be summarised in the following immediate corollary.

Corollary 4.5 (Full Abstraction for Linda-core) *Let A, B be two Linda-core processes, $Sat([\![A]\!]) = Sat([\![B]\!])$ iff, for any context $C[\bullet]$, $\mathcal{O}(C[A]) = \mathcal{O}(C[B])$ holds.*

4.5 Full Abstraction for Linda-inp

Now we move to consider the language Linda-inp where we can test for the absence of a message in the store by using the primitive *inp*. As underlined in Section 4.1, such a possibility augments the expressive power of the language. In semantic terms this means that we can construct more powerful contexts, thus allowing to discriminate processes which were identified by Linda-core contexts. As a simple example, consider the two processes $A = out(a).out(b)$ and $B = out(b).out(a)$. These processes cannot be distinguished (w.r.t. the observables $\mathcal{O}$) by any Linda-core contexts, indeed the corresponding traces $out(a) \cdot out(b)$ and $out(b) \cdot out(a)$ are identified by the saturation operation. However, the context $C[\bullet] = \bullet \mid in(a).(inp(b)?out(nok) : out(ok))$ allows to distinguish them, since it allows to check that a is present and b is absent in the store. Indeed we have that $(\emptyset, ok \in \mathcal{O}(C[A]) \setminus \mathcal{O}(C[B]))$. This example shows that a fully abstract semantics for Linda-inp must induce a finer equivalence on processes than Sat or, in other terms, that a less abstract operation has to be used to saturate sequences. However the Denotational semantics provided in Section 4.3.2 is not fully abstract. In fact, consider the two processes $A = inp(a)?\mathbf{0} : \mathbf{0}$ and $B = in(a) + A$: these two processes cannot be distinguish by any context, yet they have a different denotational semantics. Thus we need the following definition.

Definition 4.7 (Saturation for Linda-inp) *Let $T \subseteq \mathcal{S}$ be a set of traces. We define the inp-saturation of T as the set $Sat_2(T)$ which is obtained by performing the following steps (in this order) on T:*

1. *all the traces in T of the form $t \cdot \overline{in}(a) \cdot u$ with $u \neq \epsilon$ are removed;*

2. *all the $\overline{in}(x)$ actions in all traces are replaced by $\overline{inp}(x)$ (for any x).*

Condition 1 ensures that we obtain correct traces once we have performed the transformation in 2. In fact, $\overline{in}(a)$ comes from the evaluation of $in(a)$, when a is not present. Since such an evaluation is suspended, it is not correct to assume that some action can be performed later. Thus, before transforming $\overline{in}(a)$ into $\overline{inp}(a)$ (and therefore moving from the $eval_1$ of Definition 4.3 to $eval_2$ of Definition 4.5) we have to delete these traces. The correctness of the saturation is stated by the following proposition.

Proposition 4.2 *Given a process P,*

$$\mathcal{O}(P) = \{(\mathcal{M}_0, eval_2(s, \mathcal{M}_0)) | s \in Sat_2([\![P]\!])\}.$$

Proof: Immediate. □

Now, as before, we are ready to state that the abstract semantics is correct under any context w.r.t. the chosen observation criteria.

Theorem 4.6 (Correctness for Linda-inp) *Let A, B be two Linda-inp processes, if $Sat_2([\![A]\!]) = Sat_2([\![B]\!])$ then, for every context $C[\bullet]$, $\mathcal{O}(C[A]) = \mathcal{O}(C[B])$ holds.*

Proof: We will first prove $\mathcal{O}(C[A]) \subseteq \mathcal{O}(C[B])$. Let $(\mathcal{M}_0, \mathcal{M}_1) \in \mathcal{O}(C[A])$ then following from Theorem 4.3 there exists $s \in [\![C[A]]\!]$ such that $\mathcal{M}_1 = eval_2(s, \mathcal{M}_0)$. Since the denotational semantics we provide is compositional $s = c \;\tilde{\circ}\; t$ for some suitable $\tilde{\circ}$, where $c \in [\![C[\bullet]]\!]$ and $t \in [\![A]\!]$.

Applying the rules in Definition 4.7 we can construct a trace t' such that $eval_2(t, \mathcal{M}_0) = eval_2(t', \mathcal{M}_0)$. Hence $t' \in Sat_2([\![A]\!])$. Now since $Sat_2([\![A]\!]) = Sat_2([\![B]\!])$, $t' \in Sat_2([\![B]\!])$ two cases could arise: (1) $t' \in [\![B]\!]$ hence $s \in [\![C[B]]\!]$ and $(\mathcal{M}_0, \mathcal{M}_1) \in \mathcal{O}(C[B])$. Or (2) $t' \notin [\![B]\!]$ therefore there exists $u \in [\![B]\!]$ where some of the actions $\overline{in}$ have been replaced with $\overline{inp}$ and $eval_2(t, \mathcal{M}_0) = eval_2(u, \mathcal{M}_0)$. Hence by induction on the structure of c it can be easily proved that $eval_2(c \;\tilde{\circ}\; u, \mathcal{M}_0) = \mathcal{M}_1$ and therefore $(\mathcal{M}_0, \mathcal{M}_1) \in \mathcal{O}(C[B])$.

The other set inclusion $\mathcal{O}(C[B]) \subseteq \mathcal{O}(C[A])$ is symmetrical. □

And finally to obtain full abstraction we need now to prove the converse of the above theorem. This is the content of the following.

Theorem 4.7 *Let A, B be two Linda-inp processes, if $Sat_2([\![A]\!]) \neq Sat_2([\![B]\!])$ then there exists a context $C[\bullet]$ such that $\mathcal{O}(C[A]) \neq \mathcal{O}(C[B])$.*

Proof: Suppose that there exists $t \in Sat_2([\![A]\!]) \setminus Sat_2([\![B]\!])$ and consider a generic $s \in Sat_2([\![B]\!])$. Since $s \neq t$ by hypothesis, we can assume that t and s have the following form:

Let $t = r \cdot \alpha_1(x_1) \cdots \alpha_n(x_n)$ and $s = r \cdot \beta_1(y_1) \cdots \beta_m(y_m)$ where the common prefix r can also be empty and $\alpha_1, \ldots, \alpha_n, \beta_1, \ldots, \beta_m \in \mathcal{A}$ with $\alpha_1 \neq \beta_1$.

The proof is by cases, where we analyse the first different actions α_1 and β_1 in the sequences t and s. In each case we will construct a context $C[\bullet]$ which allows to distinguish A and B (that is, a context such that $\mathcal{O}(C[A]) \neq \mathcal{O}(C[B])$). As in the proof of Theorem 4.5, if $in(a_1), in(a_2), \ldots, in(a_n)$ are all the input actions which appear, in this order, in the sequence r then $InComp(r)$ denotes the sequence $out(a_1) \cdot out(a_2) \cdots out(a_n)$. Furthermore, in order to further simplify the notation, in the following we will use these assumptions:

$c_1 = InComp(r)$

c_2 is a sequence consisting of as many $in(a)$ as the $out(a)$ in r

c_3 is a sequence consisting of as many $in(b)$ as the $out(b)$ in r

We have then the following cases:

1. $t = r \cdot out(a) \cdot t_1$ and $s = r$; In this case $C[\bullet] = \bullet \mid c_1.c_2.in(a).out(ok)$ allows to distinguish A and B.

2. $t = r \cdot in(a) \cdot t_1$ and $s = r$; then $C[\bullet] = out(a).\bullet \mid c_1.c_2.inp(a)?out(ok) : out(no)$ is the distinguishing context.

3. $t = r \cdot out(a) \cdot t_1$ and $s = r \cdot out(b) \cdot s_1$. We have the following sub-cases:

(a) If the number of $out(a)$ in t is different from the number of $out(a)$ in s then it can be easily proved that there is a context that distinguishes the two programs (essentially it is a context that *counts* the occurrences of the $out(a)$). Similarly if we are considering the b's. The following is an example.

Example 4.1 If $t = out(a) \cdot in(a) \cdot out(a) \cdot out(b)$ and $s = out(b) \cdot out(a)$ then we can build the distinguishing context

$$C[\bullet] = \bullet \mid in(a).out(a).inp(a)?out(ok) : out(no)$$

□

(b) Now suppose that the number of $out(a)$ (or $out(b)$) is the same in t and s. If in t_1 or in s_1 there is an input action again it is easy to provide a distinguishing context, either by blocking the execution of the rest of the trace after the input or by querying the store for the presence/absence of messages in the store. The following provide an example.

Example 4.2 If $t = out(a) \cdot in(b) \cdot out(b)$ and $s = out(b) \cdot in(b) \cdot out(a)$ then we can consider the distinguishing context

$$C[\bullet] = \bullet \mid in(a).out(b).inp(b)?out(ok) : out(no)$$

□

(c) If in t_1 and in s_1 there are only outputs then either there is an output action that it is not present in one of the two traces, and in this case it is straightforward to build a distinguishing context, or the output actions of a sequence are a permutation of output actions of the other sequence; also in this case it is easy to construct a context that distinguishes the two processes by checking the presence of a message and the absence of the other one, as shown by the following.

Example 4.3 If $t = out(a) \cdot out(b)$ and $s = out(b) \cdot out(a)$ then the distinguishing context $C[\bullet] = \bullet \mid in(a).inp(b)?out(ok) : out(no)$ (as seen in the initial part of this Section). □

4. $t = r \cdot out(a) \cdot t_1$, $s = r \cdot in(b) \cdot s_1$ and $s' = r \cdot \overline{inp}(b) \in Sat_2([\![B]\!])$. It suffices to consider $C[\bullet] = \bullet \mid c_1.c_2.c_3.in(a).out(ok)$.

5. $t = r \cdot out(a) \cdot t_1$, $s = r \cdot in(b) \cdot s_1$ and $s' = r \cdot \overline{inp}(b) \cdot s_2 \in Sat_2([\![B]\!])$. The following situations may arise:

   (a) if $out(a) \notin s_2$ then $C[\bullet] = \bullet \mid c_1.c_2.c_3.in(a).out(ok)$;

   (b) if $in(b) \notin t_1$ then $C[\bullet] = out(b).\bullet \mid c_1.inp(b)?out(ok) : out(no)$;

   (c) otherwise the only significant case is when $s' = r \cdot \overline{inp}(b) \cdot out(a) \cdot t_1$ and therefore a suitable context can be constructed observing that the order of the actions is different (i.e. b is consumed in two different positions). This is shown in the following.

   Example 4.4 If $t = out(a)$ and $s = in(b) \cdot out(b) \cdot out(a)$ recalling that $s' = \overline{inp}(b) \cdot out(a)$ we can build the distinguishing context

   $$C[\bullet] = out(b).\bullet \mid in(b).out(b).inp(b)?out(ok) : out(no)$$

   □

6. $t = r \cdot in(a) \cdot t_1$, $s = r \cdot in(b) \cdot s_1$ and $s = r \cdot \overline{inp}(b) \in Sat_2(B)$. In this case $C[\bullet] = out(a).\bullet \mid c_1.c_2.c_3.inp(a)?out(ok) : out(no)$.

7. $t = r \cdot in(a) \cdot t_1$, $s = r \cdot in(b) \cdot s_1$ and $s' = r \cdot \overline{inp}(b) \cdot s_2 \in Sat_2([\![B]\!])$. We should here distinguish between the following further cases

   (a) if $out(a) \notin t_1$ and $in(a) \notin s_2$ then $C[\bullet] = out(a).\bullet \mid c_1.c_3$;

   (b) otherwise the worst possible scenario happens when $s_2 = in(a) \cdot t_1$ and t_1 and s_1 are "symmetrical" in a and b. As already shown in some preceding cases, when the order of the actions changes it is always possible to find a distinguishing context. This is shown in the following, last example.

   Example 4.5 Let $A = inp(a)?(out(a).in(b).out(b)) : (in(b).out(b))$, and $B = inp(b)?\ (out(b).in(a).out(a)) : (in(a).out(a))$ thus $Sat_2([\![A]\!]) = \{in(a) \cdot out(a) \cdot in(b) \cdot out(b), \overline{inp}(a) \cdot in(b) \cdot out(b), \ldots\}$ and $Sat_2([\![B]\!]) = \{in(b) \cdot$

$out(b) \cdot in(a) \cdot out(a), \overline{inp}(b) \cdot in(a) \cdot out(a), \ldots\}$ and the following context can distinguish between the two programs:

$$C[\bullet] = \bullet \mid inp(a)?out(ok1) : (inp(b)?out(ok2) : out(no))$$

□

8. There are some remaining cases, where the two sequences are different because of a $\overline{inp}$ action. However, due to the construction of our semantics, $r \cdot \overline{inp}(x) \in Sat_2([\![A]\!])$ iff $r \cdot in(x) \cdot s \in Sat_2([\![A]\!])$. Therefore we can omit to consider the sequence $r \cdot \overline{inp}(x)$ and just consider the case of the sequence $r \cdot in(x) \cdot s$, which is included above.

This completes the proof. □

The previous two theorems can be summarised in the following immediate corollary.

Corollary 4.6 (Full Abstraction for Linda-inp) *Let A and B be two Linda-inp processes, $Sat_2([\![A]\!]) = Sat_2([\![B]\!])$ iff, for any context $C[\bullet]$, $\mathcal{O}(C[A]) = \mathcal{O}(C[B])$ holds.*

4.6 Conclusions and Related work

We have studied the full abstraction problem for two variants of the Linda paradigm. For the first one, the Linda-core language, we have provided a trace semantics which is fully abstract w.r.t. the input/output notion of observables. This has been obtained by using a suitable abstraction in order to identify different traces which do not represent meaningful operational differences. The second language, Linda-inp, allows also checking for the absence of information. The augmented expressive power of this language permits us to obtain a full abstraction result by using a much simpler abstraction.

In the specific context of Linda, full abstraction has been previously investigated by Brogi and Jaquet in [16] which used also techniques inspired by Horita et al. in

[74]. The results in [16] are completely different from ours, since in such a paper a semantics based on sequences is shown to be fully abstract with respect to a notion of observable which consider traces of computations. We prefer to consider a coarser notion of observables, consisting in the input/output behaviour, which accounts for a "black box" use of processes. Clearly our notion of observables leads to a more difficult full abstraction result, being the denotational model based on traces.

Results similar to ours have been obtained in the context of concurrent constraint programming (CCP) by De Boer and Palamidessi [37], however this language differs from Linda since it does not allow to remove information from the store. This monotonic nature of CCP makes its semantic treatment simpler, hence the results in [37] cannot be applied directly to the languages we consider here. Also Brookes [17] provides a trace model and a full abstraction result for a shared variable parallel language which is substantially different from Linda. The same applies to the results in [74].

More generally, full abstraction results have been provided for many concurrent languages and in quite various settings, which however are different from the case we consider here. In fact, even though our Linda-core language can be seen as a fragment of asynchronous CCS (and therefore of asynchronous π-calculus), all the full abstraction results available for these languages consider different observational equivalences from ours. Probably the closer work in this sense is [14] by Boreale et al., where full abstraction of a trace semantics w.r.t. may testing equivalence has been studied. Note however that may testing is different from the observational equivalence that we consider (which is based on the input-output behaviour). For example, the processes $in(a).in(b)$ and $in(b).in(a)$ are may testing equivalent (see [14]) while they are not equivalent in our case, since they can be distinguished by the context $C[\bullet] = out(a) \mid \bullet$.

Several other papers consider barbed equivalences and their relations with bisimulation, (notably [4] for asynchronous π-calculus and [23] for Linda-like process algebras) which, as previously mentioned, are completely different from the equivalence we consider. It is also worth noticing that the construct *inp*, which is not available in

π-calculus and in CCS, change considerably the semantics of the language, thus for Linda-inp one cannot use existing results for CCS or π-calculus. For example, [23] shows that in presence of *inp* the coarse congruence contained in barbed equivalence is a new, specific congruence called inp-bisimulation while for the core language it is the usual one.

Recently, full abstraction results for π-calculus with contextual equivalence [108] and for Java-like languages with testing equivalence have been obtained in [79] (by considering weak bisimulation) and in [78] (by using a model based on traces). Also in these cases the considered equivalences are different from ours.

We plan to extend our results by considering the full abstraction issue for the language Klaim [39], which is based on Linda and supports distribution and mobility.

Chapter 5

Multiplicity Matters

The inevitable end of multiple chiefs is that they fade and disappear for lack of unity.

Napoleon Bonaparte

In the previous chapter, one can notice that the expressiveness gap between the two languages analysed (the operators in Linda-inp permit to count the occurrences of the messages in the store while this is not possible in Linda-core) reflects in an easier fully abstract model. Here we want to generalise this observation by studying other languages where multiplicity matters. To this aim we consider Multiset Rewriting Systems, which are sets of rules that specify how to transform a multiset into another multiset. The key issue is to compare the expressiveness of dialects obtained by bounding the number of elements in the head of the rules. The language considered is CHR.

In this chapter, we first analyse the Turing completeness of the language then we show that despite the previous results, multiple heads do augment the expressive power of the language: we prove that, under certain reasonable assumptions, it is not possible to encode the CHR language (with multi-headed rules) into a single headed language while preserving the semantics of programs. tics of programs. Moreover we show that also the number of elements in the heads of rules matters, as the

CHR language allowing at most n elements in the heads is more expressive than the language allowing at most m elements, with $m < n$.

5.1 Introduction

Constraint Handling Rules (CHR) [59, 60] is a committed-choice declarative language which has been originally designed for writing constraint solvers and which is nowadays a general purpose language. A CHR program consists of a set of multi-headed guarded (simplification and propagation) rules which allow one to rewrite constraints into simpler ones until a solved form is reached. The language is parametric w.r.t. an underlying constraint theory CT which defines the meaning of basic built-in constraints.

The presence of multiple heads is a crucial feature which differentiates CHR from other existing committed choice (logic) languages. Many examples in the vast literature on CHR provide empirical evidence for the claim that such a feature is needed in order to obtain reasonably expressive constraint solvers in a reasonably simple way (see the discussion in [60]). However this claim was not supported by any formal result, so far.

In this chapter we prove that multiple heads do indeed augment the expressive power of CHR. Since we know that CHR is Turing powerful [121], we first show that CHR with single heads, called CHR_1 in what follows, is also Turing powerful, provided that the underlying constraint theory allows the equality predicate (interpreted as pattern matching) and that the signature contains at least one function symbol (of arity greater than zero). This result is certainly not surprising; however it is worth noting that, as we prove later, when considering an underlying (constraint theory defined over a) signature containing finitely many constant symbols and no function symbol CHR (with multiple heads) is still Turing complete, while this is not the case for CHR_1.

This provide a first separation result which is however rather weak, since usual constraint theories used in CHR do allow function symbols. Moreover computability

theory is not always the right framework for comparing the expressive power of concurrent languages, since often one has to compare languages which are Turing powerful (see also the discussion in Chapter 1).

Hence in the second part of the chapter we compare the expressive power of CHR and CHR_1 by using the notion of language encoding, first formalised in [38, 120, 124]. Intuitively, a language $\mathcal{L}$ is more expressive than a language $\mathcal{L}'$ or, equivalently, $\mathcal{L}'$ can be encoded in $\mathcal{L}$, if each program written in $\mathcal{L}'$ can be translated into an $\mathcal{L}$ program in such a way that: 1) the intended observable behaviour of the original program is preserved (under some suitable decoding); 2) the translation process satisfies some additional restrictions which indicate how easy this process is and how reasonable the decoding of the observables is. For example, typically one requires that the translation is compositional w.r.t. (some of) the syntactic operators of the language [38].

We prove that CHR cannot be encoded into CHR_1 under the following three assumptions. First we assume that the observable properties to be preserved are the constraints computed by a program for a goal, more precisely we consider data sufficient answers and qualified answers. Since these are the two typical CHR observables for most CHR reference semantics, assuming their preservation is rather natural. Secondly we require that both the source CHR language and the target CHR_1 share the same constraint theory defining built-in constraints. This is also a natural assumption, as CHR programs are usually written to define a new (user-defined) predicate in terms of the existing built-in constraints. Finally we assume that the translation of a goal is compositional w.r.t. conjunction of goals, that is, we assume that $[\![A, B]\!]_g = [\![A]\!]_g, [\![B]\!]_g$ for any conjunctive goal A, B, where $[\![\ ]\!]_g$ denotes the translation of a goal. We believe this notion of compositionality to be reasonable as well, since essentially it means that the translated program is not specifically designed for a single goal. It is worth noticing that we do not impose any restriction on the translation of the program rules.

From this main separation result follows that CHR cannot be encoded in (constraint) logic programs nor in pure Prolog. This does not conflict with the fact that

there exist many CHR to Prolog compilers: it simply means that these compilers do not satisfy our assumptions (typically, they do not translate goals in a compositional way).

Then we consider the class of CHR programs which have qualified answers and trivial data sufficient answers only. Roughly, data sufficient answers are the results of terminating computations which contain built-in constraints only, while qualified answers can contain also some user-defined constraints, that is, some constraints which are defined by the program rules (rather than by the underlying theory). Trivial data sufficient answers are those identical to the original goal. We show that the previous separation result holds also when considering this class of programs. This has some relevance because qualified answers are the main observable property considered in the CHR semantics, and programs having qualified answers only are common.

Next we show that also the number of atoms (greater than one) affects the expressive power of the language. In fact we prove that, under some slightly stronger assumptions on the translation of goals, there exists no encoding of CHR_n into CHR_m (CHR with at most n (m respectively) atoms in the head of the rules), for $m < n$.

The remainder of this chapter is organised as follows. Section 5.2 introduces the languages under consideration. We then provide the encoding of RAMs in CHR_1 and discuss the Turing completeness of this language in Section 5.3. Sections 5.4 and 5.5 contain the separation results while Section 5.6 concludes by discussing some related works. Some of the results shown in this chapter has previously appeared in [66].

5.2 Preliminaries

In this section we give an overview of CHR syntax and operational semantics following [60].

5.2.1 CHR constraints and notation

We first need to distinguish the constraints handled by an existing solver, called built-in (or predefined) constraints, from those defined by the CHR program, called user-defined (or CHR) constraints. Therefore we assume that the signature contains two disjoint sets of predicate symbols for built-in and CHR constraints.

Definition 5.1 (Built-in constraint) *A* built-in constraint c *is defined by:*

$$c ::= a \mid c \wedge c \mid \exists_x c$$

where a is an atomic built-in constraint (an atomic constraint is a first-order atomic formula). For built-in constraints we assume given a (first order) theory CT which describes their meaning.

Definition 5.2 (User-defined constraint) *A* user-defined (or CHR) constraint *is a conjunction of atomic user-defined constraints.*

We use c, d to denote built-in constraints, h, k to denote CHR constraints and a, b, f, g to denote both built-in and user-defined constraints (we will call these generally constraints). The capital versions of these notations will be used to denote multisets of constraints. We also denote by `false` any inconsistent conjunction of constraints and with `true` the empty multiset of built-in constraints.

We will use "," rather than $\wedge$ to denote conjunction and we will often consider a conjunction of atomic constraints as a multiset of atomic constraints: We prefer to use multisets rather than sequences (as in the original CHR papers) because our results do not depend on the order of atoms in the rules. In particular, we will use this notation based on multisets in the syntax of CHR.

The notation $\exists_V \phi$, where V is a set of variables, denotes the existential closure of a formula ϕ w.r.t. the variables in V, while the notation $\exists_{-V} \phi$ denotes the existential closure of a formula ϕ with the exception of the variables in V which remain unquantified. $Fv(\phi)$ denotes the free variables appearing in ϕ.

Moreover, if $\bar{t} = t_1, \ldots t_m$ and $\bar{t'} = t'_1, \ldots t'_m$ are sequences of terms then the notation $p(\bar{t}) = p'(\bar{t'})$ represents the set of equalities $t_1 = t'_1, \ldots, t_m = t'_m$ if $p = p'$, and it is undefined otherwise. This notation is extended in the obvious way to sequences of constraints.

5.2.2 Syntax

A CHR program is defined as a sequence of two kinds of rules: simplification and propagation (some papers consider also simpagation rules, since these are abbreviations for propagation and simplification rules we do not need to introduce them). Intuitively, simplification rewrites constraints into simpler ones, while propagation adds new constraints which are logically redundant but may trigger further simplifications.

Definition 5.3 *A CHR* simplification *rule has the form:*

$$r \ @ \ H \Leftrightarrow C \mid B$$

while a CHR propagation *rule has the form:*

$$r \ @ \ H \Rightarrow C \mid B,$$

where r is a unique identifier of a rule, H (the head) is a (non-empty) multiset of user-defined constraints, C (the guard) is a possibly empty multiset of built-in constraints and B is a possibly empty multiset of (built-in and user-defined) constraints.

A CHR program *is a finite set of CHR simplification and propagation rules.*

In the following when the guard is `true` we omit `true|`. Also the names of rules are omitted when not needed. A CHR *goal* is a multiset of (both user-defined and built-in) constraints. An example of a CHR Program is shown in next Section: Example 5.1.

Solve	$\dfrac{CT \models c \wedge d \leftrightarrow d' \text{ and c is a built-in constraint}}{\langle (c, G), K, d\rangle \longrightarrow \langle G, K, d'\rangle}$
Introduce	$\dfrac{\text{h is a user-defined constraint}}{\langle (h, G), K, d\rangle \longrightarrow \langle G, (h, K), d\rangle}$
Simplify	$\dfrac{H \Leftrightarrow C \mid B \in P \quad x = Fv(H) \quad CT \models d \rightarrow \exists_x((H = H') \wedge C)}{\langle G, H' \wedge K, d\rangle \longrightarrow \langle B \wedge G, K, H = H' \wedge d\rangle}$
Propagate	$\dfrac{H \Rightarrow C \mid B \in P \quad x = Fv(H) \quad CT \models d \rightarrow \exists_x((H = H') \wedge C)}{\langle G, H' \wedge K, d\rangle \longrightarrow \langle B \wedge G, H' \wedge K, H = H' \wedge d\rangle}$

Table 5.1: The standard transition system for CHR

5.2.3 Operational semantics

We describe now the operational semantics of CHR by slightly modifying the transition system defined in [60]. We use a transition system $T = (Conf, \longrightarrow)$ where configurations in $Conf$ are triples of the form $\langle G, K, d\rangle$, where G are the constraints that remain to be solved, K are the user-defined constraints that have been accumulated and d are the built-in constraints that have been simplified.

An *initial configuration* has the form $\langle G, \emptyset, \emptyset\rangle$ while a *final configuration* has either the form $\langle G, K, \texttt{false}\rangle$ when it is *failed*, or the form $\langle \emptyset, K, d\rangle$ when it is successfully terminated because there are no applicable rules.

Given a program P, the transition relation $\longrightarrow \subseteq Conf \times Conf$ is the least relation satisfying the rules in Table 5.1 (for the sake of simplicity, we omit indexing the relation with the name of the program). The **Solve** transition allows to update the constraint store by taking into account a built-in constraint contained in the goal. The **Introduce** transition is used to move a user-defined constraint from the goal to the CHR constraint store, where it can be handled by applying CHR rules.

The transitions **Simplify** and **Propagate** allow to rewrite user-defined con-

straints (which are in the CHR constraint store) by using rules from the program. As usual, in order to avoid variable name clashes, both these transitions assume that all variables appearing in a program clause are fresh ones. Both the **Simplify** and **Propagate** transitions are applicable when the current store (d) is strong enough to entail the guard of the rule (C), once the parameter passing has been performed (this is expressed by the equation $H = H'$). Note that, due to the existential quantification over the variables x appearing in H, in such a parameter passing the information flow is from the actual parameters (in H') to the formal parameters (in H), that is, it is required that the constraints H' which have to be rewritten are an instance of the head H[1]. The difference between **Simplify** and **Propagate** lies in the fact that while the former transition removes the constraints H' which have been rewritten from the CHR constraint store, this is not the case for the latter.

Given a goal G, the operational semantics that we consider observes the final stores of computations terminating with an empty goal and an empty user-defined constraint. Following the terminology of [60], we call such observables *data sufficient answers*.

Definition 5.4 *[Data sufficient answers [60]] Let P be a program and let G be a goal. The set $\mathcal{SA}_P(G)$ of data sufficient answers for the query G in the program P is defined as:*

$$\mathcal{SA}_P(G) = \{\exists_{-Fv(G)} d \mid \langle G, \emptyset, \emptyset\rangle \longrightarrow^* \langle \emptyset, \emptyset, d\rangle \not\longrightarrow\}.$$

We also consider the following different notion of answer, obtained by computations terminating with a user-defined constraint which does not need to be empty.

Definition 5.5 *[Qualified answers [60]] Let P be a program and let G be a goal. The set $\mathcal{QA}_P(G)$ of qualified answers for the query G in the program P is defined as:*

$$\mathcal{QA}_P(G) = \{\exists_{-Fv(G)} (K \wedge d) \mid \langle G, \emptyset, \emptyset\rangle \longrightarrow^* \langle \emptyset, K, d\rangle \not\longrightarrow\}.$$

[1]This means that the equations $H = H'$ express pattern matching rather than unification.

Both previous notions of observables characterise an input/output behaviour, since the input constraint is implicitly considered in the goal. Clearly in general $\mathcal{SA}_P(G) \subseteq \mathcal{QA}_P(G)$ holds, since data sufficient answers can be obtained by setting $K = \emptyset$ in Definition 5.5.

Note that in presence of propagation rules, the abstract (naive) operational semantics that we consider here introduces redundant infinite computations (because propagation rules do not remove user defined constraints). It is possible to define different operational semantics (see [2] and [47]) which avoids these infinite computations by allowing to apply at most once a propagation rule to the same constraints. The results presented here hold also in case these more refined semantics are considered, essentially because the number of applications of propagations rules does not matter. We refer here to the naive operational semantics because it is much simpler than those in [2] and [47].

An example can be useful to see what kind of programs we are considering here. The following program implements the sieve of Eratosthenes to compute primes.

Example 5.1 The following CHR program which consists of three simplifications rules, given a goal $upto(N)$ with N natural number, computes all prime numbers up to N: the first two rules generate all the possible candidates as prime numbers, while the third one removes all the incorrect information.

$$upto(1) \Leftrightarrow \texttt{true}$$
$$upto(N) \Leftrightarrow N > 1 \mid prime(N), upto(N-1)$$
$$prime(X), prime(Y) \Leftrightarrow X \texttt{ mod } Y = 0 \mid prime(Y)$$

For example suppose that the goal is $upto(4)$ then the following is a possible evolution of the program:

$$\langle upto(4), \emptyset, \emptyset \rangle \longrightarrow^* \langle \emptyset, (prime(4), prime(3), prime(2)), \emptyset \rangle$$

From the goal $upto(4)$ we, first, generate all possible candidates, then we apply the third rule that checks for every couple of constraints $prime(X), prime(Y)$ if X is

divisible by Y and in this case restores in the pool of constraints only the constraint $prime(Y)$ (said otherwise we remove the constraint $prime(X)$).

$$\langle \emptyset, (prime(4), prime(3), prime(2)), \emptyset \rangle \longrightarrow^* \langle \emptyset, (prime(3), prime(2)), \emptyset \rangle$$

Since there are no applicable rules $\langle \emptyset, (prime(3), prime(2)), \emptyset \rangle$ is a final configuration. Note that this is a qualified answer and the program with this goal has no data sufficient answers. □

In the following we study several CHR dialects defined by setting a limit in the number of the atoms present in the head of rules.

Definition 5.6 (CHR dialects) *CHR_n defines a CHR language where the number of atoms in the head of the rules is at most n.*

5.3 On the Turing completeness of CHR

In this section we discuss the Turing completeness of CHR_1 by taking into account also the underlying constraint theory. In order to show the Turing completeness of a language we encode RAMs (see Section 2.1) into it.

We first show that CHR_1 is Turing powerful, provided that the constraint theory allows the built-in $=$ (interpreted as pattern matching) and that the underlying signature contains at least a function symbol (of arity one) and a constant symbol. This result is obtained by providing an encoding $[\![\ ]\!] : Machines \rightarrow CHR$ of a RAM $M(v_0, v_1)$ in CHR as shown in Figure 5.1: Every rule takes as input the program counter and the two registers and updates the state according to the instruction in the obvious way. The variable X is used for outputting the result at the end. Note that, due to the pattern matching mechanism, a generic goal $i(p_i, s, t, X)$ can fire at most one of the two rules encoding the $DecJump$ instruction (in fact, if s is a free variable no rule in the encoding of $DecJump(r_1, p_l)$ is fired).

Without loss of generality we can assume that the counters are initialised with 0, hence the encoding of a machine M with n instructions has the form:

$$[\![M(0,0)]\!] := \{[\![Instruction_1]\!], \ldots, [\![Instruction_n]\!]\}$$

$$
\begin{array}{ll}
[\![p_i : Halt]\!] := & i(p_i, R_1, R_2, X) \Leftrightarrow X = R_1 \\
[\![p_i : Succ(r_1)]\!] := & i(p_i, R_1, R_2, X) \Leftrightarrow i(p_{i+1}, succ(R_1), R_2, X) \\
[\![p_i : Succ(r_2)]\!] := & i(p_i, R_1, R_2, X) \Leftrightarrow i(p_{i+1}, R_1, succ(R_2), X) \\
[\![p_i : DecJump(r_1, p_l)]\!] := & i(p_i, 0, R_2, X) \Leftrightarrow i(p_l, 0, R_2, X) \\
 & i(p_i, succ(R_1), R_2, X) \Leftrightarrow i(p_{i+1}, R_1, R_2, X) \\
[\![p_i : DecJump(r_2, p_l)]\!] := & i(p_i, R_1, 0, X) \Leftrightarrow i(p_l, R_1, 0, X) \\
 & i(p_i, R_1, succ(R_2), X) \Leftrightarrow i(p_{i+1}, R_1, R_2, X)
\end{array}
$$

Figure 5.1: RAM encoding in CHR_1

(note that the initial values of the register are not considered in the encoding of the machine: they will be used in the initial goal, as shown below). The following theorem states the correctness of the encoding. The proof is immediate.

Theorem 5.1 *A RAM $M(0,0)$ halts with output k if and only if the goal $i(1,0,0,X)$ in the program $[\![M(0,0)]\!]$ has a data sufficient answer $X = k$.*

It is worth noting that the presence of a function symbol ($succ()$ in our case) is crucial in order to encode natural numbers and therefore to obtain the above result. Indeed, as we prove below, when considering a signature containing only a finite number of constant symbols the language CHR_1, differently from the case of CHR, is not Turing powerful. To be more precise, assume that CT defines only the $=$ symbol (to be interpreted as pattern matching, as in the previous case) and assume that such a theory is defined over a signature containing finitely many constant symbols and no function symbol (of arity > 0). Let us call $CT_\emptyset$ the resulting theory.

We first observe that, when considering $CT_\emptyset$, CHR_1 is computationally equivalent to ground CHR_1 w.r.t. termination, where ground CHR_1 indicates the CHR_1 language obtained by considering only ground rules and goals. Here and in the following $Ground(E)$ denotes the set all ground instances of the syntactic expression E. Obviously a ground CHR_1 program can be seen as a propositional program, that is, a program where each predicate symbol has arity 0 and where guards are not

present (they are always satisfied or falsified, of course). More precisely, we have the following lemma.

Lemma 5.1 *Let P be a CHR_1 program on $CT_\emptyset$ and G be a goal. Then the existence of a terminating computation for G in P can be decided iff it can be decided the existence of a terminating computation for a ground goal $G' \in Ground(G)$ in the ground program $Ground(P)$.*

Proof: Immediate, by observing that for each (finite) derivation for G in P which uses the rules $r_1, \ldots, r_n$ we can use suitable ground instances of $r_1, \ldots, r_n$ to obtain a derivation for a suitable ground instance G' of G. The thesis then follows from the fact that, due to our assumption on $CT_\emptyset$, both $Ground(P)$ and $Ground(G)$ are finite sets. □

Next we show that ground CHR_1 is computationally equivalent to P/T nets (see Section 2.3).

Lemma 5.2 *Ground CHR_1 is computationally equivalent to P/T nets w.r.t. termination.*

Proof: Let P be a ground CHR_1 program, we can build the corresponding P/T net in the following way: the set of places corresponds to the set of ground atomic CHR_1 constraints, the set of transitions correspond to the set of rules, and given a rule

$$\texttt{rule} \ @ \ H \Leftrightarrow B$$

the corresponding transition takes as input the constraints in H and the output corresponds to the constraints in B (note that we can assume that guards are missing, since they are either true or false). The initial marking is built by placing a token to the place correspondent to the ground atomic CHR_1 constraints present in the ground goal G. It is easy, then, to prove that this construction is correct. In particular, G has a terminating computation in P iff the computation for the initial marking corresponding to G terminates in the net corresponding to P. □

Now we are ready to state the desired result.

$$[\![p_i : Halt]\!]_2 := i(p_i, R_1, R_2, X) \Leftrightarrow X = R_1$$

$$[\![p_i : Succ(r_1)]\!]_2 := i(p_i, R_1, R_2, X) \Leftrightarrow s(R_1, SuccR_1), i(p_{i+1}, SuccR_1, R_2, X)$$

$$[\![p_i : Succ(r_2)]\!]_2 := i(p_i, R_1, R_2, X) \Leftrightarrow s(R_2, SuccR_2), i(p_{i+1}, R_1, SuccR_2, X)$$

$$[\![p_i : DecJump(r_1, p_l)]\!]_2 :=$$

$$i(p_i, R_1, R_2, X), s(PreR_1, R_1) \Leftrightarrow i(p_{i+1}, PreR_1, R_2, X)$$

$$zero(R_1), i(p_i, R_1, R_2, X) \Leftrightarrow i(p_l, R_1, R_2, X), zero(R_1)$$

$$[\![p_i : DecJump(r_2, p_l)]\!]_2 :=$$

$$i(p_i, R_1, R_2, X), s(PreR_2, R_2) \Leftrightarrow i(p_{i+1}, R_1, PreR_2, X)$$

$$zero(R_2), i(p_i, R_1, R_2, X) \Leftrightarrow i(p_l, R_1, R_2, X), zero(R_2)$$

Figure 5.2: RAM encoding in CHR on $CT_\emptyset$

Theorem 5.2 *CHR_1 on $CT_\emptyset$ is not Turing complete.*

Proof: Immediate from the two previous lemmata by observing that in P/T Nets termination is decidable. □

On the other hand, CHR (with multiple heads) is still Turing powerful also when considering the theory $CT_\emptyset$. Indeed, as we show in Figure 5.2, we can encode RAMs into CHR (defined on $CT_\emptyset$). The basic idea here is that to encode the values of the registers we use chains (conjunctions) of atomic formulas of the form $s(R_1, SuccR_1)$, $s(SuccR_1, SuccR_1') \ldots$ (recall that R_1, $SuccR_1$, $SuccR_1'$ are variables and we have countably many variables; moreover recall that the CHR computation mechanism avoid variables capture by using fresh names for variables each time a rule is used).

It is also worth noting that for the correctness of the encoding it is essential that pattern matching rather than unification is used when applying rules (this ensures that in the case of the decrement only one of the two instructions can match the goal and therefore can be used). The correctness of the encoding is stated by the following theorem whose proof is immediate.

Theorem 5.3 *A RAM $M(0,0)$ halts with output $k > 0$ (or $k = 0$) if and only if the goal $zero(R_1) \wedge zero(R_2) \wedge i(1, R_1, R_2, X)$ in the program $[\![M(0,0)]\!]_2$ produces a*

qualified answer

$$\exists_{-X,R_1}(X = R_1 \ \wedge \ s(R_1, SuccR_1^1) \bigwedge_{i=1\ldots k-1} (SuccR_1^i, SuccR_1^{i+1}))$$

(or $\exists_{-X,R_1}(x = R_1 \wedge zero(R_1))$*).*

Previous theorems state a separation result between CHR and CHR_1, however this is rather weak since the real implementations of CHR usually consider a non-trivial constraint theory which includes function symbols. Therefore we are interested in proving finer separation results which hold for Turing powerful languages. This is the content of the following section.

5.4 Separating CHR and CHR_1

In this section we consider a generic non-trivial constraint theory CT. We have seen that in this case both CHR and CHR_1 are Turing powerful, which means that in principle one can always encode CHR into CHR_1. The question is how difficult and how acceptable such an encoding is and this question can have important practical consequences: for example, a distributed algorithm can be implemented in one language in a reasonably simple way and cannot be implemented in another (Turing powerful) language, unless one introduces rather complicated data structures or loses some compositionality properties (see [125]).

We prove now that, when considering *acceptable encodings* and generic goals whose components can share variables, CHR cannot be embedded into CHR_1 while preserving data sufficient answers. As a corollary we obtain that also qualified answers cannot be preserved.

First we have to formally define what an acceptable encoding is. We define a *program encoding* as any function $[\![\ ]\!] : \mathcal{P}_{CHR} \rightarrow \mathcal{P}_{CHR_1}$ which translates a CHR program into a (finite) CHR_1 program ($\mathcal{P}_{CHR}$ and $\mathcal{P}_{CHR_1}$ denote the set of CHR and CHR_1 programs, respectively). To simplify the treatment we assume that both the source language CHR and the target language CHR_1 use the same built-in

constraints semantically described by a theory CT (actually this assumption could be relaxed). Note that we do not impose any other restriction on the program translation (which, in particular, could also be non compositional).

Next we have to define how the initial goal of the source program has to be translated into the target language. Here we require that the translation is compositional w.r.t. the conjunction of atoms, as mentioned in the introduction. Moreover since both CHR and CHR_1 share the same CT we assume that the built-ins present in the goal are left unchanged. These assumptions essentially mean that our encoding respects the structure of the original goal and does not introduce new relations among the variables which appear in the goal. Finally, as mentioned before, we are interested in preserving data sufficient and qualified answers. Hence we have the following definition where we denote by $\mathcal{G}_{CHR}$ and $\mathcal{G}_{CHR_1}$ the class of CHR and CHR_1 goals, respectively (we differentiate these two classes because, for example, a CHR_1 goal could use some user defined predicates which are not allowed in the goals of the original program[2]). Note that the following definition is parametric w.r.t. a class $\mathcal{G}$ of goals: clearly considering different classes of goals could affect our encodability results. Such a parameter will be instantiated when the notion of acceptable encoding will be used.

Definition 5.7 (Acceptable encoding) *Let $\mathcal{G}$ be a class of CHR goals. An* acceptable encoding *(of CHR into CHR_1, for the class of goals $\mathcal{G}$) is a pair of mappings $[\![\ ]\!] : \mathcal{P}_{CHR} \to \mathcal{P}_{CHR_1}$ and $[\![\ ]\!]_g : \mathcal{G}_{CHR} \to \mathcal{G}_{CHR_1}$ which satisfy the following conditions:*

- *$\mathcal{P}_{CHR}$ and $\mathcal{P}_{CHR_1}$ share the same CT;*
- *for any goal $(A, B) \in \mathcal{G}_{CHR}$, $[\![A, B]\!]_g = [\![A]\!]_g, [\![B]\!]_g$ holds. We also assume that the built-ins present in the goal are left unchanged;*

[2]This means that in principle the signatures of (language of) the original and the translated program are different.

- *Data sufficient (qualified) answers are preserved for the class of goals $\mathcal{G}$, that is, for all $G \in \mathcal{G} \subseteq \mathcal{G}_{CHR}$, $\mathcal{SA}_P(G) = \mathcal{SA}_{[\![P]\!]}([\![G]\!]_g)$ ($\mathcal{QA}_P(G) = \mathcal{QA}_{[\![P]\!]}([\![G]\!]_g)$) holds.*

Note that, since we consider goals as multisets, with the second condition here we are not requiring that the order of atoms in the goals is preserved by the translation: We are only requiring that the translation of A, B is the conjunction of the translation of A and of B. Weakening this condition by requiring that the translation of A, B is some form of composition of the translation of A and of B does not seem reasonable, as conjunction is the only form for goal composition available in these languages.

We are now ready to prove our separation results, next section considers only data sufficient answers.

5.4.1 Separating CHR and CHR_1 by considering data sufficient answers

In order to prove our first separation result we need the following lemma which states a key property of CHR_1 computations. Essentially it says that if the conjunctive G, H with input constraint c produces a data sufficient answer d, then when considering one component, say G, with the input constraint d we obtain the same data sufficient answer. Moreover the same answer can be obtained, either for G or for H, also starting with an input constraint c' weaker than d.

Lemma 5.3 *Let P be a CHR_1 program and let (c, G, H) be a goal, where c is a built-in constraint, G and H are multisets of CHR constraints and $V = Fv(c, G, H)$. Assume that (c, G, H) in P has the data sufficient answer d. Then the following holds:*

- *Both the goals (d, G) and (d, H) have the same data sufficient answer d.*
- *If $CT \models c \not\rightarrow d$ then there exists a built-in constraint c' such that $Fv(c') \subseteq V$, $CT \models c' \not\rightarrow d$ and either (c', G) or (c', H) has the data sufficient answer d.*

Proof: The proof of the first statement is straightforward (since we consider single headed programs). In fact, since the goal (c, G, H) has the data sufficient answer d in P, the goal (d, G) can either answer d or can produce a configuration where the user defined constraints are waiting for some guards to be satisfied in order to apply a rule r, but since the goal contains all the built-in constraints in the answer all the guards are satisfied letting the program to answer d.

We prove the second statement. Let

$$\delta = \langle (c, G, H), \emptyset, \emptyset \rangle \longrightarrow^* \langle \emptyset, \emptyset, d' \rangle \not\longrightarrow$$

be the derivation producing the data sufficient answer $d = \exists_{-V} d'$ for the goal (c, G, H).

By definition of derivation and since by hypothesis $CT \models c \not\rightarrow d$, δ must be of the form

$$\langle (c, G, H), \emptyset, \emptyset \rangle \longrightarrow^* \langle (c_1, G_1), S_1, d_1 \rangle \longrightarrow \langle (c_2, G_2), S_2, d_2 \rangle \longrightarrow^* \langle \emptyset, \emptyset, d' \rangle \not\twoheadrightarrow,$$

where for $i \in [1, 2]$, c_i and d_i are built-in constraints such that $CT \models c_1 \wedge d_1 \not\rightarrow d$ and $CT \models c_2 \wedge d_2 \rightarrow d$. We choose $c' = \exists_{-V}(c_1 \wedge d_1)$. By definition of derivation and since P is a CHR_1 program, the transition

$$\langle (c_1, G_1), S_1, d_1 \rangle \longrightarrow \langle (c_2, G_2), S_2, d_2 \rangle$$

must be a rule application of a single headed rule r, which must match with a constraint k that was derived (in the obvious sense) by either G or H. Without loss of generality, we can assume that k was derived from G. By construction c' suffices to satisfy the guards needed to reproduce k, which can then fire the rule r, after which all the rules needed to let the constraints of G disappear can fire. Therefore we have that

$$\langle (c', G), \emptyset, \emptyset \rangle \longrightarrow^* \langle \emptyset, \emptyset, d'' \rangle \not\twoheadrightarrow$$

where $CT \models \exists_{-V} d'' \leftrightarrow \exists_{-V} d' (\leftrightarrow d)$ and then the thesis. □

Note that Lemma 5.3 is not true anymore if we consider (multiple headed) CHR programs. Indeed if we consider the program P consisting of the single rule

$$\texttt{rule} \ @ \ H, H \Leftrightarrow \texttt{true} \mid c$$

then the goal (H, H) has the data sufficient answer c in P, but for each constraint c' the goal (H, c') has no data sufficient answer in P. With the help of the previous lemma we can now prove our main separation result. The idea of the proof is that any possible encoding of the rule

$$\mathtt{r} \ @ \ H, G \Leftrightarrow \mathtt{true} \mid c$$

into CHR_1 would either produce more answers for the goal H (or G), or would not be able to provide the answer c for the goal H, G.

Theorem 5.4 *Let $\mathcal{G}$ be a class of goals such that if H is an head of a rule then $H \in \mathcal{G}$. When considering data sufficient or qualified answers, there exists no acceptable encoding of CHR in CHR_1 for the class $\mathcal{G}$.*

Proof: We first consider data sufficient answers. The proof is by contradiction. Consider the program P consisting of the single rule

$$\mathtt{r} \ @ \ H, G \Leftrightarrow \mathtt{true} \mid c$$

and assume that $[\![P]\!]$ is the translation of P in CHR_1. Assume also that c (restricted to the variables in H, G) is not the weakest constraint, i.e. assume that there exist d such that $CT \models d \not\rightarrow \exists_{-V} c$ where $V = Fv(H, G)$. Note that this assumption does not imply any loss of generality, as we consider non trivial constraint systems containing at least two different constraints.

Since the goal (H, G) has the data sufficient answer $\exists_{-V} c$ in the program P and since the encoding preserves data sufficient answers the goal $[\![(H, G)]\!]_g$ has the data sufficient answer $\exists_{-V} c$ also in the program $[\![P]\!]$. From the compositionality of the translation of goals and the previous Lemma 5.3 it follows that there exists a constraint c' such that $Fv(c') \subseteq V$, $CT \models c' \not\rightarrow \exists_{-V} c$ and either the goal $[\![(c', H)]\!]_g$, or the goal $[\![(c', G)]\!]_g$ has the data sufficient answer c in the encoded program $[\![P]\!]$. However neither (c', H) nor (c', G) has any data sufficient answer in the original program P. This contradicts the fact that $[\![P]\!]$ is an acceptable encoding for P, thus concluding the proof for data sufficient answers. The thesis for qualified answers

$\texttt{reflexivity}$ @ $Lessequal(X,Y) \Leftrightarrow X = Y \mid \texttt{true}$

$\texttt{antisymmetry}$ @ $Lessequal(X,Y), Lessequal(Y,X) \Leftrightarrow X = Y$

$\texttt{transitivity}$ @ $Lessequal(X,Y), Lessequal(Y,Z) \Rightarrow Lessequal(X,Z)$

Figure 5.3: A program for defining $\leq$ in CHR

follows immediately from the previous part, as qualified answers contain the set of data sufficient answers. □

The hypothesis made on the class of goals $\mathcal{G}$ is rather weak, as typically heads of rules have to be used as goals. As an example of the application of the previous theorem consider the program (from [60]) contained in Figure 5.3 which allows one to define the user-defined constraint *Lessequal* (to be interpreted as $\leq$) in terms of the only built-in constraint $=$ (to be interpreted as syntactic equality). For example, given the goal $\{Lessequal(A,B), Lessequal(B,C), Lessequal(C,A)\}$ after a few computational steps the program will answer $A = B, B = C, C = A$. Now for obtaining this behaviour it is essential to use multiple heads, as already claimed in [60] and formally proved by previous theorem. In fact, following the lines of the proof of Theorem 5.4, one can show that if a single headed program P' is any translation of the program in Figure 5.3 which produces the correct answer for the goals above, then there exists a subgoal which has an answer in P' but not in the original program.

5.4.2 Separating CHR and CHR_1 by considering qualified answers

The proof of theorem 5.4 cannot be used for programs which have qualified answers and trivial data sufficient answers only (trivial answers are those identical to the goal). Nevertheless, since qualified answers are the most interesting ones for CHR programs, one could wonder what happens when considering these programs.

Here we prove that also when considering this class of programs CHR cannot be

encoded into CHR_1. Actually, the proof of this result is somehow easier to obtain since the multiplicity of atomic formulas here is important. In fact, if $u(x, y)$ is a user-defined constraint, the meaning of $u(x, y)$, $u(x, y)$ does not necessarily coincide with that one of $u(x, y)$. This is well known also in the case of logic programs (see any article on the S-semantics of logic programs): consider, for example, the program:

$$u(x, y) \Leftrightarrow x = a \qquad u(x, y) \Leftrightarrow y = b$$

which is essentially a pure logic program written with the CHR syntax. Notice that when considering an abstract operational semantics, as the one that we consider here, the presence of commit-choice does not affect the possible results. For example, in the previous program when reducing the goal $u(x, y)$ one can always choose (non deterministically) either the first or the second rule.

Now the goal $u(x, y), u(x, y)$ in such a program can have the (data sufficient) answer $x = a, y = b$ while this is not the case for the goal $u(x, y)$ which has either the answer $x = a$ or the answer $y = b$ (of course, using guards one can make more significative examples). Thus, when considering user-defined predicates, it is acceptable to distinguish $u(x, y), u(x, y)$ from $u(x, y)$, i.e. to take into account the multiplicity. This is not the case for "pure" built-in constraints, since the meaning of a (pure) built-in is defined by a first order theory CT in terms of logical consequences, and from this point of view $b \wedge b$ is equivalent to b.

In order to prove our result we need first the following straightforward Lemma which states that, when considering single headed rules, if the goal is replicated then there exists a computation where at every step a rule is applied twice. Hence it is easy to observe that if the computation will terminate producing a qualified answer which contains a user-defined constraint, then such a constraint is replicated.

Lemma 5.4 *Let P be a CHR_1 program. If (G, G) is a goal whose evaluation in P produces a qualified answer (c, H) containing the atomic user-defined constraint k, then the goal (c, G, G) has a qualified answer containing (k, k).*

Hence we can prove the following separation result.

Theorem 5.5 *Let $\mathcal{G}$ be the class of all possible goals and let us restrict to a class of CHR programs which, for any goal G, have no data sufficient answers different from G. When considering qualified answers there exists no acceptable encoding of CHR into CHR_1 for the class $\mathcal{G}$.*

Proof: The proof will proceed by contradiction. Let P be a program consisting of a single rule:

$$\texttt{r} \ @ \ H, H \Leftrightarrow \texttt{true} \mid k$$

where k is an atomic user-defined constraint. The goal (H, H) in P has a qualified answer k (note that P has no data sufficient answer).

Suppose that there exists an acceptable encoding of P. Hence the goal $[\![(H, H)]\!]_g$ in $[\![P]\!]$ has a qualified answer k (with the built-in constraint `true`). Since the compositionality hypothesis imply that $[\![(H, H)]\!]_g = [\![H]\!]_g, [\![H]\!]_g$, from Lemma 5.4 it follows that $[\![(H, H)]\!]_g$ in program $[\![P]\!]$ has also a qualified answer (k, k), but this answer cannot be obtained in the program with multiple heads thus contradicting one of the hypothesis on the encoding. Therefore such an encoding cannot exist. □

5.4.3 Separation result for weak acceptable encodings

Given previous negative results we are now interested in seeing whether CHR can be encoded in CHR_1 under a weaker preservation of answers, as specified in Definition 5.8. That is, we intend to see whether we can obtain an encoded program which for some (translated) goals possibly computes a superset of the answers obtained in the original program by the original goals.

Definition 5.8 *Let $\mathcal{G}$ be a class of CHR goals. An* acceptable encoding *(of CHR into CHR_1, for the class of goals $\mathcal{G}$) is a pair of mappings $[\![\ ]\!] : \mathcal{P}_{CHR} \rightarrow \mathcal{P}_{CHR_1}$ and $[\![\ ]\!]_g : \mathcal{G}_{CHR} \rightarrow \mathcal{G}_{CHR_1}$ which satisfy the following conditions:*

- *$\mathcal{P}_{CHR}$ and $\mathcal{P}_{CHR_1}$ share the same CT;*
- *for any goal $(A, B) \in \mathcal{G}_{CHR}$, $[\![A, B]\!]_g = [\![A]\!]_g, [\![B]\!]_g$ holds. We also assume that the built-ins present in the goal are left unchanged;*

- *Data sufficient (qualified) answers are weakly preserved for the class of goals $\mathcal{G}$, that is, for all $G \in \mathcal{G}$, $\mathcal{SA}_P(G) \subseteq \mathcal{SA}_{[\![P]\!]}([\![G]\!]_g)$ ($\mathcal{QA}_P(G) \subseteq \mathcal{QA}_{[\![P]\!]}([\![G]\!]_g)$) holds.*

Note that with weakly acceptable encodings we relax the condition on preservation of data sufficient (qualified) answers and we admit translations of programs and goals which compute a superset of the answers of the original program and goal. This point deserves some further explanation: in some cases it could happen that the translated program computes the same answers of the original program on some goals, while on some other goals it computes more answers. For example, consider the case of a CHR program P which is intended to be used with a conjunctive goal A, B: in case we evaluate an atomic goal, say A, in P, typically one get no (significant) answer, because of multiple heads. On the other hand, in the translated program $[\![P]\!]$ the goal $[\![A]\!]_g$ could produce some answers. These however are not interesting for our purposes, since the intended goals for the translated programs are of the form $[\![A]\!]_g, [\![B]\!]_g$. Of course, the specific class of interesting goals depends on the program and therefore it is difficult to formalise the problem in general. Hence in this case we could be interested in weakly acceptable encodings.

Here we show that also in this weaker sense CHR cannot be embedded into CHR_1. First of all we need to define more precisely what it means for a goals to be share-free:

Definition 5.9 *Consider a goal (c, H, G) where c is a built-in constraint and H and G are (multisets of) user defined predicates. Such a goal is called share-free iff*

- *H and G do not share variables;*
- *$c = c_1 \wedge c_2$ and $CT \models c \leftrightarrow \exists_{-Fv(H)} c_1 \wedge \exists_{-Fv(G)} c_2$.*

If a goal is not share-free it is called sharing goal.

This separation result uses the property of CHR_1 derivations stated by the following lemma. Here and in the following, when we say that a constraint c does not affect the variables V, formally we mean that $CT \models (\exists_V c) \leftrightarrow c$.

Lemma 5.5 *Let P be a CHR$_1$ program and let (G, H) be a share-free goal, where G and H contain user defined predicates (i.e. CHR constraints). Assume that (G, H) in P has the data sufficient answer d. Then the following holds:*

- *for any data sufficient answer c of G in P, c does not affect the variables in H;*
- *for any data sufficient answer c' of H in P, c' does not affect the variables in G;*
- *G in P has the data sufficient answer c and H in P has the data sufficient answer c' such that $CT \models c \wedge c' \leftrightarrow d$ holds.*

Proof: Straightforward by observing that in CHR$_1$ (and in CHR) computations all rules applied are renamed in order to avoid variable clashes. Then, since G and H do not share any variables and since CHR$_1$ heads are singletons, it follows that any rule applied to G and to the goals derived (in the obvious sense) from G cannot affect the variables in H and vice versa. This implies the thesis. □

We have then the following result .

Theorem 5.6 *Let $\mathcal{G}$ be the class of share-free goals and assume that we consider data sufficient answers. Moreover assume that, for any pair of goals A and B, if A and B do not share variables then $[\![A]\!]_g$ and $[\![B]\!]_g$ do not share variables. Then there exists no weak acceptable encoding of CHR into CHR$_1$ for the class $\mathcal{G}$.*

Proof: Let us consider the program P consisting of a single rule

$$\mathtt{r} \ @ \ H(x), I(y) \Leftrightarrow \mathtt{true} \mid b(x, y)$$

where b is any built-in predicate that relate x and y, such that

$$CT \models (\exists_{-x} b(x, y) \wedge \exists_{-y} b(x, y)) \not\leftrightarrow b(x, y)$$

As a mean of contradiction suppose that there exists a weak acceptable encoding $[\![P]\!]$ of such a program.

Consider the initial goal $(H(x), I(y))$ which has the data sufficient answer $b(x, y)$ in P. Now consider the encoded program $[\![P]\!]$ and the encoded goal $[\![(H(x), I(y))]\!]_g$. By definition of weak acceptable encoding it follows that $[\![(H(x),\ I(y))]\!]_g$ in $[\![P]\!]$ produces the data sufficient answer $b(x, y)$.

From Lemma 5.5 and the hypothesis on $[\![\ ]\!]_g$ (see Definition 5.7) it follows that there exists a data sufficient answer c for the goal $[\![H(x)]\!]_g$ in $[\![P]\!]$ and d for the goal $[\![I(y)]\!]_g$ in $[\![P]\!]$ such that c does not affect the variable y, d does not affect the variable x and $CT \models c \wedge d \leftrightarrow b(x, y)$ holds. However this is a contradiction, since $b(x, y)$ is a predicate relating x and y, hence it cannot be equivalent (in CT) to a conjunction $c \wedge d$ where c does not affect y and d does not affect x. This contradicts the existence of a weak acceptable encoding thus concluding the proof. □

Note that in the previous theorem we assume that the translation (of goals) does not introduce capture of variables. This ensure us that the translation process treats correctly the variable names: of course, these can be changed in the translation, however the identification of variables having different names in different goals has to be avoided.

We do not know whether Theorem 5.6 (and therefore previous corollary) holds when considering the class of sharing goals only. However, since share-free goals are a subset of all possible goals, obviously in case of weakly acceptable encodings previous theorem can be stated by considering the class of all possible goals. Hence we have the following obvious corollary.

Corollary 5.1 *Let $\mathcal{G}$ be the class of all possible goals. When considering data sufficient answers or qualified answers there exists no weakly acceptable encoding of CHR in CHR_1 for the class $\mathcal{G}$.*

5.4.4 A note on logic programs and Prolog

(Constraint) Logic programming and Prolog are programming languages quite different from CHR, mainly because they are sequential ones, without any guard mechanism and commit operator. Nevertheless, since many CHR implementations are

built on top of a Prolog system, by using a compiler which translates CHR programs to Prolog, it is meaningful to compare these sequential languages with CHR.

Note that here, following the general terminology (see for example [5]), a (constraint) logic program is a set of (definite) clauses, to be interpreted operationally in terms of SLD-resolution, thus using a non deterministic computational model. Real logic programming systems eliminate such a non determinism by choosing a specific selection rule (for selecting the atom in the goals to be evaluated) and a specific rule for searching the SLD-tree.

Following [5] we call pure Prolog a logic programming language which uses the leftmost selection rule and the depth-first search (this corresponds to consider clauses top-down, according to the textual ordering in the program). Implemented Prolog systems are extensions of pure Prolog obtained by considering specific built-ins for arithmetic, control etc. Some of these built-ins have a non logical nature, which complicates their semantics.

All our technical lemmata about CHR_1 can be stated also for (constraint) logic programming and pure Prolog (the proofs are similar, modulo some minor adjustments). Hence our separation results hold also when considering these languages rather than CHR_1. These can be summarised as follows.

Corollary 5.2 *Let $\mathcal{G}$ be a class of goals such that if H is an head of a rule then $H \in \mathcal{G}$. When considering data sufficient answers or qualified answers there exists no acceptable encoding of CHR in constraint logic programming nor in pure Prolog for the class $\mathcal{G}$.*

As mentioned in the introduction, previous result does not conflict with the fact that there exist many CHR to Prolog compilers: it simply means that, when considering pure Prolog, these compilers do not satisfy our assumptions (typically, they do not translate goals in a compositional way). Moreover real Prolog systems use several non logical built-in's, which are out of the scope of previous results.

5.5 A hierarchy of languages

After having shown that multiple heads increase the expressive power w.r.t. the case of single heads, it is natural to ask whether considering a different number of atoms in the heads makes any difference. In this section we show that this is indeed the case, since we prove that, for any $n > 1$, there exists no encoding of CHR_{n+1} into CHR_n. Thus, depending on the number of atoms in the heads, we obtain a chain of languages with increasing expressive power.

In order to obtain this generalisation we need to strengthen the requirement on acceptable encodings given in Definition 5.7. More precisely, we now require that, for any goal G, the translation $[\![G]\!]_g$ is the identity function. This accounts for a "black box" use of the program: we do not impose any restriction on the program encoding, provided that the interface remains unchanged. We have then the following result.

Theorem 5.7 *Let $\mathcal{G}$ be the class of all possible goals and assume that the goal translation is the identity. When considering data sufficient answers there exists no acceptable encoding of CHR_{n+1} in CHR_n for the class $\mathcal{G}$.*

Proof: Let P be the following CHR_{n+1} program:

$$\texttt{rule} \ @ \ h_1 \dots h_{n+1} \Leftrightarrow \texttt{true} \mid d$$

where d is a built-in constraint such that $Fv(d) \subseteq V$ and $CT \models d \not\leftrightarrow \texttt{false}$, where $V = Fv(h_1 \dots h_{n+1})$. Hence given the goal $G = h_1 \dots h_{n+1}$ the program P has the data sufficient answer d.

Observe that every goal with at most n user defined constraints has no data sufficient answer in P. Now as a mean of contraction let $[\![P]\!]$ be a suitable encoding in CHR_n and consider a run of G in $[\![P]\!]$ with final configuration $\langle \emptyset, \emptyset, d' \rangle$, where $CT \models \exists_{-V}(d') \leftrightarrow d$:

$$\delta = \langle G, \emptyset, \emptyset \rangle \rightarrow^* \langle H_i, G_i, d_i \rangle \rightarrow \langle H_{i+1}, G_{i+1}, d_{i+1} \rangle \rightarrow^* \langle \emptyset, \emptyset, d' \rangle,$$

where, without loss of generality, we can assume that in the derivation δ, for any configuration $\langle G', K', c' \rangle$ we can use either a Simplify or a Propagate transition only

if G' does not contain built-ins and G_i is the last goal to be reduced in the run by using either a Simplify or a Propagate transition, therefore G_i has at most n user-defined constraints, $H_i = \emptyset$ and there is such a rule $r \in [\![P]\!]$ such that r is the last rule used in δ. Since d is a built-in constraint, r can be of the following form $H \Leftrightarrow C \mid C'$. In this case $H_i = G_{i+1} = \emptyset$, H_{i+1} contains only built-in predicates, G_i and H have at most n user defined constraints. Then

$$CT \models d_i \rightarrow \exists_{Fv(H)}((G_i = H) \wedge C)$$

$$CT \models (d_i \wedge C' \wedge (G_i = H)) \not\leftrightarrow \texttt{false}$$

and $CT \models d \leftrightarrow \exists_{-V}(d_i \wedge C' \wedge (G_i = H))$.

By construction the goal (G_i, d_i) has the data sufficient $\exists_{-Fv(G_i,d_i)}(d_i \wedge C' \wedge (G_i = H))$ in $[\![P]\!]$.

But the goal (G_i, d_i) has no data sufficient answer in P thus contradicting the fact that $[\![P]\!]$ was an acceptable encoding for P. □

Similarly as before, we generalise the previous theorem to the case where the program has qualified answers and trivial data sufficient answers only.

Theorem 5.8 *Let $\mathcal{G}$ be the class of all possible goals (and let us restrict to a class of CHR programs which, for any goal G, have no qualified answers different from G). When considering qualified answers there exists no acceptable encoding of CHR_{n+1} in CHR_n for the class $\mathcal{G}$.*

Proof: Let P be the following CHR_{n+1} program:

$$\texttt{rule} \ @ \ h_1 \ldots h_{n+1} \Leftrightarrow \mathit{true} \mid k$$

where k is an atomic user defined constraint such that $Fv(k) \subseteq V$, where $V = Fv(h_1 \ldots h_{n+1})$. Hence given the goal $G = h_1 \ldots h_{n+1}$ the program P has only the qualified answer k and since k is an atomic user defined constraint, we have that $k \neq (h_1 \ldots h_{n+1})$.

Observe that every goal with at most n user defined constraints has only itself as qualified answer in P.

Now as a mean of contraction let $[\![P]\!]$ be a suitable encoding in CHR_n then since the encoded program has to preserve all the answers in the original P, every ground goal $[\![G_n]\!]_G$ with at most n user defined constraints has a qualified answer G_n in $[\![P]\!]$.

Therefore, if we denote by $G_n = h_1 \dots h_n$, by previous observation and by definition of qualified answers, we have that there exists two derivations

$$\langle [\![G_n]\!]_G, \emptyset, \emptyset \rangle \rightarrow^* \langle \emptyset, G'_n, d \rangle \not\rightarrow$$

and

$$\langle [\![h_{n+1}]\!]_G, \emptyset, \emptyset \rangle \rightarrow^* \langle \emptyset, h'_{n+1}, d' \rangle \not\rightarrow,$$

such that $CT \models G_n \leftrightarrow \exists_{-Fv([\![G_n]\!]_G)}(G'_n \wedge d)$ and $CT \models h_{n+1} \leftrightarrow \exists_{-Fv([\![h_{n+1}]\!]_G)}(h'_{n+1} \wedge d')$.

Without loss of generality, we can assume that

$$Fv(G'_n, d) \cap Fv(h'_{n+1}, d') \subseteq Fv([\![G_n]\!]_G) \cap Fv([\![h_{n+1}]\!]_G).$$

Now consider the goal G, from what previously said we have that:

$$\langle [\![G]\!]_G, \emptyset, \emptyset \rangle \rightarrow^* \langle [\![h_{n+1}]\!]_G, G'_n, d \rangle$$

but we also know that $\langle [\![h_{n+1}]\!]_G, \emptyset, \emptyset \rangle \rightarrow^* \langle \emptyset, h'_{n+1}, d' \rangle \not\rightarrow$ and this cannot be prevented by any step in the previous run, thus we obtain:

$$[\![G]\!]_G \rightarrow \langle \emptyset, (G'_n, h'_{n+1}), d \wedge d' \rangle,$$

where $CT \models G \leftrightarrow \exists_{-Fv([\![G]\!]_G)}(G'_n \wedge h'_{n+1} \wedge d \wedge d')$. Since G is not a qualified answer for the goal G in P and since $[\![P]\!]$ is a suitable encoding of P in CHR_n, we have that there exists $\{h'_{j_1}, \dots h'_{j_s}\} \subseteq \{G'_n, h'_{n+1}\}$, with $s \leq n$, such that $\langle \emptyset, (h'_{j_1}, \dots h'_{j_s}), d \wedge d' \rangle \rightarrow \langle G', H', d'' \rangle$ in $[\![P]\!]$.

Then, since $CT \models G \leftrightarrow \exists_{-Fv([\![G]\!]_G)}(G'_n \wedge h'_{n+1} \wedge d \wedge d')$, we have that

$$CT \models h_{j_1}, \dots h_{j_s} \leftrightarrow \exists_{-Fv([\![h_{j_1}, \dots h_{j_s}]\!]_G)}(h'_{j_1}, \dots h'_{j_s} \wedge d \wedge d')$$

and therefore $h_{j_1}, \ldots h_{j_s}$ is not a qualified answer for $[\![h_{j_1}, \ldots h_{j_s}]\!]_G$ in $[\![P]\!]$ (since it is always possible to make another derivation step from $h_{j_1}, \ldots h_{j_s}$ in $[\![P]\!]$).

But, by previous observations, the same goal has itself as answer in P thus contradicting the fact that $[\![P]\!]$ was an acceptable encoding for P. □

Remark 5.9 *Note that if we consider only the class of goals that are "reasonable" for a given program, i.e. goals that respects the intended meaning of the program, previous theorems are not valid anymore. Indeed in this restricted sense in the program*

$$\mathtt{rule} \ @ \ h_1 \ldots h_{n+1} \Leftrightarrow true \mid k$$

any goal different from $h_1 \ldots h_{n+1}$ is not a reasonable goal since there are no feasible answers related to it.

Hence in this case all CHR_n dialects for $n > 2$ are as expressive as CHR_2, indeed it is possible to provide the following encoding.

Every rule

$$\mathtt{rule} \ @ \ h_0 \ldots h_n \Leftrightarrow C \mid B$$

is translated into

$$\begin{aligned} r_1 \ @ \ h_0, h_1 &\Leftrightarrow i_1 \\ r_2 \ @ \ h_2, i_1 &\Leftrightarrow i_2 \\ &\ldots \\ r_n \ @ \ h_n, i_{n-1} &\Leftrightarrow C \mid B \end{aligned}$$

where $i_1, \ldots, i_n$ are fresh user-defined constraints that cannot be used in goals and that are only introduce for encoding the program.

We think that this restriction is too strong thus we prefer to consider the class $\mathcal{G}$ of all possible goals.

5.6 Conclusions and Related works

In this chapter we have studied the expressiveness of CHR in terms of language encoding. We have proved that multiple heads augment the expressive power of the language, indeed we have shown that CHR cannot be encoded in CHR with single heads under quite reasonable assumptions. These results are shown to hold also for (constraint) logic programming and pure Prolog. Then we extended this result by showing that CHR is more expressive than CHR with single heads also when considering programs which allow qualified answers only. More precisely we have proved that, when considering this class of programs, CHR cannot be encoded into CHR_1 assuming that goals are translated in a compositional way and that both the languages use the same theory for built-in constraints. Finally we have shown that, under some slightly stronger assumptions, in general the number of atoms in the head of rules affects the expressive power of the language. In fact we have proved that CHR_n cannot be encoded into CHR_m, with $n > m$.

There exists a very large literature on the expressiveness of concurrent languages, however there are only few papers which consider the expressive power of CHR: A recent study is [121], where the authors show that it is possible to implement any algorithm in CHR in an efficient way, i.e. with the best known time and space complexity. This result is obtained by introducing a new model of computation, called the CHR machine, and comparing it with the well-known Turing machine and RAM machine models. Earlier works by Frühwirth [62, 61] studied the time complexity of simplification rules for naive implementations of CHR. In this approach an upper bound on the derivation length, combined with a worst-case estimate of (the number and cost of) rule application attempts, allows to obtain an upper bound of the time complexity. The aim of all these works is clearly completely different from ours, even though it would be interesting to compare CHR and CHR_1 in terms of complexity.

When moving to other languages, somehow related to our paper is the work by Zavattaro [130] where the coordination languages Gamma [9] and Linda [64] are compared in terms of expressive power. Since Gamma allows multiset rewriting it

reminds CHR multiple head rules, however the results of [130] are rather different from ours, since a process algebraic view of Gamma and Linda is considered where the actions of processes are atomic and do not contain variables. On the other hand, our results depend directly on the presence of logic variables in the CHR model of computation. Relevant for our approach is also [38] which introduces the original approach to language comparison based on encoding, even though in this paper rather different languages with different properties are considered.

A similar comparison is done in [85], where Laneve and Vitale show that a language for modelling molecular biology, called κ-calculus (which will be analysed in the next chapter), is more expressive than a restricted version of the calculus, called nano-κ, which is obtained by restricting to "binary reactants" only (that is, by allowing at most two process terms in the left hand side of rules, while n terms are allowed in κ). This result is obtained by showing that, under some specific assumptions, a particular (self-assembling) protocol cannot be expressed in nano-κ, thus following a general technique which allows to obtain separation results by showing that (under some specific hypothesis) a problem can be solved in a language and not in another one (see also [100] and [125]).

This technique is rather different from the one we used, moreover also the assumption on the translation used in [85] are different from ours. Nevertheless, since κ (and nano-κ) can be easily translated in CHR, it would be interesting to see whether some results can be exported from a language to another. We left this as future work. We are also planning to investigate what happens when the source and the target CHR languages have different theories for the built-ins: we believe that some results hold also in this more general case, however some technical details need to be spelt out.

We also plan to investigate what happens when considering translation of CHR into real Prolog systems (with non logical built-ins). Some of the properties that we used in our technical lemmata in this case do not hold anymore, however we believe that also in this case we can establish separation results similar to those shown in section 5.4.4.

Chapter 6

Graphs Rewriting Systems - a Hierarchy

There is grandeur in this view of life, with its several powers, having been originally breathed into a few forms or into one; and that, whilst this planet has gone cycling on according to the fixed law of gravity, from so simple a beginning endless forms most beautiful and most wonderful have been, and are being, evolved.

Charles Darwin
The Origin of Species

This chapter keeps focusing on rewriting systems. Moreover the language analysed is similar to some extent to CHR. But instead of investigating on the expressiveness of multiple heads here we study the expressiveness of several dialects obtained by allowing or forbidding certain kinds of rules.

Here we analyse the expressive power of some dialects of the κ-calculus by focusing on the thin boundary between decidability and undecidability for problems like reachability and coverability.

6.1 Introduction

In recent years we are witnesses of an increasing interest in applications of specification languages used in concurrency as formal models of biological systems. Languages like Petri nets, term rewriting, and process calculi are becoming common idioms for fostering the cooperation between researchers working in biology and computer science [12, 29, 34, 45, 50, 53, 106, 54, 69, 107, 123].

Qualitative analysis like reachability [45, 106] and symbolic model checking [52], and static analysis like abstract interpretation [33] can be used for validation and optimisation (e.g. detection of dead rules and dependencies) of models that are used by biologists for experiments in silico (e.g. stochastic simulations). However, general purpose decision procedures are not always applicable to validate formal models of biological systems. Indeed, the level of granularity used in modelling biological mechanisms can dramatically influence the expressive power of the resulting formal languages, as in the case of the passage from basic chemistry (that may be modelled by Petri nets) to bio-chemistry (that requires binding sites, thus becoming Turing-complete) [133]. For this reason, as in other applications of concurrency, an important foundational issue is the study of dialects for which qualitative analysis is computable in an effective way and the isolation of minimal fragments in which it is proved to be impossible.

In this chapter, we investigate the boundary between decidability and undecidability of qualitative analysis of biological systems. As a formal model for our analysis, we consider the κ calculus [34]. κ is a formalism for modelling molecular biology where molecules are terms with internal state and sites, bonds are represented by names that label sites, and reactions are represented by rewriting rules.

For example, $EGFR[tk^0](1^z)$ represents a molecule of species $EGFR$ that is not phosphorilated – the internal state tk is 0 – and that is bond to another molecule – its site 1 is labelled with a name z.

The reaction in Fig. 6.1 defines the first step of the *Receptor Tyrosine Kinase* (RTK) growth factor EGF (a dimeric form of EGF binds two receptors $EGFR$, thus

phosphorylating the tyrosine kinase site – tk switches from 0 to 1). Briefly a kinase is a type of enzyme that transfers phosphate groups from high-energy donor molecules, such as ATP (Adenosine-5'-triphosphate a nucleotide) to specific target molecules (substrates); the process is called phosphorylation. Kinase enzymes that specifically phosphorylate tyrosine amino acids are called tyrosine kinases. RTK then acts as a receptor which dimerizes upon ligand binding. This induces a signalling cascade that helps regulating a number of cellular processes. For example in the activation of the epidermal growth factor (EGF). EGF plays an important role in the regulation of cell growth, proliferation, and differentiation by binding to its receptor EGFR. This reaction is rendered by the following κ rule:

$$\begin{aligned} &EGF(1^x + 2^y), EGF(1^x + 2^z), EGFR(1^y), \\ &EGFR[tk^0](1^z) \rightarrowtail \ EGF(1^x + 2^y), EGF(1^x + 2^z), \\ &\qquad\qquad EGFR(1^y), EGFR[tk^1](1^z) \end{aligned} \tag{6.1}$$

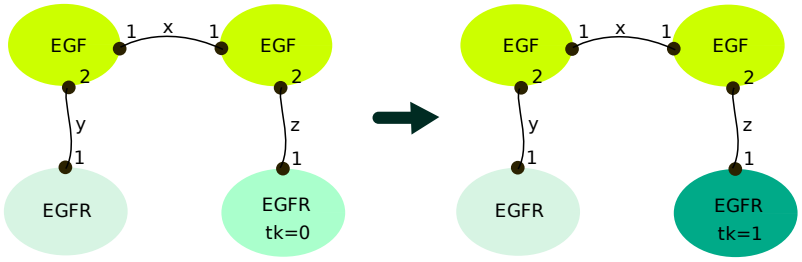

Figure 6.1: Representation of the κ-rule (6.1)

A recent contribution turns out to be rather close to the present one [33]. Using abstract interpretation (abstracting away from the multiplicity of molecules – always considered unbounded – and from the exact structure of molecular complexes) authors design an efficient algorithm for computing the set of reachable complexes in a fragment of κ with a *local rule set* and over-approximating the set of reachable complexes in the general case.

As a matter of fact, classical problems, such as reachability and coverability, turn out to be undecidable in κ. Therefore one is either compelled to design approximated

analyses or to study these properties in dialects of κ. We choose the second direction, thus yielding a number of precise analyses that do not abstract away either from the multiplicity of molecules or from the exact structure of complexes. To this aim, we consider a number of κ dialects that, as we discuss in the following, take inspiration from biological phenomena such as the *molecular self-assembly* [128] or the *DNA branch migration* [102]. These dialects are ordered into a lattice by the sublanguage relation – see Figure 6.2.

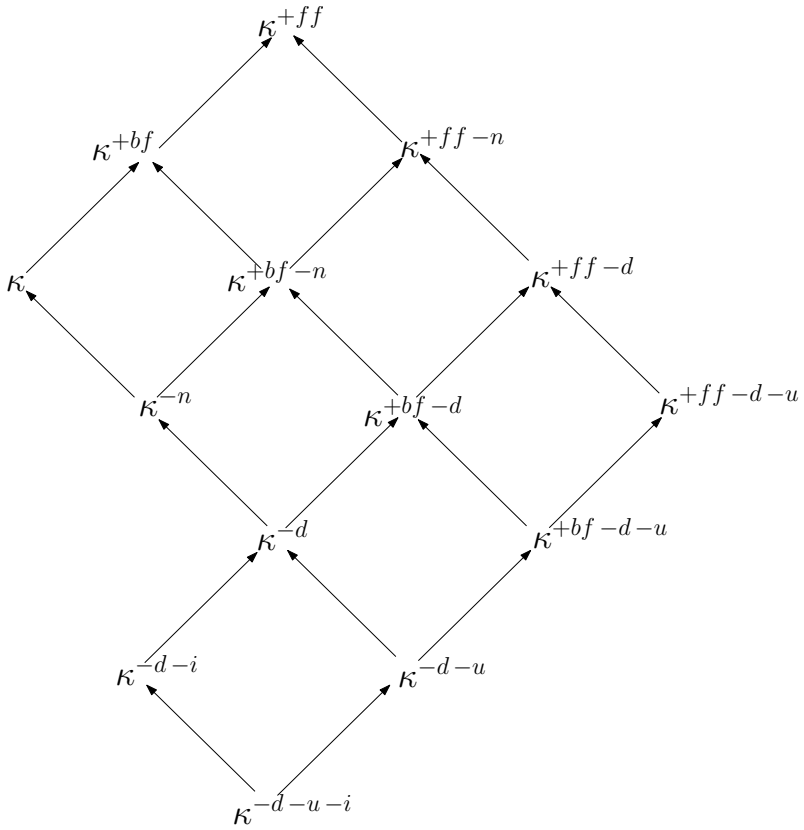

Figure 6.2: The κ lattice

Let us unravel the lattice with the restrictions imposed to κ to obtain the sublanguages κ^{-n}, κ^{-d}, and $\kappa^{-d\,-u}$. The calculus κ^{-n} follows by removing any form of destruction of molecules (the molecules never decrease). This fragment naturally models those systems where molecules always keep their "identity" even when they are part of a complex because, for example, they can subsequently dissociate from

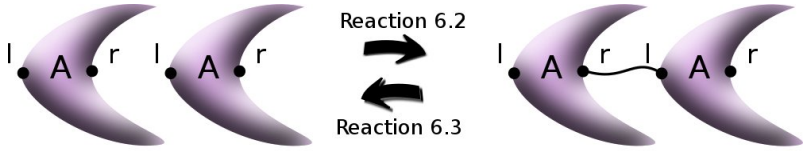

Figure 6.3: Linear bidirectional polymerisation

the complex. This is the case of polymers, that is chemical structures obtained by joining monomers that react on complementary surfaces. A simple polymerisation – the linear bidirectional one, where the complementary surfaces of monomers are two (that we respectively call l and r in the following) – is modelled by the following κ^{-n} rules depicted in Fig. 6.3:

$$A(r), A(l) \rightarrowtail A(r^x), A(l^x) \tag{6.2}$$

$$A(r^x), A(l^x) \rightarrowtail A(r), A(l) \tag{6.3}$$

The reaction (6.2) defines polymerisation (the creation of a bond between two monomers with free complementary surfaces); (6.3) defines depolymerization (the destruction of the bond, but not of the monomers).

The additional restriction yielding κ^{-d} is the one that disallows the removal of bonds (depolymerizations are forbidden). This restriction is inspired by molecular self-assembly, which is a process where molecules, initially unbound, adopt a defined arrangement. The DNA-origami method is a popular example of self-assembly that allows to create arbitrary two-dimensional shapes, such as Borromean rings [87], using DNA. In κ^{-d} self-assembly is directly enforced because bonds cannot be broken.

The last dialect along this axis, called κ^{-d-u}, is obtained by considering molecules without internal states. In several cases such states are not useful. An example is the DNA self-assembly governed by the Watson-Crick complementary base pairing [126].

We also consider two other subcalculi that forbid destructions of molecules and bonds: κ^{-d-i} and κ^{-d-u-i}. These dialects are obtained from κ^{-d} and κ^{-d-u}, respectively, by restricting reductions to those that never verify the connectedness of

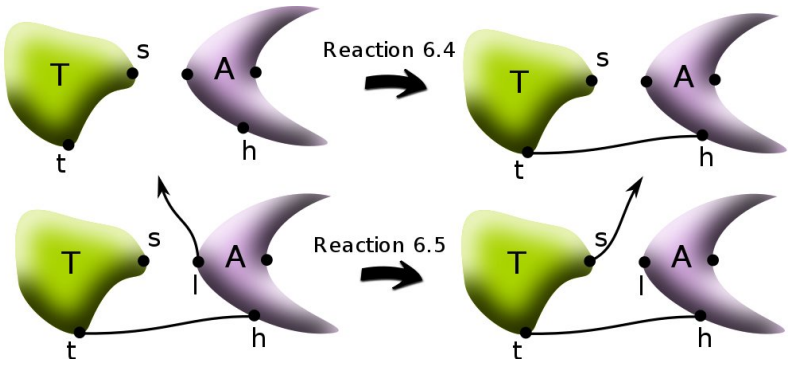

Figure 6.4: Bond Flipping

reactants. For example, the polymerisation (6.2) is a reaction of this type. It turns out that the Watson-Crick complementary base pairing may be defined in κ^{-d-u-i}.

Our analysis also takes into account a different axis. In [32] a new reaction rule has been introduced, called *exchange*. According to this reaction, the interaction between two molecules may flip a bond from one to the other. For example, the reader may consider the case where a thief molecule T may connect to a third site of the monomer A and steals the polymer connected to the site l of A (see also Fig. 6.4):

$$T(t+s), A(h) \rightarrowtail T(t^x+s), A(h^x) \tag{6.4}$$

$$T(t^x+s), A(h^x+l^y) \rightarrowtail T(t^x+s^y), A(h^x+l) \tag{6.5}$$

(reaction 6.5 is an example of bond flipping).

Bond flipping allows us to model other interesting DNA systems, such as those based on branch migration used to create, for instance, a nanoscale biped walking along a DNA strand [129]. The calculi including bond flipping are made evident with the superscript $+bf$. Finally, we consider also a more liberal form of flipping, called free flipping (see Figure 6.6), in which flipping can occur also between two unbound molecules. With free flipping, the thief molecule T can steal the polymer to a monomer without previously connecting to it:

$$T(s), A(l^y) \rightarrowtail T(s^y), A(l) \tag{6.6}$$

For all of the 14 dialects of κ we investigate three problems: the *Reachability Problem* (RP), the *Simple Coverability Problem* (SCP) and the *Coverability Problem* (CP).

The RP is the decision problem associated to the existence of a derivation (simulation) from an initial solution to a target. As shown in [45, 52, 106], this problem is of high relevance for validation of formal models of biological systems.

The SCP is the decision problem associated to the existence of a derivation from an initial solution to a target with given components, regardless of their multiplicity. SCP is a generalisation of the decision problem associated to the static analysis considered in [33].

Finally, CP is the decision problem associated to the existence of a derivation from an initial solution to a target that contains given components: CP is a generalisation of RP that can naturally be used to formulate structural properties of biological networks without need of specifying an entire target solution.

Our results about the (un)decidability of RP, SCP, and CP in the κ lattice are illustrated in Figure 6.5.

The undecidability results are proved by modelling Turing complete formalisms in the calculi, while the decidability results are proved by reduction to decidable properties in finite state systems or Petri-nets. As far as the undecidability results are concerned, the most surprising one is the undecidability of CP in $\kappa^{-d\,-u}$. We prove that this very poor fragment of κ – in which molecules have no state and bonds cannot be neither destroyed nor flipped – is powerful enough to encode RAM Machines [96], a Turing complete formalism. It is also interesting to observe that this result about $\kappa^{-d\,-u}$ relies on the possibility to test at least the presence of bonds. In fact, κ^{-d-u-i} is no longer Turing complete because CP is decidable for this fragment (CP allows one to test whether a certain complex, for instance representing the termination of a computation, can be produced).

While the dialects that include $\kappa^{-d\,-u}$ are Turing complete, many of them retain decidable SCP and/or RP properties. These facts, apparently contrasting with Turing universality of the calculi, are consequences of the following monotonic prop-

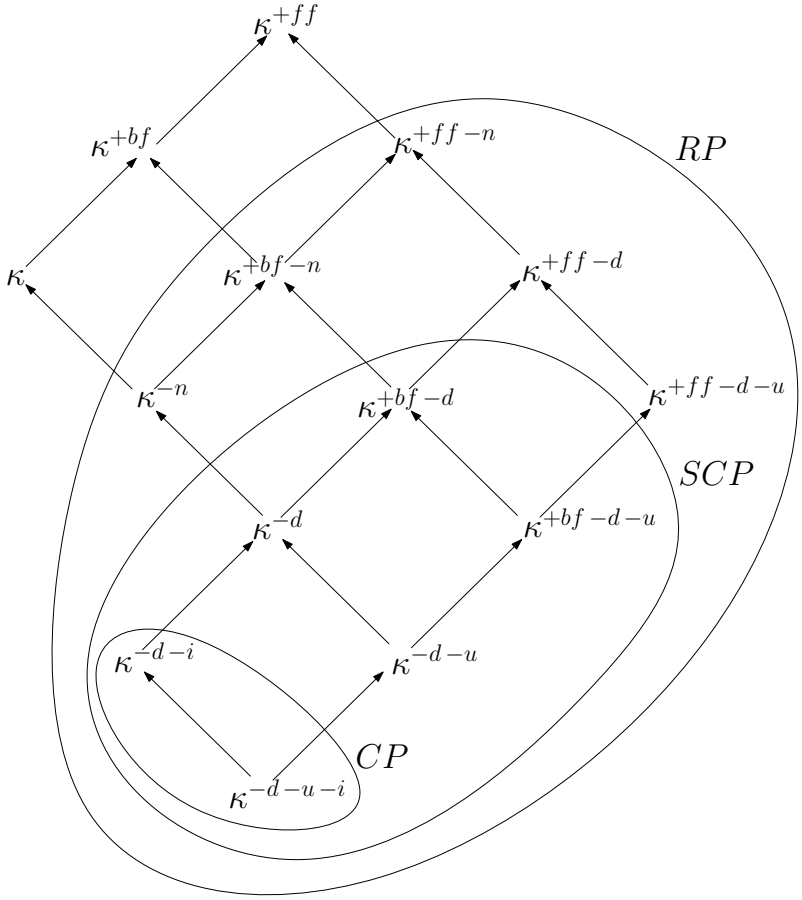

Figure 6.5: The κ lattice and the (un)decidability of RP, SCP, CP

erties: reactions cannot decrease either (i) the total number of molecules in the solution or (ii) the size of the complexes in the solution. In the calculi satisfying the form of monotonicity (i) we show that it is possible to compute an upper-bound to the number of molecules in the solutions of interest for the analysis of RP. In this way, we reduce our analysis to a finite state system. For the calculi satisfying the form of monotonicity (ii) we show that it is possible to compute an upper-bound to the size of the complexes in the solutions of interest for the analysis of SCP. In this case, even if it is not possible to reduce to a finite state system (because there is no upper-bound to the number of instances of the complexes in the solutions of interest), we can reduce to Petri-nets in which reachability and coverability are decidable.

The chapter is organised as follows: Section 6.2 recalls κ, its fragments and the needed terminology. Section 6.3 discusses the separation results between the fragments of κ. Section 6.4 discusses related contributions in literature. Section 6.5 concludes with few final remarks.

6.2 Preliminaries

This section introduces κ and its dialects, together with the terminology that is necessary in the sequel.

6.2.1 κ-calculi

Two countable sets of *species* $A, B, C, \ldots$, and of *bonds* $x, y, z, \ldots$ are assumed. Species are sorted according to the number of sites $a, b, c, \ldots$ and fields $h, i, j, \ldots$ they possess. Sites may be either bound to other sites or unbound, i.e. not connected to other sites. The configuration of sites are defined by partial maps, called *interfaces* and ranged over by σ, ρ, ... The interfaces associate to sites either a bond or a special empty value ε, which models the fact that the site is unbound.

For instance, if A is a species with three sites, $(2 \mapsto x; 3 \mapsto \varepsilon)$ is one of its interfaces. This map is written $2^x + 3$ (the ε is always omitted). We notice that this σ does not define the state of the site 1, which may be bound or not. Such (proper) partial maps are used in reaction rules in order to abstract from sites that do not play any role in the reactions (similar for evaluations, see below). In the following, when we write $\sigma + \sigma'$ we assume that the domains of σ and σ' are disjoint. The functions $dom(\cdot)$ and $ran(\cdot)$ return the domain and the range of a function.

Fields represent the internal state of a species. The values of fields are also defined by partial maps, called *evaluations*, ranged over by u, v, ... For instance, if A is a species with three fields, $\{1 \mapsto 5; 2 \mapsto 0; 3 \mapsto 4\}$, shortened into $1^5 + 2^0 + 3^4$, is a possible evaluation. We assume there are finitely many internal states, that is every field is mapped into a finite set of values. As for interfaces, $u + v$, we implicitly assume that the domains of u and v are disjoint.

Definition 6.1 *A* molecule $A[u](\sigma)$ *is a term where* u *and* σ *are a total evaluation and a total interface of* A.
Solutions, *ranged over by* S, T, ..., *are defined by:* $S ::= A[u](\sigma) \mid S, S$.
Bonds in solutions occur at most twice; in case bonds occur exactly twice the solution is proper.
A pre-solution *is a sequence of terms* $A[u](\sigma)$ *where* u *and* σ *are partial functions and bonds occur at most twice. A pre-solution is* proper *if (similarly as before) bonds occur exactly twice.*
The set of bonds in S *is denoted* $\texttt{bonds}(S)$.

In the rest of the paper the composition operator "," is assumed to be associative, so $(S, S'), S''$ is equal to $S, (S', S'')$ (therefore parentheses will be always omitted).

Let $\sigma \leq \sigma'$ if $dom(\sigma) = dom(\sigma')$ and, for every i, if $\sigma(i) \neq \varepsilon$ then $\sigma(i) = \sigma'(i)$ (the two interfaces may differ on sites mapped to the empty value ε by σ as σ' may map such sites to bonds).

Reactions have the shape $\mathsf{L} \rightarrowtail \mathsf{R}$, where L and R are pre-solutions called reactants and products, respectively. The general shape of reactions is defined in the next definition. Following [32], we extend the definition of [34] with exchange reactions, thus the calculus is an extension of the κ-calculus.[1]

Definition 6.2 *Reactions of the* κ^{+ff} *calculus – the* κ *calculus with free flipping rules – are either* creations C, *or* destructions D, *or* exchanges E.
The format of creations is

$$A_1[u_1](\sigma_1), \ldots, A_n[u_n](\sigma_n) \rightarrowtail A_1[u'_1](\sigma'_1), \ldots, A_n[u'_n](\sigma'_n), B_1[v_1](\phi_1), \ldots, B_k[v_k](\phi_k)$$

where, for every i, $dom(u_i) = dom(u'_i)$, $\sigma_i \leq \sigma'_i$, *and* v_i *and* ϕ_i *are total. Reactants and products are proper.*
The format of destructions is

$$A_1[u_1](\sigma_1), \ldots, A_n[u_n](\sigma_n) \rightarrowtail A_{i_1}[u'_{i_1}](\sigma'_{i_1}), \ldots, A_{i_m}[u'_{i_m}](\sigma'_{i_m})$$

[1] Another difference with [34] is that we allow newly produced molecules unbound from existing ones.

where $i_1, \ldots, i_m$ *is an ordered sequence in* $[1 \ldots n]$*, for every* i_j*,* $dom(u_{i_j}) = dom(u'_{i_j})$*,* $\sigma_{i_j} \geq \sigma'_{i_j}$*, and if* $i_j \notin \{i_1, \ldots, i_m\}$ *then* σ_{i_j} *is total. Reactants and products are proper.*

The format of exchanges is

$$A[u](a^x + \sigma), B[v](b + \rho) \;\rightarrowtail\; A[u'](a + \sigma), B[v'](b^x + \rho)$$

where $ran(\sigma) = ran(\rho)$*.*

Creations may change state, produce new bonds between two unbound sites, or synthesise new molecules. Destructions behave the other way around. Exchanges are reminiscent of the π calculus because they define a migration of a bond from one reactant to the other. We distinguish two types of exchanges: the one occurring between connected molecules, called *(connected) bond flipping*, and the one occurring between disconnected molecules, called *free (bond) flipping*. These are illustrated below:

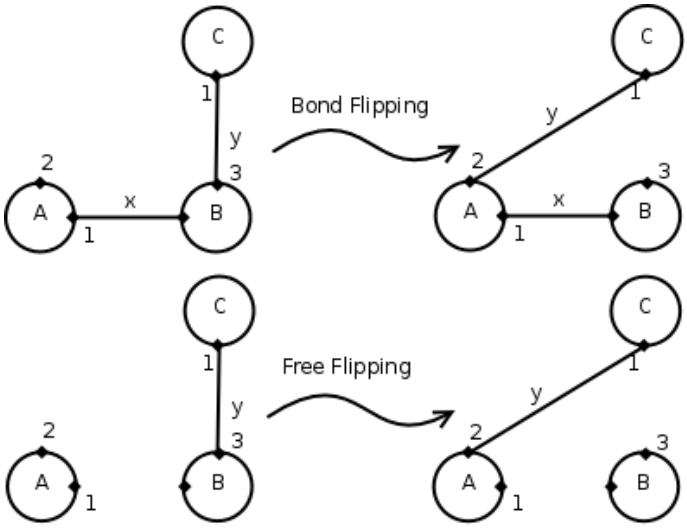

Figure 6.6: Bond flipping and free flipping

The operational semantics of κ^{+ff} calculus uses the following two definitions:

- the *structural equivalence* between solutions, denoted $\equiv$, is the least one satisfying (we remind that solutions are already quotiented by associativity of ","):
  - $S, T \equiv T, S$;
  - $S \equiv T$ if there exists an injective renaming $\imath$ on bonds such that $S = \imath(T)$.
- $A_1[u_1 + u_1'](\sigma_1 \circ \imath + \sigma_1'), \ldots, A_n[u_n + u_n'](\sigma_n \circ \imath + \sigma_n')$ is an $(\imath, u_1' \cdots u_n', \sigma_1', \cdots, \sigma_n')$ *instance* of $A_1[u_1](\sigma_1), \ldots, A_n[u_n](\sigma_n)$ if $\imath$ is an injective renaming on bonds and the maps $u_j + u_j'$ and $\sigma_j \circ \imath + \sigma_j'$ are total with respect to the species A_j.

Definition 6.3 *The reduction relation of the κ^{+ff} calculus, written $\rightarrow$, is the least one satisfying the rules:*

- *let $\mathsf{L} \rightarrowtail \mathsf{R}$ be a reaction of κ^{+ff}, S be an $(\imath, \widetilde{u}, \widetilde{\sigma})$-instance of L, and T be an $(\imath, \widetilde{u'}, \widetilde{\sigma'})$-instance of R. Then $S \rightarrow T$;*
- *let $S \rightarrow T$ and $(\mathtt{bonds}(T) \setminus \mathtt{bonds}(S)) \cap \mathtt{bonds}(R) = \emptyset$, then $S, R \rightarrow T, R$;*
- *let $S \equiv S'$, $S' \rightarrow T'$, and $T' \equiv T$, then $S \rightarrow T$.*

The κ^{+ff} calculus groups several sub-calculi that have in turn simpler formats of rules. We have already depicted in Figure 6.5 the fragments we study. We move from κ^{+ff} along two different axes:

1. we restrict reactions by letting $i_m = n$ in destructions (forbidding cancellations of molecules), the superscript $-n$; removing destructions, the superscript $-d$; removing destructions and considering species with emptyset of sites (removing fields), the superscript $-d-u$; removing destructions, fields, and such that no bond occurs in the left-hand side of creations and exchanges, except the flipping one, the superscript $-d-u-i$;
2. we restrict exchanges by allowing bond-flipping only, the superscript $+bf$, and by removing exchanges, no superscript $+bf$ or $+ff$.

Some of the combinations are empty. For example, a calculus without checks of bonds and with cancellation of bonds is meaningless as, in order to remove one bond, it is necessary to test its presence first.

6.2.2 Decision problems for qualitative analysis.

A first basic qualitative property is whether a solution eventually produces "something relevant" or not. Clearly this "something relevant" can be defined in a variety of ways. In this paper we consider its formalisation in terms of reachability and coverability, two standard properties which have been extensively investigated in many concurrent formalisms. Few preliminary notions are required.

Definition 6.4 (Complex) *Given a proper solution, a* complex *is a sub-solution that is connected (there is a path of bonds connecting every pair of molecules therein) and proper. Two complexes in a solution are* equal *if they are structurally equivalent.*

Let $\mathtt{S}(S)$ be the set of different complexes in S; let also $\rightarrow^*$ be the transitive and reflexive closure of $\rightarrow$.

Definition 6.5 RP: *the* reachability problem of T from a proper solution S *checks the existence of R such that $S \rightarrow^* R$ and $R \equiv T$;*

SCP: *the* simple coverability problem of T from a proper solution S *checks the existence of R such that $S \rightarrow^* R$ and $\mathtt{S}(R) = \mathtt{S}(T)$ and $R \equiv T, T'$, for some T';*

CP: *the* coverability problem of T from a proper solution S *checks the existence of R such that $S \rightarrow^* R$ and $R \equiv T, T'$, for some T'.*

6.3 (Un)Decidability Results for κ dialects

In this section we study the (un)decidability of RP, SCP, and CP in the κ lattice of Figure 6.5. The overall results represented in that figure are the consequences

of theorems that we detail in the remainder of this section. For each decidability region – one for RP, one for SCP, and one for CP – we prove that the corresponding property is decidable in the top language of the region and undecidable in the bottom language(s) among those not included in the region.

We separate the presentation of our results in two subsections, the first one is devoted to decidability, the latter to undecidability.

6.3.1 Decidability results

The proofs of decidability follow by reduction to decidable problems in either finite state systems or P/T nets (see Section 2.3).

Our first positive result is for the $\kappa^{+ff\,-n}$ fragment.

Theorem 6.1 *RP is decidable in* $\kappa^{+ff\,-n}$.

Proof: We reduce RP to the reachability problem in a finite state system. Let $\mathcal{R}$ be a set of $\kappa^{+ff\,-n}$ reactions and let S and T be two proper solutions. We notice that, in order for $S \rightarrow^* T$, all intermediary solutions traversed by the computation must have a number of molecules which is less or equal to the number n_T of molecules in T. This is because in $\kappa^{+ff\,-n}$ it is not possible to delete molecules.

Let $\mathcal{A}$ be the set of species occurring either in S or in a rule of $\mathcal{R}$. Let also $\mathsf{set}^T(\mathcal{A})$ be the set of (proper) solutions with a number of molecules less than n_T. This set is finite *up-to structural equivalence* because the number of sites and fields of species is finite, the values of fields is finite, and the possible combinations of bonds is finite, as well. By mapping every solution R to its canonical representative in the structural equivalence class, called $[R]$, we can build a *finite state system* FSS_T such that, by Definition 6.3, given two solutions in $\mathsf{set}^T(\mathcal{A})$, $R \rightarrow R'$ if and only if $[R] \rightarrow [R']$. We conclude the proof by observing that $S \rightarrow^* T$ if and only if $[S] \rightarrow^* [T]$, and this latter property is decidable in FSS_T. □

The next result is about the decidability of SCP in $\kappa^{+bf\,-d}$. This follows from the property that, in $\kappa^{+bf\,-d}$, the connectedness of two molecules can never be broken.

Lemma 6.1 *Let S and T be two proper solutions of the $\kappa^{+bf\,-d}$ calculus such that $S \rightarrow T$. If there exists a path of bonds connecting two molecules in S – i.e. the two molecules are* connected *– then the two molecules are still connected in T (possibly with a different path).*

Proof: Bonds can only be created and flipped in $\kappa^{+bf\,-d}$. In particular, in this last case, a flip occurs if the affected molecules – not only the reactants – are already connected (see the top picture of Figure 6.6). This entails the property of the lemma. □

Theorem 6.2 *SCP is decidable for $\kappa^{+bf\,-d}$.*

Proof: We reduce to the *target marking reachability* problem for P/T nets, which is decidable [27]. This problem amounts to checking, given a P/T net P and a target marking m_t, whether a marking m is reachable in P such that $m(p) = 0$ for every place p such that $m_t(p) = 0$, and $m(p') \geq m_t(p')$ for every other place p'.

Let $\mathcal{R}$ be a set of $\kappa^{+bf\,-d}$ reactions and S, T and R be proper solutions such that $\mathtt{S}(T) = \mathtt{S}(R)$ and $R \equiv T, R'$, for some solution R'. Let n_T be the maximum number of molecules of a complex in T.

As a consequence of Lemma 6.1, if $S \rightarrow^* R$, then the complexes occurring in every intermediary solution traversed by the computation have a number of molecules smaller or equal to n_T.

Let $\mathcal{A}$ be the set of species occurring either in S or in a rule of $\mathcal{R}$, and let $\mathsf{SET}^T(\mathcal{A})$ be the set of complexes composed of at most n_T molecules belonging to the species in $\mathcal{A}$. As in the proof of Theorem 6.1, this set $\mathsf{SET}^T_{\equiv}(\mathcal{A})$ is finite if taken up-to structural equivalence.

We define the following P/T net. The places are the elements of $\mathsf{SET}^T_{\equiv}(\mathcal{A})$. We build the transitions in two steps. Given a rule $\rho : \mathsf{L} \rightarrowtail \mathsf{R}$, we first define RED_ρ as the least set containing all reductions $S_1, \cdots, S_n \rightarrow S'_1, \cdots, S'_m$ such that:

i) S_i and $S'_j \in \mathsf{SET}^T_{\equiv}(\mathcal{A})$ for every i and j;

ii) the reduction is obtained by applying Definition 6.3 that instantiates ρ with a proof-tree $\mathtt{PT}$,

iii) for every i, S_i is directly involved in the reduction (i.e. at least one molecule of its is an instance of a term in L in the unique leaf of $\mathtt{PT}$).

Condition (iii) ensures that set RED_ρ is finite up to structural equivalence. Indeed, we have that n is less or equal than the number of terms in L, m is less or equal than the number of terms in R, and $\mathsf{SET}^T_{\equiv}(\mathcal{A})$ is finite. For each rule ρ and each reduction $S_1, \cdots, S_n \to S'_1, \cdots, S'_m$ in RED_ρ we build a P/T transition with pre-set $[S_1], \ldots, [S_n]$ and post-set $[S'_1], \ldots, [S'_m]$. Let m_S and m_T be the initial and final markings corresponding to S and T, respectively. The above P/T net faithfully reproduces the possible computations of S that traverse solutions retaining complexes composed of at most n_T molecules. This allows us to reduce SCP of S to the target marking reachability of m_T in the above P/T net, which is decidable. □

Our last decidability result regards κ^{-d-i}.

Theorem 6.3 *CP is decidable in κ^{-d-i}.*

Proof: We reduce to the coverability problem in P/T net. Let $\mathcal{R}$ be a set of κ^{-d-i} reactions and S, T and R be proper solutions such that $R \equiv T, R'$ for some solution R'. Let n_T be the maximum number of molecules of a complex in T.

As in the proof of Theorem 6.2, let $\mathcal{A}$ be the set of species occurring either in S or in a rule of $\mathcal{R}$, and let $\mathsf{SET}^T(\mathcal{A})$ be the set of complexes composed of at most n_T molecules belonging to the species in $\mathcal{A}$. The set $\mathsf{SET}^T_{\equiv}(\mathcal{A})$ is finite.

We define the following P/T net. Places are elements of $\mathsf{SET}^{T,+}_{\equiv}(\mathcal{A})$ that extends $\mathsf{SET}^T_{\equiv}(\mathcal{A})$ with the places $\widehat{A}[u](\sigma)$, for every species $A \in \mathcal{A}$, every evaluation u, and with partial functions σ mapping every site to ε (properly speaking, $\widehat{A}$ is not a molecule because σ cannot be partial). Note that the number of places is finite because the additional places $\widehat{A}[u](\sigma)$, with respect to the P/T net already discussed in Theorem 6.2, is finite (A is taken from the finite set $\mathcal{A}$, the possible evaluations u are finite and similarly for σ).

Transitions are defined in two steps. Given a rule $\rho : \mathsf{L} \rightarrowtail \mathsf{R}$, we first define RED^+_ρ as the least set containing all reductions $S_1, \cdots, S_n \to S'_1, \cdots, S'_m$ such that:

i) S_i and S'_j are complexes composed only of molecules belonging to species in $\mathcal{A}$ (possibly with size greater than n_T);

ii) the reduction is obtained by applying Definition 6.3 that instantiates ρ with a proof-tree $\mathtt{PT}$,

iii) for every i, S_i is directly involved in the reduction (i.e. at least one molecule of its is an instance of a term in L in the unique leaf of $\mathtt{PT}$).

Note that, unlike the proof of Theorem 6.2, RED^+_ρ can be infinite as we do not impose any restriction to the sizes of S_i. Nevertheless, it is possible to group transitions in RED^+_ρ into finitely many different groups. For every $S_1, \cdots, S_n \to S'_1, \cdots, S'_m$ in RED^+_ρ, we let a transition with pre-set given by the following places

- $[S_i]$ if S_i has no more than n_T molecules;
- $\widehat{A_1}[u_1](\sigma_1), \cdots, \widehat{A_{m'}}[u_{m'}](\sigma_{m'})$ if S_i has more than n_T molecules and the molecules of S_i that participate to the reduction (the ones that instantiate the terms in L in the unique leaf of $\mathtt{PT}$) are

  $$A_1[u_1](\sigma_1 + \rho_1), \cdots, A_{m'}[u_{m'}](\sigma_{m'} + \rho_m)$$

  with $\varepsilon \notin ran(\rho_i)$ and $\{\varepsilon\} = ran(\sigma_i)$

and post-set given by the following places

- $[S'_j]$ if S'_j has no more than n_T molecules;
- $\widehat{A_1}[u_1](\sigma_1), \cdots, \widehat{A_{m'}}[u_{m'}](\sigma_{m'})$ if S'_j has more than n_T molecules and the molecules of S'_j that participate to the reduction (the ones that instantiate the terms in R in the unique leaf of $\mathtt{PT}$) are

  $$A_1[u_1](\sigma_1 + \rho_1), \cdots, A_{m'}[u_{m'}](\sigma_{m'} + \rho_m)$$

  with $\varepsilon \notin ran(\rho_i)$ and $\{\varepsilon\} = ran(\sigma_i)$.

The set of transitions is finite because both the pre-sets and the post-sets use places in $\mathsf{SET}^{T,+}_{\equiv}(\mathcal{A})$ and their cardinality is less or equal to the number of terms in L and R, respectively.

Let m_S and m_T be the initial and final markings corresponding to S and T, respectively. This P/T net *does not* faithfully represent all the complexes that can be produced by computations starting from S. In fact, while every complex with cardinality less than n_T is represented by a place, this is not the case for complexes bigger than n_T. When such a complex is created, the net removes the structure of bonds, and considers only the states and the free sites of its molecules. However, this information is sufficient for the coverability analysis in the κ^{-d-i}-calculus because, by Lemma 6.1, the size of a complex cannot decrease, thus complexes larger than n_T cannot directly produce the complexes of interest for the analysis but can only trigger reactions necessary in order to reach such complexes. As in the κ^{-d-i}-calculus the bond names cannot be tested in the reactants of a reaction, the loss of this information for large structures is not problematic.

This construction allows us to reduce the coverability problem for κ^{-d-i} to the coverability of the marking m_T in P/T net, which is decidable. □

6.3.2 Undecidability results

Our undecidability results follow by reducing to undecidable problems such as the halting problem for RAMs (see Section 2.1)

Here we consider RAMs in which registers are initially set to zero and where the instruction 0 is Halt. Our first negative result is for reachability of a solution in κ.

Theorem 6.4 *RP is undecidable in κ.*

Proof: We reduce the termination problem for RAMs to RP.

Let M be a RAM with n instructions. To encode it in κ we use five species:

1. P is the program counter; it retains one field with values in $[0, \ldots, n]$ and no site;

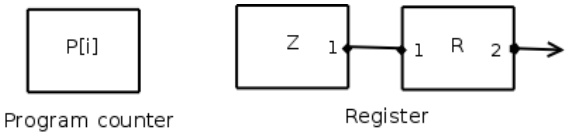

Figure 6.7: Species for the encoding

2. Z_1 and Z_2, both with one site, represent the value 0;

3. R_1 and R_2, both with two sites, represent the unity to be added to or removed from registries.

Let $j, l \in [0..n]$ and let $i \in \{1, 2\}$. The encoding $[\![\cdot]\!]_\kappa$ is defined in Figure 6.8. It turns

$$[\![j : \mathsf{Succ}(R_i)]\!]_\kappa = \begin{cases} P[1^j], Z_i(1) \rightarrowtail P[1^{j+1}], Z_i(1^x), R_i(1^x + 2) \\ P[1^j], R_i(2) \rightarrowtail P[1^{j+1}], R_i(2^x), R_i(1^x + 2) \end{cases}$$

$$[\![j : \mathsf{DecJump}(R_i, l)]\!]_\kappa = \begin{cases} P[1^j], Z_i(1) \rightarrowtail P[1^l], Z_i(1) \\ P[1^j], Z_i(1^x), R_i(1^x + 2) \rightarrowtail P[1^{j+1}], Z_i(1) \\ P[1^j], R_i(2^x), R_i(1^x + 2) \rightarrowtail P[1^{j+1}], R_i(2) \end{cases}$$

$$[\![j : \mathsf{Halt}]\!]_\kappa = \begin{cases} P[1^j], Z_1(1), Z_2(1) \rightarrowtail P[1^0], Z_1(1), Z_2(1) \\ P[1^j], Z_i(1^x), R_i(1^x + 2) \rightarrowtail P[1^j], Z_i(1) \\ P[1^j], R_i(2^x), R_i(1^x + 2) \rightarrowtail P[1^j], R_i(2) \end{cases}$$

Figure 6.8: Enconding RAMs in κ.

out that the RAM halts if and only if the solution $P[1^0], Z_1(1), Z_2(1)$ is reachable from the initial state. Therefore we conclude that RP is undecidable in κ. □

Theorem 6.5 *SCP is undecidable in κ^{-n}.*

Proof: We reduce the termination problem for RAMs to SCP in κ^{-n}. For this, we modify the encoding of RAMs used for κ in the previous theorem as follows:

- a binary field to the species R_1 and R_2 is added. When this field is zero, the molecule is considered garbage, otherwise it is a valid one.

Without loss of generality, we assume that the two registers are incremented at least once.

The encoding $[\![\cdot]\!]_{\kappa^{-n}}$ is defined in Figure 6.9.

$$[\![j : \mathsf{Succ}(R_i)]\!]_{\kappa^{-n}} = \begin{cases} P[1^j], Z_i(1) \rightarrowtail P[1^{j+1}], Z_i(1^x), R_i[1^1](1^x + 2) \\ P[1^j], R_i(2) \rightarrowtail P[1^{j+1}], R_i(2^x), R_i[1^1](1^x + 2) \end{cases}$$

$$[\![j : \mathsf{DecJump}(R_i, l)]\!]_{\kappa^{-n}} = \begin{cases} P[1^j], Z_i(1) \rightarrowtail P[1^l], Z_i(1) \\ P[1^j], Z_i(1^x), R_i[1^1](1^x + 2) \rightarrowtail \\ \qquad\qquad P[1^{j+1}], Z_i(1), R_i[1^0](1 + 2) \\ P[1^j], R_i(2^x), R_i[1^1](1^x + 2) \rightarrowtail \\ \qquad\qquad P[1^{j+1}], R_i(2), R_i[1^0](1 + 2) \end{cases}$$

$$[\![j : \mathsf{Halt}]\!]_{\kappa^{-n}} = \begin{cases} P[1^j], Z_1(1), Z_2(1) \rightarrowtail P[1^0], Z_1(1), Z_2(1) \\ P[1^j], Z_i(1^x), R_i[1^1](1^x + 2) \rightarrowtail \\ \qquad\qquad P[1^j], Z_i(1), R_i[1^0](1 + 2) \\ P[1^j], R_i(2^x), R_i[1^1](1^x + 2) \rightarrowtail \\ \qquad\qquad P[1^j], R_i(2), R_i[1^0](1 + 2) \end{cases}$$

Figure 6.9: Encoding RAMs in κ^{-n}.

Namely, the increment refines the previous encoding by setting to 1 the field of the new R; the decrement, rather than removing one molecule at the end of the register, which is not allowed in κ^{-n}, removes the bond and resets the field to zero; the halt operation turns every molecule R to garbage. We now observe that any solution that contains the complexes in the target solution

$$T = P[1^0], Z_1(1), Z_2(1), R_1[1^0](1 + 2), R_2[1^0](1 + 2)$$

encodes a halting configuration. Thus, termination of a RAM can be reduced to SCP for the corresponding κ^{-n} encoding and for the target solution T. □

We observe that, without using fields and destructions, as in $\kappa^{-d\,-u}$, it is not possible to reuse the encoding scheme of Theorems 6.4 and 6.5.

In the following theorem we prove that the use of creations in molecules without fields is sufficient to make the CP problem undecidable.

Theorem 6.6 *CP is undecidable in* $\kappa^{-d\,-u}$.

Proof: We define an encoding of RAMs by using constructions on species with emptysets of fields. Instructions are implemented by species P_j with a site 1 that may be bound to a molecule of species D. When this happens, the instruction is disabled. A further species *Halt* with no sites will represent a terminating state.

Registers are implemented by grids of increasing height (see Figure 6.10). The

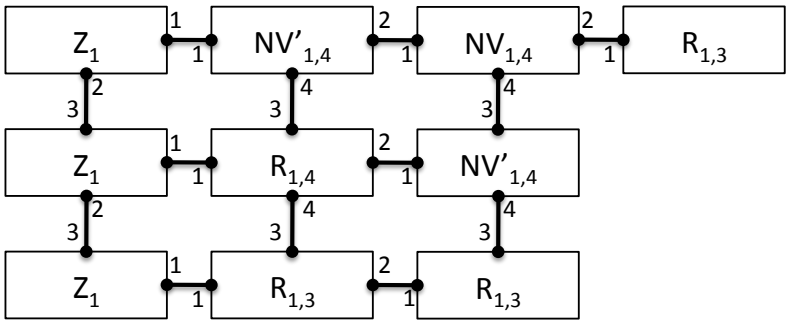

Figure 6.10: Grid representing the register R_1.

first column consists of Z_i molecules with three sites; the other nodes of the grid, called *register molecules*, are either $R_{i,j}$ molecules or $NV_{i,j}$ or $NV'_{i,j}$ molecules, $i \in \{1, 2\}$ and j ranging over instruction numbers, all retaining 4 sites. The meaningful part of the grid is the topmost row: the number of molecules $R_{i,j}$ therein represents the value of the corresponding register while the other rows represent previous values (we add a new row when performing a decrement). For instance, the register in Figure 6.10 contains the value 1 obtained after two increments –performed by the instruction with index 3–, two decrements –performed by the instruction with index

4–, and a subsequent increment –performed by the instruction with index 3. The encoding of $[\![j : \mathsf{Inc}(R_{i,j})]\!]_{\kappa^{-d-u}}$ increases the topmost row of the grid with a molecule $R_{i,j}$. The encoding of an increment $[\![j : \mathsf{Succ}(R_i)]\!]_{\kappa^{-d-u}}$ is defined by the three rules

$$\begin{aligned} P_j(1), Z_i(1) \rightarrowtail & P_j(1^x), D(1^x), Z_i(1^y), \\ & R_{i,j}(1^y + 2 + 3 + 4), P_{j+1}(1) \\ P_j(1), R_{i,j'}(2+3) \rightarrowtail & P_j(1^x), D(1^x), R_{i,j'}(2^y + 3), \\ & R_{i,j'}(1^y + 2 + 3 + 4), P_{j+1}(1) \\ P_j(1), NV_{i,j'}(2+3) \rightarrowtail & P_j(1^x), D(1^x), NV_{i,j'}(2^y + 3), \\ & R_{i,j'}(1^y + 2 + 3 + 4), P_{j+1}(1) \end{aligned}$$

The encoding of $[\![j : \mathsf{DecJump}(R_i, l)]\!]_{\kappa^{-d-u}}$ is more complex. The key idea is to copy the topmost row of the grid (from left to right according to the graphical representation of the grid in Figure 6.10) reducing, if possible, the number of molecules $R_{i,j'}$. This is obtained replacing the first encountered $R_{i,j'}$ molecule with the molecule $NV'_{i,j}$. If there is no such molecule available, all molecules in the new topmost row will be of species $NV_{i,j}$ (i.e. the kind of molecule used to copy molecules of species $NV_{i,j'}$ or $NV'_{i,j'}$). The copy records j in the second index of the register molecules: in this way, when the copy is finished (i.e. the new instance of Z_i is produced) it is possible to release the molecule representing the next instruction, that is P_{j+1} in case the decrement succeeded, P_s otherwise. The encoding of a decrement

$$[\![j : \mathsf{DecJump}(R_i, l)]\!]_{\kappa^{-d-u}}$$

is reported in Figure 6.11.

The encoding $[\![j : \mathsf{Halt}]\!]_{\kappa^{-d-u}}$ simply produces the $Halt$ molecule and is defined by the rule

$$P_j(1) \rightarrowtail P_j(1^x), D(1^x), Halt$$

The encoding satisfies the following property: the RAM halts if and only the solution $P_1(1), Z_1(1+2+3), Z_2(1+2+3)$ in the corresponding κ^{-d-u} encoding can produce molecule *Halt*. Thus, termination of RAMs is reduced to CP with target solution $T = Halt$. Therefore the undecidability of CP in κ^{-d-u}. □

$$P_j(1), Z_i(1) \rightarrowtail P_j(1^x), D(1^x), Z_i(1^y), P_l(1)$$
$$P_j(1), R_{i,j'}(2+3) \rightarrowtail P_j(1^x), D(1^x), R_{i,j'}(2+3^y), NV'_{i,j}(1+2+3+4^y)$$
$$P_j(1), X_{i,j'}(2+3) \rightarrowtail P_j(1^x), D(1^x), X_{i,j'}(2+3^y), NV_{i,j}(1+2+3+4^y)$$
$$(X \in \{NV, NV'\})$$
$$X_{i,j}(1+4^x), Y_{i,j'}(1^y+3^x), W_{i,j'}(2+4^y) \rightarrowtail X_{i,j}(1^z+4^x), Y_{i,j'}(1^y+3^x),$$
$$W_{i,j'}(2^u+4^y), X_{i,j}(1+2^z+3+4^u)$$
$$(X, W \in \{NV, NV'\},\ Y \in \{NV, NV'\})$$
$$NV_{i,j}(1+4^x), Y_{i,j'}(1^y+3^x), R_{i,j'}(2+4^y) \rightarrowtail NV_{i,j}(1^z+4^x), Y_{i,j'}(1^y+3^x),$$
$$R_{i,j'}(2^u+4^y), NV'_{i,j}(1+2^z+3+4^u)$$
$$(Y \in \{NV, NV'\})$$
$$NV'_{i,j}(1+4^x), Y_{i,j'}(1^y+3^x), R_{i,j'}(2+4^y) \rightarrowtail NV'_{i,j}(1^z+4^x), Y_{i,j'}(1^y+3^x),$$
$$R_{i,j'}(2^u+4^y), R_{i,j}(1+2^z+3+4^u)$$
$$(Y \in \{NV, NV'\})$$
$$R_{i,j}(1+4^x), Y_{i,j'}(1^y+3^x), R_{i,j'}(2+4^y) \rightarrowtail R_{i,j}(1^z+4^x), Y_{i,j'}(1^y+3^x),$$
$$R_{i,j'}(2^u+4^y), R_{i,j}(1+2^z+3+4^u)$$
$$(Y \in \{R, NV, NV'\})$$
$$R_{i,j}(1+4^x), Y_{i,j'}(1^y+3^x), X_{i,j'}(2+4^y) \rightarrowtail R_{i,j}(1^z+4^x), Y_{i,j'}(1^y+3^x),$$
$$X_{i,j'}(2^u+4^y), NV'_{i,j}(1+2^z+3+4^u)$$
$$(X \in \{NV, NV'\}, Y \in \{R, NV, NV'\})$$
$$R_{i,j}(1+4^x), Y_{i,j'}(1^y+3^x), Z_{i,j'}(2^y+3) \rightarrowtail R_{i,j}(1^z+4^x), Y_{i,j'}(1^y+3^x),$$
$$Z_{i,j'}(2^y+3^u), Z_{i,j}(1^u+2^z+3), P_{j+1}(1)$$
$$(Y \in \{R, NV, NV'\})$$
$$NV'_{i,j}(1+4^x), Y_{i,j'}(1^y+3^x), Z_{i,j'}(2^y+3) \rightarrowtail NV'_{i,j}(1^z+4^x), Y_{i,j'}(1^y+3^x),$$
$$Z_{i,j'}(2^y+3^u), Z_{i,j}(1^u+2^z+3), P_{j+1}(1)$$
$$(Y \in \{R, NV, NV'\})$$
$$NV_{i,j}(1+4^x), Y_{i,j'}(1^y+3^x), Z_{i,j'}(2^y+3) \rightarrowtail NV_{i,j}(1^z+4^x), Y_{i,j'}(1^y+3^x),$$
$$Z_{i,j'}(2^y+3^u), Z_{i,j}(1^u+2^z+3), P_l(1)$$
$$(Y \in \{R, NV, NV'\})$$

Figure 6.11: Encoding of decrement instructions $[\![j : \mathsf{DecJump}(R_i, l)]\!]_{\kappa^{-d-u}}$ in κ^{-d-u}.

Our last negative result is for the fragment $\kappa^{+ff\,-d\,-u}$.

Theorem 6.7 *SCP is undecidable in* $\kappa^{+ff\,-d\,-u}$.

Proof: We proceed as described in Theorem 6.6 assuming, without loss of generality as in Theorem 6.5, that the two registers are incremented at least once. The new construction adds rules that come into play in case the $Halt$ molecule is produced. In fact, the molecule $Halt$ triggers the "destruction" of the grids representing the registers: one molecule is produced for each end of each bond, and the bond is passed to such new molecule. Following this approach we know a priori the exact structure of the structures that will be available at the end of the computation in case of RAM termination. □

6.4 Related work

In this section we discuss some related works by first focusing on formal models specifically proposed for describing biological systems and then considering more generally the fields of term/graph rewriting and process calculi.

As we said in the Introduction, the closest work to this contribution is [33] where a syntactic restriction entailing a form of SCP is proposed. This restriction – κ with local rule sets – is orthogonal to the ones proposed in this chapter. It does not cover the reachability analysis of finite structures with recurrent patterns, such as finite polymers. In these cases, the analysis in [33] yields an over-approximation of the reachable complexes. How much reasonable is this over-approximation is not clear.

Apart from κ, the literature reports several proposals for describing (and reasoning on) biological systems, which use a variety of formal tools, including process calculi, term/graph rewriting, (temporal) logic, and rule based languages. However, the expressive power of most of these formalisms is the one of Petri nets. Therefore, the decidability of reachability and coverability problems is an immediate consequence of the corresponding results on Petri nets.

Formalisms whose expressive power is similar to κ, miss results analogous to those contained in this chapter. For example, the biochemical abstract machine Biocham

[53, 54] is a rule-based model similar to κ. However reactions are constrained to specify completely the reagent solution, unlike κ where reactions partially specify reactants and products. It is worth noticing that the Biocham constraint do not allow finite descriptions of rules creating polymers of arbitrary length. As a consequence, when considering purely qualitative aspects, i.e. removing kinetic quantities, the Biocham can be reduced to a classical Petri net [53].

Another rule-based model for describing and analysing biological processes is Pathway Logic [50, 123]. This model is based on rewrite logic, which allows to describe biological entities and their relations at different levels of abstractions and granularity by using elements of an algebraic data type (to describe states) and rewrite rules (to describe transitions between states and therefore behaviours). Even though Pathway Logic models of biological processes are developed in Maude system, which is Turing complete, yet the analysis of biological systems uses the, so called, Pathway Logic Assistant for representing models in terms of Petri Nets [123]. Therefore, also in this case, the relevant decidability results derive from the analogous results on Petri nets. This is the case also for the model used in [69].

A different model, based on graph transformation has been proposed by Blinov et al. [12]. However, in this case, the relevant properties (e.g. membership of a given species in a reaction network) are semi-decidable and we are not aware of suitable restrictions on the general model that ensure decidability for some of them.

As regards the fields of term/graph rewriting and process calculi, we have not find results from which we can derive immediately those we have obtained for κ. In particular, for term rewriting systems, the reductions to Petri net reachability can be applied to decide reachability for associative-commutative ground term rewriting (AC) [90] and for Process Rewrite Systems (PRS) [89]. However, AC and PRS are more expressive than Petri nets, but strictly less expressive than Turing machines [89]. On the other hand our positive results are given for fragments of κ that are Turing-complete. As such, the set of derivatives of a κ solution may not be a regular set of terms. Thus, decision procedures based on tree automata like those proposed in fragments of non-ground term rewriting [31, 35, 77, 112] cannot be applied to the

κ-lattice.

Decidability results for reachability in process calculi like Mobile Ambients, Boxed Ambients, and Bio-ambients are given in [13, 26, 27, 43, 132]. These results are obtained for fragments (or for weak semantics) that ensure the monotonicity of the generated ambient structures.

In addition they consider process calculi (Mobile/Boxed/Bio Ambients) which operate on tree-like structures and without fresh name generation. This contrasts with the dialect of κ of Figure 6.5, that operate on (possibly cyclic) graph-structures and admit dynamic creation of new names (bonds).

Concerning Graph Rewriting Systems (GRS) there exist folk theorems about reachability that state its undecidability in full-fledged GRS and its decidability for GRS in which rules do not add new nodes. We are not aware of (un)decidability results for decision problems like reachability and coverability in graph rewriting systems with features similar to those considered in our κ-lattice. The only specific results we are aware of are those given for reachability in context-free graph grammars [46] and for coverability in GRS that are *well-structured* with respect to the graph minor relation [81]. However, we consider here more general rules than those of context-free graph grammars. Furthermore, we do not see how to apply the decision procedure proposed in [81] to languages in the κ-lattice that, in general, do not enjoy strict compatibility with respect to the graph minor ordering.

6.5 Conclusions

We have investigated three decidability problems for several κ dialects. These problems allow one to check whether, starting from a given initial solution, a sequence of reactions described in the κ formalism produces a solution having some specific features. Hence our results, summarized in Figure 6.5, can be seen as a first step in the direction of qualitative analysis of κ calculus.

Besides presenting techniques for qualitative analysis, we also characterise the computational power of κ-like biologically inspired models. In this respect, the main

result is that we can remove bond and molecule destruction and the internal state of molecules from κ without losing Turing completeness (see the modelling of 2 Counter Machines presented in " the proof of Theorem 6.6). On the contrary, if we remove the possibility to test the presence of one bond in a reaction, the calculus is no longer Turing universal (see Theorem 6.3).

Our work can be extended along at least two lines. First, several other fragments of κ can be considered for a similar investigation. Notably `nano`κ that admits at most two reactants. In particular, our encoding of a 2CM into κ^{-d-i} uses ternary (at the left hand side) rules and we conjecture that a 2CM cannot be encoded faithfully into κ^{-d-i} with binary rules only.

Second, there are several other interesting properties to investigate, for example a form of coverability where one admits complexes strictly larger than the original ones. In this perspective, we plan to exploit the theory of well structured transition systems as done in [81] to prove decidability of coverability w.r.t. the graph minor relation in classes of graph rewriting systems.

Chapter 7

Concluding Remarks

Parlare oscuramente lo sa fare ognuno,
ma chiaro pochissimi.

Galileo Galilei
(Considerazioni al Tasso)

In this thesis we have studied four different approaches for studying expressiveness issues. We have chosen four languages with different aims and behaviours, and have shown how to apply the techniques in different environments.

We began with the analysis of a synchronous language $\mathrm{CCS}_!^{-\omega}$ and we have explored its expressiveness w.r.t. the existence of faithful encodings of grammars of types 1,2 and 3 in the Chomsky Hierarchy. The leitmotiv that guided us in the analysis was to study the expressiveness gap given by one source of non-determinism. Non-determinism is, indeed, one of the characteristic features of concurrent languages and the language we analysed is not the only one presenting such a behaviour. It would be certainly interesting to discuss different sources of non determinism in concurrency from the expressiveness point of view. Unfortunately this is not trivial as non-determinism is intrinsic to the language i.e. it is not introduced by a specific syntactic operator and therefore it cannot be isolated without introducing unnatural definitions.

We have then moved to an asynchronous language by analysing the full abstraction problem for two Linda-like languages: Linda-core and Linda-inp. We have first defined a trace-based denotational semantics and then we have obtained fully abstract semantics for both languages by using suitable saturations. Full abstraction was one of the techniques inherited from the λ-calculus and its use in concurrency is, nowadays, source of discussion (see for example the discussion in [10]). In fact, apart from the way we have applied it, it is sometimes used for proving correctness of encodings (or more generally, as a way of transporting behavioural equivalences from one language to another). In this context the main criticism is that full abstraction is too strong a tool for comparison and might rule out possible terms of the language that are instead recognised correct by tools as bisimulation or testing semantics.

Next we have returned to typical techniques for concurrent calculi: language encoding and decidability of properties. For the first one, we have chosen a language that is not generally considered as a concurrent calculus even if it presents some *parallel* characteristics. In fact, multiple heads of CHR can be viewed as a way of handling a group of atoms simultaneously. Indeed we show that this feature of the language increases the expressiveness: When considering generic constraint theories and under some rather reasonable assumptions it is not possible to encode CHR (with multi-headed rules) into CHR_1 while preserving the semantics of programs. Moreover we have also shown that restricting the number of atoms in the head of the rules generates a hierarchy of languages with increasing expressiveness.

Finally, we have analysed a language similar but simpler than CHR: the κ-calculus. By restricting the shape of the rewriting rules of the calculus, we have obtained several dialects. We have analysed their expressive power by focusing on the decidability and undecidability for problems like reachability and coverability. Apart from the hierarchy of languages we have obtained, such a technique can be interesting also because it can set the type of "questions" that are feasible for a language: for instance, does it make sense to check if a particular state of the system is reachable? Or reversing the problem one can ask whether there exists a

convenient restriction of a calculus such that a particular property is decidable.

In the next section we discuss some possible future works that can be tackled following the same lines of this dissertation.

7.1 Future developments

Possible future developments have already been discussed at the end of every chapter. Here we want to add some more general directions related to the idea of exploring expressiveness techniques for concurrent formalisms.

For instance, we have not mentioned one of the areas of concurrency with a strong impact on real systems: *Model Checking*. Model checking [30] is an automatic technique used for verifying concurrent systems, which has been efficiently implemented and it is currently used by industry in the design of (mostly) embedded systems. The idea is to analyse if a suitable model of the system satisfies given properties defined in a proper modal logic [86]: i.e. can the system reach a possibly dangerous state?

Such a technique can clearly be used as a reasoning technique. Its key elements are: (1) the representations of the systems (the model) and (2) a suitable logic for expressing interesting properties. Recently Delzanno and Gabbrielli [42] have considered a denotational semantics based on traces for a language similar to Linda and provide a compact representation of them by means of constraints. As such, their semantics resembles the one proposed in Chapter 4. This representation could represent a suitable model; therefore since the system is formalised using constraint logic programming one can think of defining a proper logic and thus testing properties on the system using a constraint solver. It would be interesting, then, to see if this proposal has a place in the model checking setting.

Another topic that has not been touched in this dissertation is the computational complexity of decidable properties, that is, the cost of deciding whether or not a certain property is satisfied by the system under analysis. This has been studied for example for bisimulation: Mayr in [88] shows that strong bisimulation of Basic

Parallel Processes is co-NP-hard or more generally Balcázar et al. [8] prove that bisimilarity over finite labelled transition systems is P-complete. Therefore it could be interesting to see if such analysis applies to properties like the ones studied in Chapter 6. Indeed, as mentioned above, knowing that a property is decidable can have some significance for other purposes. Thus being able to determine the computational complexity can help studying the feasibility of a given problem: i.e. if checking the reachability of a state is exponential in time means that in practice the problem is not tractable.

Related to complexity, one could also try something a bit more ambitious. In the last decade there has been a growing interest in finding proper restrictions to the λ-calculus in order to prove that all the definable functions belong to a given complexity class, for example polynomial time (see [72] for a survey on the subject). The idea would be to obtain something similar for concurrent languages. To this aim one should decide first which metric to use (since there is no agreement on how to calculate complexity of concurrent systems) and then try to find proper restrictions in order to obtain interesting classes of languages. Notice that one has also to define which the interesting problems are: for instance, instead of focusing on the number of steps one can think in terms of the number of communications needed to accomplish a certain action, therefore bounding the capacity of the transmissions means.

References

[1] Martín Abadi and Andrew D. Gordon. Reasoning about cryptographic protocols in the spi calculus. In *CONCUR*, volume 1243 of *Lecture Notes in Computer Science*, pages 59–73. Springer, 1997.

[2] Slim Abdennadher. Operational semantics and confluence of constraint propagation rules. In *Principles and Practice of Constraint Programming*, pages 252–266, 1997.

[3] Parosh Aziz Abdulla, Giorgio Delzanno, and Laurent Van Begin. Comparing the expressive power of well-structured transition systems. In Jacques Duparc and Thomas A. Henzinger, editors, *CSL*, volume 4646 of *Lecture Notes in Computer Science*, pages 99–114. Springer, 2007.

[4] Roberto M. Amadio, Ilaria Castellani, and Davide Sangiorgi. On bisimulations of the asynchronous π-calculus. *Theoretical Computer Science*, 195(2):291–324, 1998.

[5] Krzysztof R. Apt. *From logic programming to Prolog.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[6] Jesús Aranda, Cinzia Di Giusto, Mogens Nielsen, and Frank D. Valencia. CCS with replication in the chomsky hierarchy: The expressive power of divergence. In *APLAS*, volume 4807 of *Lecture Notes in Computer Science*. Springer, 2007.

[7] Jos C. M. Baeten and Flavio Corradini. Regular expressions in process algebra. In *LICS '05*, pages 12–19, Washington, DC, USA, 2005. IEEE Computer Society.

[8] José Balzacar, Joaquim Gabarro, and Miklos Santha. Deciding bisimilarity is P-complete. *Formal aspects of computing*, 4(6 A):638–648, 1992.

[9] Jean-Pierre Banâtre and Daniel Le Métayer. Programming by multiset transformation. *Communications of the ACM*, 36(1):98–111, 1993.

[10] Romain Beauxis, Catuscia Palamidessi, and Frank D. Valencia. On the asynchronous nature of the asynchronous pi-calculus. In *Concurrency, Graphs and Models*, volume 5065 of *LNCS*, pages 473–492. Springer, 2008.

[11] Jan A. Bergstra and Jan Willem Klop. Process algebra for synchronous communication. *Information and Control*, 60(1-3):109–137, 1984.

[12] Michael L. Blinov, Jin Yang, James R. Faeder, and William S. Hlavacek. Graph theory for rule-based modeling of biochemical networks. In *Transactions on Computational Systems Biology VII*, volume 4230 of *LNCS*, pages 89–106, 2006.

[13] Iovka Boneva and Jean-Marc Talbot. When ambients cannot be opened. *Theor. Comput. Sci.*, 333(1-2):127–169, 2005.

[14] Michele Boreale, Rocco De Nicola, and Rosario Pugliese. Trace and testing equivalence on asynchronous processes. *Information and Computation*, 172(2):139–164, 2002.

[15] Gérard Boudol. Asyncrony and the π-calculus. (note). Technical report, Rapport de Recherche 1702, INRIA, Sophia-Antipolis., 1992.

[16] Antonio Brogi and Jean-Marie Jaquet. Modeling coordination via asynchronous communication. In *Proceedings of the Second Int.l Conference on Coordination Languages and Models*, pages 238–255, London, UK, 1997. Springer-Verlag.

[17] Stephen D. Brookes. Full abstraction for a shared-variable parallel language. *Information and Computation*, 127(2):145–163, 1996.

[18] Olaf Burkart, Didier Caucal, Faron Moller, and Bernhard Steffen. *Verification on infinite structures*, chapter 9, pages 545–623. Elsevier, North-Holland, 2001.

[19] Nadia Busi, Maurizio Gabbrielli, and Gianluigi Zavattaro. Replication vs. recursive definitions in channel based calculi. In *Thirtieth International Colloquium on Automata, Languages and Programming (ICALP'03)*, volume 2719 of *Lecture Notes in Computer Science*, pages 133–144. Springer-Verlag, 2003.

[20] Nadia Busi, Maurizio Gabbrielli, and Gianluigi Zavattaro. Comparing recursion, replication, and iteration in process calculi. In *Thirtyfirst International Colloquium on Automata, Languages and Programming (ICALP'04)*, volume 3142 of *Lecture Notes in Computer Science*, pages 307–319. Springer-Verlag, 2004.

[21] Nadia Busi, Maurizio Gabbrielli, and Gianluigi Zavattaro. On the expressive power of recursion, replication, and iteration in process calculi. *To appear in Mathematical Structures in Computer Science*, 2008.

[22] Nadia Busi, Roberto Gorrieri, and Gianluigi Zavattaro. Three semantics of the output operation for generative communication. In *Proceedings of the Second Int.l Conference on Coordination Languages and Models*, pages 205–219, London, UK, 1997. Springer-Verlag.

[23] Nadia Busi, Roberto Gorrieri, and Gianluigi Zavattaro. A process algebraic view of linda coordination primitives. *Theoretical Computer Science*, 192(2):167–199, 1998.

[24] Nadia Busi, Roberto Gorrieri, and Gianluigi Zavattaro. On the Turing equivalence of Linda coordination primitives. *Theoretical Computer Science*, 230(1-2):260–261, 2000.

[25] Nadia Busi and Gianluigi Zavattaro. On the expressive power of movement and restriction in pure mobile ambients. *Theoretical Computer Science*, 322(3):477–515, 2004.

[26] Nadia Busi and Gianluigi Zavattaro. Deciding reachability in mobile ambients. In *ESOP*, volume 3444 of *Lecture Notes in Computer Science*, pages 248–262, 2005.

[27] Nadia Busi and Gianluigi Zavattaro. Reachability analysis in boxed ambients. In *ICTCS*, volume 3701 of *Lecture Notes in Computer Science*, pages 143–159, 2005.

[28] Luca Cardelli and Andrew D. Gordon. Mobile ambients. *Theoretical Computer Science*, 240(1):177–213, 2000.

[29] Luca Cardelli and Gianluigi Zavattaro. On the Computational Power of Biochemistry. In *Third International Conference on Algebraic Biology (AB'08)*, volume 5147 of *Lecture Notes in Computer Science*, pages 65–80. Springer-Verlag, 2008.

[30] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model checking*. MIT Press, Cambridge, MA, USA, 1999.

[31] Jean-Luc Coquidé, Max Dauchet, Rémi Gilleron, and Sándor Vágvölgyi. Bottom-up tree pushdown automata and rewrite systems. In *RTA*, volume 488 of *LNCS*, pages 287–298, 1991.

[32] Alberto Credi, Marco Garavelli, Cosimo Laneve, Sylvain Pradalier, Serena Silvi, and Gianluigi Zavattaro. Modelizations and simulations of nano devices in nanok calculus. In *CMSB 2007*, volume 4695 of *Lecture Notes in Computer Science*, pages 168–183, 2007.

[33] Vincent Danos, Jérôme Feret, Walter Fontana, and Jean Krivine. Abstract interpretation of cellular signalling networks. In *VMCAI*, volume 4905 of *LNCS*, pages 83–97, 2008.

[34] Vincent Danos and Cosimo Laneve. Formal molecular biology. *Theoretical Computer Science*, 325(1):69–110, 2004.

[35] Max Dauchet and Sophie Tison. The theory of ground rewrite systems is decidable. In *LICS*, pages 242–248, 1990.

[36] Frank S. de Boer, Maurizio Gabbrielli, and Maria Chiara Meo. A timed linda language and its denotational semantics. *Fundamenta Informaticae*, 63(4), 2004.

[37] Frank S. de Boer and Catuscia Palamidessi. A fully abstract model for concurrent constraint programming. In *Proceedings TAPSOFT/CAAP '91: vol 1*, pages 296–319, New York, NY, USA, 1991. Springer-Verlag New York, Inc.

[38] Frank S. de Boer and Catuscia Palamidessi. Embedding as a tool for language comparison. *Information and Computation*, 108(1):128–157, 1994.

[39] Rocco De Nicola, Gian Luigi Ferrari, and Rosario Pugliese. Klaim: A kernel language for agents interaction and mobility. *IEEE Transactions on Software Engineering*, 24(5):315–330, 1998.

[40] Rocco De Nicola and Matthew C. B. Hennessy. Testing equivalence for processes. In Josep Díaz, editor, *Automata, Languages and Programming, 10th Colloquium*, volume 154 of *Lecture Notes in Computer Science*, pages 548–560, Barcelona, Spain, 18–22 July 1983. Springer-Verlag.

[41] Giorgio Delzanno, Cinzia Di Giusto, Maurizio Gabbrielli, Cosimo Laneve, and Gianluigi Zavattaro. A qualitative analysis of formal molecular biology. Technical report, 2008.

[42] Giorgio Delzanno and Maurizio Gabbrielli. Compositional verification of asynchronous processes via constraint solving. In *ICALP*, volume 3580 of *Lecture Notes in Computer Science*, pages 1239–1250. Springer, 2005.

[43] Giorgio Delzanno and Roberto Montagna. On reachability and spatial reachability in fragments of bioambients. *Electronic Notes in Theoretical Computer Science*, 171(2):69–79, 2007.

[44] Cinzia Di Giusto and Maurizio Gabbrielli. Full abstraction for Linda. In *ESOP '08*, volume 4960 of *Lecture Notes in Computer Science*, pages 78–92. Springer, 2008.

[45] David L. Dill, Merrill Knapp, Pamela Gage, Carolyn L. Talcott, Keith Laderoute, and Patrick Lincoln. The pathalyzer: A tool for analysis of signal transduction pathways. In *SBRG*, volume 4023 of *LNCS*, pages 11–22. Springer, 2005.

[46] Frank Drewes, Hans-Jörg Kreowski, and Annegret Habel. Hyperedge replacement, graph grammars. In *Handbook of Graph Grammars*, pages 95–162. World Scientific, 1997.

[47] Gregory J. Duck, Peter J. Stuckey, Maria J. García de la Banda, and Christian Holzbaur. The refined operational semantics of constraint handling rules. In *ICLP*, Lecture Notes in Computer Science, pages 90–104, 2004.

[48] Catherine Dufourd, Alain Finkel, and Philippe Schnoebelen. Reset nets between decidability and undecidability. In Kim G. Larsen, Sven Skyum, and Glynn Winskel, editors, *Proceedings of the 25th International Colloquium on Automata, Languages and Programming (ICALP'98)*, volume 1443 of *Lecture Notes in Computer Science*, pages 103–115, Aalborg, Denmark, July 1998. Springer.

[49] James Dugundji and Andrzej Granas. *Fixed Point Theory*. Springer-Verlag New York, Inc., 2003.

[50] Steven Eker, Merrill Knapp, Keith Laderoute, Patrick Lincoln, José Meseguer, and M. Kemal Sönmez. Pathway logic: Symbolic analysis of biological signaling. In *Pacific Symposium on Biocomputing*, pages 400–412, 2002.

[51] Javier Esparza and Mogens Nielsen. Decidability Issues for Petri Nets-a Survey. *Bulletin of the European Association for Theoretical Computer Science*, 52:245–262, 1994.

[52] François Fages. Symbolic model-checking for biochemical systems. In *ICLP*, volume 2916 of *LNCS*, page 102. Springer, 2003.

[53] François Fages and Sylvain Soliman. Formal cell biology in biocham. In *SFM*, volume 5016 of *LNCS*, pages 54–80, 2008.

[54] Francois Fages, Sylvain Sollman, and Nathalie Chabrier-Rivier. Modelling and querying interaction networks in the biochemical abstract machine biocham. *Journal of Biological Physics and Chemistry*, 4:64–73, 2004.

[55] Alain Finkel and Philippe Schnoebelen. Well-structured transition systems everywhere! *Theoretical Computer Science*, 256(1-2):63–92, 2001.

[56] Thomas Forster. *Logic, Induction and Sets*. Cambridge University Press, 2003.

[57] Cédric Fournet and Georges Gonthier. The reflexive cham and the join-calculus. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 372–385, New York, NY, USA, 1996. ACM.

[58] Eric Freeman, Ken Arnold, and Susanne Hupfer. *JavaSpaces Principles, Patterns, and Practice*. Addison-Wesley Longman Ltd., Essex, UK, UK, 1999.

[59] Thom W. Frühwirth. Introducing simplification rules. Technical report, 1991.

[60] Thom W. Frühwirth. Theory and practice of constraint handling rules. *Journal of Logic Programming*, 37(1-3):95–138, 1998.

[61] Thom W. Frühwirth. As time goes by II: More automatic complexity analysis of concurrent rule programs. *Electronic Notes in Theoretical Computer Science*, 59(3), 2001.

[62] Thom W. Frühwirth. As time goes by: Automatic complexity analysis of simplification rules. In *8th International Conference on Principles of Knowledge Representation and Reasoning, Toulouse, France*, 2002.

[63] David Gelernter. Generative communication in linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

[64] David Gelernter and Nicholas Carriero. Coordination languages and their significance. *Communications of the ACM*, 35(2):96, 1992.

[65] Pablo Giambiagi, Gerardo Schneider, and Frank D. Valencia. On the expressiveness of infinite behavior and name scoping in process calculi. In *FoSSaCS 2004*, pages 226–240, 2004.

[66] Cinzia Di Giusto, Maurizio Gabbrielli, and Maria Chiara Meo. Expressiveness of multiple heads in chr. In *SOFSEM 2009*, volume 5404 of *Lecture Notes in Computer Science*, pages 205–216. Springer, 2009.

[67] Daniele Gorla. Comparing communication primitives via their relative expressive power. *Information and Computation*, 206(8):931–952, 2008.

[68] Daniele Gorla. Towards a unified approach to encodability and separation results for process calculi. In *CONCUR*, volume 5201 of *Lecture Notes in Computer Science*, pages 492–507. Springer, 2008.

[69] Monika Heiner, David Gilbert, and Robin Donaldson. Petri nets for systems and synthetic biology. In *SFM*, volume 5016 of *LNCS*, pages 215–264, 2008.

[70] Matthew C. B. Hennessy and Gordon Plotkin. Full abstraction for a simple parallel programming language. In J. Bečvář, editor, *Mathematical Foundations of Computer Science 1979*, volume 74 of *Lecture Notes in Computer Science*, pages 108–120, Olomouc, Czechoslovakia, 3–7 September 1979. Springer-Verlag.

[71] Charles Antony Richard Hoare. *Communicating Sequential Processes*. Prentice Hall International Series in Computer Science, 1985.

[72] Martin Hofmann. Programming languages capturing complexity classes. *SIGACT News*, 31(1):31–42, 2000.

[73] Kohei Honda and Mario Tokoro. An object calculus for asynchronous communication. In *ECOOP*, volume 512 of *Lecture Notes in Computer Science*, pages 133–147, 1991.

[74] Eiichi Horita, Jan W. de Bakker, and Jan J. M. M. Rutten. Fully abstract denotational models for nonuniform concurrent languages. *Information and Computation*, 115(1):125–178, 1994.

[75] Hans Huttel and Jiri Srba. Recursion vs. replication in simple cryptographic protocols. In *SOFSEM'05*, volume 3381 of *Lecture Notes in Computer Science*, pages 175–184. Springer-Verlag, 2005.

[76] IBM. Tspaces. `http://www.almaden.ibm.com/cs/TSpaces/index.html`.

[77] Florent Jacquemard. Decidable approximations of term rewriting systems. In *RTA*, volume 1103 of *LNCS*, pages 362–376, 1996.

[78] Alan Jeffrey and Julian Rathke. Java jr. : Fully abstract trace semantics for a core java language. volume 3444 of *Lecture Notes in Computer Science*, pages 423–438. Springer-Verlag, 2005.

[79] Alan Jeffrey and Julian Rathke. Full abstraction for polymorphic pi-calculus. *Theoretical Computer Science*, 2007. To appear.

[80] Bengt Jonsson. A model and proof system for asynchronous networks. In *Proceedings of the fourth annual ACM symposium on Principles of distributed computing*, pages 49–58, New York, NY, USA, 1985. ACM Press.

[81] Salil Joshi and Barbara König. Applying the graph minor theorem to the verification of graph transformation systems. In *CAV*, volume 5123 of *LNCS*, pages 214–226, 2008.

[82] Paris C. Kanellakis and Scott A. Smolka. CCS expressions finite state processes, and three problems of equivalence. *Information and Computation*, 86(1):43–68, 1990.

[83] Antonín Kučera and Petr Jančar. Equivalence-checking on infinite-state systems: Techniques and results. *Theory and Practice of Logic Programming*, 6(3):227–264, 2006.

[84] Ivan Lanese, Jorge A. Pérez, Davide Sangiorgi, and Alan Schmitt. On the expressiveness and decidability of higher-order process calculi. In *LICS '08: Proceedings of the 2008 23rd Annual IEEE Symposium on Logic in Computer Science*, pages 145–155, Washington, DC, USA, 2008. IEEE Computer Society.

[85] Cosimo Laneve and Antonio Vitale. Expressivity in the k family. In *MFPS XXIV*, 2008.

[86] Zohar Manna and Amir Pnueli. *The temporal logic of reactive and concurrent systems*. Springer-Verlag New York, Inc., New York, NY, USA, 1992.

[87] C. Mao, W. Sun, and N.C. Seeman. Assembly of Borromean rings from DNA. *Nature*, 386(6621):137 –138, 1997.

[88] Richard Mayr. On the Complexity of Bisimulation Problems for Basic Parallel Processes. *Lecture Notes in Computer Science*, pages 329–341, 2000.

[89] Richard Mayr. Process rewrite systems. *Inf. Comput.*, 156(1-2):264–286, 2000.

[90] Richard Mayr and Michael Rusinowitch. Reachability is decidable for ground ac rewrite systems. In *Infinity '98*, pages 53–64, 1998.

[91] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980.

[92] Robin Milner. A complete inference system for a class of regular behaviours. *Journal of Computer and System Sciences*, 28(3):439–466, 1984.

[93] Robin Milner. *Communication and concurrency*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, 1989.

[94] Robin Milner. *Communicating and mobile systems: the π-calculus*. Cambridge University Press, New York, NY, USA, 1999.

[95] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, part I and II. *Information and Computation*, 100(1):1–40 and 41–77, 1992.

[96] Marvin Minsky. *Computation: finite and infinite machines*. Prentice Hall, 1967.

[97] James H. Morris. *Lambda-Calculus Models of Programming Languages*. PhD thesis, MIT, 1968.

[98] Uwe Nestmann. Welcome to the jungle: A subjective guide to mobile process calculi. In Christel Baier and Holger Hermanns, editors, *CONCUR*, volume 4137 of *Lecture Notes in Computer Science*, pages 52–63. Springer, 2006.

[99] Mogens Nielsen, Catuscia Palamidessi, and Frank D. Valencia. On the expressive power of concurrent constraint programming languages. In *PPDP 2002*, pages 156–167. ACM Press, October 2002.

[100] Catuscia Palamidessi. Comparing the expressive power of the synchronous and asynchronous *pi*-calculi. *Mathematical Structures in Computer Science*, 13(5):685–719, 2003.

[101] Catuscia Palamidessi and Frank D. Valencia. Recursion vs replication in process calculi: Expressiveness. *Bulletin of the EATCS*, 87:105–125, 2005.

[102] I.G. Panyutin and P. Hsieh. The kinetics of spontaneous DNA branch migration. *Proc. National Academy of Science USA*, 91(6):2021–2025, 1994.

[103] David Park. Concurrency and automata on infinite sequences. In *Proceedings of the 5th GI-Conference on Theoretical Computer Science*, pages 167–183, London, UK, 1981. Springer-Verlag.

[104] Joachim Parrow. Trios in concert. In Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*, pages 621–637. MIT Press, 2000.

[105] Joachim Parrow. Expressiveness of process algebras. *Electronic Notes in Theoretical Computer Science*, 209:173–186, 2008.

[106] Mor Peleg, Iwei Yeh, and Russ B. Altman. Modelling biological processes using workflow and petri net models. *Bioinformatics*, 18(6):825–837, 2002.

[107] Andrew Phillips and Luca Cardelli. Efficient, correct simulation of biological processes in the stochastic pi-calculus. In *Computational Methods in Systems Biology*, volume 4695 of *Lecture Notes in Computer Science*, pages 184–199. Springer, September 2007.

[108] Benjamin C. Pierce and Davide Sangiorgi. Behavioral equivalence in the polymorphic pi-calculus. In *Proceedings of the 24th ACM SIGPLAN-SIGACT*, pages 242–255, New York, NY, USA, 1997. ACM Press.

[109] Gordon Plotkin. Lcf considered as a programming language. *Theoretical Computer Science*, (5):223–255, 1977.

[110] Damien Pous. Using bisimulation proof techniques for the analysis of distributed abstract machines. *Theoretical Computer Science*, 402(2-3):199–220, 2008.

[111] Wolfgang Reisig. *Petri nets: an introduction*. Springer-Verlag New York, Inc., New York, NY, USA, 1985.

[112] Kai Salomaa. Deterministic tree pushdown automata and monadic tree rewriting systems. *J. Comput. Syst. Sci.*, 37(3):367–394, 1988.

[113] Davide Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms*. PhD thesis CST–99–93, Department of Computer Science, University of Edinburgh, 1992.

[114] Davide Sangiorgi. Bisimulation for higher-order process calculi. *Information and Computation*, 131(2):141–178, 1996.

[115] Davide Sangiorgi. π-calculus, internal mobility, and agent-passing calculi. *Theoretical Computer Science*, 167(1-2):235–274, 1996.

[116] Davide Sangiorgi. A theory of bisimulation for the π-calculus. *Acta Informatica*, 33(1):69–97, 1996.

[117] Davide Sangiorgi. On the bisimulation proof method. *Mathematical Structures in Computer Science*, 8(5):447–479, 1998.

[118] Davide Sangiorgi. On the origins of bisimulation and coinduction. *To appear in TOPLAS*, 2008.

[119] Davide Sangiorgi and David Walker. *PI-Calculus: A Theory of Mobile Processes*. Cambridge University Press, New York, NY, USA, 2001.

[120] Ehud Y. Shapiro. The family of concurrent logic programming languages. *ACM Comput. Surv.*, 21(3):413–510, 1989.

[121] Jon Sneyers, Tom Schrijvers, and Bart Demoen. The computational power and complexity of Constraint Handling Rules. In T. Schrijvers and T. Frühwirth, editors, *Proceedings of the 2nd Workshop on Constraint Handling Rules (CHR'05)*, number CW 421 in Dept. Computer Science, Technical report, pages 3–17, Sitges, Spain, October 2005.

[122] Joseph Stoy. *Denotational semantics: The Scott-Strachey approach to programming language theory*. MIT Press, 1977.

[123] Carolyn L. Talcott. Pathway logic. In *SFM*, volume 5016 of *LNCS*, pages 21–53, 2008.

[124] Frits W. Vaandrager. Expressive results for process algebras. In *Proceedings of the REX Workshop on Sematics: Foundations and Applications*, pages 609–638, London, UK, 1993. Springer-Verlag.

[125] Maria Grazia Vigliotti, Iain Phillips, and Catuscia Palamidessi. Tutorial on separation results in process calculi via leader election problems. *Theoretical Computer Science*, 388(1-3):267–289, 2007.

[126] J.D. Watson and F.H.C. Crick. A Structure for Deoxyribose Nucleic Acid. *Nature*, 171:737–738, 1953.

[127] George Wells, Peter Clayton, and Alan G. Chalmers. A Comparison of Linda Implementations in Java. In Peter H. Welch and Andrè W. P. Bakkers, editors, *Communicating Process Architectures 2000*, pages 63–76, sep 2000.

[128] G.M. Whitesides, J.P. Mathias, and C.T. Seto. Molecular self-assembly and nanochemistry – a chemical strategy for the synthesis of nanostructures. *Science*, 254:1312–1319, 1991.

[129] P. Yin, M. Harry, M.T. Choi, R. Colby, R. Calvert, and N.A. Pierce. Programming biomolecular self-assembly pathways. *Nature*, 451:318–322, 2008.

[130] Gianluigi Zavattaro. On the incomparability of gamma and linda. Technical report, Amsterdam, The Netherlands, 1998.

[131] Gianluigi Zavattaro. Towards a hierarchy of negative test operators for generative communication. *Electronic Notes in Theoretical Computer Science*, 16(2), 1998.

[132] Gianluigi Zavattaro. Reachability analysis in bioambients. *Electron. Notes Theor. Comput. Sci.*, 227:179–193, 2009.

[133] Gianluigi Zavattaro and Luca Cardelli. Termination problems in chemical kinetics. In *CONCUR*, volume 5201 of *Lecture Notes in Computer Science*, pages 477–491, 2008.