A Logic-Based Approach to Model Checking of Parameterized and Infinite-State Systems

by

Marco Bozzano
A Logic-Based Approach to Model Checking of Parameterized and Infinite-State Systems

by

Marco Bozzano

January, 2002
Dottorato di Ricerca in Informatica
Dipartimento di Informatica e Scienze dell’Informazione
Università degli Studi di Genova

DISI, Univ. di Genova
via Dodecaneso 35
I-16146 Genova, Italy
http://www.disi.unige.it/

Ph.D. Thesis in Computer Science

Submitted by Marco Bozzano
DISI, Univ. di Genova
bozzano@disi.unige.it

Date of submission: January 2002

Title: A Logic-Based Approach to Model Checking of Parameterized and Infinite-State Systems

Advisors: Maurizio Martelli
DISI, Univ. di Genova
martelli@disi.unige.it

Giorgio Delzanno
DISI, Univ. di Genova
giorgio@disi.unige.it

Ext. Reviewers: Jean-Marc Andreoli
Xerox Research Centre Europe, France
Jean-Marc.Andreoli@xrce.xerox.com

Maurizio Gabbielli
Dipartimento di Scienze dell’Informazione
Univ. di Bologna, Italy
gabri@cs.unibo.it

Frank Pfenning
Department of Computer Science
Carnegie Mellon University, U.S.A.
fp@cs.cmu.edu
Abstract

The aim of this thesis is to investigate the problem of verification for concurrent systems. In particular, a major problem in verification is that of validating systems, e.g. protocols, which are parametric, in the sense that the number of entities taking part in a given run is not fixed a priori. Typically, such kind of systems are also infinite-state, in that they use data structures containing possibly unbounded data values.

In this thesis we tackle the problem of verification using a logical approach. In particular, the leading thread of this work will be a specification language based on a fragment of Girard’s linear logic, which we will show to have direct connections with classical formalisms like high-level nets or rewriting. By combining the power of logical connectives with the flexibility of rewriting, we are able to nicely model local and global transitions, and to elegantly express new data generation. Reasoning on heterogeneous domains can also be achieved via specialized constraint solvers.

We show how this language can be used both for the specification and the analysis of parametric systems. In particular, we present a verification procedure which resembles classical symbolic model checking algorithms for infinite-state systems, and is well-suited to study system properties like safety, e.g. mutual exclusion. Technically, our verification procedure uses a fixpoint computation strategy which is based on a new bottom-up semantics for a fragment of linear logic. We illustrate our methodology presenting different examples coming from concurrency theory, like a parameterized version of the ticket mutual-exclusion protocol, and from security, like authentication protocols.
To my family
Ilza, I’m no good at being noble, but it doesn’t take much to see that the problems of three little people don’t amount to a hill of beans in this crazy world. Someday you’ll understand that. Now, now . . . Here’s looking at you, kid.

(H. Bogart as Rick Blaine in Casablanca (1942))
Acknowledgements

I am very grateful to the people that helped me during all these years. First of all, I want to thank my supervisor (and advisor), Maurizio Martelli, for his precious advice and support, and especially for his friendly guidance during the four years of my PhD program. I would also like to thank Eugenio Moggi (the coordinator of my PhD program), Giuseppe Rosolini and all my colleagues at DISI.

I wish to thank, in particular, Catherine Meadows and Frank Pfenning for giving me the opportunity to spend some months at the Naval Research Laboratory, in Washington D.C., and at Carnegie Mellon University, in Pittsburgh. Special thanks to Iliano Cervesato, Catherine Meadows and all the staff at NRL, including Paul Syverson, Ira Moskowitz, and Li Wu Chang. A special greeting to Matteo Pradella. I am grateful to Frank Pfenning for giving me the opportunity to visit one among the best computer science departments in the world, and for the several discussions I had with him. Special thanks and greetings to Jeff Polakow, David Walker, Andreas Abel, Brigitte Pientka, and also to the Italian community in Pittsburgh. I also wish to acknowledge some fruitful discussions I had with Jack Minker, Dale Miller and Catuscia Palamidessi.

I would also like to thank the reviewers of this thesis, Jean-Marc Andreoli, Maurizio Gab- brielli, and Frank Pfenning, for their helpful and pertinent comments and suggestions.

Finally, I wish to express my deepest gratitude to Giorgio Delzanno, who has also been advisor of this thesis, for his continuous and invaluable advice. Giorgio introduced me to the literature on verification and model checking, and shared with me the work of the last four years. I’ve been knowing Giorgio since the time I was an undergraduate and I really appreciated working with him. I must thank him for teaching me to be more confident in myself and a bit more enthusiastic in my daily work. It’s probably not excessive to say that the present thesis would have never been conceivable without his help and support.
# Table of Contents

List of Figures 6

List of Tables 9

Chapter 1 Introduction 10

1.1 Motivations .................................................. 10
1.2 Contribution of the Thesis .................................. 11
1.3 Plan of the Thesis .................................................. 15
1.3.1 Structure of the Thesis .................................. 16

Chapter 2 Preliminaries 18

2.1 Functions and Orders ........................................... 18
2.2 Multisets ...................................................... 19
2.3 Signatures and Algebras ..................................... 20
2.4 Substitutions and Multiset Unifiers ......................... 20
2.5 Basics of Logic Programming ................................ 21

I Linear Logic as a Unifying View of Concurrency 24

Chapter 3 Linear Logic 25

3.1 Full Linear Logic ............................................... 25
3.1.1 Programming in Linear Logic ............................ 27
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2</td>
<td>The Language LO</td>
<td>29</td>
</tr>
<tr>
<td>3.3</td>
<td>Extensions of LO</td>
<td>34</td>
</tr>
<tr>
<td>3.3.1</td>
<td>LO with $\mathbf{1}$ and $\otimes$</td>
<td>34</td>
</tr>
<tr>
<td>3.3.2</td>
<td>LO with Universal Quantification</td>
<td>36</td>
</tr>
<tr>
<td></td>
<td><strong>Chapter 4</strong> LO, Multiset Rewriting and Petri Nets</td>
<td>40</td>
</tr>
<tr>
<td>4.1</td>
<td>Multiset Rewriting over Atomic Formulas</td>
<td>40</td>
</tr>
<tr>
<td>4.2</td>
<td>Place/Transition Nets</td>
<td>41</td>
</tr>
<tr>
<td>4.2.1</td>
<td>A Producer/Consumer Example</td>
<td>43</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Place Invariants</td>
<td>44</td>
</tr>
<tr>
<td>4.3</td>
<td>From Petri Nets to Linear Logic</td>
<td>46</td>
</tr>
<tr>
<td>4.4</td>
<td>Petri Nets with Transfer Arcs</td>
<td>48</td>
</tr>
<tr>
<td>4.4.1</td>
<td>A Readers/Writers Example</td>
<td>49</td>
</tr>
<tr>
<td>4.4.2</td>
<td>Broadcast Protocols</td>
<td>50</td>
</tr>
<tr>
<td></td>
<td><strong>Chapter 5</strong> First-Order LO, First-Order Multiset Rewriting and Coloured Petri Nets</td>
<td>53</td>
</tr>
<tr>
<td>5.1</td>
<td>Multiset Rewriting Systems over First-Order Atomic Formulas</td>
<td>53</td>
</tr>
<tr>
<td>5.2</td>
<td>CP-nets</td>
<td>55</td>
</tr>
<tr>
<td>5.2.1</td>
<td>An Example: the Dining Philosophers</td>
<td>57</td>
</tr>
<tr>
<td>5.3</td>
<td>From CP-nets to Linear Logic</td>
<td>58</td>
</tr>
<tr>
<td>5.4</td>
<td>Multiset Rewriting with Universal Quantification</td>
<td>61</td>
</tr>
<tr>
<td></td>
<td><strong>II</strong> Model Checking for Linear Logic Specifications</td>
<td>65</td>
</tr>
<tr>
<td></td>
<td><strong>Chapter 6</strong> A Backward Approach to Model Checking</td>
<td>66</td>
</tr>
<tr>
<td>6.1</td>
<td>Transition Systems and Verification</td>
<td>67</td>
</tr>
<tr>
<td>6.1.1</td>
<td>Structured Transition Systems</td>
<td>69</td>
</tr>
<tr>
<td>6.1.2</td>
<td>Symbolic Verification</td>
<td>73</td>
</tr>
</tbody>
</table>
6.2 Ensuring Termination ........................................ 76
6.2.1 The Theory of Well-Quasi-Orderings ............... 76
6.2.2 Well-Structured Transition Systems .................. 78
6.3 Verification in Linear Logic ................................. 79
6.4 Related Work .............................................. 80

Chapter 7 Bottom-Up Evaluation of Propositional LO Programs 82
7.1 A Bottom-Up Semantics for LO ......................... 83
7.2 An Effective Semantics for LO ......................... 88
7.3 Examples .................................................. 96
7.3.1 A Producer/Consumer Example .................... 97
7.3.2 A Petri Net for Mutual Exclusion ................. 99
7.4 Related Work .............................................. 102

Chapter 8 Refining LO Resource Management: LO with 1 104
8.1 A Bottom-Up Semantics for LO 1 ....................... 105
8.2 ConstraintSemantics for LO 1 ......................... 108
8.3 Bottom-Up Evaluation for LO 1 ....................... 114
8.4 An Example: Readers/Writers ......................... 117
8.4.1 Verification of the Readers/Writers Protocol .... 120
8.5 Related Work .............................................. 122

Chapter 9 Reasoning on Specialized Domains: LO with Constraints 124
9.1 Enriching LO With Constraints ......................... 125
9.1.1 Constraint Systems .................................. 125
9.1.2 The Language LO(C) ................................ 127
9.2 An Effective Semantics for LO(C) ................. 129
9.3 Ensuring Termination .................................. 142
9.3.1 A Dynamic Abstraction from DC to NC ........... 147
9.4 An Example: The Ticket Protocol .......................... 148
  9.4.1 Specifying the Ticket Protocol ......................... 149
  9.4.2 Verifying the Ticket Protocol .......................... 151
9.5 Related Work ............................................. 154

Chapter 10 Bottom-Up Evaluation of First-Order LO Programs 157
  10.1 A Proof-system for LO_\forall .............................. 158
  10.2 A Bottom-Up Semantics for LO_\forall ...................... 161
  10.3 An Effective Semantics for LO_\forall ..................... 167
  10.4 Ensuring Termination ..................................... 181
  10.5 An Example: A Distributed Test-and-Lock Protocol ........ 183
  10.6 Related Work ............................................. 187

Chapter 11 A Case-Study: Security Protocols 189
  11.1 Introduction ............................................. 190
  11.2 Some Background on Authentication ....................... 191
    11.2.1 Cryptographic Prerequisites .......................... 192
    11.2.2 A Classification of Attacks ......................... 193
    11.2.3 The Dolev-Yao Intruder Model ....................... 193
    11.2.4 An Informal Protocol Notation ...................... 194
  11.3 Specifying Authentication Protocols .................... 196
  11.4 Examples ................................................ 198
    11.4.1 Millen’s \textit{ffgg} Protocol ........................ 198
    11.4.2 The Needham-Schroeder Protocol .................... 202
    11.4.3 Corrected Needham-Schroeder ....................... 205
    11.4.4 The Otway-Rees Protocol ............................ 206
    11.4.5 Experimental Results ............................... 209
  11.5 Future Work ............................................. 210
11.6 Related Work .............................................. 211

Chapter 12 Conclusions ....................................... 214
  12.1 Future Work ........................................... 215

Bibliography ...................................................... 218

Appendix A Verification Tool .................................. 233
  A.1 General Description ..................................... 233
  A.2 Implementation Notes ................................... 234
  A.3 How It Works ............................................ 234
  A.4 Experimental Environment ............................ 236
  A.5 Future Work ............................................ 236
# List of Figures

1.1 A simple Petri net representing a semaphore .................................. 12
3.1 A one-sided, dyadic proof system for full linear logic ...................... 27
3.2 A proof system for LO ................................................................. 32
3.3 Examples of LO proofs ............................................................... 33
3.4 A proof system for LO$_1$ ............................................................. 34
3.5 Examples of LO$_1$ proof ............................................................. 36
3.6 A proof system for LO$_\forall$ ......................................................... 37
3.7 An example of LO$_\forall$ proof ...................................................... 38
4.1 A producer/consumer net .............................................................. 43
4.2 Matrix and initial marking representation ....................................... 45
4.3 Reachability as provability in propositional LO ............................ 48
4.4 A readers/writers net ................................................................. 49
4.5 A readers/writers broadcast protocol ........................................... 51
5.1 The dining philosophers net ......................................................... 57
5.2 Reachability as provability in first-order LO .................................. 60
5.3 Reachability as provability in LO$_\forall$ ........................................ 63
6.1 An abstract verification procedure for reachability ......................... 69
6.2 Illustrating the covering problem: a Petri net for mutual exclusion ...... 70
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.3</td>
<td>A verification procedure for covering</td>
<td>73</td>
</tr>
<tr>
<td>7.1</td>
<td>A proof system for propositional LO</td>
<td>84</td>
</tr>
<tr>
<td>7.2</td>
<td>Symbolic fixpoint computation</td>
<td>94</td>
</tr>
<tr>
<td>7.3</td>
<td>Another example of LO derivation</td>
<td>95</td>
</tr>
<tr>
<td>7.4</td>
<td>Fixpoint computed for the producer/consumer example</td>
<td>97</td>
</tr>
<tr>
<td>7.5</td>
<td>A petri net for mutual exclusion: example trace</td>
<td>99</td>
</tr>
<tr>
<td>7.6</td>
<td>Fixpoint computed for the mutual exclusion example</td>
<td>100</td>
</tr>
<tr>
<td>7.7</td>
<td>Fixpoint computation for the mutual exclusion example: first step</td>
<td>101</td>
</tr>
<tr>
<td>8.1</td>
<td>A proof system for propositional LO&lt;sub&gt;1&lt;/sub&gt;</td>
<td>105</td>
</tr>
<tr>
<td>8.2</td>
<td>Symbolic fixpoint computation for an LO&lt;sub&gt;1&lt;/sub&gt; program</td>
<td>115</td>
</tr>
<tr>
<td>8.3</td>
<td>Invalidation phase for the readers/writers example</td>
<td>119</td>
</tr>
<tr>
<td>8.4</td>
<td>Fixpoint computed using invariant strengthening for the readers/writers protocol</td>
<td>121</td>
</tr>
<tr>
<td>9.1</td>
<td>A proof system for LO(C)</td>
<td>129</td>
</tr>
<tr>
<td>9.2</td>
<td>High-level description of the ticket protocol</td>
<td>149</td>
</tr>
<tr>
<td>9.3</td>
<td>A multi-client, single-server version of the ticket protocol</td>
<td>150</td>
</tr>
<tr>
<td>9.4</td>
<td>A multi-client, multi-server version of the ticket protocol</td>
<td>151</td>
</tr>
<tr>
<td>9.5</td>
<td>Multi-client ticket protocol: example trace</td>
<td>152</td>
</tr>
<tr>
<td>10.1</td>
<td>A proof system for non ground semantics of LO&lt;sub&gt;∀&lt;/sub&gt;</td>
<td>160</td>
</tr>
<tr>
<td>10.2</td>
<td>A trace violating mutual exclusion for the test-and-lock protocol</td>
<td>184</td>
</tr>
<tr>
<td>10.3</td>
<td>A test-and-lock protocol: example trace</td>
<td>184</td>
</tr>
<tr>
<td>10.4</td>
<td>Fixpoint computed for the test-and-lock protocol</td>
<td>185</td>
</tr>
<tr>
<td>10.5</td>
<td>Fixpoint computed using invariant strengthening for the test-and-lock protocol</td>
<td>186</td>
</tr>
<tr>
<td>11.1</td>
<td>Specification of the &lt;i&gt;ffgg&lt;/i&gt; protocol</td>
<td>199</td>
</tr>
<tr>
<td>11.2</td>
<td>Intruder theory for the &lt;i&gt;ffgg&lt;/i&gt; protocol</td>
<td>200</td>
</tr>
</tbody>
</table>

7
11.3 A parallel session attack to the ffgg protocol ........................................ 201
11.4 Specification of the Needham-Schroeder protocol ................................... 203
11.5 Intruder theory for the Needham-Schroeder protocol .............................. 204
11.6 An attack to the Needham-Schroeder protocol ...................................... 204
11.7 Specification of the Otway-Rees protocol .............................................. 207
11.8 Intruder theory for the Otway-Rees protocol ........................................ 208
11.9 An attack to the Otway-Rees protocol .................................................. 209

A.1 Pseudo-algorithm for bottom-up evaluation ............................................. 235
List of Tables

3.1 The set of linear logic connectives ........................................ 26

6.1 A parallel between reachability problems and linear logic semantics .... 79

9.1 Validating the ticket protocol: experimental results ....................... 153

11.1 Analysis of authentication protocols: experimental results ............... 209

12.1 A summary of the experiments carried out in the thesis ................. 216
Chapter 1

Introduction

The contribution of this thesis is twofold. First of all, we present a specification language, based on a fragment of Girard’s linear logic, which is suitable for the specification of concurrent systems, and is closely related to well-known formalisms like Petri nets, high-level Petri nets or rewriting systems. The core of the language provides logical primitives to express, e.g., communication by rendezvous, broadcast, and fresh name generation, and can be extended with specialized constraint solvers to reason on different data domains. Furthermore, we enrich this language with a novel computational model based on an alternative bottom-up semantics for linear logic programs, which we show to have applications for the verification of parameterized and infinite-state systems.

We have obtained some original results, both theoretical and practical. From the theoretical side, we have isolated fragments of first order linear logic for which provability is decidable. From the practical side, we have derived an automated verification procedure for an interesting class of safety properties, and some novel results concerning the validation of parameterized systems.

1.1 Motivations

In recent years several attempts have been made in order to develop tools for the automated verification of concurrent and distributed systems. Specifically, one of the more challenging research areas is related to the verification of parameterized systems. By parameterized system, we mean any collection of an arbitrary but finite number of components interacting via synchronous or asynchronous communication. Examples of parameterized systems naturally arise in different areas of computer science like, e.g., operating systems, network computing, or web applications. For instance, communication and authentication protocols
are a key ingredient for applications involving remote access or e-commerce.

Very often, most of these applications are highly critical. They must comply with a number of security requirements, and therefore their design and implementation is always difficult and error-prone. As a meaningful example, we could mention the case of Needham-Schroeder authentication protocol [NS78], an apparently very simple protocol which only recently (seventeen years after its publication) has been shown to be flawed [Low95]. For these reasons, the availability of possibly automatic tools for the formal specification and analysis of such kind of systems seems to be indispensable.

Some interesting results have been obtained for the verification of the class of parameterized systems with finite-state components [ACJT96, FS01, PRZ01, APR+01]. In many practical cases, verification problems for this kind of systems can be reduced to problems on Petri nets, by applying a counting abstraction that simply forgets local data while keeping track of the number of processes in a given state. The original properties can then be verified on the resulting model by means of reachability procedures [GS92, EN98, EFM99, Del00]. However, very often such kind of systems are also infinite-state, in that they use data structures containing possibly unbounded data values. Some more results have been obtained for the verification of classes of infinite-state systems consisting of a fixed number of components with possibly unbounded data. As an example, in [BGP97, Del00, DP99, Fri99], constraints are used to symbolically represent and manipulate infinite collections of states. The new frontier seems therefore to be the verification of systems which are both parametric and infinite-state.

1.2 Contribution of the Thesis

In this thesis we will present a uniform logical environment for the specification and analysis of concurrent systems. In particular, we will isolate a fragment of Girard’s linear logic [Gir87] corresponding to the language LO as defined in [AP91b] and some extensions of its. We will show that this language has close connections with classical formalisms in concurrency theory like (coloured) Petri nets or multiset rewriting systems over atomic formulas. Our work has in fact been inspired by the multiset-rewriting-like logic described in [CDL+99]. The core of the language provides logical primitives to express local communication and synchronization between processes, and to implement broadcast communication. Furthermore, universal quantification can be used as an elegant way to express fresh name generation.

Building upon previous works for verification of parameterized systems [AJ98, Del00, FS01, AJ01b], we will study a backward verification procedure, resembling usual symbolic model checking algorithms, which can be used to validate systems specified in our language. This procedure works by computing the backward reachability set of a given system starting
from a symbolic representation of the set states violating the property under consideration.

To exemplify, consider the simple Petri net [Rei85] drawn below, which represents a semaphore enforcing mutual exclusion for accessing an ideal (shared) resource. We have $K$ processes (we have drawn three of them in Figure 1.1), represented by tokens, i.e., black spots, which can be in state idle or use (i.e., using the resource). The semaphore is in turn represented by a token which can assume one of the two values locked and unlocked. Initially all processes are idle and the semaphore is unlocked. A configuration of the system is a multiset, for instance the initial configuration drawn in Figure 1.1 is represented by the multiset \{idle, idle, idle, unlocked\}. A process can get control of the resource (transition Enter) by locking the semaphore, provided the semaphore is currently unlocked. A process can release the resource (transition Leave) by unlocking the semaphore. These transitions can be encoded in linear logic as the following two program clauses (let $P$ be the resulting program):

\[
\text{Enter: } \text{idle} \otimes \text{unlocked} \leftarrow \text{use} \otimes \text{locked} \\
\text{Leave: } \text{use} \otimes \text{locked} \leftarrow \text{idle} \otimes \text{unlocked}
\]

The connective $\otimes$ (multiplicative linear disjunction) acts as a multiset constructor, whereas $\leftarrow$ (reversed linear implication) can be viewed as a rewriting operator (i.e., the left-hand side can be rewritten into the right-hand side). For instance, the first clause can transform the initial configuration into \{idle, idle, use, locked\}.

Suppose we want to formally verify that the above property of mutual exclusion holds. Our verification strategy works as follows. The set of states violating the mutual exclusion property are exactly those in which there are at least two processes using the resource. Therefore we can symbolically represent them by the multiset \{use, use\} (note that, in the
general case with \(K\) initially idle processes, this is a symbolic representation for an infinite set of system configurations). From a logical point of view, this is the counterpart of the linear logic axiom

\[
\text{Unsafe: } \text{use} \quad \not\Rightarrow \quad \text{use} \quad \rightarrow \quad \top
\]

which, intuitively, states that any configuration containing at least two tokens \text{use} is provable, according to the following proof scheme

\[
\begin{align*}
    & P \vdash \top, \Delta \\
    \Rightarrow & \quad P \vdash \text{use}, \text{use}, \Delta \\
    \text{Unsafe}
\end{align*}
\]

where \(\Delta\) is any multiset of tokens.

In order to verify if an unsafe configuration is reachable, we can compute the predecessor operator. If a process is in state \text{use}, necessarily it must have got access to the resource by executing the transition \text{Enter}, and the semaphore must have been unlocked (also, the semaphore must be currently locked). The corresponding set of predecessor states is given by \{\text{use, idle, unlocked}\}. The concept of reachability between Petri net configurations is the counterpart of the notion of provability in linear logic. For example, the following proof states that the configuration \{\text{use, idle, unlocked}\} is backward reachable from \{\text{use, use, locked}\}:

\[
\begin{align*}
    & P \vdash \top, \text{locked} \\
    \Rightarrow & \quad P \vdash \text{use}, \text{use}, \text{locked} \\
    \text{Unsafe}
\end{align*}
\]

Note that the configuration \{\text{use, use, locked}\} is unsafe, i.e., violates mutual exclusion. The above proof shows that the configuration \{\text{use, idle, unlocked}\} is also unsafe.

Using a similar kind of reasoning, we can compute the reflexive and transitive closure of the predecessor operator, which yields the following (symbolic representations of) states:

\{\text{use, use}\} \quad \{\text{use, idle, unlocked}\} \quad \{\text{idle, unlocked, idle, unlocked}\}

None of the above states is a legal initial configuration of the system (remember that we require all processes to be initially idle, furthermore we allow only one unlocked token). In other words, any legal initial configuration will never lead to a configuration violating mutual exclusion. The property of mutual exclusion is therefore verified. From a logical point of view, computing the closure of the predecessor operator amounts to computing the fixpoint of a suitable operator.

This simple idea, illustrated for a specification given via a Petri net, in this thesis will be extended in order to reason about systems which can be parametric in several dimensions.
For instance, we can extend the above specification in order to allow more than one resource. An atomic formula \( \text{sem}(x, \text{locked}) \) or \( \text{sem}(x, \text{unlocked}) \) can be used to represent a semaphore for a resource named \( x \). We can then verify mutual exclusion for any number of resources. We can also verify more complex protocols, which are infinite-state even for fixed values of the parameters. Our approach therefore provides a methodology to lift traditional verification techniques for Petri nets to more complex specification languages like coloured Petri nets [Jen97].

From a logical viewpoint, the verification procedure sketched above is the counterpart of a new, alternative, bottom-up semantics for a fragment of linear logic. The advantages of using a backward reachability procedure, as opposed to a forward one, are clearly advocated in [FS01, AJ01b]. In particular, computability results rely on the possibility of finding a finite representation for infinite set of states which are upward-closed. This can be done by specifying the minimality requirements that a given state should satisfy (remember the previous example, in which the multiset \( \{\text{use}, \text{use}\} \) represents every configuration with at least two processes in state use). The resulting specification methodology turns out to be well-suited to specify an interesting class of safety properties (e.g. mutual exclusion in concurrency theory, confidentiality in authentication) for parameterized and open systems.

From a logical perspective, this backward verification procedure relies on a symbolic evaluation of first-order programs based on unification, and makes use of suitable subsumption checks in order to relieve the state-explosion problem. In addition, techniques based on static analysis of programs provide further criteria to prune the search space.

Our framework has been intentionally designed as a modular (as opposed to monolithic) system. In fact, we have decided to leave the logical substratum, which basically consists of a multiset rewriting formalism, reasonably simple. The core of the system can then be enriched in different ways. For instance, reasoning on specialized and heterogeneous domains can be carried out by interfacing the system with different constraint systems, as in traditional constraint programming [JM94]. This solution is clearly more flexible, because the knowledge and reasoning capabilities which are specific of every particular domain can be delegated to specialized constraint solvers. In this way, we have been able to specify and analyze protocols which are both parametric and infinite-state (e.g. using integer-valued data variables).

As a result of our analysis, we have also obtained some results concerning decidability of the provability relation for some fragments of first-order linear logic (with or without constraints). From the verification viewpoint, these results are the counterpart of verification problems which, consequently, can be shown to be decidable. We have used these results as a termination guarantee for our verification procedure based on backward reachability. As an example, we have been able to analyze and validate, to our knowledge for the first time, a parameterized version of a classical mutual-exclusion protocol called the ticket protocol. It is remarkable that this protocol is both parametric and also infinite-state, in that it is
specified using integer variables which can get arbitrary values (even for a formulation of the protocol with a fixed number of processes, e.g. with two processes). Another interesting application field concerns authentication protocols. We will show that our methodology can be used either to find attacks or to verify correctness of security protocols. Our verification procedure, based on backward reachability, can automatically find attacks following complex patterns, like parallel attacks, without encoding any prior knowledge about the kind of attacks to look for.

Finally, we must mention that, from a logical standpoint, our work follows the established tradition of fixpoint, bottom-up semantics for (constraint) logic programs [Llo87, JM94, GDL95]. In particular, we have characterized the analogous of the so-called C-semantics [FLMP93, BGLM94], originally defined for Horn clause logic, for a subset of first-order linear logic including universal quantification in goals. As a side result, we have also obtained a non ground semantics for Horn programs enriched with universal quantification in goals. We believe that our work could also have an impact on the field of linear logic programming. This line of research can be carried on by considering either more complex linear logic languages [And92, HM94, Mil96, DM01], or more complex observational semantics, like the S-semantics of [FLMP93, BGLM94].

1.3 Plan of the Thesis

Broadly speaking, the thesis is divided into two parts. The first one presents the logical background for the rest of the thesis, and discusses the relationship between linear logic and classical alternative formalisms for the specification of concurrent systems, like (coloured) Petri nets and multiset rewriting systems.

The second part includes the original contribution of this thesis. Its core is structured as a collection of four chapters dealing with a different fragment or extension of the language LO [AP91b]. Every chapter deals with the definition of the bottom-up semantics for the corresponding fragment of linear logic, contains all the relevant proofs, and terminates with some examples. The four fragments or extensions of LO which we will be analyzed are: propositional LO; an extension of LO with the constant 1 and the $\otimes$ connective in goals, called LO₁; a first-order formulation of LO with constraints, called LO(C); and finally a first-order formulation of LO admitting universal quantification over goals, called LOₚ.

As a general guideline, we have included formal proofs and results stating correctness and completeness of the bottom-up semantics for each of the four different fragments mentioned above, even though some overlap may arise. For instance, the bottom-up semantics for propositional LO can be seen as an instance of the bottom-up semantics for LOₚ programs. However, the semantics for propositional programs can be greatly simplified, and it also enjoys computability results which do not extend, in general, to the first-order case. We
also believe that the relevant proofs, which are much simpler in the propositional case, can help the reader to better understand the extensions presented in the following chapters. We have therefore chosen to include them in the corresponding chapter.

A more detailed description of the structure of the thesis is given below.

1.3.1 Structure of the Thesis

We describe below how the thesis is concretely structured. Some chapters are based or extend previously published works. For the convenience of the reader, a list of the relevant papers is given at the end of this section.

First of all, in Chapter 2 we give some preliminaries and introduce some notations needed in the rest of the thesis.

The purpose of Part I is to present some background on linear logic and to substantiate our view of linear logic as a unifying view of concurrency. Specifically, this part is structured as follows. First of all, we introduce the basic concepts pertaining to linear logic in Chapter 3. In particular, we present in more detail the language LO and some proper extensions of its. In Chapter 4 we discuss the relationship between linear logic, and in particular the language LO, and other formalisms for the specification of concurrent systems, namely Petri nets, multiset rewriting systems over atomic formulas, and broadcast protocols. In Chapter 5 we extend the previous ideas by considering first-order linear logic theories and relating them to extensions of Petri nets, like coloured Petri nets, and first-order multiset rewriting systems.

Part II contains the original contribution of this thesis. In Chapter 6, first of all we discuss the backward verification approach which we follow in this thesis, we explain the connection with bottom-up semantics of linear logic programs, and we present a general theory which can be used to prove termination (decidability) results. Chapter 7 deals with the propositional fragment of LO. The chapter is based on [BDM00, BDM01b, BDM02], and contains some simple examples illustrating our methodology. Chapter 8 considers the language LO₁, an (apparently) simple extension of propositional LO which, however, has the effect of breaking the decidability property of the provability relation which holds instead for propositional LO. At the end of the chapter we show how broadcast primitives can be simulated in this logical fragment. The chapter extends previous results in [BDM00, BDM02]. In Chapter 9 we consider the language LO(𝐶), i.e., LO enriched with constraints. The analysis of the so-called ticket protocol, which we mentioned in Section 1.2, is included in this chapter. Some preliminary results concerning the contents of this chapter appeared in [BDM01a, BD02]. Finally, in Chapter 10, we extend the previous results to the fragment LO₂, a first-order formulation of LO with universal quantification over goals. We present as example the analysis of a test-and-lock mutual exclusion proto-
Chapter 11 presents some applications in the field of security, in particular it deals with the specification and analysis of authentication protocols, and should be considered as the logical prosecution of Chapter 10. Some preliminary results concerning the contents of this chapter have been presented in [Boz01].

Finally, we draw some conclusions in Chapter 12.

Publication List


Chapter 2

Preliminaries

In this chapter we introduce some notions and terminology which will be used throughout the thesis.

2.1 Functions and Orders

We will use the notation \( f : A \rightarrow B \) to denote a (total) function from the set \( A \) to the set \( B \). \( \mathbb{N} \) will denote the set of natural numbers, whereas \( \mathbb{N}_0 \) will stand for \( \mathbb{N} \setminus \{0\} \), and \( \mathbb{Z} \) will denote the set of integer numbers. A binary relation \( \leq \) on a set \( A \) is said to be a quasi-order (or preorder) if it is reflexive and transitive. The pair \((A, \leq)\) is called a preset. If in addition \( \leq \) is symmetric, it is an equivalence relation, while if it is antisymmetric it is a partial order and \((A, \leq)\) is called a poset. Given a poset \((A, \leq)\), an element \( a \in A \) is an upper [lower] bound of a subset \( X \subseteq A \) iff \( x \leq a [a \leq x] \) for all \( x \in X \). The least upper bound (l.u.b.) of \( X \), if it exists, is an upper bound of \( X \) which is minimum w.r.t. to \( \leq \) among the set of upper bounds of \( X \). Similarly, the greatest lower bound (g.l.b.) is defined as the maximum among the lower bounds of \( X \). A chain is a function \( C : \mathbb{N} \rightarrow A \) such that \( C(i) \leq C(i + 1) \) for all \( i \in \mathbb{N} \). A poset \( A \) is called a complete lattice iff least upper bounds and greatest lower bounds exist for every (non empty) subset of \( A \).

Given a complete lattice \( A \) with partial ordering \( \leq \), l.u.b. \( \bigvee \), and least element \( \bot \), a function \( f : A \rightarrow A \) is said to be monotonic iff \( a \leq b \) implies \( f(a) \leq f(b) \) for all \( a, b \in A \); \( f \) is continuous iff for every chain \( C : \mathbb{N} \rightarrow A \) we have that \( f(\bigvee C) = \bigvee (f(C)) \), where it is meant that \( \bigvee C \) stands for \( \bigvee_{i \in \mathbb{N}} C(i) \) and \( f(C) \) is the chain such that \( f(C)(i) = f(C(i)) \) for every \( i \in \mathbb{N} \). A fixpoint for a function \( f : A \rightarrow A \) is any element \( x \in A \) such that \( f(x) = x \). By Knaster-Tarski fixpoint theorem [Llo87], the set of fixpoints of any monotonic function \( f : A \rightarrow A \) on a complete lattice \( A \) is a non empty complete lattice, and therefore \( f \) admits
a least fixpoint, denoted lfp(f), and a greatest fixpoint, denoted gfp(f). If in addition 
f is continuous, it holds that lfp(f) = f\uparrow_\omega, where for k ∈ N, f\uparrow_k and f\uparrow_\omega are defined as follows: f\uparrow_0 = ⊥, f\uparrow_{k+1} = f(f\uparrow_k) for all k ≥ 0, and f\uparrow_\omega = \bigcup_{k∈N} f\uparrow_k.

2.2 Multisets

Throughout the thesis, we will extensively use the concept of multiset (or bag). Intuitively, a multiset extends the notion of set by allowing multiple occurrences of the same element. Formally, a multiset with elements in D is a function M : D → N. If d ∈ D and M is a multiset on D, we say that d ∈ M if and only if M(d) > 0. For convenience, we will often use the notation for sets (allowing duplicated elements) to indicate multisets, when no ambiguity arises from the context. For instance, \{a, a, b\}, where a, b ∈ D, will denote the multiset M such that M(a) = 2, M(b) = 1, and M(d) = 0 for all d ∈ D\{a, b\}. Sometimes we will simply write a, a, b for \{a, a, b\}. Finally, given a set D, MS(D) will denote the set of multisets with elements in D. We define the following operations on multisets.

**Definition 2.1 (Operations on Multisets)** Let D be a set, M_1, M_2 ∈ MS(D), and n ∈ N.

i. \(\epsilon\) is s.t. \(\epsilon(d) = 0\) for all \(d \in D\) (empty multiset);

ii. \((M_1 + M_2)(d) = M_1(d) + M_2(d)\) for all \(d \in D\) (union);

iii. \((M_1 \setminus M_2)(d) = \max\{0, M_1(d) - M_2(d)\}\) for all \(d \in D\) (difference);

iv. \((M_1 \cap M_2)(d) = \min\{M_1(d), M_2(d)\}\) for all \(d \in D\) (intersection);

v. \((n \cdot M)(d) = nM(d)\) for all \(d \in D\) (scalar product);

vi. \(M_1 \not\subseteq M_2\) iff there exists \(d \in D\) s.t. \(M_1(d) \neq M_2(d)\) (comparison);

vii. \(M_1 \preceq M_2\) iff \(M_1(d) \leq M_2(d)\) for all \(d \in D\) (inclusion);

viii. \((M_1 \bullet M_2)(d) = \max\{M_1(d), M_2(d)\}\) for all \(d \in D\) (merge);

ix. \(|M_1| = \Sigma_{d \in D} M_1(d)\) (cardinality).

We will use the summation notation \(\sum_{i \in I} M_i\) to denote the union of a family of multisets \(M_i\), with \(i \in I\), \(I\) being a finite set. It turns out that \((MS(D), \preceq)\) has the structure of a lattice (the lattice is complete provided a greatest element is added). In particular, merge and intersection are, respectively, the least upper bound and the greatest lower bound operators with respect to multiset inclusion \(\preceq\).
2.3 Signatures and Algebras

In the following we will often need signatures to keep trace of constant, function, and possibly predicate symbols (in this case we will say “signature with predicates”) used in programs. We always assume the sets of constant, function, and predicate symbols to be mutually disjoint.

Given a signature Σ, we take for granted the definitions of Σ-algebra and term algebra over Σ. A term is said to be ground if it does not contain variables. Given a denumerable set of variable symbols \( V \) (usually noted \( x, y, z, \ldots \)), we denote by \( T^V_Σ \) the set of non ground terms over Σ and \( T_Σ \) denotes the set of ground terms over Σ. We use the notation \( t[s/x] \) to indicate the capture-free substitution of the term \( s \) for the variable \( x \in V \) in a term \( t \). We extend this notation to \( t[s/x] \), where \( s \) and \( x \) are, respectively, a vector of terms and a vector of variables, with the same number of components. Finally, we denote by \( A^V_Σ (A_Σ) \) the set of non ground (ground) atoms over Σ.

In Chapter 10 we will need to augment signatures with newly created constants, called eigenvariables. We will use the compact notation \( Σ,c \) to denote the augmentation of Σ by \( c \).

2.4 Substitutions and Multiset Unifiers

We inherit the usual concept of substitution (mapping from variables to terms) from traditional logic programming. We will always consider a denumerable set of variables \( V \), and substitutions will be usually noted \( θ, σ, τ, \ldots \). We will use the notation \( [x \mapsto t, \ldots] \), where \( x \) is a variable and \( t \) is a term, to denote substitution bindings, with nil denoting the empty substitution. The application of a substitution \( θ \) to \( F \), where \( F \) is a generic expression (e.g. a formula, a term, \ldots) will be denoted by \( Fθ \). A substitution \( θ \) is said to be grounding for \( F \) if \( Fθ \) is ground, in this case \( Fθ \) is called a ground instance of \( F \). Composition of two substitutions \( θ \) and \( σ \) will be denoted \( θ \circ σ \), e.g. \( F(θ \circ σ) \) stands for \( (Fθ)σ \). We will indicate the domain of a substitution \( θ \) by \( Dom(θ) \), and will say “\( θ \) defined on a signature \( Σ \)” meaning that \( θ \) can only map variables in \( Dom(θ) \) to terms in \( T^V_Σ \). Substitutions are ordered with respect to the ordering \( ≤ \) defined in this way: \( θ ≤ τ \) if and only if there exists a substitution \( σ \) s.t. \( τ = θ \circ σ \). If \( θ ≤ τ \), θ is said to be more general than τ; if \( θ ≤ τ \) and \( τ ≤ θ \), θ and τ are said to be equivalent. Finally, \( FV(F) \), for an expression \( F \), will denote the set of free variables of \( F \), and \( θ|_W \), where \( W \subseteq V \), will denote the restriction of \( θ \) to \( Dom(θ) \cap W \).

We will need the notion of most general unifier (m.g.u.). The definition of most general unifier is somewhat delicate. In particular, different classes of substitutions (e.g. idempo-
tent substitutions) have been considered for defining most general unifiers. We refer the reader to [Ede85, LMM88, Pal90] for a discussion. Most general unifiers form a complete lattice with respect to the ordering $\leq$, provided a greatest element is added. For our purposes, we do not choose a particular class of most general unifiers, we only require the operation of least upper bound of two substitutions w.r.t $\leq$ to be defined and effective. The least upper bound of $\theta_1$ and $\theta_2$ (see [Pal90] for its definition) will be used in Chapter 10 and indicated $\theta_1 \uparrow \theta_2$. We assume $\uparrow$ to be commutative and associative.

Example 2.2 Let us consider a signature with a constant symbol $a$ and a function symbol $f$. Let $\mathcal{V}$ be a denumerable set of variables, and $u, v, w, \ldots \in \mathcal{V}$. Let $\theta_1 = [x \mapsto u, y \mapsto f(u')], \theta_2 = [x \mapsto v, y \mapsto f(a)],$ and $\theta_3 = [x \mapsto f(w), y \mapsto z]$. Then we have that $\theta_1 \leq \theta_2$, $\theta_1, \theta_3$ and $\theta_2, \theta_3$ are not comparable w.r.t. $\leq$, and, modulo renaming, $\theta_1 \uparrow \theta_2 \uparrow \theta_3 = [x \mapsto f(x'), y \mapsto f(a)]$. \hfill \Box

**Multiset Unifiers.** We need to lift the definition of most general unifier from expressions to multisets of expressions. Namely, given two multisets $\mathcal{A} = \{a_1, \ldots, a_n\}$ and $\mathcal{B} = \{b_1, \ldots, b_n\}$ (note that $|\mathcal{A}| = |\mathcal{B}|$), we define a most general unifier of $\mathcal{A}$ and $\mathcal{B}$, written $m.g.u.(\mathcal{A}, \mathcal{B})$, to be the most general unifier (defined in the usual way) of the two vectors of expressions $\langle a_1, \ldots, a_n \rangle$ and $\langle b_1, \ldots, b_n \rangle$, where $\{i_1, \ldots, i_n\}$ is a permutation of $\{1, \ldots, n\}$. Depending on the choice of the permutation, in general there will be more than one way to unify two given multisets (the resulting class of m.g.u. in general will include unifiers which are not equivalent). We will use the notation $\theta = m.g.u.(\mathcal{A}, \mathcal{B})$ to denote any unifier which is non deterministically picked from the set of most general unifiers of $\mathcal{A}$ and $\mathcal{B}$.

Example 2.3 Let us consider a signature with a constant symbol $a$, a function symbol $f$ and a predicate symbol $p$. Let $\mathcal{V}$ be a denumerable set of variables, and $v, w, x, \ldots \in \mathcal{V}$. Let $\mathcal{A} = \{p(x, x), p(f(y), y)\}$ and $\mathcal{B} = \{p(v, a), p(w, z)\}$. The multisets $\mathcal{A}$ and $\mathcal{B}$ can be unified in two different ways. By unifying $p(x, x)$ with $p(v, a)$ and $p(f(y), y)$ with $p(w, z)$, we get, modulo renaming, the m.g.u. $\theta_1 = [x \mapsto a, v \mapsto a, w \mapsto f(y), z \mapsto y]$. By switching the order of atom unifications, we get instead, again modulo renaming, the m.g.u. $\theta_2 = [w \mapsto x, z \mapsto x, v \mapsto f(y), y \mapsto a]$. As it can be readily verified, neither $\theta_1 \leq \theta_2$ nor $\theta_2 \leq \theta_1$ hold. We also have that $\mathcal{A}\theta_1 = \mathcal{B}\theta_1 = \{p(a, a), p(f(y), y)\}$, while $\mathcal{A}\theta_2 = \mathcal{B}\theta_2 = \{p(x, x), p(f(a), a)\}$. \hfill \Box

### 2.5 Basics of Logic Programming

We recall here some basic notions in Logic Programming (see for instance [Llo87, Apt90]).
**Syntax.** A (positive or negative) **literal** is an atom or the negation of an atom. A **clause** is a disjunction of literals, the set of positive literals is called the **head** and the set of negative literals the **body**. Clauses with at most one positive literal are called **Horn clauses**, a clause with exactly one positive literal is called **program clause** or **definite clause**, and usually noted $H \leftarrow B_1, \ldots, B_n$, where $H$ is the head and $B_1, \ldots, B_n$ the body literals (stripped of the negation symbol). A definite clause with empty body is called **unit clause**, whereas a Horn clause with empty head is called **goal** or **negative clause**.

A logic program is a finite and non empty set of program clauses.

**Semantics.** Given a program $P$ over a signature $\Sigma$, with non empty set of constants, the **Herbrand universe** of $P$ is the set of all ground terms over $\Sigma$, while the **Herbrand base** of $P$ is the set of ground atoms over $\Sigma$. By **Herbrand interpretation** we mean an interpretation for $P$ such that its domain is the Herbrand universe, constants are assigned to themselves, and every $n$-ary function symbol $f$ is assigned to the function mapping any sequence of terms $t_1, \ldots, t_n$ to $f(t_1, \ldots, t_n)$. A Herbrand interpretation is uniquely determined by a subset of the Herbrand base fixing the interpretation of predicate symbols.

A logic program $P$ can be equipped with three kinds of semantics:

- **operational semantics**, usually given via the notion of SLD-resolution (see [Llo87, Apt90]) or via a suitable proof-system;

- **model-theoretic semantics**, defined via the notion of satisfiability of a program w.r.t. a given (Herbrand) interpretation;

- **fixpoint semantics**, given by the least fixpoint of a suitable immediate consequence operator, usually denoted $T_P$.

A suitable notion of equivalence between the three different kinds of semantics can be proved [Llo87, Apt90]. In the rest of the thesis, we will be mainly interested in the fixpoint semantics. The $T_P$ operator maps Herbrand interpretations (i.e., subsets of the Herbrand base) to Herbrand interpretations, and is defined as follows: $H \in T_P(I)$ if and only if $H \leftarrow B_1, \ldots, B_n$ is a ground instance of a clause in $P$, and $B_1, \ldots, B_n \in I$. $T_P$ can proved to be monotonic and continuous, therefore it admits a least fixpoint $\text{lfp}(T_P) = T_P^\uparrow_\omega$, which coincides with the least Herbrand model of $P$.

We conclude by mentioning that the above scheme can be extended in order to take into consideration more refined notions of observables [FLMP93, BGLM94]. In particular, the so-called C-semantics formalizes the notion of non-ground success set of a logic program, whereas the so-called S-semantics correspond to the computed answer substitution semantics. The C-semantics is defined by considering the non-ground Herbrand base (i.e., the set of non-ground atoms over a signature $\Sigma$) and the corresponding notion of non-ground
Herbrand interpretation. The operational semantics is defined as the set of non-ground atoms which have an *SLD-refutation* with *empty answer substitution* (see [Llo87, Apt90]). An extended $T_P$ operator can be defined, which, as usual, turns out to be monotonic and continuous, therefore admitting a least fixpoint $\text{lfp}(T_P) = T_P\upharpoonright_{\omega}$.

In Chapter 10 we will extend the C-semantics of [FLMP93, BGLM94] to a subset of linear logic programs.
Part I

Linear Logic as a Unifying View of Concurrency
Chapter 3

Linear Logic

Linear logic [Gir87] has been introduced as a resource-oriented refinement of classical logic. Intuitively, the idea is to constrain the number of times a given assumption (resource occurrence) can be used inside a deduction for a given goal formula. This peculiar resource management together with the possibility of naturally modeling the notion of state, make linear logic an appealing formalism to reason about concurrent and dynamically changing systems.

After giving a very brief summary of some aspects related to linear logic and linear logic programming in Section 3.1, in Section 3.2 we will focus on the language LO [AP91b] and some of its extensions, which will be the leading thread of the present thesis work.

3.1 Full Linear Logic

In classical logic one usually thinks about the construction of a proof as the process of finding a logical deduction for a goal starting from a given set of assumptions. In linear logic one may think of assumptions as multisets of resources, each of which can be used just once. This view of assumptions imposes a shift in the concept of proof as well. Building a proof in linear logic is the process of finding a deduction which transforms all the given assumptions (and only them) into something else (the conclusion). This view of proof construction is related to the notion of causal implication. To exemplify, a reaction in chemistry is a typical example of a process that can be naturally modeled in linear logic (everybody knows that reactions must be correctly balanced).

Another feature which is intrinsic to linear logic is the possibility of modeling concurrent executions (e.g. chemical reactions which happen simultaneously). This is witnessed by several papers relating linear logic with different formalisms for concurrency, e.g. [EW90,
Table 3.1: The set of linear logic connectives

\[
\begin{array}{|c|cccccc|}
\hline
\text{classical} & t & f & \land & \lor & \top & \bot \\
\hline
\text{multiplicative} & 1 & \bot & \otimes & \oplus & \top & \bot \\
\text{additive} & \top & 0 & \& & \oplus & \top & \bot \\
\hline
\hline
\hline
\end{array}
\]

ALPT93, Mil92, Cer94, Laf95, CDKS00]. For instance, in [Mil92] a translation of $\pi$-calculus into linear logic is presented. The translation yields a proof system in linear logic whose provability relation is the counterpart of the $\pi$-calculus reduction relation, and can be used, e.g., to prove structural equivalence of $\pi$-calculus agent expressions.

Linear logic can be considered as the endpoint of a proof-theoretical investigation of subsystems of classical logic. In particular, the analysis involves the so called structural rules, i.e., exchange, weakening and contraction. In fact, the weakening and contraction rules are not compatible with the notion of resource sketched above: the weakening rule allows one to ignore a given assumption, while the contraction rule is responsible for duplication of assumptions. Removing the structural rules of weakening and contraction from classical sequent calculus has the unavoidable consequence of splitting the set of classical connectives into two different classes, called additive and multiplicative. In the usual sequent calculus presentation, additive and multiplicative connectives differ with respect to context management (context sharing vs. context splitting, respectively). While in classical logic both presentations, additive and multiplicative, are provably equivalent, the removal of structural rules makes them not equivalent anymore.

Although removing the structural rules seems to be unavoidable in this setting, it would be unreasonable to remove them altogether. In other words, a way to recover the power of classical logic is necessary. To this aim, two further connectives, the so-called exponentials, one dual with the other, are introduced. Intuitively, every formula preceded by an exponential behaves like the corresponding classical one (e.g. an hypothesis can be used any number of times or not used at all in a proof). This mechanism re-introduces classical connectives in a controlled way. A summary of linear logic connectives is presented in Table 3.1. We note that it is also possible to define a linear negation (denoted $\perp$ in the following), which turns out to be involutive (i.e., $(F^\perp)^\perp = F$) and constructive at the same time.

In Figure 3.1 we present an example of a sequent system for full linear logic. It is an
example of one-sided sequent system (two-sided versions are possible \cite{Tro92}). It is also a dyadic sequent system, according to the terminology of \cite{And92}. In fact, sequents of the form $\vdash \Theta : \Gamma$ are divided into two parts $\Theta$ and $\Gamma$, which are multisets of formulas. $\Theta$ is the so-called unbounded part, while $\Gamma$ is the bounded part. In other words, formulas in $\Theta$ must be implicitly considered as exponentiated (i.e., preceded by ?) and thus can be reused any number of times, while formulas in $\Gamma$ must be used exactly once. The different behaviour of multiplicative rules w.r.t. additive rules (i.e., context splitting vs. context sharing) should be evident. For instance, the reader can compare the different context management in rules $\otimes$ and $\&$. In rule $\forall$, $c$ stands for a new constant, while in rule $\exists$, $t$ stands for an arbitrary term. The rules for implications are missing but can be recovered by means of logical equivalences like $F \rightarrow G \equiv F^\perp \not\exists G$.

### 3.1.1 Programming in Linear Logic

In recent years a number of fragments of linear logic have been proposed as a logical foundation for extensions of logic programming. Several new programming or specification languages, like LO \cite{AP91b} (see Section 3.2), LinLog \cite{And92}, ACL \cite{KY95}, Lolli \cite{HM94}, Lygon \cite{HP94}, Forum \cite{Mil96} and $\mathcal{E}_{hhj}$ \cite{DM01} (a subset of Forum) have been proposed with the aim of enriching traditional logic programming languages with a well-founded notion of state and with aspects of concurrency. For instance, LO has been applied to model coordination languages \cite{And96}, Forum has been used for specifying the sequential and
pipelined operational semantics of the DLX machine [Chi95], and $\mathcal{E}_{hhf}$ to model object-oriented and deductive databases [BDM97], multi-agent systems [BDM+99], and object calculi [BDLM00].

Further research efforts have tried to incorporate linear logic within logical frameworks. A logical framework can be seen as a meta-language for specifying and reasoning about different formalisms represented as deductive systems, such as various logics, natural deduction inference systems and sequent calculi, programming languages, abstract machines, and so on. In [MMP96, McD97], the authors present a logical framework based on extensions of linear logic and intuitionistic logic incorporating a notion of definition and induction on natural numbers. Such framework has been applied, for instance, to prove simulation and bisimulation judgments in abstract transition systems, and properties such as subject reduction and determinacy of evaluation in the language PCF. A different approach is the one followed in [Pfe01], where a logical framework is presented, based on a constructive type theory incorporating dependent types, according to the so-called judgments-as-types representation. These ideas have been incorporated in the system Twelf [PS99] and exploited to prove, for instance, various properties in the area of Mini-ML, natural deduction and category theory. A linear logical extension is considered in [CP02].

The operational semantics of linear logic languages is usually given via a sequent-calculi presentation of the corresponding fragment of linear logic. Special classes of proofs have been studied in order to limit the non-determinism intrinsic in the process of proof construction. In particular, the focusing proofs of [And92] and the uniform proofs of [Mil96] allow one to consider cut-free and goal-driven proof systems that are still complete with respect to provability in linear logic.

The key issue underlying the notion of goal-driven provability is the concept of synchronous and asynchronous connectives [And92]. Asynchronous connectives ($\bot$, $\exists$, $?$, $\top$, $\&$, $\forall$) require no choice from the proof search procedure, and only introduce don't care non-determinism, while synchronous connectives ($1$, $\otimes$, $!$, $0$, $\oplus$, $\exists$) require the search procedure to make a committed choice and introduce don't know non-determinism. Specifically, the focusing proofs of [And92] have the following features:

- if the current goal contains some asynchronous formulas, they are immediately decomposed: any of them can be immediately and randomly selected as the principal formula for decomposition;
- if the current goal contains only synchronous formulas, one of them must be selected non deterministically for processing, but as soon as this choice is made, the proof can focus on it, i.e., it can strip all layers of synchronous connectives from it.

The above restrictions on the proof search procedure induce a proof normalization which does not affect completeness of the provability relation. In order to enforce this behaviour
of the proof search procedure, a variant of the proof system of Figure 3.1 based on focusing proofs (called triadic) is presented in [And92].

When dealing with specific languages based on linear logic, the focusing proofs of [And92] can be furtherly refined and tailored to the particular logical fragment under consideration. For instance, focusing proofs have a simple and direct computational interpretation for the fragment called LinLog, presented in [And92]. LinLog is based on a subset of linear logic for which a sound mapping from full linear logic, preserving focusing proofs, exists. LinLog formulas extend the syntax of definite clauses and goals of traditional Horn logic, by allowing clauses with multiple atoms (connected by \(\otimes\)) in the head, and goals built over a subset of linear connectives. The computational interpretation of LinLog is particularly suited for concurrency and object-oriented programming (program clauses are called methods in [And92]).

In this thesis, we will focus our attention on a fragment of LinLog called LO [AP91b]. The reason we selected LO is that we were looking for a relatively simple linear logic language with a focusing-proof presentation, state-based computations and aspects of concurrency. As we will demonstrate in Chapter 4, LO programs are at least as expressive as Petri nets or multiset rewriting systems over atomic formulas. In practice, LO has been successfully applied to model, e.g., concurrent object-oriented languages [AP91b], and coordination languages based on the Linda model [And96]. In this thesis we will show that LO, enriched with a computational model based on bottom-up evaluation, can provide a valuable environment for the specification and validation of protocols.

For more details on the material of this section, we refer the reader to [Gir87, Tro92, CD97] for an introduction to linear logic, [MNPS91] and [Mil96] for a discussion on, respectively, uniform proofs in intuitionistic and in linear logic, and [And92] for the related concept of focusing proofs and for LinLog. Some background material on classical logic and sequent calculus can be found in [Gal86]. In the next section we present the language LO.

### 3.2 The Language LO

LO [AP91b] is a logic programming language based on a fragment of LinLog [And92]. Its mathematical foundations lie on a proof-theoretical presentation of a fragment of linear logic defined over the linear connectives \(\rightarrow\) (linear implication, we use the reversed notation \(H \rightarrow G\) for \(G \rightarrow H\)), \& (additive conjunction), \(\otimes\) (multiplicative disjunction), and the constant \(\top\) (additive identity). In this section we present the proof-theoretical semantics of LO, which corresponds to the usual top-down operational semantics for traditional logic programming languages like Prolog. For the sake of precision, we present a slight extension of LO by admitting the constant \(\perp\) in goals and clause heads. Following [AP91b], we give these definitions.
Definition 3.1 (Atomic Formulas) Let $\Sigma$ be a signature with predicates including a set of constant and function symbols $L$ and a set of predicate symbols $P$, and let $V$ be a denumerable set of variables. An atomic formula over $\Sigma$ and $V$ has the form $p(t_1, \ldots, t_n)$ (with $n \geq 0$), where $p \in P$ and $t_1, \ldots, t_n$ are (non ground) terms in $T^V_{\Sigma}$.

We are now ready to define $G$-formulas, which correspond to goals to be evaluated in a given program.

Definition 3.2 (Goal Formulas) Let $\Sigma$ be a signature with predicates and $V$ a denumerable set of variables. The class of $G$-formulas (goal formulas) over $\Sigma$ and $V$ is defined by the following grammar

$$G ::= G \not\in G | G \& G | A | \top | \bot$$

where $A$ stands for an atomic formula over $\Sigma$ and $V$. A multiset of goal formulas will be called a context hereafter.

The following definitions introduce $D$-formulas, which correspond to multiple-headed program clauses.

Definition 3.3 (Head Formulas) Let $\Sigma$ be a signature with predicates and $V$ a denumerable set of variables. The class of $H$-formulas (head formulas) over $\Sigma$ and $V$ is defined by the following grammar

$$H ::= A \not\in \ldots \not\in A | \bot$$

where $A$ stands for an atomic formula over $\Sigma$ and $V$.

Definition 3.4 (Clauses and Programs) Let $\Sigma$ be a signature with predicates and $V$ a denumerable set of variables. The class of $D$-formulas (program clauses) over $\Sigma$ and $V$ is defined by the following grammar

$$D ::= \forall (H \circ G) | D \& D$$

where $H$ and $G$ are, respectively, a head formula and a goal formula over $\Sigma$ and $V$, and $\forall (H \circ G)$ stands for $\forall x_1 \ldots x_k. (H \circ G)$, with $\{x_1, \ldots, x_k\} = FV(H \circ G)$. An LO program over $\Sigma$ and $V$ is a $D$-formula over $\Sigma$ and $V$.

For the sake of simplicity, in the following we usually omit universal quantifiers in program clauses, i.e., we consider free variables as being implicitly universally quantified.

Definition 3.5 (LO Sequents) Let $\Sigma$ be a signature with predicates and $V$ a denumerable set of variables. An LO sequent has the form $P \vdash G_1, \ldots, G_k$, where $P = D_1 \& \ldots \& D_n$ is an LO program and $G_1, \ldots, G_k$ is a context, i.e., a multiset of goals, over $\Sigma$ and $V$. 
Remark 3.6 Given an LO program \( P = D_1 \& \ldots \& D_n \), in the rest of the thesis we often find it convenient to view \( P \) as the set of clauses \( D_1, \ldots, D_n \). Formally, this is justified by the logical equivalence between \(! (D_1 \& \ldots \& D_n)\) and \(! D_1 \otimes \ldots \otimes ! D_n\).

According to remark 3.6, the left-hand side of an LO sequent can be seen as the set of clauses \( D_1, \ldots, D_n \), while the right-hand side is a multiset of goals. Structural rules (exchange, weakening and contraction) are allowed on the left-hand side, while on the right-hand side only the rule of exchange is allowed (for the fragment under consideration, it turns out that the rule of weakening is admissible, as stated in Proposition 3.9, while contraction is forbidden). The LO sequent \( P \vdash G_1, \ldots, G_k \) stands for the following one-sided linear logic sequent (see Figure 3.1):
\[
\vdash D_1 \perp, \ldots, D_n \perp : G_1, \ldots, G_k.
\]

We remind that the formulas on the left of ‘:’ can be used in a proof an arbitrary number of times. In other words, an LO program can also be viewed as a set of reusable clauses.

We now define provability in LO.

Definition 3.7 (Ground Instances) Let \( \Sigma \) be a signature with predicates and \( \mathcal{V} \) a denumerable set of variables. Given an LO program \( P \) over \( \Sigma \) and \( \mathcal{V} \), the set of ground instances of \( P \), denoted \( \text{Gnd}(P) \), is defined as follows:
\[
\text{Gnd}(P) = \{(H \leftarrow G) \theta \mid \forall (H \leftarrow G) \in P \text{ and } \theta \text{ is a grounding substitution for } H \leftarrow G\}
\]

The execution of a multiset of \( G \)-formulas \( G_1, \ldots, G_k \) in \( P \) corresponds to a goal-driven proof for the two-sided LO sequent \( P \vdash G_1, \ldots, G_k \). According to this view, the operational semantics of LO is given via the uniform (focusing) proof system presented in Figure 3.2, where \( P \) is a set of clauses, \( A \) is a multiset of atomic formulas, and \( \Delta \) is a multiset of \( G \)-formulas. We have used the notation \( \hat{H} \), where \( H \) is a linear disjunction of atomic formulas \( a_1 \& \ldots \& a_n \), to denote the multiset \( a_1, \ldots, a_n \) (by convention, \( \hat{\perp} = \epsilon \), where \( \epsilon \) is the empty multiset).

Definition 3.8 (LO provability) Let \( \Sigma \) be a signature with predicates and \( \mathcal{V} \) a denumerable set of variables. Given an LO program \( P \) and a goal \( G \) over \( \Sigma \) and \( \mathcal{V} \), we say that \( G \) is provable from \( P \) if there exists a proof tree, built over the proof system of Figure 3.2, with root \( P \vdash G \), and such that every branch is terminated with an instance of the \( \top_r \) axiom.

The concept of uniformity applied to LO requires that the right rules \( \top_r, \&_r, \&_r, \perp_r \) have priority over \( bc \), i.e., \( bc \) is applied only when the right-hand side of a sequent is a multiset of atomic formulas (as suggested by the notation \( A \) in Figure 3.2). The proof system of
Figure 3.2 is a specialization of more general uniform proof systems for linear logic like Andreoli’s focusing proofs [And92] and Forum [Mil96]. Rule $bc$ is analogous to a backchaining (resolution) step in traditional logic programming languages. Note that according to the concept of resolution explained above, $bc$ can be executed only if the right-hand side of the current LO sequent consists of atomic formulas. As an instance of rule $bc$, we get the following proof fragment, which deals with the case of clauses with empty head:

\[
\begin{align*}
\vdash P \vdash A, G, & \quad P \vdash A, bc \quad \text{provided } P \vdash H, A
\end{align*}
\]

Given that clauses with empty head are always applicable in atomic contexts, the degree of non-determinism they introduce in proof search is usually considered unacceptable [Mil96] and in particular they are forbidden in the original presentation of LO [AP91b]. However, the computational model we are interested in, i.e., bottom-up evaluation, does not suffer this drawback. Clauses with empty head often allow more flexible specifications (see for instance Chapter 11).

LO clauses having the form $H \rightarrow \top$ play the same role as axioms (i.e., unit clauses) for Horn programs. In fact, when a backchaining step over such a clause is possible, we get a successful (branch of) computation, independently of the current context $A$, as shown in the following proof scheme:

\[
\begin{align*}
\vdash P \vdash \top, A & \quad P \vdash \top, A, bc \quad \text{provided } H \rightarrow \top \in \text{Gnd}(P)
\end{align*}
\]
Proposition 3.9 (Admissibility of the Weakening Rule) Given an LO program $P$ and two multisets of goals $\Delta, \Delta'$ such that $\Delta \preceq \Delta'$, if $P \vdash \Delta$ then $P \vdash \Delta'$.

Proof By simple induction on the structure of LO proofs. $\square$

Admissibility of the weakening rule makes LO an affine fragment of linear logic [Kop95]. Note that all structural rules are admissible on the left hand side (i.e., on the program part) of LO sequents.

Example 3.10 Let $P$ be the (propositional) LO program consisting of the clauses

1. $a \vdash b \& c$
2. $b \vdash (d \& c) \& f$
3. $c \& d \vdash \top$
4. $e \& c \vdash b \& c$
5. $c \& f \vdash \top$

where $a, b, c, d, e, f$ are propositional symbols, and consider an initial goal $e, e$. A proof for this goal is shown on the left-hand side of Figure 3.3, where we have denoted by $bc^{(i)}$ the application of the backchaining rule over clause number $i$ of $P$. The proof proceeds as follows. Using clause 4, to prove $e, e$ we have to prove $b \& c$, which, by LO $\&$ rule, reduces to prove $b, c$. At this point we can backchain over clause 2, and we get the new goal $d \& f, c$. By applying $\&$ rule, we get two separate goals $d \& e, c$ and $f, c$. The first, after a reduction via $\&$ rule, is provable by means of clause (axiom) 3, while the latter is provable directly by clause (axiom) 5. Note that $\top$ succeeds in a non-empty context.
(i.e., containing \( e \)) in the left branch. A similar proof shows that the goal \( a \) is also provable from \( P \). By Proposition 3.9, provability of \( e, e \) and \( a \) implies provability of any multiset of goals \( e, e, \Delta \) and \( a, \Delta \), for every multiset of goals \( \Delta \).

\[ \square \]

**Example 3.11** Let \( P \) be the following (first-order) LO program (variables are considered externally quantified in clauses)

1. \( p(x) \leftarrow q(y) \& r(x) \)
2. \( q(b) \leftarrow \top \)
3. \( r(a) \leftarrow \top \)

where \( p, q, r \) are predicate symbols, \( a, b \) are constant symbols, and \( x, y \) are variables. A proof for the goal \( p(a) \) is given on the right-hand side of Figure 3.3. Backchaining on clause 1 is carried out by considering its ground instance \( p(a) \leftarrow q(b) \& r(a) \).

\[ \square \]

### 3.3 Extensions of LO

As shown in [And92], the logical fragment underlying the language LO can be extended in order to take into consideration more powerful programming constructs. In this section we will present two different extensions of LO, which will be discussed later on. Specifically, in Section 3.3.1 we present the language LO enriched with the constant \( 1 \) and the multiplicative conjunction \( \otimes \) (see also Chapter 8), and in Section 3.3.2 we present the language LO enriched with universal quantification over goals (see also Chapter 10).

#### 3.3.1 LO with \( 1 \) and \( \otimes \)

In this section we consider an extension of LO where goal formulas range over the \( G \)- formulas of Section 3.2, the multiplicative conjunction \( \otimes \) and the constant \( 1 \), according to
the following grammar:

\[
\begin{align*}
E & ::= E \otimes E \mid E \& E \mid A \mid \top \mid \bot \\
G & ::= E \mid G \otimes G \mid 1 \\
H & ::= A \otimes \ldots \otimes A \mid \bot \\
D & ::= \forall (H \circ G) \mid D \& D
\end{align*}
\]

The class \(E\) stands for elementary goals, and it is introduced in order to prevent occurrences of the synchronous connective \(\otimes\) to appear inside the scope of synchronous connectives (\(\otimes\) and \(\&\)). As shown in [And92], this is necessary in order to preserve the restriction of the backchaining rule to atomic contexts.

In particular, in this fragment we admit clauses of the form \(H \circ 1\). We name this language \(LO_1\), and use the notation \(P \vdash_1 \Delta\) for \(LO_1\) sequents. A proof system for \(LO_1\) is presented in Figure 3.4. It is obtained from the proof system of Figure 3.2 by adding a rule for \(\otimes\) and a rule for the constant \(1\). The meaning of the new kind of clauses \(H \circ 1\) is given by the following inference scheme:

\[
\begin{align*}
\frac{P \vdash_1 \hat{H}}{P \vdash_1 1} &_r \\
\frac{P \vdash_1 H}{} &_{bc}
\end{align*}
\]

provided \(H \circ 1 \in Gnd(P)\)

Note that there cannot be other resources in the right-hand side of the lower sequent except for \(\hat{H}\). In other words, the head of the clause must match the current context exactly in order for rule \(bc\) to be fired. Thus, in contrast with \(\top\), the constant \(1\) intuitively introduces the possibility of counting resources. As a result, the monotonicity property stated in Proposition 3.9, i.e., admissibility of the weakening rule, does not hold anymore for the fragment \(LO_1\).

**Example 3.12** Let us consider the following \(LO_1\) program \(P\):

1. \(a \otimes \text{transf} \circ 1 \otimes \text{transf}\)
2. \(\text{transf} \circ \bot \& \text{check}\)
3. \(\text{check} \otimes b \circ \text{check}\)
4. \(\text{check} \otimes c \circ \text{check}\)
5. \(\text{check} \circ 1\)

A proof for the goal \(a, a, c, \text{transf}\) is given on the left-hand side of Figure 3.5, where, for simplicity, we have incorporated applications of the \(\otimes\), \(\&\), and \(\bot\) rules into backchaining steps. As we will see in Chapter 4, \(LO\) programs can be used to encode Petri nets. According to this view, every propositional symbol like \(a, b, c\) denotes one place, and occurrences
Figure 3.5: Examples of LO₁ proof

of symbols denote tokens in the corresponding place. In this view, the above program can be seen as specifying an operation transf which transfers all tokens in place a to place b. The first clause specifies the transfer of a single token from a to b, and it is supposed to be used as many times as the number of initial tokens in a. The second clause starts an auxiliary branch of the computation which checks that all tokens have been moved to b. Note that using 1 in clause 5 ensures that the check cannot succeed if there are tokens left in a.

Example 3.13 Let us consider the following LO₁ program P:

1. \( c \otimes c \leftarrow \top \)
2. \( a \otimes b \leftarrow 1 \)
3. \( b \leftarrow a \otimes c \)

A proof for the goal \( b, b, c \) is given on the right-hand side of Figure 3.5.

3.3.2 LO with Universal Quantification

In this section we consider an extension of LO where the G-formulas of Section 3.2 are enriched with universal quantification, according to the following grammar:

\[
G ::= \diamond G | G \& G | \forall x.G | A | \top | \bot \\
H ::= A \diamond \ldots \diamond A | \bot \\
D ::= \forall (H \leftarrow G) | D \& D
\]
This extension is inspired by multiset rewriting with universal quantification [CDL+99] (see Section 5.4). The resulting language will be called LO∀.

We now reformulate the proof-theoretical semantics of Section 3.2. A proof system for LO∀ can be directly obtained by slightly modifying the proof system presented in Figure 3.2. Specifically, we need to make signatures explicit and add a rule for the universal quantifier. The resulting proof system is shown in Figure 3.6. Now, sequents assume the form $P \vdash \Sigma, \Delta$, where $P$ is a program (as usual, a set of reusable clauses), $\Delta$ is a multiset of goals, and $\Sigma$ is a signature. The signature $\Sigma$ contains at least the constant, function, and predicate symbols of program $P$, and can dynamically grow thanks to rule $\forall_r$. The idea is that all formulas on the right-hand side of a sequent are implicitly assumed to range over $\Sigma$, i.e., only the symbols declared in $\Sigma$ can appear in formulas.

Rule $\forall_r$ is responsible for signature augmentation. Every time rule $\forall_r$ is fired, a new constant $c$ is added to the current signature, and the resulting goal is proved in the new signature. This behaviour is standard in logic programming languages [MNPS91]. Rule $bc$ denotes a backchaining (resolution) step (as usual $\hat{H}$ is the multiset consisting of the atoms in the disjunction $H$, with $\hat{\bot} = \epsilon$). According to the usual concept of uniformity, $bc$ can be executed only if the right-hand side of the current sequent consists of atomic formulas. Rules $\top_r$, $\exists_r$, $\&_r$, and $\bot_r$ are the same as in standard LO.

We can formulate the following proposition, which is analogous to Proposition 3.9.

**Proposition 3.14 (Admissibility of the Weakening Rule)** Given a program $P$, a signature $\Sigma$, and two multisets of goals $\Delta, \Delta'$ such that $\Delta \preceq \Delta'$, if $P \vdash \Sigma, \Delta$ then $P \vdash \Sigma, \Delta'$.

**Proof** By simple induction on the structure of proofs. \[\square\]

**Example 3.15** Let $\Sigma$ be a signature with a constant symbol $a$, a function symbol $f$ and predicate symbols $p, q, r, s$. Let $\mathcal{V}$ be a denumerable set of variables, and $u, v, w, \ldots \in \mathcal{V}$. 37
Let $P$ be the program

1. $r(w) \leftarrow q(f(w))$
2. $s(z) \leftarrow \forall x. p(f(x))$
3. $\bot \leftarrow q(u) \& r(v)$
4. $p(x) \otimes q(x) \leftarrow \top$

The goal $s(a)$ is provable from $P$. The corresponding proof is shown in Figure 3.7. Note that the notion of ground instance is now relative to the current signature. For instance, backchaining over clause 3 is possible because the corresponding signature contains the constant $c$ (generated one level below by the $\forall_r$ rule), and therefore $\bot \leftarrow q(f(c)) \& r(c)$ is a valid instance of clause 3.

**Note.** Subject of Part II of this thesis will be the definition of an alternative semantics for LO, based on bottom-up evaluation, and the investigation of the connection between provability in linear logic and verification (model checking). In particular, we will consider the fragment corresponding to propositional LO in Chapter 7, its extension with the constant $1$ and the connective $\otimes$ (called LO$_1$) in Chapter 8, a formulation of LO with constraints (called LO($\mathcal{C}$)) in Chapter 9, and a first-order formulation of LO with universal quantification (called LO$_\forall$) in Chapter 10. In this thesis, we follow the original formulation of the language LO, as given for instance in [AP91b]. We remark that papers like [AP91a] rely on a more complex computational interpretation for LO involving broadcast communication. In the following, we will use the name LO and its variations as a convenient shorthand to indicate the fragment of linear logic this language is based on, even though the correspondence between the semantics we are interested in and the actual language may be questionable.
Summary of the Chapter. In this chapter we have briefly illustrated the main ideas underlying linear logic. We have presented in more detail the language LO, its definition and a proof-theoretical, sequent-style, semantics for it. A bottom-up semantics for different fragments of LO will be the subject of Part II.

In the next chapter we will demonstrate the potentialities of linear logic for specifying concurrent systems, by discussing its relationship with well-known formalisms like rewriting systems and Petri nets.
Chapter 4

LO, Multiset Rewriting and Petri Nets

Petri nets [Rei85] and their extensions [Jen97] have proved to be a powerful formalism for the specification and validation of concurrent systems. In this chapter, we give a brief overview of Petri nets and we discuss their relationship with linear logic and specifically with the language LO of Section 3.2. In order to study this connection, we first present the theory of multiset rewriting systems, which can be used to give an alternative definition for Petri nets. After showing the connection between Petri nets and multiset rewriting systems, we show how the latter can be translated into LO theories. We conclude the chapter discussing an extension of Petri nets, namely Petri nets with transfer arcs.

4.1 Multiset Rewriting over Atomic Formulas

As discussed in [Cer94], multisets and multiset rewriting systems can be used to give an alternative definition for the static and dynamic aspects of Petri nets. For the sake of precision, in the following we will deal with multiset rewriting systems over atomic formulas as defined in [Cer94], as opposed to general rewriting systems [Mes92, DP01] or AC theories [RV95]. The definition is as follows.

**Definition 4.1 (Multiset Rewriting Systems over Atomic Formulas)** Let $S$ be a finite set (alphabet). A **multiset rewrite rule** $\mu$ over $S$ is a pair, written $\mu_1 \rightarrow \mu_2$, where $\mu_1$ and $\mu_2$ are two multisets over $S$ called the **antecedent** and the **consequent** of $\mu$, respectively. A **multiset rewriting system** over $S$ is a set $R$ of multiset rewrite rules over $S$.
Definition 4.2 (Configuration) Let \( R \) be a multiset rewriting system over \( S \). A configuration is a multiset \( M \) over \( S \).

Given a configuration \( M \), a rewrite rule can fire at \( M \) provided the corresponding antecedent is contained in \( M \). Firing is accomplished by substituting the consequent for the antecedent in \( M \), thus getting a new multiset \( M' \). This is formally stated in the following definition.

Definition 4.3 (Firing) Let \( R \) be a multiset rewriting system over \( S \), \( M \) a multiset over \( S \), and \( \mu = \mu_1 \rightarrow \mu_2 \in R \) a rule.

i. We say that \( \mu \) is \( M \)-enabled if \( \mu_1 \preceq M \);

ii. if \( \mu \) is \( M \)-enabled, a multiset \( M' \) is the result of firing \( \mu \) at \( M \), written \( M \triangleright^\mu M' \), if \( M' = (M \setminus \mu_1) + \mu_2 \); we write \( M \triangleright M' \) if \( M \triangleright^\mu M' \) for some \( \mu \), and we use \( \triangleright^\ast \) to denote the reflexive and transitive closure of \( \triangleright \).

The semantics of multiset rewriting systems is formalized by the following notion of reachability set.

Definition 4.4 (Reachability Set) Let \( R \) be a multiset rewriting system over \( S \). Given a multiset \( M_I \) over \( S \), called initial configuration, we define the reachability set of \( R \), denoted \( \text{Reach}(R) \), as follows: \( \text{Reach}(R) = \{ M | M_I \triangleright^\ast M \} \).

In the next section we will present the theory of Petri nets. The mutual relations between Petri nets, multiset rewriting systems and linear logic theories will be discussed in Section 4.3.

4.2 Place/Transition Nets

In the following we shall be interested mainly in Petri nets with infinite place capacities. This is not a serious limitation, however, as arbitrary Petri nets can be encoded into Petri nets with infinite place capacities by means of the well-known operation of complementation [Rei85].

The traditional definition of Petri net (with infinite place capacities) is as follows.

Definition 4.5 (P/T Nets) A P/T net is a tuple \( N = \langle P, T, A, W \rangle \), where
i. $P$ and $T$ are two finite and disjoint sets, whose elements are called, respectively, places and transitions;

ii. $A \subseteq (P \times T) \cup (T \times P)$ is a set of arcs, defined by a binary relation called the flow relation;

iii. $W : A \rightarrow \mathbb{N}_0$ is the weight function.

Definition 4.6 (Preset and Postset) Given a Petri net $N = \langle P, T, A, W \rangle$ and $t \in T$, we define $\cdot t = \{p \in P \mid (p, t) \in A\}$ and $t \cdot = \{p \in P \mid (t, p) \in A\}$. We call $\cdot t$ and $t \cdot$, respectively, the preset and postset of the transition $t$.

Definition 4.7 (Marking) A marking for a Petri net $N = \langle P, T, A, W \rangle$ is a multiset over $P$, i.e., a mapping $\mathcal{M} : P \rightarrow \mathbb{N}$.

Petri nets can be conveniently represented as labelled graphs, as in the producer/consumer example in Figure 4.1 (see also Section 4.2.1). In the graphical representation, places are drawn as circles, while transitions are drawn as squares. The flow relation is represented by arcs connecting places and transitions. Every arc is labelled with its own weight; labels with weight 1 are usually omitted for simplicity. Markings are represented by drawing as many tokens (i.e., black spots) into places, as the corresponding multiplicity. According to the token metaphor, firing of a transition $t$ causes tokens flow from places in $\cdot t$ to places in $t \cdot$, according to the weight of the corresponding arcs. A transition $t$ can only fire if enough tokens are present in the places in $\cdot t$. This is formalized by the following definition.

Definition 4.8 (Firing) Let $N = \langle P, T, A, W \rangle$ be a Petri net, $t \in T$ a transition, and $\mathcal{M}$ a marking for $N$:

i. We say that $t$ is $\mathcal{M}$-enabled iff $\mathcal{M}(p) \geq W(p, t)$ for all $p \in \cdot t$;

ii. if $t$ is $\mathcal{M}$-enabled, a marking $\mathcal{M}'$ is the result of firing $t$ at $\mathcal{M}$, written $\mathcal{M}[t] \mathcal{M}'$, iff for all $p \in P$

$$
\mathcal{M}'(p) = \begin{cases} 
\mathcal{M}(p) - W(p, t) & \text{iff } p \in \cdot t \setminus t \\
\mathcal{M}(p) + W(t, p) & \text{iff } p \in t \setminus \cdot t \\
\mathcal{M}(p) - W(p, t) + W(t, p) & \text{iff } p \in \cdot t \cap t \\
\mathcal{M}(p) & \text{otherwise}
\end{cases}
$$

we write $\mathcal{M}[\ast] \mathcal{M}'$ if $\mathcal{M}[t] \mathcal{M}'$ for some $t$, and we use $[\ast]$ to denote the reflexive and transitive closure of $\cdot$.
Similarly to multiset rewriting systems, we can give the following definition for the reachability set.

**Definition 4.9 (Reachability Set)** Let $N = (P, T, A, W)$ be a Petri net. Given a marking $M_I$ for $N$, called the initial marking, we define the reachability set of $N$, denoted $\text{Reach}(N)$, as follows: $\text{Reach}(N) = \{ M | M_I[\ast]M \}$.

### 4.2.1 A Producer/Consumer Example

A simple example representing a producer/consumer net, with initial marking, is shown in Figure 4.1. The producer, represented by the left cycle, produces and releases one item at a time and puts it in a buffer. The buffer has a limited capacity of five items. This is enforced by representing the buffer by means of two places: place `buffer` contains the items released and not yet consumed, while place `free` represents the available positions in the buffer. The consumer, represented by the right cycle, acquires and consumes two items at a time, as specified by the label 2 on the arcs from `buffer` to `Acquire` and from `Acquire` to `free`.

Formally, the set of places is given by $P = \{ \text{releasing, producing, buffer, free, acquiring, consuming} \}$, and the set of transitions by $T = \{ \text{Produce, Release, Acquire, Consume} \}$. The set of arcs includes for instance (Produce, releasing), (releasing, Release), and so on. Every arc has weight 1 except for the arcs (buffer, Acquire) and (Acquire, free) which have weight 2. The preset of Release is given by $\{ \text{releasing, free} \}$, while its postset is given by $\{ \text{producing, buffer} \}$. In Figure 4.1, the following initial marking $M_I$ = $\{ \text{releasing, buffer, buffer, buffer, free, free, acquiring} \}$.
is drawn, i.e., we have $M_I(\text{releasing}) = 1$, $M_I(\text{producing}) = 0$, $M_I(\text{buffer}) = 3$, $M_I(\text{free}) = 2$, $M_I(\text{acquiring}) = 1$, $M_I(\text{consuming}) = 0$. We can conveniently use the following vector notation: $\langle 1, 0, 3, 2, 1, 0 \rangle$, where each position represents the number of tokens in each place, ordered as above.

The following transitions are $M_I$-enabled in Figure 4.1: Release and Acquire. By firing the transition Release, we get that $M_I[\text{Release}]M'\', with $M'\' = \langle 0, 1, 4, 1, 1, 0 \rangle$, according to the above vector notation. Similarly, by firing transition Acquire we get $M_I[\text{Acquire}]M''\', with $M''\' = \langle 1, 0, 1, 4, 0, 1 \rangle$. As a further example, we can fire transition Consume in $M''\'$ and get $M_I[\text{Consume}]M''\'''\', with $M''\'''\' = \langle 0, 1, 4, 1, 0 \rangle$. Transition Acquire is not $M''\'''\'$-enabled, because only one item is available in the buffer.

4.2.2 Place Invariants

The notion of place invariant is useful to perform a static analysis of a Petri net. For instance, given a Petri net $N$ with set of places $P$, one can be interested in finding subsets of places $S \subseteq P$ such that their joint total token count does not change whatever transition may fire. This is a typical example of a property which can be statically proved by means of the place invariant analysis. Place invariants can be used to study properties of Petri nets like liveness or boundedness [Rei85].

In order to introduce the concept of place invariant, we first discuss a linear algebra representation for Petri nets.

**Definition 4.10 (Matrix Representation)** Let $N = \langle P, T, A, W \rangle$ be a Petri net.

i. Given a transition $t \in T$, we define the vector $t(p)$ as follows:

$$t(p) = \begin{cases} W(t,p) & \text{iff } p \in t^* \setminus t^* \\ -W(p,t) & \text{iff } p \in t^* \setminus t^* \\ W(t,p) - W(p,t) & \text{iff } p \in t^* \cap t^* \\ 0 & \text{otherwise} \end{cases}$$

ii. we define the matrix $N : P \times T \rightarrow \mathbb{Z}$ as follows: $N(p,t) = t(p)$.

Note that the above matrix representation is unambiguous (i.e., different Petri nets have different representations) only if we assume $t^* \cap t^* = \emptyset$ for every $t \in T$. Every marking of a Petri net can also be represented as a vector, as shown in Section 4.2.1. Let $M$ denote the vector corresponding to the multiset $M$, i.e., $M(p) = M(p)$ for every $p \in P$. The following result holds.
<table>
<thead>
<tr>
<th>Produce</th>
<th>Release</th>
<th>Acquire</th>
<th>Consume</th>
<th>(M_I)</th>
</tr>
</thead>
<tbody>
<tr>
<td>releasing</td>
<td>1</td>
<td>−1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>producing</td>
<td>−1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>buffer</td>
<td>0</td>
<td>1</td>
<td>−2</td>
<td>0</td>
</tr>
<tr>
<td>free</td>
<td>0</td>
<td>−1</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>acquiring</td>
<td>0</td>
<td>0</td>
<td>−1</td>
<td>1</td>
</tr>
<tr>
<td>consuming</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>−1</td>
</tr>
</tbody>
</table>

Figure 4.2: Matrix and initial marking representation

**Proposition 4.11** Let \(N = \langle P, T, A, W \rangle\) be a Petri net, \(t \in T\) a transition, and \(M, M'\) two markings for \(N\). If \(t\) is \(M\)-enabled, then \(M[t]M'\) iff \(M' = M + t\).

**Proof** By definitions.

**Example 4.12** Consider the producer/consumer Petri net in Figure 4.1. The corresponding matrix and initial marking vector are shown in Figure 4.2. The initial marking \(M_I\) is given by the vector \(\langle 1, 0, 3, 2, 1, 0 \rangle\), and the vector corresponding to transition Release is \(\langle -1, 1, 1, -1, 0, 0 \rangle\). As Release is \(M_I\)-enabled and \(\langle 1, 0, 3, 2, 1, 0 \rangle + \langle -1, 1, 1, -1, 0, 0 \rangle = \langle 0, 1, 4, 1, 1, 0 \rangle\), we have that \(M_I[\text{Release}]M, M\) corresponding to the vector \(\langle 0, 1, 4, 1, 1, 0 \rangle\).

We have the following definition for place invariants. In the following, let \(N^T\) denote the transpose of matrix \(N\) and \(i^T\) the transpose of vector \(i\), let \(*\) be the matrix-vector or vector-vector product, and \(\mathbf{0}\) a null vector.

**Definition 4.13 (Place Invariants)** Let \(N = \langle P, T, A, W \rangle\) be a Petri net. A place vector \(i: P \rightarrow \mathbb{Z}\) is called a place invariant of \(N\) iff \(N^T * i = 0\).

Place invariants are defined as solutions \(x\) of the linear algebraic system of equations \(N^T * x = 0\). Let \(\text{Reach}(N)\) be the set \(\{M | M \in \text{Reach}(N)\}\). Place invariants have the following property.

**Proposition 4.14 (Properties of Place Invariants [STC98])** Let \(N = \langle P, T, A, W \rangle\) be a Petri net, and \(M_I\) an initial marking for \(N\). If \(i\) is a place invariant of \(N\), then \(\text{Reach}(N) \subseteq \{x | i^T * x = i^T * M_I\}\).
Intuitively, reachable markings must satisfy the equation $i^T \cdot x = i^T \cdot M$ for every place invariant $i$. In other words, place invariants provide us with an over-approximation of the set of reachable markings of a given Petri net. As a special case, solutions $i$ with elements in \{0,1\} are characteristic vectors of sets of places with constant token count. We also have the following result.

**Proposition 4.15** Let $N = \langle P, T, A, W \rangle$ be a Petri net, $i_1$ and $i_2$ two place invariants of $N$, and let $z \in \mathbb{Z}$. Then $i_1 + i_2$ and $z \cdot i_1$ are also place invariants of $N$.

**Proof** By definition. \hfill \Box

Tools for automatically computing place invariants are available. For instance, specialized libraries for invariant computation are available with GreatSPN [CFGR95].

**Example 4.16** Let us consider the producer/consumer net of Figure 4.1 again. The corresponding matrix is shown in Figure 4.2. Let $x = \langle x_1, \ldots, x_6 \rangle$, $x_1, \ldots, x_6$ corresponding to places releasing, ..., consuming in the order of Figure 4.2. Now, the associated algebraic system is

\[
\begin{align*}
    x_1 - x_2 &= 0 \\
    -x_1 + x_2 + x_3 - x_4 &= 0 \\
    -2x_3 + 2x_4 - x_5 + x_6 &= 0 \\
    x_5 - x_6 &= 0
\end{align*}
\]

which reduces to

\[
\begin{align*}
    x_1 &= x_2 \\
    x_3 &= x_4 \\
    x_5 &= x_6
\end{align*}
\]

Every place vector satisfying the above equations is a place invariant. For instance, the vectors $\langle 1, 0, 0, 0, 0, 0 \rangle$, $\langle 0, 0, 1, 0, 0, 0 \rangle$, and $\langle 0, 0, 0, 0, 1, 1 \rangle$ are place invariants. Being characteristic vectors, we have that the total token count of the corresponding places is always constant and equals the initial marking count. Therefore we have proved the following properties: for every reachable marking $M$, $M(\text{releasing}) + M(\text{producing}) = 1$, $M(\text{buffer}) + M(\text{free}) = 5$, and $M(\text{acquiring}) + M(\text{consuming}) = 1$. It follows, e.g., that $M(\text{buffer}) \leq 5$ (the buffer cannot contain more than five elements), $M(\text{releasing}) \leq 1$, and so on. According to Proposition 4.15, every linear combination of the above place vectors is still a place invariant. For instance, $\langle 1, 1, 1, 1, 1, 1 \rangle$ is a place invariant, i.e., the total number of tokens in the net of Figure 4.1 is constant. \hfill \Box

In Part II, we will often use place invariants to optimize the performance of our verification tool (see Appendix A).

### 4.3 From Petri Nets to Linear Logic

After the informal discussion in the previous section, the existence of a connection between multiset rewriting systems and Petri nets should be evident. In fact, markings are exactly
multisets of places, whereas transitions can be seen as multiset rewrite rules, in accordance to the token metaphor. The following definitions make this connection explicit.

**Definition 4.17 (Petri Nets as Multiset Rewriting Systems)** Given a Petri net \( N = \langle P, T, A, W \rangle \), we define the multiset rewriting system \( R(N) \) over \( P \) as follows:

\[
R(N) = \left\{ \sum_{p \in \cdot t} W(p, t) \cdot p \longrightarrow \sum_{p \in t'} W(t, p) \cdot p \mid t \in T \right\}
\]

We remind that \( W(p, t) \cdot p \) denotes the scalar product between \( W(p, t) \in N \) and the singleton multiset \( \{p\} \).

The following proposition states soundness and completeness of the Petri nets translation into multiset rewriting systems. The proof can be found in [Cer94].

**Proposition 4.18** Let \( N = \langle P, T, A, W \rangle \) be a Petri net and \( M, M' \in MS(P) \) two markings over \( P \). Then \( M \models^* M' \) in \( N \) if and only if \( M \triangleright^* M' \) in \( R(N) \).

After giving a translation from Petri nets to multiset rewriting systems, we now show how multiset rewriting systems can be specified in linear logic, and in particular in the language LO presented in Section 3.2. As a result, we also get a translation of Petri Nets into linear logic.

**Definition 4.19 (Multiset Rewriting Systems as LO Theories)** Let \( R \) be a multiset rewriting system over a set \( S \). We define the LO theory \( P(R) \) over the signature \( \Sigma = S \), as follows:

\[
P(R) = \{ H \circ - G \mid \hat{H} \rightarrow \hat{G} \in R \}
\]

We recall that, given a \( \boxdot \) disjunction of atomic formulas \( F = a_1 \boxdot \ldots \boxdot a_n \), \( \hat{F} \) stands for \( a_1, \ldots, a_n \), while \( \bot \) stands for the empty multiset \( \epsilon \).

Soundness and completeness of the above translation is stated in the following proposition. We omit the proof, which is a slight variation of the one presented in [Cer94] (see also [Cer95]) and can be developed using the same methodology.

**Proposition 4.20 (Reachability as Provability in Propositional LO)** Let \( R \) be a multiset rewriting system over a set \( S \), \( M, M' \in MS(S) \) two multisets over \( S \), and \( H, G \) the (possibly empty) \( \boxdot \) disjunctions of atomic formulas such that \( \hat{H} = M' \) and \( \hat{G} = M \). Then \( M \triangleright^* M' \) in \( R \) if and only if \( P(R), H \circ 1 \vdash_1 G \).
Example 4.21 Let us consider the Petri net for the producer/consumer example (see Figure 4.1). Its translation yields the following multiset rewriting system (one rule for each transition in \( T = \{ \text{Produce, Release, Acquire, Consume} \} \), respectively):

\[
\begin{align*}
\{ \text{producing} \} & \rightarrow \{ \text{releasing} \} \\
\{ \text{releasing, free} \} & \rightarrow \{ \text{producing, buffer} \} \\
\{ \text{buffer, buffer, acquiring} \} & \rightarrow \{ \text{free, free, consuming} \} \\
\{ \text{consuming} \} & \rightarrow \{ \text{acquiring} \}
\end{align*}
\]

which corresponds to the following LO program \( P \):

1. \( \text{producing} \circ \rightarrow \text{releasing} \)
2. \( \text{releasing} \circ \text{free} \circ \rightarrow \text{producing} \circ \text{buffer} \)
3. \( \text{buffer} \circ \text{buffer} \circ \text{acquiring} \circ \rightarrow \text{free} \circ \text{free} \circ \text{consuming} \)
4. \( \text{consuming} \circ \rightarrow \text{acquiring} \)

By firing, in the following order, the transitions Release, Produce, and Acquire, we have that \( \mathcal{M}_I[\ast] \mathcal{M} \), where \( \mathcal{M}_I \) is the initial marking drawn in Figure 4.1, and \( \mathcal{M} \) is the marking corresponding to the vector \( \langle 1, 0, 2, 3, 0, 1 \rangle \) (according to the notation of Section 4.2.1). Let \( D \) be the following \( \text{LO}_1 \) clause:

\[
\text{releasing} \circ \text{buffer} \circ \text{buffer} \circ \text{free} \circ \text{free} \circ \text{free} \circ \text{free} \circ \text{consuming} \circ \rightarrow 1
\]

Figure 4.3 (where applications of the \( \circ \) rules have been incorporated into backchaining steps, labelled as usual) shows the \( \text{LO}_1 \) proof corresponding to \( \mathcal{M}_I[\ast] \mathcal{M} \), according to Proposition 4.20.

4.4 Petri Nets with Transfer Arcs

We conclude this chapter by discussing an extension of Petri nets, namely Petri nets with \textit{transfer arcs}. After presenting an example, we will discuss the connection with the so-
called broadcast protocol model [EN98]. We will give a definition and a formal semantics for broadcast protocols. We will go back to this topic in Chapter 8. In particular, in Section 8.4 we will discuss the connection between the broadcast protocol model and linear logic.

4.4.1 A Readers/Writers Example

Figure 4.4 shows a net representing a set of (identical) processes which compete for a shared resource (e.g. a memory location). Processes can be reading or writing the resource, or otherwise be idle (correspondingly, we have the places reading, writing, and idle). In Figure 4.4, we have drawn an initial marking in which we have three reading processes and two idle ones.

Processes are allowed to concurrently read the given resource, while writing must be exclusive (only one write operation and no read ones at a time). This constraint is enforced by using an invalidation strategy which works as follows. Every time a read request is sent (transition Read_{req}), the requesting process is granted the read access after all the writing processes (in this case at most one) are moved from writing to idle. In a similar way, a write request (transition Write_{req}) necessitates moving all processes currently in state reading or writing to state idle. Two further transitions, End_{read} and End_{write}, are straightforward, and allow a reading or writing process to go back to state idle. In Figure 4.4 we have drawn ordinary Petri net arcs in the usual way, while dotted lines (which come in pairs) represent transfer arcs. For instance, the pair of dotted lines from reading to
Write_req and from Write_req to idle represent the transfer of all processes in state reading to state idle which is performed upon reception of a write request. Intuitively, the effect of a transfer arc is to move all tokens in a given place to another place. Transfer arcs do not affect enabling of the corresponding transition.

There is a strong connection between Petri nets with transfer arcs and broadcast protocols. In the next section we will present the definition of broadcast protocol and we will make the ideas presented above more precise by showing a formal semantics for this latter model.

### 4.4.2 Broadcast Protocols

Broadcast protocols [EN98, EFM99] are systems composed of a finite but arbitrarily large set of identical processes, which can communicate by rendezvous (a message is exchanged between two processes) or broadcast (a process sends a message to all the other processes). Typical examples of systems which can be modeled as broadcast protocols are bus-based hardware protocols, e.g. cache coherence protocols.

**Definition 4.22 (Broadcast Protocol)** A broadcast protocol is a triple \( B = (S, L, R) \), where

1. \( S \) is a finite set of states;

2. \( L \) is a finite set of labels, defined in the following way. Given three finite and mutually disjoint sets \( \Sigma_l, \Sigma_r \) and \( \Sigma_b \), \( L \) is the union of \( \Sigma_i \) (the set of local labels), two sets \( (\Sigma_r \times \{?\}) \) and \( (\Sigma_r \times \{!\}) \) (the sets of input and output rendezvous labels), and two sets \( (\Sigma_b \times \{??\}) \) and \( (\Sigma_b \times \{!!\}) \) (the sets of input and output broadcast labels). We will write \( \Sigma_L \) for \( \Sigma_l \cup \Sigma_r \cup \Sigma_b \), and we will shorten \( (a, ?) \) to \( a? \), and similarly for the other labels;

3. \( R \subseteq S \times L \times S \) is a set of transitions. A transition \( (s, l, s') \) will be written \( s \xrightarrow{l} s' \). \( R \) must satisfy the following property: for every \( a \in \Sigma_b \) and state \( s \in S \), there exists a state \( s' \in S \) such that \( s \xrightarrow{a??} s' \).

Intuitively, the last condition on the set of transitions guarantees that every process is always willing to receive a broadcasted message. For the sake of simplicity, we will make two further assumptions on the set \( L \): for each state \( s \) and label \( a?? \), there is exactly one state \( s' \) such that \( s \xrightarrow{a??} s' \) (determinism); each label of the form \( a, a? a! \) and \( a!! \) appears in exactly one transition.

**Example 4.23** The readers/writers net presented in Figure 4.4 can be translated into the broadcast protocol model, as shown in Figure 4.5 (where we have not drawn the initial
marking). The are three states, reading, writing and idle, as usual. A process willing to read broadcasts an (output) message Read_req!! to all the other processes. Upon receipt of the correspondent input message Read_req??, processes currently reading remain in state reading, while writing processes become idle. The transition for a write request is similar, with the difference that both reading and writing processes go to state idle. Transitions End_read and End_write are described by local labels, and their meaning is similar to ordinary Petri net transitions. No rendezvous transitions are needed in this example.

We now present the semantics of broadcast protocols.

**Definition 4.24 (Configuration)** A configuration for a broadcast protocol $B = \langle S, L, R \rangle$ is a multiset over $S$, i.e., a mapping $C : S \rightarrow \mathbb{N}$.

The notion of configuration is analogous to that of marking for Petri nets. Suppose $S = \{s_1, \ldots, s_n\}$. Then, a configuration $C$ can be equivalently seen as a vector $\langle c_1, \ldots, c_n \rangle$, where $c_i = C(s_i)$ (the multiplicity of $s_i$ in $C$) indicates how many processes are in state $s_i$ [EFM99].

**Definition 4.25 (Firing)** Let $B = \langle S, L, R \rangle$ be a broadcast protocol, $S = (s_1, \ldots, s_n)$, $a \in \Sigma_L$, and $C$ a configuration for $B$. We say that $C'$ is the result of firing $a$ at $C$, written $C \xrightarrow{a} C'$, iff
i. \( s_i \xrightarrow{a} s_j \in R \), with \( C(s_i) > 0 \), and \( C' = (C \setminus \{s_i\}) + \{s_j\} \);

ii. \( s_i \xrightarrow{a'} s_j \in R \) and \( s_k \xrightarrow{a''} s_l \in R \), with \( C(s_i) > 0 \) and \( C(s_k) > 0 \), and \( C' = (C \setminus \{s_i, s_k\}) + \{s_j, s_l\} \);

iii. \( s_i \xrightarrow{a!!} s_j \in R \), with \( C(s_i) > 0 \), and \( C' \) can be computed from \( C \) in the following three steps:

\[
C_1 = C \setminus \{s_i\}
\]

\[
C_2(s_k) = \sum_{s_l \mid s_l \xrightarrow{a''} s_k} C_1(s_l) \quad (\text{for } k : 1, \ldots, n)
\]

\[
C' = C_2 + \{s_j\}
\]

The first two cases in the previous definition are for local and rendezvous transitions, and similar to ordinary Petri net transitions. The last case is for broadcast communication. Intuitively, the sending process leaves \( s_i \), all other processes receive the broadcast and move to their destinations, and finally the sending process reaches \( s_j \). This is similar to a Petri net transition having pairs of transfer arcs for all places and in addition an ordinary pair of arcs, connecting \( s_i \) and \( s_j \), for the sending process. From a graphical point of view, note that in Figure 4.4 pairs of transfer arcs which are self loops (e.g. corresponding to the arc Read_req?? for the state reading in Figure 4.5) are omitted. It should be evident that broadcast protocols can be seen as a special case of Petri nets with transfer arcs. The semantics of these latter nets can be given similarly.

**Example 4.26** Consider the readers/writers net in Figure 4.5 and the configuration \( C \), corresponding to the initial marking of Figure 4.4, with 3 readers and 2 idle processes, i.e., \( C(\text{reading}) = 3 \), \( C(\text{writing}) = 0 \), \( C(\text{idle}) = 2 \). Using the vector notation, \( C \) is given by \( \langle 3, 0, 2 \rangle \). Then \( C \xrightarrow{\text{Read}_\text{req}} C' \), with \( C' \) given by \( \langle 4, 0, 1 \rangle \); \( C' \xrightarrow{\text{Write}_\text{req}} C'' \), with \( C'' \) given by \( \langle 0, 1, 4 \rangle \); \( C'' \xrightarrow{\text{End}_\text{write}} C''' \), with \( C''' \) given by \( \langle 0, 0, 5 \rangle \).

**Summary of the Chapter.** In this chapter we have introduced the theories of multiset rewriting systems over atomic formulas and Petri nets, we have discussed their definition and semantics. We have analyzed the mutual relations between linear logic, multiset rewriting systems and Petri nets. We have illustrated the main ideas of this connection by means of classical examples, like producers/consumers and readers/writers. We have also mentioned the broadcast protocol model. In the next chapter we will carry on the discussion by analyzing the relations between first-order linear logic theories, first-order rewriting systems and coloured Petri nets.
Chapter 5

First-Order LO, First-Order Multiset Rewriting and Coloured Petri Nets

According to the token metaphor discussed in Chapter 4, Petri nets dynamics can be described by means of tokens flowing from places to places. While the tokens considered so far are just black spots, i.e., indistinguishable from one another, a natural extension is to allow tokens to carry values, like in the so-called theory of Coloured Petri Nets (CP-nets for short) [Jen97]. The logical counterpart of CP-nets is to consider multiset rewriting over first-order atoms, and first-order LO theories, as opposed to propositional ones. In this chapter we summarize the basic concepts pertaining to CP-nets and we discuss their relationship with multiset rewriting over first-order atoms and first-order linear logic theories. We conclude by showing how enriching logic theories with universal quantification can provide a way to generate new values.

5.1 Multiset Rewriting Systems over First-Order Atomic Formulas

The theory of multiset rewriting systems presented in Section 4.1 can be extended in a straightforward way, by considering a set of rules built over first-order atomic formulas, and their ground instances. Given a denumerable set of variables \( \mathcal{V} \) and a signature \( \Sigma \) including a set of constant and function symbols and a set of predicate symbols, atomic formulas have the form \( p(t_1, \ldots, t_n) \), where \( p \) is a predicate symbol and \( t_i \) are terms in \( T^{\mathcal{V}}_\Sigma \), for \( i : 1, \ldots, n \).

Definition 5.1 (Multiset Rewriting Systems over First-Order Atoms) Let \( \Sigma \) be a signature with predicates and \( \mathcal{V} \) a denumerable set of variables. A multiset rewrite
**rule** \( \mu \) over \( \Sigma \) and \( \mathcal{V} \) is a pair, written \( \mu_1 \rightarrow \mu_2 \), where \( \mu_1 \) and \( \mu_2 \) are two multisets of (non ground) atomic formulas in \( A^\Sigma_{\mathcal{V}} \), called the **antecedent** and the **consequent** of \( \mu \), respectively. A **multiset rewriting system** over \( \Sigma \) and \( \mathcal{V} \) is a set \( R \) of multiset rewrite rules over \( \Sigma \) and \( \mathcal{V} \).

The dynamic behaviour of a multiset rewriting system over first-order atomic formulas can be described by considering the ground instances of the set of its rewrite rules. Given a signature \( \Sigma \), a possible way to achieve this is to consider a \( \Sigma \)-algebra \( D_\Sigma \) to provide a set of values (domain) and to evaluate the function and constant symbols in \( \Sigma \). By analogy with Herbrand models for logic programs, in the following we assume that values are provided by \( T_\Sigma \), i.e., the ground term algebra over \( \Sigma \). Therefore, values are simply ground terms. The set of ground instances of a rule \( \mu \) are obtained as applications of the rule itself with an evaluation \( \sigma \), mapping variables in \( \mu \) to terms in \( T_\Sigma \). We denote by \( \mu_\sigma \) this application.

**Definition 5.2 (Configuration)** Let \( R \) be a multiset rewriting system over \( \Sigma \) and \( \mathcal{V} \). A **configuration** is a multiset \( \mathcal{M} \) of ground atomic formulas in \( A_\Sigma \).

Given a configuration \( \mathcal{M} \), a rewrite rule can fire at \( \mathcal{M} \) (modulo an evaluation \( \sigma \)) provided the corresponding antecedent is contained in \( \mathcal{M} \). Firing is accomplished by substituting the consequent for the antecedent in \( \mathcal{M} \), thus getting a new multiset \( \mathcal{M}' \). This is formally stated in the following definition.

**Definition 5.3 (Firing)** Let \( R \) be a multiset rewriting system over \( \Sigma \) and \( \mathcal{V} \), \( \mathcal{M} \) a configuration, \( \mu = \mu_1 \rightarrow \mu_2 \in R \) a rule, and \( \sigma \) an evaluation for the variables in \( \mu \).

i. We say that \( \mu \) is **\( \mathcal{M} \)-enabled** with \( \sigma \) if \( \mu_1\sigma \preceq \mathcal{M} \);

ii. if \( \mu \) is \( \mathcal{M} \)-enabled with \( \sigma \), a multiset \( \mathcal{M}' \) is the result of **firing** \( \mu \) at \( \mathcal{M} \) with \( \sigma \), written \( \mathcal{M} \triangleright^{\mu,\sigma} \mathcal{M}' \), if \( \mathcal{M}' = (\mathcal{M}\backslash\mu_1\sigma) + \mu_2\sigma \); we write \( \mathcal{M} \triangleright \mathcal{M}' \) if \( \mathcal{M} \triangleright^{\mu,\sigma} \mathcal{M}' \) for some \( \mu \) and \( \sigma \), and we use \( \triangleright^* \) to denote the reflexive and transitive closure of \( \triangleright \).

The usual notion of reachability set can be extended as follows.

**Definition 5.4 (Reachability Set)** Let \( R \) be a multiset rewriting system over \( \Sigma \) and \( \mathcal{V} \). Given a multiset of values \( \mathcal{M}_I \), called **initial configuration**, we define the **reachability set** of \( R \), denoted \( \text{Reach}(R) \), as follows: \( \text{Reach}(R) = \{ \mathcal{M} \mid \mathcal{M}_I \triangleright^* \mathcal{M} \} \).

In the next section we will present the theory of coloured Petri nets. The mutual relations between coloured Petri nets, multiset rewriting systems over first-order atoms and first-order linear logic theories will be discussed in Section 5.3.
5.2 CP-nets

A (simplified) definition for Coloured Petri Nets (CP-nets) [Jen97] is as follows.

**Definition 5.5 (CP-Nets)** A CP-net is a tuple $N = \langle S, P, T, A, C, E \rangle$ where

i. $S$ is a set of **types** (i.e., domains enriched with operations defined over them);

ii. $P$ and $T$ are two finite and disjoint sets, whose elements are called, respectively, **places** and **transitions**;

iii. $A \subseteq (P \times T) \cup (T \times P)$ is a set of arcs, defined by a binary relation called the **flow** relation;

iv. $C : P \to S$ is a function called the **color function**, which associates every place with a type;

v. $E$ is a function defined from $A$ into expressions, called the **arc expression** function, which associates every arc with a multiset of expressions. Expressions must be of the correct type, i.e., if $a = (p, t)$ or $a = (t, p)$ then $E(a)$ must have the type of multisets with elements of type $C(p)$.

By analogy with first-order multiset rewriting theories, the definition of $S$ can be given by means of a (multi-sorted) algebra, defined over a signature $\Sigma$ including constant and function symbols. In this case, arcs may be labelled with multisets of expressions in $T^V_\Sigma$, i.e., the term algebra over $\Sigma$ and a set of variables $V$.

For convenience, in the following definition we overload the notations for presets and postsets used for ordinary Petri nets.

**Definition 5.6 (Preset and Postset)** Given a CP-net $N = \langle S, P, T, A, C, E \rangle$ and $t \in T$, we define $\cdot t = \{ p \in P \mid (p, t) \in A \}$ and $t \cdot = \{ p \in P \mid (t, p) \in A \}$. We call $\cdot t$ and $t \cdot$, respectively, the **preset** and **postset** of the transition $t$.

We now give the definition of binding and marking. The notion of binding is the counterpart of the notion of substitution for free variables in first-order logic.

**Definition 5.7 (Bindings and Markings)** Let $N = \langle S, P, T, A, C, E \rangle$ be a CP-net.

i. A **binding** for a transition $t \in T$ is a function mapping variables in $t$ (i.e., variables in $E(a)$, with $a = (p, t)$ or $a = (t, p)$ for some place $p$) to elements in $S$;
ii. a token element is a pair \((p, c)\), where \(p \in P\) and \(c \in C(p)\);

iii. a binding element is a pair \((t, b)\), where \(t \in T\) and \(b\) is a binding for \(t\);

iv. a marking is a multiset of token elements.

Given a binding \(b\) and an expression \(F\), in the following we will use the notation \(F^<b>\) to denote the application of \(b\) to \(F\). To exemplify, suppose a CP-net is defined by means of a \(\Sigma\)-algebra \(D_\Sigma\) and \(F\) is a term in \(T_\Sigma^V\). In this case, \(F^<b>\) is the evaluation of \(F\) obtained by evaluating variables in \(F\) according to \(b\), and constant and function symbols according to \(D_\Sigma\). For instance, if we choose \(D_\Sigma\) to be the ground term algebra \(T_\Sigma\), \(b\) must map variables to ground terms, and \(F^<b>\) is the usual term evaluation (i.e., constant and function symbols are evaluated to themselves). Evaluation can be straightforwardly extended to multisets of expressions. Finally, we will use the notation \(M(p)\), where \(M\) is a marking and \(p\) a place, to denote the multiset \(C\) over \(C(p)\) given by \(C(c) = M(p,c)\), for every \(c \in C(p)\).

Similarly to Petri nets, CP-nets can be represented as labelled graphs. An example is shown in Figure 5.1 (see also Section 5.2.1). As usually, places are drawn as circles and transitions as squares. Every arc \(a\) is labelled with the multiset of expressions \(E(a)\). Note that the multiset \(E(a)\) extends the weight function of traditional Petri nets. In fact, in Petri nets there is only one kind of token, and we can see the weight function as giving the multiplicity of this token in the corresponding multiset. Markings can be represented by drawing as many tokens, i.e., elements in \(S\), inside the corresponding place. The token metaphor can be extended to CP-nets. Firing of a transition is accomplished by first binding the free variables specified over the arcs of the transition itself to elements in \(S\), thus getting a binding element, and then by letting elements flow from places to places, provided the binding element is enabled. We have the following definition.

**Definition 5.8 (Firing)** Let \(N = \langle S, P, T, A, C, E \rangle\) be a CP-net, \((t, b)\) a binding element, and \(M\) a marking for \(N\).

i. We say that \((t, b)\) is \(M\)-enabled iff \(E(p, t)^<b> \preceq M(p)\) for all \(p \in t\).

ii. if \((t, b)\) is \(M\)-enabled, a marking \(M'\) is the result of firing \((t, b)\) at \(M\), written \(M[t, b]M'\), iff for all \(p \in P\)

\[
M'(p) = \begin{cases} 
M(p) \setminus E(p, t)^<b> & \text{iff } p \in t \setminus t^* \\
M(p) + E(t, p)^<b> & \text{iff } p \in t^* \setminus t \\
(M(p) \setminus E(p, t)^<b>) + E(t, p)^<b> & \text{iff } p \in t^* \cap t^* \\
M(p) & \text{otherwise}
\end{cases}
\]

we write \(M[\ast]M'\) if \(M[t, b]M'\) for some \(t\) and \(b\), and we use \(\ast\) to denote the reflexive and transitive closure of \(\ast\).
The usual notion of reachability set can be extended as follows.

**Definition 5.9 (Reachability Set)** Let $N = \langle S, P, T, A, C, E \rangle$ be a CP-net. Given a marking $M_I$ for $N$, called the initial marking, we define the reachability set of $N$, denoted $\text{Reach}(N)$, as follows: $\text{Reach}(N) = \{ M | M_I [ \ast ] M \}$.

### 5.2.1 An Example: the Dining Philosophers

In Figure 5.1 we show a net representing the well-known dining philosophers problem. We have a set of $n$ philosophers sitting around a (round) table and alternating periods in which they think to periods in which they eat. Each philosopher has a dish of spaghetti in front of him/her and two forks lie on each side of the dish (every fork is in common between two neighbouring dishes). A philosopher wishing to eat needs to take both forks, the one to the right and the one to the left of his/her dish. If either fork has already been taken, a philosopher must wait until both forks are available. Once he/she has finished eating, a philosopher puts the forks back onto the table and starts thinking again.

In Figure 5.1, we show an example for $n = 5$ philosophers, named $p_i$, $i : 0, \ldots, 4$. The forks are named $f_i$, $i : 0, \ldots, 4$, $f_i$ being the fork to the left of philosopher $p_i$, and $f_{(i+1 \mod 5)}$ the fork to the right. The set of places is $P = \{ \text{thinking, forks, eating} \}$, containing, respectively, the set of thinking philosophers, the set of available forks, and the set of eating philosophers. We have two distinct types, namely a type $S_p$ for philosophers and a
type $S_f$ for forks. The functions $l$ and $r$ map philosophers to the respective left and right fork, as defined above. The transition $\text{Get}_{\text{forks}}$ moves a philosopher $x$ from thinking to eating, provided his/her forks $l(x)$ and $r(x)$ are available. The transition $\text{Put}_{\text{forks}}$ moves a philosopher $x$ from eating to thinking, releasing forks $l(x)$ and $r(x)$.

Initially, all philosophers are thinking and all forks are on the table. In Figure 5.1 the following marking is shown:

$$M = \{(\text{thinking}, p_1), (\text{thinking}, p_3), (\text{thinking}, p_4), (\text{forks}, f_4), (\text{eating}, p_0), (\text{eating}, p_2)\},$$

i.e., we have that $M(\text{thinking}) = \{p_1, p_3, p_4\}$, $M(\text{forks}) = \{f_4\}$, and $M(\text{eating}) = \{p_0, p_2\}$. Let $b_1$ be the binding such that $b_1(x) = p_0$. The binding element $(\text{Put}_{\text{forks}}, b_1)$ is enabled in $M$. We have that $M[\text{Put}_{\text{forks}}, b_1]M'$, where $M'(\text{thinking}) = \{p_0, p_1, p_3, p_4\}$, $M'(\text{forks}) = \{f_0, f_1, f_4\}$, and $M'(\text{eating}) = \{p_2\}$. As the reader can verify, the transition $\text{Get}_{\text{forks}}$ is not enabled in $M$ for any binding $b$, because only one fork is available (at most two philosophers can eat at the same time). As another example, let $b_2$ be the binding such that $b_2(x) = p_4$. The binding element $(\text{Get}_{\text{forks}}, b_2)$ is enabled in $M'$, and $M'[\text{Get}_{\text{forks}}, b_2]M''$, where $M''(\text{thinking}) = \{p_0, p_1, p_3\}$, $M''(\text{forks}) = \{f_1\}$, and $M''(\text{eating}) = \{p_2, p_4\}$.

### 5.3 From CP-nets to Linear Logic

In this section we will discuss the connections between CP-nets, first-order multiset rewriting and first-order linear logic theories. Basically, we can use the same ingredients as in Section 4.3, where we formally stated a similar connection between ordinary Petri nets, rewriting and propositional linear logic theories. However, the correspondence between CP-nets as defined in [Jen97] and first-order multiset rewriting (or linear logic theories) is not perfect. In particular, our linear logic theories as presented so far are untyped (though a type theory could certainly be introduced), and furthermore in [Jen97] CP-nets are provided with a rich ML-like language for defining operations on type values. For these reasons, and having in mind the purpose of the present thesis work, we will feel free to keep the tone of this discussion somewhat informal. In particular, we will demonstrate the connection between CP-nets and multiset rewriting over first-order atoms by directly showing the encoding of the dining philosophers example of Section 5.2.1.

**Example 5.10** Consider the dining philosopher CP-net of Figure 5.1. The idea is to take the set of places as the set of predicate symbols of the corresponding theory. We have three places which we can encode by means of the predicate symbols thinking, forks and eating. We need at least two function symbols $l$ and $r$ to encode the left and right fork functions, and we can also use two further term constructors $p$ and $f$ to simulate the two types for philosophers and forks, respectively. We can use natural numbers to encode philosopher
indices. To exemplify, the terms $p(0), \ldots, p(n)$ will denote the set of $n$ philosophers, and $f(0), \ldots, f(n)$ the set of forks. The initial marking presented in Figure 5.1 corresponds to the following multiset:

$$\mathcal{M} = \{\text{thinking}(p(1)), \text{thinking}(p(3)), \text{thinking}(p(4)), \text{forks}(f(4)), \text{eating}(p(0)), \text{eating}(p(2))\}.$$  

The transition Get_forks can be encoded by means of the following rewrite rule, where $x$ is a variable:

$$\{\text{thinking}(p(x)), \text{forks}(l(p(x))), \text{forks}(r(p(x)))\} \rightarrow \{\text{eating}(p(x))\}$$

and, similarly, the following rewrite rule encodes the transition Put_forks:

$$\{\text{eating}(p(x))\} \rightarrow \{\text{thinking}(p(x)), \text{forks}(l(p(x))), \text{forks}(r(p(x)))\}$$

Now, we need to define rules to implement the $l$ and $r$ functions, e.g. using the following encoding:

$$\{\text{forks}(l(p(x)))\} \rightarrow \{\text{forks}(f(x))\}$$
$$\{\text{forks}(r(p(x)))\} \rightarrow \{\text{forks}(f(x + 1 \mod n))\}$$
$$\{\text{forks}(f(x))\} \rightarrow \{\text{forks}(l(p(x)))\}$$
$$\{\text{forks}(f(x + 1 \mod n))\} \rightarrow \{\text{forks}(r(p(x)))\}$$

where the $\mod$ operator is defined via a suitable arithmetic theory. A simple alternative is to consider a set of rules, one for each possible value of $x$, as suggested in Figure 5.1 (in this case the set of values is finite). In general, we need to enrich our rewriting theory with clauses for all the operations defined in the original CP-net.

The connection between multiset rewriting systems over first-order atomic formulas and first-order LO theories can be stated by analogy with the propositional case of Section 4.3.

**Definition 5.11 (First-Order Multiset Rewriting Systems as LO Theories)** Let $R$ be a multiset rewriting system over $\Sigma$ and $\mathcal{V}$. We define the LO theory $\mathcal{P}(R)$ (over $\Sigma$ and $\mathcal{V}$), as follows:

$$\mathcal{P}(R) = \{\forall (H \leftarrow G) \mid \widehat{H} \rightarrow \widehat{G} \in R\}$$

As usual, the notation $\forall (H \leftarrow G)$ stands for the universal quantification of clause $H \leftarrow G$ over its free variables, whereas $a_1 \overline{\varphi} \ldots \overline{\varphi} a_n$ stands for $a_1, \ldots, a_n$, with $\overline{\varphi} = \epsilon$. An analogous soundness and completeness result holds.

**Proposition 5.12 (Reachability as Provability in First-Order LO)** Let $R$ be a multiset rewriting system over $\Sigma$ and $\mathcal{V}$, $\mathcal{M}, \mathcal{M}' \in \mathcal{MS}(\mathcal{A}_\Sigma)$ two configurations, and $H$, $G$ the (possibly empty) $\overline{\varphi}$-disjunctions of ground atomic formulas such that $\widehat{H} = \mathcal{M}'$ and $\widehat{G} = \mathcal{M}$. Then $\mathcal{M} \vdash^* \mathcal{M}'$ in $R$ if and only if $\mathcal{P}(R), H \leftarrow 1 \vdash_1 G$.  

59
We can collect the multiset rewriting rules of Example 5.10 into the following convention: a formula like \( \text{thinking}(p(0), p(1), p(3)), \text{forks}(f(1)), \text{eating}(p(2), p(4)) \) is shown in Figure 5.2. We have used the following first-order LO program

\[
\begin{align*}
P, D \vdash \text{thinking}(p(0), p(1), p(3), p(4)), &\text{forks}(r(p(4)), f(1), l(p(4))), \text{eating}(p(2)) \\
P, D \vdash \text{thinking}(p(0), p(1), p(3), p(4)), &\text{forks}(f(0), f(1), l(p(4))), \text{eating}(p(2)) \\
P, D \vdash \text{thinking}(p(0), p(1), p(3), p(4)), &\text{forks}(f(0), r(p(0)), f(4)), \text{eating}(p(2)) \\
P, D \vdash \text{thinking}(p(0), p(1), p(3), p(4)), &\text{forks}(l(p(0)), r(p(0)), f(4)), \text{eating}(p(2)) \\
P, D \vdash \text{thinking}(p(1), p(3), p(4)), &\text{forks}(f(4)), \text{eating}(p(0), p(2))
\end{align*}
\]

Figure 5.2: Reachability as provability in first-order LO

Example 5.13 We can collect the multiset rewriting rules of Example 5.10 into the following first-order LO program \( P \) (where \( x \) is implicitly universally quantified):

1. \( \text{thinking}(p(x)) \not\exists \text{forks}(l(p(x))) \not\exists \text{forks}(r(p(x))) \imp \text{eating}(p(x)) \)
2. \( \text{eating}(p(x)) \imp \text{thinking}(p(x)) \not\exists \text{forks}(l(p(x))) \not\exists \text{forks}(r(p(x))) \)
3. \( \text{forks}(l(p(x))) \imp \text{forks}(f(x)) \)
4. \( \text{forks}(r(p(x))) \imp \text{forks}(f(x + 1 \mod n)) \)
5. \( \text{forks}(f(x)) \imp \text{forks}(l(p(x))) \)
6. \( \text{forks}(f(x + 1 \mod n)) \imp \text{forks}(r(p(x))) \)

Let \( D \) be the following LO\(_1\) clause:

\[
\text{thinking}(p(0)) \not\exists \text{thinking}(p(1)) \not\exists \text{thinking}(p(3)) \not\exists \text{forks}(f(1)) \not\exists \text{eating}(p(2)) \not\exists \text{eating}(p(4)) \imp 1.
\]

The derivation corresponding to firing \( M[\text{Put}_\text{forks}(b_1), M'[\text{Get}_\text{forks}(b_2), M''] \) of Section 5.2.1 (according to Proposition 5.12) is shown in Figure 5.2. We have used the following convention: a formula like \( \text{thinking}(p_1, \ldots, p_n) \) is a shorthand for the multiset \( \text{thinking}(p_1), \ldots, \text{thinking}(p_n) \), and similarly for \( \text{forks}(\ldots) \) and \( \text{eating}(\ldots) \). As usual, applications of the \( \not\exists \) rules have been incorporated into backchaining steps. \( \square \)
5.4 Multiset Rewriting with Universal Quantification

We conclude this chapter by discussing an extension of first-order multiset rewriting with universal quantification. In the proof as computation interpretation of logic programs, universal quantification is a logical operator which provides a way to generate new values. From a logical perspective, this view of universal quantification is based on its proof-theoretical semantics in intuitionistic (and also linear) logic [MNPS91]. We will define first-order multiset rewriting with universal quantification taking inspiration from [CDL+99], where a similar logic fragment, called MSR, is defined. In [CDL+99], MSR is used for the specification and analysis of security protocols, a topic we will discuss in Chapter 11.

Given the direct relationship between (first-order) multiset rewriting and (first-order) linear logic (in particular the fragment LO, as shown in Section 4.3), it should be evident that multiset rewriting with universal quantification is the counterpart of LO with universal quantification (see Section 3.3.2). As an aside, we remark that in [CDL+99] the logic MSR is compared to a fragment of linear logic which turns out to be dual with respect to ours, and therefore existential quantification is used in place of universal quantification.

The following definitions extend those presented in Section 5.1. As new values (constants) can be generated, multiset rewriting systems are defined on dynamically growing signatures, which we must keep trace of. Given a signature (with predicates) $\Sigma$ and a set of new constants $C$ (not appearing in $\Sigma$), we will use the notation $\Sigma \cup C$ to denote the addition of the constants in $C$ to the set of constants in $\Sigma$.

**Definition 5.14 (Quantified Multiset Rewriting Systems)** Let $\Sigma$ be a signature with predicates and $V$ a denumerable set of variables. A quantified multiset rewrite rule $\mu$ over $\Sigma$ and $V$ is a pair, written $\mu_1 \rightarrow \forall x.\mu_2$, where $\mu_1$ and $\mu_2$ are two multisets of (non ground) atomic formulas in $A^\Sigma_V$, called the antecedent and the consequent of $\mu$, respectively, and $x$ is a vector of variables $\langle x_1, \ldots, x_k \rangle$, with $x_i \in V$ for $i : 1, \ldots, k$. A quantified multiset rewriting system is a set $R$ of multiset rewrite rules over $\Sigma$ and $V$.

The dynamic behaviour of a quantified multiset rewriting system can be described by considering the ground instances of the set of its rewrite rules, which can be obtained as explained in the following. First of all, note that the notion of instance (evaluation of terms to base values) is relative to the current signature, i.e., if the signature is augmented with a new constant $c$, we must provide a new value to evaluate $c$. By analogy with Section 5.1, given a signature $\Sigma_1$ we assume that the set of values are provided by $T_{\Sigma_1}$, i.e., the ground term algebra over $\Sigma_1$. We need to extend the notion of configuration as follows.

**Definition 5.15 (Configuration)** Let $R$ be a quantified multiset rewriting system over
\[ \Sigma \text{ and } \mathcal{V}. \] A configuration is a pair, written \( \langle M, \Sigma_1 \rangle \), where \( \Sigma_1 \) is a signature such that \( \Sigma \subseteq \Sigma_1 \) and \( M \in M\Sigma(T_{\Sigma_1}) \) is a multiset of terms.

Instantiation of a rule works as follows. Given a rule \( \mu = \mu_1 \rightarrow \forall x, \mu_2 \), with \( x = \langle x_1, \ldots, x_k \rangle \), and a configuration \( \langle M, \Sigma \rangle \), we first substitute new constants \( c_1, \ldots, c_k \) (not appearing in \( \Sigma_1 \)) for the bound variables \( x \). We then substitute values (terms in \( T_{\Sigma_1} \)) for the free variables in \( \mu \). Firing will yield a new configuration \( \langle M', \Sigma_2 \rangle \), with \( \Sigma_2 = \Sigma_1 \cup \{c_1, \ldots, c_k\} \). Given an evaluation \( \sigma \) of the free variables in \( \mu \), we denote by \( \mu \sigma \) the application of \( \sigma \) to \( \mu \). As explained above, the notion of evaluation is relative to a signature. We have the following definition.

**Definition 5.16 (Firing)** Let \( R \) be a quantified multiset rewriting system over \( \Sigma \) and \( \mathcal{V} \), \( \langle M, \Sigma_1 \rangle \) a configuration, \( \mu = \mu_1 \rightarrow \forall x, \mu_2 \in R \) a rule, with \( x = \langle x_1, \ldots, x_k \rangle \), and \( \sigma \) an evaluation mapping free variables in \( \mu \) to terms in \( T_{\Sigma_1} \).

1. We say that \( \mu \) is \( \langle M, \Sigma_1 \rangle \)-enabled with \( \sigma \) if \( \mu_1 \sigma \not\prec M \);
2. if \( \mu \) is \( \langle M, \Sigma_1 \rangle \)-enabled with \( \sigma \), a multiset \( \langle M', \Sigma_2 \rangle \) is the result of firing \( \mu \) at \( \langle M, \Sigma_1 \rangle \) with \( \sigma \), written \( \langle M, \Sigma_1 \rangle \triangleright^{\mu, \sigma} \langle M', \Sigma_2 \rangle \), if \( c = \langle c_1, \ldots, c_k \rangle \) is a vector of new constants (not appearing in \( \Sigma_1 \)), \( \Sigma_2 = \Sigma_1 \cup \{c_1, \ldots, c_k\} \), and \( M' = (M \setminus \mu_1 \sigma) + (\mu_2[c/x]) \sigma \); we write \( \langle M, \Sigma_1 \rangle \triangleright \langle M', \Sigma_2 \rangle \sigma \triangleright^{\mu, \sigma} \langle M', \Sigma_2 \rangle \) for some \( \mu \) and \( \sigma \), and we use \( \triangleright^\ast \) to denote the reflexive and transitive closure of \( \triangleright \).

**Example 5.17** Let \( \Sigma \) be a signature with two constant symbols \( a \) and \( b \), one function symbol \( f \) and two predicate symbols \( p, q \). Let \( \mathcal{V} \) be a denumerable set of variables and \( w, x, y \in \mathcal{V} \). Let \( \mu \) be the rule

\[ \{p(x), q(f(y))\} \rightarrow \forall w. \{p(f(x)), q(y), q(w)\} \]

and \( M = \{p(f(a)), p(b), q(f(b))\} \). Then we can fire rule \( \mu \) at \( \langle M, \Sigma \rangle \) with the evaluation \( \sigma \) s.t. \( \sigma(x) = \sigma(y) = b \). Let \( c \) be a new constant, and \( \Sigma_1 = \Sigma \cup \{c\} \). Then we get \( \langle M, \Sigma \rangle \triangleright^{\mu, \sigma} \langle M', \Sigma_1 \rangle \), where \( M' = \{p(f(a)), p(f(b)), q(b), q(c)\} \).

The connection between first-order multiset rewriting systems and first-order LO theories can be extended to quantified multiset rewriting systems and \( \text{LO}_\forall \) theories, as follows.

**Definition 5.18 (Quantified Multiset Rewriting Systems as \( \text{LO}_\forall \) Theories)** Let \( R \) be a quantified multiset rewriting system over \( \Sigma \) and \( \mathcal{V} \). We define the \( \text{LO}_\forall \) theory \( \mathcal{P}(R) \) (over \( \Sigma \) and \( \mathcal{V} \)), as follows:

\[ \mathcal{P}(R) = \{ \forall (H \leftarrow \forall x.G) \mid \hat{H} \rightarrow \forall x.\hat{G} \in R \} \]

62
A soundness and completeness result holds. Note that, in order to state the following proposition, we need to consider provability in LO\(_∀\) augmented with the rule 1\(_r\) (see Sections 3.3.1 and 3.3.2).

**Proposition 5.19 (Reachability as Provability in LO\(_∀\))** Let \( R \) be a quantified multiset rewriting system over \( \Sigma \) and \( \mathcal{V} \), \( C_1 \) and \( C_2 \) two configurations such that \( C_1 = \langle M, \Sigma_1 \rangle \) and \( C_2 = \langle M', \Sigma_2 \rangle \), and \( H, G \) the (possibly empty) \#disjunctions of ground atomic formulas such that \( \widehat{H} = M' \) and \( \widehat{G} = M \). Then \( C_1 \triangleright^* C_2 \) in \( R \) if and only if \( P(R) \) 1\(_{r}(D) \), \( \circ\vdash_{\Sigma,1} G \).

**Example 5.20** Let \( \Sigma \) be the signature of Example 5.17. The LO\(_∀\) program \( P \) corresponding to the rule given in Example 5.17 consists of the following clause:

1. \( p(x) \not\triangleright q(f(y)) \circ \forall w. (p(f(x)) \not\triangleright q(y) \not\triangleright q(w)) \)

Let \( D \) be the clause \( p(f(a)) \not\triangleright p(f(b)) \not\triangleright q(b) \not\triangleright q(c) \circ 1 \). The derivation corresponding to \( \langle M, \Sigma \rangle \triangleright^* \langle M', \Sigma_1 \rangle \) (see Example 5.17), with the usual conventions, is shown in Figure 5.3.

Finally, the usual notion of reachability set can be extended as follows.

**Definition 5.21 (Reachability Set)** Let \( R \) be a quantified multiset rewriting system over \( \Sigma \) and \( \mathcal{V} \). Given a configuration \( \langle M, \Sigma \rangle \), called **initial configuration**, we define the **reachability set** of \( R \), denoted \( \text{Reach}(R) \), as follows: \( \text{Reach}(R) = \{ \langle M, \Sigma \rangle \mid \langle M, \Sigma \rangle \triangleright^* \langle M, \Sigma_1 \rangle \} \).

---

**Summary of the Chapter.** In this chapter we have presented the theories of multiset rewriting over first-order atomic formulas and coloured Petri nets, an extension of the Petri net model in which tokens can carry along values. We have illustrated the connection between these formalisms and first-order linear logic theories. We have also formulated the results for an extension of multiset rewriting systems with universal quantification, and the corresponding fragment LO\(_∀\).
After substantiating our view of linear logic as a formalism for concurrency in Part I, subject of Part II will now be the definition of a bottom-up semantics for linear logic programs, which will be used as a tool for model checking of parameterized and infinite-state systems. First of all, in the next chapter we will explain in detail our approach to verification and the connection with linear logic semantics.
Part II

Model Checking for Linear Logic Specifications
Chapter 6

A Backward Approach to Model Checking

In Part I we have presented the basics of linear logic, specifically the language LO [AP91b] and some proper extensions, and we have analyzed the mutual relations between linear logic and different formalisms for the specification of concurrent systems, like Petri nets and multiset rewriting systems. This chapter is intended to provide the connection between linear logic and the problem of verification for parameterized concurrent systems, which is the focal point of this thesis work. Verification can be performed via an algorithmic procedure which is based on a backward evaluation strategy [ACJT96] and has strong similarities with symbolic model checking techniques [KMM*97b]. The purpose of this chapter is to formally motivate this approach and to discuss its applicability.

In the case of ordinary Petri nets, the verification algorithm proposed in [ACJT96] has a counterpart in the so-called Karp and Miller’s coverability graph computation [KM69]. The contribution of this thesis will be to extend this view towards richer specification languages, like LO. In particular, propositional LO contains a subset which can be used to encode Petri nets, as demonstrated in Section 4.3. Furthermore, in Section 5.3 we have shown that first-order formulations of LO have a direct connection with high-level extensions of nets, like coloured Petri nets. In the following chapters, we will discuss a verification procedure for specifications written in different fragments of linear logic, which we will apply to solve problems in application domains like concurrency theory or security protocols. We will also isolate interesting fragments for which termination of the verification procedure is guaranteed.

This chapter is structured as follows. We first present the problem of verification using a rather general formulation on transition systems. In particular, we focus on reachability problems. Then, we describe a backward verification procedure and the class of problems
this procedure is able to solve. We discuss the requirements which are needed to ensure effectiveness and a general theory providing sufficient conditions for termination. We clarify the connection between the verification procedure and bottom-up semantics of linear logic. Much of the background material for this chapter is a partial and free re-elaboration of [FS01, AJ01b].

6.1 Transition Systems and Verification

We can formulate the problem of verification for arbitrary transition systems. We need the following definitions.

**Definition 6.1 (Transition System)** A transition system is a pair $T = \langle S, R \rangle$, where $S$ is a set of states and $R \subseteq S \times S$ is a set of transitions. A transition $(s, s')$ is usually noted $s \rightarrow s'$, and the reflexive and transitive closure of the $\rightarrow$ relation is denoted $\rightarrow^*$. If $s_1 \rightarrow^* s_2$, we say that there is an execution trace from state $s_1$ to state $s_2$.

A transition system may have additional structure like, e.g., labels on transitions or initial states. As an example, the operational semantics of Petri nets, multiset rewriting systems and broadcast protocols (see Chapters 4 and 5) induces a transition system. For instance, the firing relation of Petri nets is clearly an example of transition relation on markings.

**Definition 6.2 (Predecessor and Successor)** Given a transition system $T = \langle S, R \rangle$, and $W \subseteq S$, we define the predecessor and successor operators, respectively, as follows: $\text{Pre}(W) = \{s \in S \mid s \rightarrow w, w \in W\}$ and $\text{Post}(W) = \{s \in S \mid w \rightarrow s, w \in W\}$. We denote the reflexive and transitive closure of Pre and Post by Pre$^*$ and Post$^*$.

We now discuss the concept of system verification, with particular emphasis on problems involving the notion of reachability. Informally, reachability properties can be described as follows. We are given a set of initial states and a set of final states of a given (transition) system $T$, and the problem is to verify whether some final state can be reached with an execution trace starting from an initial state. Verification of safety properties is a typical example of problem which falls into this class. Safety problems can be described in temporal logic [MP95] as follows. Given a property $\varphi$, verifying whether $\varphi$ holds for every reachable state amounts to checking whether $AG\varphi$ (always globally $\varphi$) holds. The previous expression is logically equivalent to $\neg EF\neg \varphi$ (not exists finally not $\varphi$), therefore the problem is equivalent to verifying whether $EF\psi$ holds, where $\psi$ represents the negation of the safety property under consideration. To exemplify, we may want to verify whether an undesired state (e.g., in concurrency theory, a state in which a deadlock or a violation of mutual exclusion occur) is reachable starting from a valid initial configuration.
Definition 6.3 (Reachability Problem) Let $T = \langle S, R \rangle$ be a transition system, and $s_1, s_2 \in S$ two states. We say that $s_2$ is reachable from $s_1$ if $s_1 \xrightarrow{*} s_2$. Given a set $I \subseteq S$ of initial states and a set $F \subseteq S$ of final states, the reachability problem consists in determining if $i \xrightarrow{*} f$ for some $i \in I$ and $f \in F$.

Different approaches can be used to tackle the reachability problem. An approach which is suitable for automation is to systematically and exhaustively search for all execution traces from some initial state to some final state. Clearly, the number of execution traces in general can be infinite, therefore the problem of termination must be addressed. A distinction can be made depending on the search strategy. Classical examples of strategies include forward and backward search. Mixed strategies are also possible [Hol88]. In the forward approach, the search starts from the set of initial states, while in the backward approach the search starts from the set of final states. This is explained by the following proposition.

Proposition 6.4 (Forward and Backward Search) Let $T = \langle S, R \rangle$ be a transition system, $I \subseteq S$ a set of initial states and $F \subseteq S$ a set of final states. Then the reachability problem amounts to checking whether $\text{Post}^*(I) \cap F \neq \emptyset$ (forward search), or, equivalently, whether $\text{Pre}^*(F) \cap I \neq \emptyset$ (backward search).

Proof Immediate by definitions. □

The previous proposition suggests a way to solve the reachability problem. In the case of forward search, the idea is to compute the $\text{Post}^*$ operator as transitive closure of the $\text{Post}$ operator, starting from the set $I$. Similarly, backward search works by computing $\text{Pre}^*$ as transitive closure of $\text{Pre}$, starting from $F$. At this level, the forward and backward search strategies are perfectly symmetrical. Let us fix, e.g., a backward search strategy. Intuitively, the idea is to start from $F$ and generate, in a breadth-first search style, the set of states from which a state in $F$ can be reached, using an iterative procedure. At the first step, we compute the set of states from which a state in $F$ can be reached with an execution trace of length one or zero. In general, at step $j$ we compute the set of states from which $F$ can be reached with an execution trace of length at most $j$. If the procedure converges, i.e., the sets computed at steps $j$ and $j + 1$ coincide, one checks whether the intersection with the set of initial states $I$ is empty or not.

This (very abstract) verification procedure is described in Figure 6.1. Clearly, at least two important issues must be addressed. First, the sets computed at each step in general are infinite. Therefore, in order for the procedure to work, we need a symbolic way (i.e., an assertional language) to represent infinite sets of states, and we need a version of the $\text{Pre}$ operator working on symbolic sets of states. In this way, we will be guaranteed that the computation of each step of the verification procedure is effective. Second, we need
Procedure Reachability

input $T = (S, R)$: a transition system
   $I$: a set of initial states; $F$: a set of final states
output $Pre^*(F) \cap I = \emptyset$
begin
   $W := F$; $V := \emptyset$
   while $W \not\subseteq V$ do
      if $W \cap I \neq \emptyset$ then return reachable
      else $V := W$
         $W := W \cup Pre(W)$
      endif
   endwhile
   return unreachable
end

Figure 6.1: An abstract verification procedure for reachability

(sufficient) conditions ensuring termination of the verification procedure. The first issue
will be addressed in Section 6.1.1, where we introduce the notion of structured transition
system, while the problem of termination will be discussed in Section 6.2.

6.1.1 Structured Transition Systems

In order to refine the abstract verification procedure for solving the reachability problem of
Figure 6.1, in this section we discuss transition systems which satisfy additional hypotheses.
In particular, we consider transition systems with additional structure, namely an ordering
relation, on the set of states. We have the following definition.

Definition 6.5 (Structured Transition System) A structured transition system is a
tuple $T = \langle S, R, \leq \rangle$, where $\langle S, R \rangle$ is a transition system and $\leq$ is a quasi-order on $S$.

We remind that a quasi-order is a reflexive and transitive binary relation.

Example 6.6 A multiset rewriting system (see Section 4.1) clearly induces a transition
system in which states are multisets over a set $S$. It also induces a structured transition
system where the quasi-order is given by the sub-multiset relation $\subseteq$. Thanks to the results
of Section 4.3, Petri nets are equivalent to multiset rewriting systems and therefore can be
seen in turn as structured transition systems.\[\square\]
Now, an interesting instance of the reachability problem of Section 6.1 is the following, which goes under the name of covering problem, and consists in determining if starting from a state $s$ it is possible to cover $t$, i.e., to reach a state $t'$ such that $t \leq t'$.

**Definition 6.7 (Covering Problem)** Let $T = \langle S, R, \leq \rangle$ be a structured transition system. Given a set $I \subseteq S$ of initial states and a set $F \subseteq S$ of final states, the covering problem consists in determining if there exist $i \in I$, $f \in F$, and $f' \in S$, such that $i \rightarrow^* f'$ and $f \leq f'$.

The covering problem is sometimes referred to as control-state reachability problem [FS01]. The covering problem is particularly meaningful because many safety problems in concurrency theory, e.g. mutual exclusion, can be described in this way.

**Example 6.8** Let us consider the Petri net in Figure 6.2, representing a monitor for a parameterized system with two mutually exclusive critical sections, $cs_1$ and $cs_2$. The Petri net is parametric in the number of processes. Initially, we have $K$ processes in place $waiting$, one token in $lock_1$ and one token in $lock_2$. In other words, the initial marking, drawn in Figure 6.2, is $M_I = K \cdot \{waiting\} + \{lock_1, lock_2\}$. A process wishing to enter $cs_1$ (transition Enter1) must test for the presence of processes in $cs_2$ using $lock_2$, and locks $cs_1$ using $lock_1$. Transition Enter2 is specular. Processes leave the critical section using transitions Leave1 and Leave2. Mutual exclusion requires that at most one process can be executing inside either critical section at the same time, and that the two critical sections are mutually exclusive. Accordingly, a marking violating mutual exclusion has the form $\{cs_1, cs_1\} + M$, $\{cs_1, cs_2\} + M$, or $\{cs_2, cs_2\} + M$, where $M$ is any marking. In other words, a configuration is unsafe if there are at least two processes accessing one of the two critical sections at the same time. Therefore, mutual exclusion is violated if there exists a valid initial state from which it is possible to cover a state having the form $\{cs_1, cs_1\}$,
\{cs1, cs2\}, or \{cs2, cs2\}. Verifying the problem of mutual exclusion amounts to solving a covering problem in which the set of initial states are the legal initial states of the system under consideration, whereas the set of final states are the configurations representing, so to say, the minimality violations of the property of mutual exclusion, i.e., the unsafe states.

As the notion of rewriting in transition systems is symmetrical, in principle it would be possible to reverse the role of initial and final states in Definition 6.7. However, as the previous example suggests, the formulation of the covering problem given in Definition 6.7 is quite natural. In fact, if we think about final states as the set of unsafe states of a given system, Example 6.8 suggests that any state covering an unsafe one is still unsafe. This is not true, generally speaking, for the set of initial states of a system. For instance, the configuration \(M_{I} + \{cs1, cs1\}\) in Example 6.8 is not a legal initial state, even though it covers \(M_{I}\), which is indeed a legal initial state. In other words, we can say that the set of final (unsafe) states is upward-closed (meaning that \(s\) unsafe and \(s \leq s'\) implies \(s'\) unsafe) while the set of initial states is not upward-closed.

We can now give a justification of our choice of a backward search strategy with respect to a forward one. In fact, according to Proposition 6.4, using a backward search strategy amounts to an iterative computation of the set of reachable states, starting from the set of final states. The idea is to exploit the upward-closure property of the set of unsafe states, find a suitable symbolic representation and use them as the set of final states of Definition 6.7. For instance, Example 6.8 suggests that multisets of the form \(\{cs1, cs1\}\) can be used to symbolically represent the infinite set of states given by \(\{cs1, cs1\} + M\). The resulting computation is completely independent of the set of initial states. We remark that the property of upward-closure is also crucial for reasoning about parametric systems, i.e., systems in which the number of components (e.g. the number \(K\) of processes in Example 6.8) is not fixed a priori. Verification of parametric systems is one of the goals of the present thesis work.

In the following we will try to make the above intuitions more formal. We first give the following definitions.

**Definition 6.9 (Upward-closure)** Let \(\leq\) be a quasi-order on \(S\). Then a set \(W \subseteq S\) is said to be upward-closed if \(w \in W\) and \(w \leq w'\) implies \(w' \in W\). Given \(w \in S\), we define the upward-closure of \(w\), written \(w^\uparrow\), as the set \(\{s \in S \mid w \leq s\}\). Similarly, for a subset \(W \subseteq S\) we define \(W^\uparrow = \{s \in S \mid w \leq s, \ w \in W\}\). An upward closed set is also called an ideal, and \(W^\uparrow\) is called the ideal generated by \(W\).

From the previous definition, it follows that a set \(W \subseteq S\) is upward-closed if and only if \(W = W^\uparrow\).
Definition 6.10 (Basis) Let $\preceq$ be a quasi-order on $S$, and $W \subseteq S$. A basis for $W$ is a set $\overline{W}$ such that $W = \overline{W}^\uparrow$.

The notions of upward-closed set and basis are the main ingredients of the symbolic version of the verification procedure for reachability we aim at. In particular, we are interested in finite bases. A finite basis $\overline{W}$ can in fact be used as a symbolic representation for the (generally infinite) set of states $\overline{W}^\uparrow$. Now, let us make the assumption that the set of states from which the computation begins (i.e., the set of final states $F$ of Definition 6.7) is upward-closed, and that we can find a finite basis for it. In order for the symbolic algorithm to work, we need to ensure that the sets computed at every iteration of the algorithm in Figure 6.1 are still upward-closed, and therefore they are in turn representable by means of finite bases. For this purpose, we introduce the following notion of compatibility between the transition relation and the quasi-order relation of a structured transition system.

Definition 6.11 (Upward Compatibility) Let $T = \langle S, R, \preceq \rangle$ be a structured transition system. We say that $\preceq$ is (upward) compatible with $\rightarrow$ if for all $s_1, s_2, t_1 \in S$, if $s_1 \preceq t_1$ and $s_1 \rightarrow s_2$, there exists $t_2 \in S$ such that $t_1 \rightarrow t_2$ and $s_2 \preceq t_2$.

The above notion is usually called strict upward compatibility. The definition could be weakened by requiring that $t_1 \rightarrow^* t_2$ instead of $t_1 \rightarrow t_2$, as done in [FS01]. The following result (see [FS01] for the proof) holds.

Proposition 6.12 Let $T = \langle S, R, \preceq \rangle$ be a structured transition system with upward compatibility, and $W \subseteq S$. If $W$ is upward-closed, then $\text{Pre}(W)$ is upward-closed.

We need one more assumption to ensure effectiveness of the $\text{Pre}^*$ computation, namely the effective pred-basis property.

Definition 6.13 (Effective Pred-Basis) Let $T = \langle S, R, \preceq \rangle$ be a structured transition system with upward compatibility. We say that $T$ has an effective pred-basis if there exists an algorithm accepting any state $s \in S$ and computing a finite basis of $\text{Pre}(s^\uparrow)$, denoted $\text{pb}(s)$. We extend this notation to a set of states $W \subseteq S$ as follows: $\text{pb}(W) = \bigcup_{w \in W} \text{pb}(w)$.

We have the following proposition, whose proof can be found in [FS01].

Proposition 6.14 ([FS01]) Let $T = \langle S, R, \preceq \rangle$ be a structured transition system with upward compatibility and the effective pred-basis property, and $F \subseteq S$. Given a finite basis $F^b$ for $F$, define a sequence $K_0, K_1, \ldots$ of sets as follows: $K_0 = F^b$ and $K_{n+1} = K_n \cup \text{pb}(K_n)$ for $n \in \mathbb{N}$. Then $(\bigcup_{i=0}^\infty K_i)^\uparrow = \text{Pre}^*(F)$. 

72
Procedure Covering

input $T = (S, R, \leq)$: a transition system

$I$: a set of initial states

$F$: a finite basis for the set of final states

output $Pre^*(F) \cap I = \emptyset$

begin

$W := F; \quad V := \emptyset$

while $W \neq \emptyset$ do

choose $w \in W; \quad W := W \setminus \{w\}$

if $w^\uparrow \cap I \neq \emptyset$ then return reachable

else if $\not\exists w' \in V : w^\uparrow \subseteq w'^\uparrow$ then

$V := V \cup \{w\}$

$W := W \cup pb(w)$

endif

endwhile

return unreachable

end

Figure 6.3: A verification procedure for covering

We are now ready to describe a refined version of the reachability algorithm for the covering problem, based on Proposition 6.14, which is shown in Figure 6.3. Under the hypotheses of upward compatibility and effective pred-basis property, this algorithm is effective, in the sense that any iteration of the while loop can be computed effectively. For the sake of precision, we need some assumptions more. Namely, we assume that $\leq$ is decidable, the test $w^\uparrow \cap I \neq \emptyset$ and the test $w^\uparrow \subseteq w'^\uparrow$ (insertion test) are computable for any states $w, w'$. Termination in general is not guaranteed (the algorithm could loop forever). In Section 6.2 we will discuss sufficient conditions ensuring termination.

6.1.2 Symbolic Verification

In Section 6.1.1 we have discussed the theoretical foundation of our approach for verification of parameterized systems. Its effectiveness relies on structured transition systems (i.e., endowed with a notion of upward-closed set), and on the ability to compute finite bases of sets of predecessors of any upward-closed set. We will see some instances of this general scheme in chapters 7 through 10. The approach taken there slightly differs from the general scheme presented in this chapter. We summarize below some points which should help the reader to better understand the connection.

- A verification problem, formalized as a covering problem for a given structured tran-
A transition system $T = \langle S, R, \leq \rangle$ (with upward compatibility), is given;

- verification is performed by choosing a suitable symbolic notation for (upward-closed) infinite sets of states (a constraint system in the terminology of [AJ01b]). The symbolic notation exploits the upward-closure property of the sets to represent, but, as a difference with Section 6.1.1, symbolic elements need not be elements of $S$. The connection is provided by a denotation operator, usually noted $[\cdot]$, which maps a set of symbolic elements to the corresponding (upward-closed) set in $S$;

- the finite pred-basis operator (denoted $pb(\cdot)$ in Figure 6.3) is implemented through a symbolic version of the predecessor operator, let it be $Pre$, which takes as input the symbolic representation of a set $V$ and computes the symbolic representation of the set $Pre(V)$. The following property must hold for the symbolic operator to be correct: $[Pre(W)] = Pre([W])$;

- the insertion test of Figure 6.3 now becomes $[w] \subseteq [w']$. It is implemented through a subsumption operator, usually noted $\sqsubseteq$, such that $w \subseteq w'$ implies $[w] \subseteq [w']$ \(^1\). The operator is typically extended to sets of symbolic elements as follows (i.e., pointwise): $W \sqsubseteq W'$ iff for every $w \in W$ there exists $w' \in W'$ s.t. $w \sqsubseteq w'$.

We conclude this section by discussing two different techniques that can sometimes be useful either to enforce termination or to accelerate convergence of the verification procedure of Figure 6.3. In Appendix A we discuss further how these techniques can be integrated into an automatic verification tool.

6.1.2.1 Invariant Strengthening

Suppose we have to solve a reachability problem, $I$ and $F$ being the sets of initial and final states, respectively, of a given transition system, and let us call the problem unsatisfiable in case it is not possible to find an execution trace from a state in $I$ to a state in $F$. Now, the idea of invariant strengthening is to transform the original problem by considering a larger set of final states $F'$. This technique is perfectly sound, as stated in the following proposition.

**Proposition 6.15** Let $T = \langle S, R \rangle$ be a transition system and $I, F, F' \subseteq S$. If the reachability problem for $I$ and $F'$ is unsatisfiable and $F \subseteq F'$, then the reachability problem for $I$ and $F$ is unsatisfiable.

**Proof** Immediate by definition.

\[^1\]Note that our $\sqsubseteq$ operator corresponds to the $\sqsupseteq$ operator of [AJ01b]. Here and in the following chapters we prefer to follow the classical logic programming convention.
In general, the reverse implication in the above theorem does not hold, i.e., finding an execution trace from \( I \) to \( F' \) does not guarantee that an execution trace from \( I \) to \( F \) exists. Using a parallel with temporal logic specifications, a property \( AG\varphi \) can be transformed, by invariant strengthening, into a property \( AG(\varphi \land \psi) \). Intuitively, the formula \( \psi \) should represent an invariant which is known to hold in all reachable states. Otherwise said, the expression \( \neg EF\neg \varphi \) can be transformed into \( \neg EF(\neg \varphi \lor \neg \psi) \), i.e., we can enlarge the set of final (unsafe) states with a formula \( \neg \psi \) which does not hold in any reachable state. We can refine the above proposition as follows.

**Proposition 6.16** Let \( T = \langle S, R \rangle \) be a transition system and \( I, F, F' \subseteq S \). Suppose \( F \subseteq F' \) and \( \text{Post}^*(I) \cap (F' \setminus F) = \emptyset \). Then the reachability problem for \( I \) and \( F' \) is unsatisfiable if and only if the reachability problem for \( I \) and \( F \) is unsatisfiable.

**Proof** By definitions.

We conclude this section by illustrating another technique which is similar to invariant strengthening. The idea is to dynamically enlarge the search space of the algorithm for backward verification. Suppose we have an abstraction \( \alpha \) mapping sets of states into sets of states, and such that \( W \subseteq \alpha(W) \) for every set of states \( W \). Then, at every iteration we can substitute \( \alpha(\text{Pre}(W)) \) for \( \text{Pre}(W) \). Intuitively, this technique is perfectly sound for the same reasons invariant strengthening is. We will use this strategy in Chapter 9 for validating specifications written in LO enriched with constraints. In that case, the abstraction \( \alpha \) works by relaxing constraints. Let us define the following modified predecessor operator.

**Definition 6.17** Let \( T = \langle S, R \rangle \) be a transition system, and \( I, F \subseteq S \). Consider the reachability problem for \( I \) and \( F \), and a function \( \alpha: \mathcal{P}(S) \rightarrow \mathcal{P}(S) \). Then we define the operator \( \text{Pre}_\alpha \) as follows: \( \text{Pre}_\alpha(W) = \alpha(\text{Pre}(W)) \) for every \( W \subseteq S \).

Correctness of the abstraction technique is stated in the following proposition.

**Proposition 6.18** Let \( T = \langle S, R \rangle \) be a transition system, and \( I, F \subseteq S \). Consider the reachability problem for \( I \) and \( F \), and a function \( \alpha: \mathcal{P}(S) \rightarrow \mathcal{P}(S) \) such that \( W \subseteq \alpha(W) \) for every \( W \subseteq S \). If \( \text{Pre}_\alpha^*(F) \cap I = \emptyset \), then \( \text{Pre}_\alpha^*(F) \cap I = \emptyset \).

**Proof** By definitions.
6.1.2.2 Pruning

The technique of pruning is, so to say, complementary with respect to the methodology of dynamic abstraction illustrated above. The idea is to dynamically cut the search space of the algorithm for backward verification. Suppose we have computed an over-approximation $H$ of the reachability set. At every iteration, we can intersect the set of reachable states computed so far with the set $H$. Intuitively, states not in $H$ are not reachable from the set of initial states, therefore computing their set of predecessors is completely useless. Consequently, states not in $H$ can always be discarded. Let us define the following modified predecessor operator.

**Definition 6.19** Let $T = (S, R)$ be a transition system, and $I, F, H \subseteq S$. Consider the reachability problem for $I$ and $F$, and suppose $\text{Post}^*(I) \subseteq H$. Then we define the operator $\text{Pre}_H$ as follows: $\text{Pre}_H(W) = \text{Pre}(W) \cap H$ for every $W \subseteq S$.

Correctness of the pruning technique is stated in the following proposition.

**Proposition 6.20** Let $T = (S, R)$ be a transition system, and $I, F, H \subseteq S$. Consider the reachability problem for $I$ and $F$, and suppose $\text{Post}^*(I) \subseteq H$. Then $\text{Pre}_H^*(F) \cap I = \emptyset$ if and only if $\text{Pre}_H^*(F) \cap I = \emptyset$.

**Proof** By definitions.

6.2 Ensuring Termination

In this section we discuss some sufficient conditions which ensure termination of the verification procedure of Figure 6.3. In particular, the theoretical foundation is provided by the theory of well- and better-quasi-orderings [Mil85], presented in Section 6.2.1, which gives rise to the notion of well-structured transition system [FS01], presented in Section 6.2.2.

6.2.1 The Theory of Well-Quasi-Orderings

In the following we summarize some basic definitions and results on the theory of well- and better-quasi-orderings (see for instance [Hig52, Mil85, AN00]).

We remind that a quasi-order $\leq$ on a set $A$ is a binary relation over $A$ which is reflexive and transitive. In the following it will be denoted $(A, \leq)$.
Definition 6.21 (Well-Quasi-Ordering) A quasi-order $(A, \leq)$ is a well-quasi-ordering (wqo) if for each infinite sequence $a_0 a_1 a_2 \ldots$ of elements in $A$ there exist indices $i < j$ such that $a_i \leq a_j$.

Examples of wqos are the identity relation on a finite set or the $\leq$ relation on natural numbers (but not on integers).

The following propositions state the key results involving wqos and upward-closed sets.

Proposition 6.22 ([Hig52]) Let $(A, \leq)$ be a wqo, and $X \subseteq A$. If $X$ is upward-closed, then it has a finite basis.

Proposition 6.23 Let $(A, \leq)$ be a wqo. Then every infinite sequence $I_0 \subseteq I_1 \subseteq I_2 \subseteq \ldots$ of upward-closed sets, with $I_i \subseteq A$ for every $i \in \mathbb{N}$, eventually stabilizes, i.e., there exists $k \in \mathbb{N}$ such that $I_k = I_{k+1} = I_{k+2} = \ldots$.

The notion of better-quasi-ordering (bqo) implies that of wqo, i.e., any bqo is a wqo. For the sake of simplicity, we will not present the formal definition of bqo, which can be found in [Mil85]. According to the following proposition, an entire hierarchy of bqos (and therefore of wqos) can be built starting from known ones.

In what follows, let $\hat{n}$ denote the set $\{1, \ldots, n\}$, for $n \in \mathbb{N}$. A multiset over $A$ can be represented as a mapping $w : [\hat{n}] \to A$, where $|w|$ is its cardinality. Similarly, a string over $A$ can be represented as a mapping $w : [\hat{n}] \to A$, where $|w|$ is its length.

Proposition 6.24 (Properties of Better-Quasi-Orderings)

i. Each bqo is a wqo;

ii. If $A$ is a finite set, then $(A, =)$ is a bqo;

iii. let $(A, \leq)$ be a bqo, and let $\mathcal{MS}(A)$ denote the set of finite multisets over $A$. Then, $(\mathcal{MS}(A), \leq^B)$ is a bqo, where $\leq^B$ is the quasi-order on $\mathcal{MS}(A)$ defined as follows: $w \leq^B w'$ iff there exists an injection $h : [\hat{w}] \to [\hat{w'}]$ such that $w(j) \leq w'(h(j))$ for $1 \leq j \leq |w|$;

iv. let $(A, \leq)$ be a bqo, and let $A^*$ denote the set of finite strings over $A$. Then, $(A^*, \leq^*)$ is a bqo, where $\leq^*$ is the quasi-order on $A^*$ defined in the following way: $w \leq^* w'$ iff there exists a strictly monotone (meaning that $j_1 < j_2$ iff $h(j_1) < h(j_2)$) injection $h : [\hat{w}] \to [\hat{w'}]$ such that $w(j) \leq w'(h(j))$ for $1 \leq j \leq |w|$;
v. let \((A, \leq)\) be a bqo. Then, \((\mathcal{P}(A), \sqsupseteq)\) is a bqo, where \(\mathcal{P}(A)\) is the power set of \(A\) and \(\sqsupseteq\) is the pointwise ordering defined as follows: given \(X, Y \subseteq A\), \(X \sqsupseteq Y\) iff for every \(y \in Y\) there exists \(x \in X\) s.t. \(x \leq y\)\(^2\).

The reader can verify that the ordering \(\leq^B\) of property iii coincides with the sub-multiset relation \(\preceq\) (see Definition 2.1) when \(\leq\) is interpreted as equality on a finite set \(A\), so that for instance \((\mathcal{MS}(A), \preceq)\) is a wqo. This result goes under the name of Dickson’s Lemma:

**Proposition 6.25 (Dickson’s Lemma [Dic13])** Let \(A_1, A_2, \ldots\) be an infinite sequence of multisets over the finite set \(A\). Then there exist two indices \(i\) and \(j\) such that \(i < j\) and \(A_i \preceq A_j\).

We note that properties ii, iii, and iv of Proposition 6.24 also hold for wqos. On the contrary, property v is the distinguished feature of bqos w.r.t. wqos, i.e., in general \((A, \leq)\) being a wqo does not imply that \((\mathcal{P}(A), \sqsupseteq)\) is a wqo (see Rado’s counterexample in [AN00]). The proof of property v can be found in [Mar99].

**Remark 6.26** Consider a wqo \((A, \leq)\) and the following ordering \(\sqsupseteq^*\) on \(\mathcal{P}(A)\): given \(X, Y \subseteq A\), \(X \sqsupseteq^* Y\) iff for every \(x \in X\) there exists \(y \in Y\) s.t. \(x \leq y\). Then \((\mathcal{P}(A), \sqsupseteq^*)\) is a wqo. This conclusion follows by property iii of Proposition 6.24 (which can be reformulated for sets instead of multisets) and by observing that the existence of an injection between \(X, Y \subseteq A\), satisfying the hypotheses of property iii, implies \(X \sqsupseteq^* Y\). We remark, however, that \((\mathcal{P}(A), \sqsupseteq^*)\) being a wqo does not imply that \((\mathcal{P}(A), \sqsupseteq)\) (defined in property v of Proposition 6.24) is a wqo: the ordering relations \(\sqsupseteq^*\) and \(\sqsupseteq\) are not comparable, as the reader can verify.

### 6.2.2 Well-Structured Transition Systems

Building on the concept of well-quasi-ordering of Section 6.2.1, we can now extend the definition of structured transition system as follows.

**Definition 6.27 (Well-Structured Transition System [FS01])** A well-structured transition system is a structured transition system \(T = (S, R, \leq)\) with upward compatibility, and such that \((S, \leq)\) is a well-quasi-ordering.

We have the following result.

\(^2\)See note on page 74
reachability problems | linear logic semantics
---|---
transition system | proof system
state | sequent
transition | rule instance
final state | axiom | with 1
  | with \(\top\)
reachability | provability
\(\text{Pre}\) operator | \(T_P\) operator
\(\text{Pre}^*\) operator | \(\text{lfp}(T_P)\)

Table 6.1: A parallel between reachability problems and linear logic semantics

**Proposition 6.28 ([FS01])** Let \(T = \langle S, R, \leq \rangle\) be a well-structured transition system with the effective pred-basis property and decidable \(\leq\). Then it is possible to compute a finite basis of \(\text{Pre}^*(F)\) for any upward-closed set \(F \subseteq S\) given via a finite basis. It follows that the covering problem is decidable for \(T\).

The proof of the previous proposition, which can be found in [FS01], is carried out by showing that the sequence \(K_0, K_1, \ldots\) of Proposition 6.14 stabilizes for a given \(m \in \mathbb{N}\), thanks to the well-quasi-ordering property of \(\leq\).

Proposition 6.28 and the theory of well-quasi-orderings of Section 6.2.1 therefore provide sufficient conditions for the termination of the algorithm for the covering problem presented in Figure 6.3. We will see some applications in the following chapters. For instance, termination of the bottom-up evaluation procedure for propositional LO, presented in Chapter 7, is a consequence of Dickson’s lemma 6.25.

### 6.3 Verification in Linear Logic

In this section we will explain, very informally, the connection between the reachability and covering problems presented in this chapter and provability in linear logic. This should help the reader to better understand the following chapters.

We present in Table 6.1 a succinct and suggestive view of this connection. The main idea underlying the connection is that a proof system, like the sequent systems presented in Chapter 3, can be seen as a transition system. In this view, reachability between configurations amounts to provability of a given goal from a given axiom. Axioms represent
final states, and the set of logical consequences of the axioms correspond to the backward reachability set, i.e., the set of states which are backward reachable from the set of final states. Computing the closure $P_{\text{Pre}}^*$ of the predecessor operator, starting from the set of final states, corresponds to evaluating the bottom-up semantics of a linear logic program. Evaluation of the bottom-up semantics is carried out starting from the axioms and iteratively accumulating provable goals, exactly like computation of $\text{Pre}^*$ starts from the set of final (unsafe) states and accumulates reachable states. A single state can be represented by an $\text{LO}_1$ axiom like $H \circ_{\text{pre}} \top$ (see Section 3.3.1), whereas the notion of upward-closed set of states has a counterpart in LO axioms of the form $H \circ_{\text{pre}} \top$ (see Section 3.2). In fact, we remind that in LO (without $\text{LO}_1$) provable goals are upward-closed, according to Proposition 3.9 (admissibility of the weakening rule). The counterpart of the $\text{Pre}$ operator is a classical $TP$-like fixpoint operator for logic programs [Llo87].

6.4 Related Work

In this section we briefly discuss some related work. As mentioned at the beginning of this chapter, part of the material presented here is freely taken from [FS01, AJ01b].

In [FS01], Finkel and Schnoebelen build upon previous works like [Fin87, ACJT96] and present a general theory for well-structured transition systems. They discuss a number of verification problems, among which the covering problem we have presented in this chapter, the inevitability problem and the boundedness problem. They divide resolution methods into set-saturation methods (like the one for the covering problem presented here) and tree-saturation methods, and they discuss related conditions for decidability and effectiveness. Finally, they show examples of well-structured transition systems from the fields of Petri nets, rewriting systems, process algebras, and finite state machines. In [AJ01b], Abdulla and Jonsson discuss strategies for solving the covering problem, which can be seen as an instance of the general scheme of [FS01]. They use the theory of well-quasi-orderings and show how to build an hierarchy of wqos. They focus on symbolic verification methods and show examples of different constraint systems.

The general framework of [FS01, AJ01b] has been applied to solving a number of problems for infinite-state systems. Among them, networks of timed processes [AJ98, AN01], broadcast protocols [EFM99, DEP99], cache coherence protocols [Del00], and protocols using communication over unbounded channels [AJ01a]. A discussion on the use of better-quasi-orderings and their advantage over well-quasi-orderings for verification of infinite-state systems can be found in [AN00].

Summary of the Chapter. In this chapter we have discussed the theoretical
foundation of a symbolic verification algorithm which can be considered the core of
the present thesis work. The algorithm implements a backward-style model checking
verification procedure for solving reachability problems in transition systems and it
is particularly suitable to solve the so-called covering problem. We have explained
why backward reachability is more convenient than forward reachability.

Following the general scheme presented in this chapter, in the following chapters we
will discuss bottom-up semantics for different fragments of linear logic programs. In
particular, in chapters 7 through 9 we will discuss bottom-up evaluation for proposi-
tional LO program, with or without the constant 1, first-order LO programs, and LO
programs with constraints. For propositional LO programs, we will be able to prove
termination using the well-quasi-ordering theory discussed in this chapter. We will
exemplify the backward-style approach solving different problems taken, e.g., from
concurrency theory or security protocols.
Chapter 7

Bottom-Up Evaluation of Propositional LO Programs

The proof-theoretical semantics of LO we have seen in Chapter 3 corresponds to the top-down operational semantics based on resolution for traditional logic programming languages like Prolog. We are now interested in finding a suitable definition of bottom-up semantics that can be used as an alternative operational semantics for LO. More precisely, given an LO program $P$ we would like to define a procedure to compute all goal formulas $G$ such that $G$ is provable from $P$. This procedure should enjoy the usual properties of classical bottom-up semantics, in particular its definition should be based on an effective fixpoint operator (i.e., at least every single step must be finitely computable), and it should be goal-independent. As usual in Logic Programming, goal independence is achieved by searching for proofs starting from the axioms (the unit clauses of Section 3.2) and accumulating goals which can be proved by applying program clauses to the current interpretation.

In this chapter we first deal with a propositional formulation of LO comprising the logical connectives $\lor$, $\land$, $\rightarrow$ and $\top$. As proved in Section 4.3, there is a natural translation of Petri nets into this logical fragment. In particular, a Petri net transition corresponds to an LO clause, and firing of a transition corresponds to backchaining over a given clause. In this view, the bottom-up semantics for propositional LO which we will discuss in this chapter gives us an algorithmic procedure to compute the (backward) reachability set of a given Petri net (it can be seen as an alternative to the Karp and Miller’s coverability graph construction [KM69]). Now, as explained in Chapter 6 (see in particular Section 6.3), the connection with the problem of verification is as follows. If we look at the set of axioms of a given LO program $P$ as specifying the set of unsafe (final) configurations of a given system (e.g. markings in the case of Petri nets), then computing the bottom-up semantics of an LO program amounts to computing the (backward) reachability set of the system. In this view, the immediate consequence operator (see Section 2.5) for LO programs
which we will define in this chapter, is the counterpart of the predecessor operator for the given transition system. Thanks to Proposition 3.9, LO axioms can be seen as specifying upward-closed sets of configurations, therefore evaluating the bottom-up semantics for an LO program gives us an algorithm to solve the so-called covering problem (see Definition 6.7). Proposition 3.9 is crucial to prove that the algorithm for bottom-up evaluation of LO programs is terminating. This result will follow from an argument (Dickson’s Lemma) which is based on the general theory of well-quasi-orderings discussed in Section 6.2.1. We will prove soundness and completeness results of the resulting semantics from scratch, without relying on Chapter 6. We will exemplify the connection between provability and verification, informally explained above, with the examples of Section 7.3.

Technically, the definition of the bottom-up semantics for LO will proceed as follows. The semantics is based on the definition of an extended notion of interpretation comprising multisets of atoms, and of a fixpoint operator which turns out to be monotonic and continuous over the complete lattice corresponding to the interpretation domain. Following the semantic framework of (constraint) logic programming [GDL95, JM94], we formulate the bottom-up evaluation procedure in two steps. More precisely, we first present a simple, non-effective notion of (concrete) interpretations and the corresponding definition of fixpoint operator, which we call $TP$. Thanks to Proposition 3.9, already in the propositional case there are infinitely many provable multisets of atomic formulas. To circumvent this problem, we present an extended notion of (abstract) interpretations, ordered according to the multiset inclusion relation of their elements, and we define a symbolic and effective version of the fixpoint operator, which we call $SP$. The purpose of this double definition is to ease the proof of soundness and completeness. Namely, this latter proof is carried out by proving that the fixpoint of the $TP$ operator is equivalent to the operational semantics and that $SP$ is a symbolic version of $TP$. Dickson’s Lemma [Dic13] ensures the termination of the fixpoint computation based on $SP$ for propositional LO programs.

This chapter is based on [BDM00, BDM02].

### 7.1 A Bottom-Up Semantics for LO

**Some notations.** In the following we will extensively use operations on multisets. We assume as given a fixed signature $\Sigma$ comprising a finite set of propositional symbols. Multisets over $\Sigma$ will be hereafter called facts, and noted $A, B, C, \ldots$. A multiset with (possibly duplicated) elements $a_1, \ldots, a_n \in \Sigma$ will be simply indicated as $\{a_1, \ldots, a_n\}$, overloading the usual notation for sets. We will use $\Delta, \Theta, \ldots$ to denote contexts, i.e., multisets of (possibly compound) goal formulas (a fact is a context in which every formula is atomic). Given two multisets $\Delta$ and $\Theta$, $\Delta \preceq \Theta$ indicates multiset inclusion, $\Delta + \Theta$ (or $\Delta, \Theta$ if there is no ambiguity) multiset union, and $\Delta, \{G\}$ is written simply $\Delta, G$. Finally, we remind that,
Definition 7.1 (Operational Semantics) Given a propositional LO program $P$, its operational semantics, denoted $O(P)$, is given by

$$O(P) = \{A | A \text{ is a fact and } P \vdash A\}.$$  

Note that, according to [And92], the information on provable facts from a given program $P$ is all we need to decide whether a general goal (with possible nesting of connectives) is provable from $P$ or not. In fact, provability of a compound goal can always be reduced to provability of a finite set of atomic multisets.

We will now discuss the bottom-up semantics. We give the following definitions.

Definition 7.2 (Herbrand Base $B_P$) Given a propositional LO program $P$ defined over $\Sigma$, the Herbrand base of $P$, denoted $B_P$, is given by

$$B_P = MS(\Sigma) = \{A | A \text{ is a multiset (fact) over } \Sigma\}.$$  

Definition 7.3 (Herbrand Interpretations) We say that $I \subseteq B_P$ is a Herbrand interpretation. Herbrand interpretations form a complete lattice $\langle D, \subseteq \rangle$ with respect to set inclusion, where $D = P(B_P)$.

Before introducing the formal definition of the ground bottom-up semantics, we need to define a notion of satisfiability of a context $\Delta$ in a given interpretation $I$. For this purpose, we introduce the judgment $I \models \Delta \triangleright A$, where $I$ is an interpretation, $\Delta$ is a context and $A$ is

![Figure 7.1: A proof system for propositional LO](image-url)
an output fact. The need for this judgment, with respect to the familiar logic programming setting [GDL95], is motivated by the arbitrary nesting of connectives in LO clause bodies, which is not allowed in traditional presentations of (constraint) logic programs. In $I \vdash \Delta \cdot \mathcal{A}$, $\mathcal{A}$ should be read as an output fact such that $\mathcal{A} + \Delta$ is valid in $I$. This notion of satisfiability is modeled according to the right-introduction rules of the connectives. In other words, the computation performed by the satisfiability judgment corresponds to top-down steps inside our bottom-up semantics. The notion of output fact $\mathcal{A}$ will simplify the presentation of the algorithmic version of the judgment which we will present in Section 7.2.

**Definition 7.4 (Satisfiability Judgment)** Let $P$ be a propositional LO program and $I$ an interpretation. The satisfiability judgment $\models$ is defined as follows:

\[
I \models \top, \Delta \cdot \mathcal{A}' \text{ for any fact } \mathcal{A}';
I \models \mathcal{A} \cdot \mathcal{A}' \text{ if } \mathcal{A} + \mathcal{A}' \in I;
I \models G_1 \land G_2, \Delta \cdot \mathcal{A} \text{ if } I \models G_1, \Delta \cdot \mathcal{A} \text{ and } I \models G_2, \Delta \cdot \mathcal{A};
I \models G_1 \Rightarrow G_2, \Delta \cdot \mathcal{A} \text{ if } I \models G_1, G_2, \Delta \cdot \mathcal{A};
I \models \bot, \Delta \cdot \mathcal{A} \text{ if } I \models \Delta \cdot \mathcal{A}.
\]

The satisfiability judgment $\models$ satisfies the following properties.

**Lemma 7.5** For every interpretation $I$, context $\Delta$ and fact $\mathcal{A}$,

\[
I \models \Delta \cdot \mathcal{A} \iff I \models \Delta, \mathcal{A} \cdot \epsilon.
\]

**Proof** By simple induction on $\models$ definition. \hfill \Box

**Lemma 7.6** For any interpretations $I_1, I_2, \ldots$, context $\Delta$, and fact $\mathcal{A}$,

i. if $I_1 \subseteq I_2$ and $I_1 \models \Delta \cdot \mathcal{A}$ then $I_2 \models \Delta \cdot \mathcal{A}$;

ii. if $I_1 \subseteq I_2 \subseteq \ldots$ and $\bigcup_{i=1}^{\infty} I_i \models \Delta \cdot \mathcal{A}$ then there exists $k \in \mathbb{N}$ s.t. $I_k \models \Delta \cdot \mathcal{A}$.

**Proof** The proof of i is by simple induction on $\models$ definition. The proof of ii is by induction on the derivation of $\bigcup_{i=1}^{\infty} I_i \models \Delta \cdot \mathcal{A}$.

- If $\Delta = \top, \Delta'$, obvious;
- if $\Delta$ is a fact, then $\bigcup_{i=1}^{\infty} I_i \models \Delta \triangleright A$ means $\Delta + A \in \bigcup_{i=1}^{\infty} I_i$, which in turn implies that there exists $k$ such that $\Delta + A \in I_k$, therefore $I_k \models \Delta \triangleright A$;

- if $\Delta = G_1 \land G_2, \Delta'$, then by inductive hypothesis there exist $k_1$ and $k_2$ s.t. $I_{k_1} \models G_1, \Delta' \triangleright A$ and $I_{k_2} \models G_2, \Delta' \triangleright A$. By taking $k = \max\{k_1, k_2\}$, by $i$ we have that $I_k \models G_1, \Delta' \triangleright A$ and $I_k \models G_2, \Delta' \triangleright A$, therefore $I_k \models G_1 \land G_2, \Delta' \triangleright A$, i.e., $I_k \models \Delta \triangleright A$ as required;

- if $\Delta = G_1 \nmid G_2, \Delta'$ or $\Delta = \bot, \Delta'$, the conclusion follows by a straightforward application of the inductive hypothesis.

We now come to the definition of the fixpoint operator $T_P$.

**Definition 7.7 (Fixpoint Operator $T_P$)** Given a propositional LO program $P$ and an interpretation $I$, the fixpoint operator $T_P$ is defined as follows:

$$T_P(I) = \{ \hat{H} + A \mid H \leftarrow G \in P, \ I \models G \triangleright A \}.$$ 

The following property holds.

**Proposition 7.8 (Monotonicity and Continuity)** For every propositional LO program $P$, the fixpoint operator $T_P$ is monotonic and continuous over the lattice $\langle D, \subseteq \rangle$.

**Proof** Monotonicity. Immediate from $T_P$ definition and Lemma 7.6 i.

Continuity. We prove that $T_P$ is finitary. Namely, given a chain of interpretations $I_1 \subseteq I_2 \subseteq \ldots$, $T_P$ is finitary if $T_P(\bigcup_{i=1}^{\infty} I_i) \subseteq \bigcup_{i=1}^{\infty} T_P(I_i)$. Let $A \in T_P(\bigcup_{i=1}^{\infty} I_i)$. By $T_P$ definition, there exist a clause $H \leftarrow G \in P$ and a fact $B$ s.t. $A = \hat{H} + B$ and $\bigcup_{i=1}^{\infty} I_i \models G \triangleright B$. By Lemma 7.6 ii, we have that there exists $k \in \mathbb{N}$ s.t. $I_k \models G \triangleright B$. Again by $T_P$ definition, we get $A = \hat{H} + B \in T_P(I_k) \subseteq \bigcup_{i=1}^{\infty} T_P(I_i)$.

Monotonicity and continuity of the $T_P$ operator imply, by Tarski’s Theorem, that $\text{lfp}(T_P) = T_P^{\uparrow}$. By analogy with fixpoint semantics for Horn clauses [Llo87], the fixpoint semantics of a propositional LO program $P$ is then defined as follows.

**Definition 7.9 (Fixpoint Semantics)** Given a propositional LO program $P$, its fixpoint semantics, denoted $F(P)$, is defined as follows:

$$F(P) = \text{lfp}(T_P) = T_P^{\uparrow}.$$
Intuitively, $T_P(I)$ is the set of immediate logical consequences of the program $P$ and of the facts in $I$. In fact, if we define $P_I$ as the program $\{ A \leftarrow \top \mid \hat{A} \in I \}$, the definition of $T_P$ can be viewed as the following instance of the cut rule of linear logic (see Section 3.1):

$$
\begin{array}{c}
\vdash P^\perp, P^\perp_1 : H, G^\perp \vdash P^\perp, P^\perp_1 : G, \mathcal{A} \\
\hline 
\vdash P^\perp, P^\perp_1 : H, \mathcal{A}
\end{array}
$$

which is equivalent to

$$
\begin{array}{c}
\vdash (P, P^\perp_1) : H \circ \dashv G \vdash (P, P^\perp_1) : G, \mathcal{A} \\
\hline 
\vdash (P, P^\perp_1) : H, \mathcal{A}
\end{array}
$$

Note that, since $H \circ \dashv G \in P$, the sequent $\vdash (P, P^\perp_1) : H \circ \dashv G$ is always provable in linear logic. According to this view, $F(P)$ characterizes the set of logical consequences of a program $P$.

We conclude this section by proving the following fundamental result, which states that the fixpoint semantics is sound and complete with respect to the operational semantics.

**Theorem 7.10 (Soundness and Completeness)** For every propositional LO program $P$, $F(P) = O(P)$.

**Proof** $F(P) \subseteq O(P)$. We prove that for every $k \in \mathbb{N}$ and context $\Delta$, if $T_P|_k \models \Delta \triangleright \epsilon$ then $P \vdash \Delta$. The proof is by lexicographic induction on $(k, h)$, where $h$ is the length of the derivation of $T_P|_k \models \Delta \triangleright \epsilon$.

- If $\Delta = \top, \Delta'$, obvious;
- if $\Delta$ is a fact, then $\Delta \in T_P|_k$, so that $T_P|_k \not= \emptyset$ and $k > 0$. By definition of $T_P$ we have that there exists a fact $\mathcal{A}$ and a clause $\hat{H} \circ \dashv G \in P$, such that $T_P|_{k-1} \models G \triangleright \mathcal{A}$ and $\Delta = \hat{H}. \mathcal{A}$. By Lemma 7.5 we have that $T_P|_{k-1} \models G, \mathcal{A} \triangleright \epsilon$, and then, by inductive hypothesis, $P \vdash G, \mathcal{A}$, therefore by LO bc rule, $P \vdash \hat{H}, \mathcal{A}$, i.e., $P \vdash \Delta$;
- if $\Delta = G_1 \& G_2, \Delta'$ then by inductive hypothesis $P \vdash G_1, \Delta'$ and $P \vdash G_2, \Delta'$, therefore $P \vdash G_1 \& G_2, \Delta'$ by LO \& rule;
- if $\Delta = G_1 \Rightarrow G_2, \Delta'$ or $\Delta = \bot, \Delta'$, the conclusion follows by a straightforward application of the inductive hypothesis.

$O(P) \subseteq F(P)$. We prove that for every context $\Delta$ if $P \vdash \Delta$ then there exists $k \in \mathbb{N}$ such that $T_P|_k \models \Delta \triangleright \epsilon$. The proof is by induction on the derivation of $P \vdash \Delta$.

87
- If the proof ends with an application of $\top$, then the conclusion is immediate;

- if the proof ends with an application of the $bc$ rule, then $\Delta = A_1, \ldots, A_n, A$, where $A_1, \ldots, A_n$ are atomic formulas, and there exists a clause $A_1 \nexists \ldots \nexists A_n \leftarrow G \in P$. For the uniformity of LO proofs, we can suppose $A$ to be a fact. By inductive hypothesis, we have that there exists $k$ such that $T_P \uparrow^k \models G, A\triangleright \epsilon$, then, by Lemma 7.5, $T_P \uparrow^k \models G, A\triangleright A$, which, by definition of $T_P$, in turn implies that $A_1, \ldots, A_n, A \in T_P(T_P \uparrow^k) = T_P \uparrow^{k+1}$, therefore $T_P \uparrow^{k+1} \models A_1, \ldots, A_n, A\triangleright \epsilon$, i.e., $T_P \uparrow^{k+1} \models \Delta\triangleright \epsilon$;

- if the proof ends with an application of the $\&$ rule, then $\Delta = G_1 \& G_2$, $\Delta'$ and, by inductive hypothesis, there exist $k_1$ and $k_2$ such that $T_P \uparrow_{k_1} \models G_1, \Delta'\triangleright \epsilon$ and $T_P \uparrow_{k_2} \models G_2, \Delta'\triangleright \epsilon$. Then, if $k = \max\{k_1, k_2\}$ we have, by Lemma 7.6 i and monotonicity of $T_P$, that $T_P \uparrow_{k} \models G_1, \Delta'\triangleright \epsilon$ and $T_P \uparrow_{k} \models G_2, \Delta'\triangleright \epsilon$, therefore $T_P \uparrow_{k} \models G_1 \& G_2, \Delta'\triangleright \epsilon$, i.e., $T_P \uparrow_{k} \models \Delta\triangleright \epsilon$;

- the $\forall$-case and $\perp$-case follow by a straightforward application of the inductive hypothesis.

We note that it is also possible to define a model-theoretic semantics (as for classical logic programming [GDL95]) based on the notion of least model with respect to a given class of models and partial order relation. In our setting, the partial order relation is simply set inclusion, while models are exactly Herbrand interpretations which satisfy program clauses, i.e., $I$ is a model of $P$ if and only if for every clause $H \leftarrow G \in P$ and for every fact $A$

$$I \models G \triangleright A \text{ implies } I \models H \triangleright A.$$ 

It turns out that the operational, fixpoint and model-theoretic semantics are all equivalent. We omit details. Finally, we also note that these semantics can be proved equivalent to the phase semantics for LO given in [AP91b].

### 7.2 An Effective Semantics for LO

The operator $T_P$ defined in the previous section does not enjoy one of the crucial properties we required for our bottom-up semantics, namely its definition is not effective. As an example, take the program $P$ consisting of the clause $a \leftarrow \top$. Then, $T_P(\emptyset)$ is the set of all multisets with at least one occurrence of $a$, which is an infinite set. In other words, $T_P(\emptyset) = \{B \mid a \not\preceq B\}$, where $\not\preceq$ is the multiset inclusion relation. In order to compute effectively one step of $T_P$, we have to find a finite representation of potentially infinite
sets of facts (in the terminology of [ACJT96], a constraint system). The previous example suggests us that a provable fact $A$ may be used to denote the ideal generated by $A$ (see Definition 6.9), i.e., the subset of $B_P$ defined as follows: $[A] = \{B \mid A \preceq B\}$. We extend this definition to interpretations as follows.

**Definition 7.11 (Denotation of an Interpretation)** Given an interpretation $I$, its denotation $[I]$ is the interpretation given by

$$[I] = \bigcup_{A \in I} [A].$$

Two interpretations $I$ and $J$ are said to be equivalent, written $I \simeq J$, if and only if $[I] = [J]$.

Based on this idea, we define an abstract Herbrand base where we handle every single fact $A$ as a representative element for $[A]$ (note that in the semantics of Section 7.1 the denotation of a fact $A$ is $A$ itself).

**Definition 7.12 (Abstract Interpretation Domain)** Abstract interpretations form a complete lattice $\langle \mathcal{I}, \sqsubseteq \rangle$ where

- $\mathcal{I} = \{[I]_\simeq \mid I$ is an interpretation\};
- $[I]_\simeq \sqsubseteq [J]_\simeq$ iff $[I] \subseteq [J]$, i.e., iff for every $B \in I$ there exists $A \in J$ such that $A \preceq B$;
- the least upper bound of $[I]_\simeq$ and $[J]_\simeq$, written $[I]_\simeq \sqcup [J]_\simeq$, is $[I \cup J]_\simeq$;
- the bottom and top elements are $[\emptyset]_\simeq$ and $[\epsilon]_\simeq$, respectively.

The equivalence $\simeq$ allows us to reason modulo redundancies. For instance, any $A$ is redundant in $\{\epsilon, A\}$, which, in fact, is equivalent to $\{\epsilon\}$. It is important to note that to compare two ideals we simply need to compare their generators w.r.t. the multiset inclusion relation $\preceq$. Thus, given a finite set of facts we can always remove all redundancies using a polynomial number of comparisons.

**Notation.** For the sake of simplicity, in the rest of the paper we will often identify an interpretation $I$ with its class $[I]_\simeq$. Furthermore, note that if $A \preceq B$, then $[B] \subseteq [A]$. In contrast, if $I$ and $J$ are two interpretations and $I \subseteq J$ then $[I] \subseteq [J]$.

The two relations $\preceq$ and $\sqsubseteq$ are well-quasi-orderings (see Section 6.2.1). This is a direct consequence of Proposition 6.25 (Dickson’s Lemma). This property is the key point for proving termination of the fixpoint computation for propositional LO programs. In fact, it
will allow us to prove that the computation of the least fixpoint of the symbolic formulation of the operator $T_P$ (working on abstract interpretations) is guaranteed to terminate on every input LO program.

By definition of $\subseteq$ and Proposition 6.23, the following corollary holds.

**Corollary 7.13** There are no infinite chains of interpretations $I_0 \subseteq I_1 \subseteq I_2 \subseteq \ldots$ such that for all $j,k \in \mathbb{N}$ with $j < k$, $I_k \nsubseteq I_j$.

Corollary 7.13 ensures that it is not possible to generate infinite sequences of interpretations that is *correct* and *complete* w.r.t. the ground semantics. If we find it, then we can use the corollary to prove that (for any program) its fixpoint can be reached after finitely many steps. For this purpose and using the multiset operations $\setminus$ (difference), $\bullet$ (least upper bound w.r.t. $\preceq$), and $\epsilon$ (empty multiset) defined in Section 2.2, we first define a new, abstract version, of the satisfiability relation $|=$. The intuition under the new judgment $I \models \Delta \triangleright A$, where $I$ is an (abstract) interpretation, $\Delta$ is a context, and $A$ is a fact, is that $A$ is the *minimal* fact (in a sense to be clarified) that should be added to $\Delta$ in order for $A + \Delta$ to be satisfiable in $I$.

**Definition 7.14 (Abstract Satisfiability Judgment)** Let $P$ be a propositional LO program and $I \in \mathcal{P}(B_P)$. The abstract satisfiability judgment $\models$ is defined as follows:

\begin{align*}
I \models \top, \Delta \triangleright \epsilon; \\
I \models A \triangleright B \setminus A \text{ for } B \in I; \\
I \models G_1 \& G_2, \Delta \triangleright A_1 \bullet A_2 \text{ if } I \models G_1, \Delta \triangleright A_1, I \models G_2, \Delta \triangleright A_2; \\
I \models G_1 \not\models G_2, \Delta \triangleright A \text{ if } I \models G_1, G_2, \Delta \triangleright A; \\
I \models \bot, \Delta \triangleright A \text{ if } I \models \Delta \triangleright A.
\end{align*}

Given a finite interpretation $I$ and a context $\Delta$, the previous definition gives us an *algorithm* to compute all facts $A$ such that $I \models \Delta \triangleright A$ holds.

**Example 7.15** Let us consider the following clause and interpretation:

\[ b \leftarrow (d \not\models e) \& f, \quad I = \{\{c,d\}, \{c,f\}\}. \]

We want to compute the facts $A$ for which $I \models G \triangleright A$, where $G = (d \not\models e) \& f$ is the body of the clause. From the second rule defining the judgment $\models$, we have that $I \models \{d,e\} \bullet \{c\}$, because $\{c,d\} \in I$ and $\{c,d\}\setminus\{d,e\} = \{c\}$. Therefore we get $I \models d \not\models e \triangleright \{c\}$ using the fourth
rule for $\vdash$. Similarly, we have that $I \vdash \{ f \} \triangleright \{ c \}$, because $\{ c, f \} \in I$ and $\{ c, f \} \setminus \{ f \} = \{ c \}$. By applying the third rule for $\vdash$, with $G_1 = d \not\in e$, $G_2 = f$, $A_1 = \{ c \}$, $A_2 = \{ c \}$ and $\Delta = \epsilon$, we get $I \vdash G \triangleright \{ c \}$, in fact $\{ c \} \triangleright \{ c \} = \{ c \}$. There are other ways to apply the rules for $\vdash$. In fact, we can get $I \vdash \{ d, e \} \triangleright \{ c, f \}$, because $\{ c, f \} \in I$ and $\{ c, f \} \setminus \{ d, e \} = \{ c, f \}$. Similarly, we can get $I \vdash \{ f \} \triangleright \{ c, d \}$. By considering all combinations, it turns out that $I \vdash G \triangleright A$, for every $A \in \{ \{ c \}, \{ c, f \}, \{ c, d \}, \{ c, d, f \} \}$. The information conveyed by $\{ c, f \}, \{ c, d \}, \{ c, d, f \}$ is in some sense redundant, as we will see in the following (see Example 7.26). In other words, it is not always true that the output fact of the judgment $\vdash$ is minimal (in the previous example only the output $\{ c \}$ is minimal). Nevertheless, the important point to be stressed here is that the set of possible facts satisfying the judgment, given $I$ and $G$, is finite. This will be sufficient to ensure effectiveness of the fixpoint operator.

The connection between the satisfiability judgments $\models$ and $\vdash$ is clarified by the following lemma.

**Lemma 7.16** For every $I \in \mathcal{P}(B_P)$, context $\Delta$, and fact $C$,

i. if $I \vdash \Delta \triangleright A$, then $[I] \models \Delta \triangleright A'$ for all $A'$ s.t. $A \prec A'$;

ii. if $[I] \models \Delta \triangleright A'$, then there exists $A$ such that $I \vdash \Delta \triangleright A$ and $A \prec A'$.

**Proof**

i. By induction on the derivation of $I \vdash \Delta \triangleright A$.

- $I \vdash \top, \Delta \vdash \epsilon$ and $[I] \models \top, \Delta \triangleright A'$ and $\epsilon \prec A'$ for any $A'$;

- if $I \vdash A \triangleright A'$ then $A' = B \setminus A$ for $B \in I$. Since $B \preceq (B \setminus A) + A = A' + A$, we have that $(B \setminus A) + A \in [I]$, therefore $[I] \models A \triangleright B \setminus A$, so that $[I] \models A \triangleright C$ for all $C$ s.t. $A' = B \setminus A \preceq C$, because $[I]$ is upward closed;

- if $I \vdash G_1, G_2, \Delta \triangleright A$ then $A = A_1 \bullet A_2$ and $I \vdash G_1, \Delta \triangleright A_1$ and $I \vdash G_2, \Delta \triangleright A_2$.

By inductive hypothesis, $[I] \models G_1, \Delta \triangleright B_1$ and $[I] \models G_2, \Delta \triangleright B_2$ for any $B_1, B_2$ s.t. $A_1 \preceq B_1$ and $A_2 \preceq B_2$. That is, $[I] \models G_i, \Delta \triangleright C$ for any $C \in [A_1 \bullet A_2]$ $i : 1, 2$.

It follows that $[I] \models G_1 \land G_2, \Delta \triangleright C$ for all $C \in [A_1 \bullet A_2]$;

- the $\exists$-case and $\bot$-case follow by a straightforward application of the inductive hypothesis.

ii. By induction on the derivation of $[I] \models \Delta \triangleright A'$.

- The $\top$-case follows by definition;
- if $[I] \models A \triangleright A'$ then $A' + A \in [I]$, i.e., there exists $B \in I$ s.t. $B \preceq A' + A$. Since $B \setminus A \preceq (A' + A) \setminus A = A'$, it follows that for $C = B \setminus A$, $I \models A \triangleright C$ and $C \preceq A'$;

- if $[I] \models G_1 \land G_2, \Delta \triangleright A$ then $[I] \models G_i, \Delta \triangleright A$ for $i : 1, 2$. By inductive hypothesis, there exists $A_i$ such that $A_i \preceq A$, $I \models G_i, \Delta \triangleright A_i$ for $i : 1, 2$, i.e., $I \models G_1 \land G_2, \Delta \triangleright A_1 \cup A_2$. The thesis follows noting that $A_1 \cup A_2 \preceq A$;

- the $\exists$-case and $\bot$-case follow by a straightforward application of the inductive hypothesis.

$\square$

The satisfiability judgment $\models$ also satisfies the following properties.

**Lemma 7.17** For any $I_1, I_2, \ldots \in \mathcal{P}(B_P)$, context $\Delta$, and fact $A$,

i. if $I_1 \subseteq I_2$ and $I_1 \models \Delta \triangleright A$ then there exists $A'$ such that $I_2 \models \Delta \triangleright A'$ and $A' \preceq A$;

ii. if $I_1 \subseteq I_2 \subseteq \ldots$, and $\bigcup_{i=1}^{\infty} I_i \models \Delta \triangleright A$ then there exists $k \in \mathbb{N}$ s.t $I_k \models \Delta \triangleright A$.

**Proof**

i. If $I_1 \models \Delta \triangleright A$, then by Lemma 7.16 i, $[I_1] \models \Delta \triangleright A$. Since $[I_1] \subseteq [I_2]$ then, by Lemma 7.6 i, $[I_2] \models \Delta \triangleright A$. Thus, by Lemma 7.16 ii, there exists $A' \preceq A$ s.t. $I_2 \models \Delta \triangleright A'$;

ii. If $\bigcup_{i=1}^{\infty} I_i \models \Delta \triangleright A$, then by Lemma 7.16 i, $\bigcup_{i=1}^{\infty} I_i \models \Delta \triangleright A$, i.e., as it can be readily verified from Definition 7.11 and Definition 7.12, $\bigcup_{i=1}^{\infty} [I_i] \models \Delta \triangleright A$. By Lemma 7.6 ii, there exists $k \in \mathbb{N}$ s.t. $[I_k] \models \Delta \triangleright A$. Thus, by Lemma 7.16 ii, there exists $A' \preceq A$ s.t. $I_k \models \Delta \triangleright A'$.

$\square$

We are ready now to define the abstract fixpoint operator $S_P : \mathcal{I} \rightarrow \mathcal{I}$. We will proceed in two steps. We will first define an operator working over elements of $\mathcal{P}(B_P)$. With a little bit of overloading, we will call the operator with the same name, i.e., $S_P$. As for the $S_P$ operator used in the symbolic semantics of CLP programs [JM94], the operator should satisfy the equation $[S_P(I)] = T_P([I])$ for every $I \in \mathcal{P}(B_P)$. This property ensures the soundness and completeness of the *symbolic* representation w.r.t. the concrete one.

After defining the operator over $\mathcal{P}(B_P)$, we will lift it to our abstract domain $\mathcal{I}$ consisting of the equivalence classes of elements of $\mathcal{P}(B_P)$ w.r.t. the relation $\simeq$ defined in Definition 7.11. Formally, we first introduce the following definition.

92
Definition 7.18 (Symbolic Fixpoint Operator $S_P$) Given a propositional LO program $P$ and $I \in \mathcal{P}(B_P)$, the symbolic fixpoint operator $S_P$ is defined as follows:

$$S_P(I) = \{ \hat{H} + \mathcal{A} \mid H \circ G \in P, \ I \models G \triangleright \mathcal{A} \}.$$  

The following property shows that $S_P$ is sound and complete w.r.t. $T_P$.

Proposition 7.19 For every propositional LO program $P$ and $I \in \mathcal{P}(B_P)$, $[S_P(I)] = T_P([I])$.

Proof Let $A = \hat{H}, B \in S_P(I)$ where $H \circ G \in P$ and $I \models G \triangleright B$ then, by Lemma 7.16 i, $[I] \models G \triangleright B'$ for any $B'$ s.t. $B \preceq B'$. Thus, for any $A' = \hat{H}, B'$ s.t. $A \preceq A', A' \in T_P([I])$. Vice versa, if $A \in T_P([I])$ then $A = \hat{H}, B$ where $H \circ G \in P$ and $[I] \models G \triangleright B$. By Lemma 7.16 ii, there exists $B'$ s.t. $B' \preceq B$ and $I \models G \triangleright B'$, i.e., $A' = \hat{H}, B' \in S_P(I)$ and $A' \preceq A$. □

Furthermore, the following corollary holds.

Corollary 7.20 For every propositional LO program $P$ and $I, J \in \mathcal{P}(B_P)$, if $I \simeq J$ then $S_P(I) \simeq S_P(J)$.

Proof If $I \simeq J$, i.e., $[I] = [J]$, then $T_P([I]) = T_P([J])$. Thus, by Prop. 7.19 it follows that $[S_P(I)] = [S_P(J)]$, i.e., $S_P(I) \simeq S_P(J)$. □

The previous corollary allows us to safely lift $S_P$ definition from the lattice $(\mathcal{P}(B_P), \subseteq)$ to the lattice $(\mathcal{I}, \sqsubseteq)$. Formally, we define the abstract fixpoint operator as follows.

Definition 7.21 (Abstract Fixpoint Operator $S_P$) Given a propositional LO program $P$ and an equivalence class $[I]_{\simeq}$ of $\mathcal{I}$, the abstract fixpoint operator $S_P$ is defined as follows:

$$S_P([I]_{\simeq}) = [S_P(I)]_{\simeq}$$

where $S_P(I)$ is defined in Definition 7.18.

For the sake of simplicity, in the following we will use $I$ to denote its class $[I]_{\simeq}$. The abstract fixpoint operator $S_P$ satisfies the following property.

Proposition 7.22 (Monotonicity and Continuity) For every propositional LO program $P$, the abstract fixpoint operator $S_P$ is monotonic and continuous over the lattice $(\mathcal{I}, \sqsubseteq)$.  

93
Procedure Symbolic Fixpoint
input $P$: an LO program
output $F_{sym}(P)$
begin
    $New := \{H \mid H \circ \top \in P\}$; $Old := \emptyset$
    repeat
        $Old := New$
        $New := SP(New)$
    until $New \sqsubseteq Old$
return $Old$
end

Figure 7.2: Symbolic fixpoint computation

Proof Monotonicity. For any $A = \hat{H}, B \in SP(I)$ there exists $H \circ G \in P$ s.t. $I \models G \bullet B$. Assume now that $I \sqsubseteq J$. Then, by Lemma 7.17 i, we have that $J \models G \bullet B'$ for $B' \preceq B$. Thus, there exists $A' = \hat{H}, B' \in SP(J)$ such that $A' \preceq A$, i.e., $SP(I) \sqsubseteq SP(J)$.

Continuity. We show that $SP$ is finitary. Let $I_1 \subseteq I_2 \subseteq \ldots$ be a chain of interpretations. For any $A = \hat{H}, B \in SP(\bigsqcup_{i=1}^{\infty} I_i)$ there exists $H \circ G \in P$ s.t. $\bigsqcup_{i=1}^{\infty} I_i \models G \bullet B$. By Lemma 7.17 ii, there exists $k \in \mathbb{N}$ s.t. $I_k \models G \bullet B'$ for $B' \preceq B$. Thus, $A' = \hat{H}, B' \in SP(I_k)$, i.e., $A' \in \bigsqcup_{i=1}^{\infty} SP(I_i)$ with $A' \preceq A$, i.e., $SP(\bigsqcup_{i=1}^{\infty} I_i) \sqsubseteq \bigsqcup_{i=1}^{\infty} SP(I_i)$.

Corollary 7.23 For every propositional LO program $P$, $[lfp(S_P)] = lfp(T_P)$.

Let $F_{sym}(P) = lfp(S_P)$, then we have the following main theorem.

Theorem 7.24 (Soundness and Completeness) For every propositional LO program $P$, $O(P) = F(P) = [F_{sym}(P)]$. Furthermore, there exists $k \in \mathbb{N}$ such that $F_{sym}(P) = \bigsqcup_{i=0}^{k} SP[\uparrow_k (\emptyset)]$.

Proof Theorem 7.10 and Corollary 7.23 show that $O(P) = F(P) = [F_{sym}(P)]$. Corollary 7.13 guarantees that the fixpoint of $SP$ can always be reached after finitely many steps.

The previous results give us an algorithm to compute the operational and fixpoint semantics of a propositional LO program via the operator $SP$. The algorithm is inspired by the backward reachability algorithm used in [ACJT96, FS01] to compute backwards the closure of the predecessor operator of a well-structured transition system. The algorithm in pseudo-code for computing $F(P)$ is shown in Figure 7.2. Corollary 7.13 guarantees that the algorithm always terminates and returns a symbolic representation of $O(P)$. As a corollary of Theorem 7.24, we obtain the following result.
of other judgments for clause 2, as seen in Example 7.15, for instance therefore we have, from the rule for $\&$, that

\[
\begin{array}{c}
P \vdash e, b, \top \\
P \vdash d, e, b, c \\
P \vdash d, e, b \not\in c \\
P \vdash d, e, e, e \\
P \vdash d \not\in e, e, e \\
P \vdash (d \not\in e) \& f, e, e \\
P \vdash b, e, e
\end{array}
\]

Figure 7.3: Another example of LO derivation

**Corollary 7.25** The provability of $P \vdash G$ in propositional LO is decidable.

In view of Proposition 3.9, this result can be considered as an instance of the general decidability result [Kop95] for propositional affine linear logic (i.e., linear logic with weakening).

**Example 7.26** We compute the fixpoint semantics for the program $P$ of Example 3.10, which is given below.

1. $a \rightarrow b \not\in c$
2. $b \rightarrow (d \not\in e) \& f$
3. $c \not\in d \rightarrow \top$
4. $e \not\in e \rightarrow b \not\in c$
5. $c \not\in f \rightarrow \top$

We start the computation from $S_P \uparrow_0 = \emptyset$. The first step consists in adding the multisets corresponding to program facts, i.e., clauses 3 and 5, therefore we compute

$S_P \uparrow_1 = \{\{c, d\}, \{c, f\}\}$.

Now, we can try to apply clauses 1, 2, and 4 to facts in $S_P \uparrow_1$. From the first clause, we have that $S_P \uparrow_1 \models \{b, c\} \rightarrow \{d\}$ and $S_P \uparrow_1 \models \{b, c\} \rightarrow \{f\}$, therefore $\{a, d\}$ and $\{a, f\}$ are elements of $S_P \uparrow_2$. Similarly, for clause 2 we have that $S_P \uparrow_1 \models \{d, e\} \rightarrow \{c\}$ and $S_P \uparrow_1 \models \{f\} \rightarrow \{c\}$, therefore we have, from the rule for $\&$, that $\{b, c\}$ belongs to $S_P \uparrow_2$ (we can also derive other judgments for clause 2, as seen in Example 7.15, for instance $S_P \uparrow_1 \models \{d, e\} \rightarrow \{c, f\}$), but it immediately turns out that all these judgments give rise to redundant information, i.e., facts that are subsumed by the already calculated ones). By clause 4, finally we have that $S_P \uparrow_1 \models \{b, c\} \rightarrow \{d\}$ and $S_P \uparrow_1 \models \{b, c\} \rightarrow \{f\}$, therefore $\{d, e, e\}$ and $\{e, e, f\}$ belong to $S_P \uparrow_2$. We can therefore take the following equivalence class as representative for $S_P \uparrow_2$:

$S_P \uparrow_2 = \{\{c, d\}, \{c, f\}, \{a, d\}, \{a, f\}, \{b, c\}, \{d, e, e\}, \{e, e, f\}\}$.
We can similarly calculate $S_P \uparrow^3$. For clause 1 we immediately have that $S_P \uparrow^2 \vdash \{b, c\} \bullet \epsilon$, so that $\{a\}$ is an element of $S_P \uparrow^3$; this makes the information given by $\{a, d\}$ and $\{a, f\}$ in $S_P \uparrow^2$ redundant. From clause 4 we can get that $\{e, e\}$ is another element of $S_P \uparrow^3$, which implies that the information given by $\{d, e, e\}$ and $\{e, e, f\}$ is now redundant. No additional (not redundant) elements are obtained from clause 2. We therefore can take

$$S_P \uparrow^3 = \{\{c, d\}, \{c, f\}, \{b, c\}, \{a\}, \{e, e\}\}.$$ 

The reader can verify that $S_P \uparrow^4 = S_P \uparrow^3 = F_{sym}(P)$ so that

$$O(P) = F(P) = \llbracket\{c, d\}, \{c, f\}, \{b, c\}, \{a\}, \{e, e\}\rrbracket.$$ 

We suggest the reader to compare the top-down proof for the goal $e, e$, given in Figure 3.3, and the part of the bottom-up computation which yields the same goal. The order in which the backchaining steps are performed is reversed, as expected. Moreover, the top-down computation requires to solve one goal, namely $d, e, c$, which is not minimal, in the sense that its proper subset $c, d$ is still provable. Using the bottom-up algorithm sketched above, at every step only the minimal information (in this case $c, d$) is kept at every step.

In general, this strategy has the further advantage of reducing the amount of non-determinism in the proof search. For instance, let us consider the goal $b, e, e$ (which is certainly provable because of Proposition 3.9). This goal has at least two different proofs. The first is a slight modification of the proof in Figure 3.3 (just add the atom $b$ to every sequent).

An alternative proof, obtained by changing the order of applications of the backchaining steps, is given in Figure 7.3. There are even more complicated proofs (for instance in the left branch we could rewrite $b$ again by backchaining over clause 2 rather than axiom 3). The bottom-up computation avoids these complications by keeping only minimal information at every step. We would also like to stress that the bottom-up computation is always guaranteed to terminate, as stated in Theorem 7.24, while in general the top-down computation can diverge.

\[\square\]

### 7.3 Examples

We conclude this chapter by showing how computation of the bottom-up semantics for propositional LO programs can be used to prove safety properties. In particular, we consider the producer/consumer example of Section 4.2.1 and the net for mutual exclusion of
Section 6.1.1. We use these examples to exemplify our methodology for backward verification of safety properties. Other examples will be presented in the following chapters. Verification has been performed by means of an automatic tool implementing the procedure for the covering problem described in Figure 6.3. We refer the reader to Appendix A for a description of this tool and details about the environment in which the experiments have been conducted.

7.3.1 A Producer/Consumer Example

Let us consider the Petri net of Figure 4.1, representing a producer/consumer net with a bounded capacity buffer. According to Section 4.3, translation of this net yields the following LO program:

1. producing $\circ -$ releasing
2. releasing $\&$ free $\circ -$ producing $\&$ buffer
3. buffer $\&$ buffer $\&$ acquiring $\circ -$ free $\&$ free $\&$ consuming
4. consuming $\circ -$ acquiring

If we wish to verify properties of the reachable states of the Petri net in Figure 4.1, first of all we need to give a specification of the initial states of the system. Specifically, we need

Figure 7.4: Fixpoint computed for the producer/consumer example
to encode the initial marking drawn in Figure 4.1. To this aim, we can simply introduce a
new propositional symbol (place) init and add the following LO clause:

5. init : ¬releasing ⊕ buffer ⊕ buffer ⊕ buffer ⊕ free ⊕ free ⊕ acquiring

The only purpose of init is to load the initial configuration of the system.

We can now verify a safety property using the backward reachability algorithm of Section 6.1.1. For instance, the Petri net in Figure 4.1 has the following bounded capacity property:
at any time, the buffer contains at most five items. We can formally verify that the above
specification complies with the bounded capacity property using the following strategy.
First of all, we can see the above problem as an instance of the covering problem for the
Petri net in Figure 4.1. In fact, the bounded capacity property is equivalent to saying
that the buffer will never contain more than five items. Therefore, we need to verify that
starting from the initial marking we will never reach a marking containing six or more
items in place buffer. Otherwise said, we have to verify that it is not possible to cover the
following marking: \{buffer, buffer, buffer, buffer, buffer, buffer\}. According to the discussion
of Section 6.3, we can encode this problem using the following LO axiom:

6. buffer ⊕ buffer ⊕ buffer ⊕ buffer ⊕ buffer ⊕ buffer ⊕ init ⊕ ⊤

The head of the above axiom represents the minimality violation of the safety property
under consideration. In other words, any marking with at least six items in place buffer is
unsafe. We recall that, from a logical point of view, this is a result of Proposition 3.9.

We can now run our verification tool (see Appendix A) to automatically verify if the
bounded capacity property holds. We simply need to evaluate the bottom-up semantics of
the above LO program, including the specification of the initial and of the final (unsafe)
states (i.e., we consider clauses 1 through 6), and check that the fixpoint does not contain
any legal initial state. In this case, the initial state (marking) is represented by the atom
init, therefore we simply need to verify that the marking \{init\} is not contained in the fix-
point, i.e., that \{init\} is not backward reachable. Termination of the bottom-up evaluation
is guaranteed by Lemma 7.13. Running the verification tool, we get the fixpoint shown in
Figure 7.4. The fixpoint contains 16 elements and is reached after 13 steps (iterations).
As it does not contain init, the bounded capacity property is verified.

We have used the producer/consumer example mainly to illustrate our approach and dis-
cuss our methodology for verification of safety properties. The above example is not par-
ticularly meaningful for the following reason. As discussed in Section 4.2.2 (see Example
4.16), the following invariant holds in the Petri net of Figure 4.1: the total number of
tokens in places buffer and free is always less or equal than five. The property we have
discussed above is a simple consequence of this invariant, and therefore it can be computed
statically. In the next section we present a more interesting example.
Let us consider the Petri net presented in Figure 6.2. The net represents a simple monitor for a parameterized system with two mutually exclusive critical sections (see discussion in Example 6.8). Two locks are used to protect the critical sections from unauthorized access. The net is parametric in the number of waiting processes, i.e., processes willing to enter one of two critical sections. Following the encoding of section 4.3, the above Petri net is represented by the LO program below:

1. \( \text{waiting} \rightarrow \text{lock1} \land \text{lock2} \land \text{cs1} \)
2. \( \text{waiting} \rightarrow \text{lock1} \land \text{lock2} \land \text{cs2} \)
3. \( \text{cs1} \rightarrow \text{waiting} \land \text{lock1} \)
4. \( \text{cs2} \rightarrow \text{waiting} \land \text{lock2} \)

The first two clauses encode transitions Enter1 and Enter2, whereas the last two clauses encode transitions Leave1 and Leave2, respectively.

Following the methodology illustrated in Section 7.3.1, we need to encode the initial and final (i.e., unsafe) states of the system. Initial states can be encoded by introducing a new propositional symbol \( \text{init} \) and adding the following LO clauses:

5. \( \text{init} \land \text{waiting} \)
6. \( \text{init} \land \text{lock1} \land \text{lock2} \)

The above encoding is very flexible, in that it allows us to leave the number of initial processes in place waiting (i.e., the parameter \( K \) in Figure 6.2) unspecified. In fact, by
using the first clause we can put any number of processes in place waiting, while the second clause initializes the two locks and let the processes compete for accessing the critical sections. For clarity, an example trace (LO proof fragment) is shown in Figure 7.5 (where $P$ is the LO program made up of clauses 1 through 6, and we have followed the usual notational conventions).

It remains to give the encoding of the unsafe states of the system. Specifically, a configuration is unsafe if at least two processes are accessing either of the critical sections at the same time. Namely, we have the following encoding in LO:

7. $cs_1 \not\forall cs_1 \circ \top$
8. $cs_1 \not\forall cs_2 \circ \top$
9. $cs_2 \not\forall cs_2 \circ \top$

Now, we can run our automatic verification tool (see Appendix A) to evaluate the bottom-up semantics of the above LO program (clauses 1 through 9), and check whether the fixpoint contains the configuration $\{init\}$ or not. We stress that in this case static computation of structural invariants for the Petri net of Figure 6.2 is not sufficient to prove the above safety property (see for instance [DRB01]). As an aside, we remark that the matrix representation of the Petri net in Figure 6.2 (upon which the calculus of structural invariants is based) is ambiguous, in the sense that there are cycles (e.g. the one given by the arc from $lock_2$ to Enter1 and the arc from Enter1 to $lock_2$) which are lost in the matrix representation. Running the verification tool, we get the fixpoint shown in Figure 7.6. The fixpoint contains 14 elements and is reached after 7 steps. As it does not contain $init$, the mutual exclusion property is verified.
We now show how the computation of structural invariants of Section 4.2.2, in combination with the methodology of pruning of Section 6.1.2.2, can be used to optimize the above fixpoint computation. Computing the place invariants for the Petri net in Figure 6.2 yields, among the others, the following properties: for every reachable marking \( M \),
\[
M(\text{lock}1) + M(\text{cs}1) = 1 \quad \text{and} \quad M(\text{lock}2) + M(\text{cs}2) = 1
\]
(see for instance [DRB01]). This in turn implies that configurations containing at least one token in \( \text{lock}1 \) and one token in \( \text{cs}1 \) (or, respectively, one token in \( \text{lock}2 \) and one token in \( \text{cs}2 \)) are unreachable and can therefore be discarded (pruned). Using dynamic pruning (this requires a slight modification of the fixpoint operator, see Section 6.1.2.2), this time the verification algorithm converges immediately at the first iteration. In Figure 7.7 we show the partial result computed at the second iteration of the algorithm without using dynamic pruning. With respect to the initial set of unsafe states, only two elements (the last two) are computed. These elements do not satisfy the above structural invariants, therefore they can be dynamically pruned. The fixpoint consists of the remaining configurations.

Using our verification tool, we have verified that the mutual exclusion property holds for the Petri net in Figure 6.2, independently of the number of initial processes. In other words, we have verified a safety property for a parametric system. We conclude this section by showing how even more sophisticated specifications are possible in linear logic. Specifically, we can support dynamic generation of waiting processes. Let us consider the following modified LO specification for the set of initial states:

\[
\begin{align*}
5'. \ init & \leftarrow \ demon \ \& \ \text{lock}1 \ \& \ \text{lock}2 \\
6'. \ \ demon & \leftarrow \ demon \ \& \ \text{waiting}
\end{align*}
\]

Initialization of the system directly creates the two locks and a demon. The demon is used to generate waiting processes at run-time. In other words, new waiting processes can be dynamically injected in the system at any time after the initialization phase. Evaluating the bottom-up semantics for the LO program obtained by substituting the above clauses 5’ and 6’ for, respectively, clauses 5 and 6, we can prove that the mutual exclusion property still holds for the above specification. The fixpoint contains 20 elements and is reached after 7 steps. Using dynamic pruning, the fixpoint computation converges immediately in the same way as for the original specification.
7.4 Related Work

The inspiration for our work (see [BDM02]) originally came from decidability results, based on the theory of well-quasi-orderings, for verification algorithms for validation of parameterized systems via a backward reachability analysis [ACJT96, AJ98, FS01] (see also Section 6.4). Specifically, in this chapter we have defined a bottom-up evaluation algorithm which solves the so-called covering problem (see [FS01]) for transition systems defined via the linear logic language LO [AP91b]. The decidability result for propositional LO provability, which we obtain here as a corollary, is not new. In particular, it is consequence of decidability of linear affine logic [Kop95]. Given that Petri nets can be encoded in LO (see Section 4.3), our bottom-up evaluation algorithm can be seen as an alternative to the Karp and Miller’s coverability graph construction for Petri nets [KM69] (which, however, as a difference with ours, is a forward construction). Here, we also deal with further connectives like the additive conjunction of linear logic.

From a logic programming perspective, our bottom-up semantics is clearly an extension of the traditional fixpoint semantics for Horn logic [Llo87] to a fragment of linear (affine) logic. As a difference, we need to add special rules, handled via the satisfiability judgment \( \models \) to handle formulas built over the logical connectives \( \otimes, \& , \top \) and \( \bot \).

Other sources of inspiration came from linear logic programming. In [HW98], the authors present an abstract deductive system for the bottom-up evaluation of linear logic programs. The left introduction, weakening and cut rules are used to compute the logical consequences of a given formula. The proof system the authors propose should be thought of as a general scheme to which evaluation of the specific language at hand should conform. The scheme is general enough to allow for different computational strategies, mixing bottom-up and top-down evaluation, breadth-first or depth-first search, eager or lazy evaluation. The satisfiability relations we use in the definition of the fixpoint operators correspond to top-down steps within their bottom-up evaluation scheme. The framework is discussed for a more general fragment than LO, in particular dynamically changing programs are supported. However, they do not provide an effective fixpoint operator as we did in the case of LO, and they do not discuss computability issues for the derivability relation.

In [APC97], Andreoli, Pareschi and Castagnetti present a partial evaluation scheme for propositional LO, which works as follows. Given an initial goal \( G \), they use a construction similar to Karp and Miller’s coverability graph [KM69] for Petri Nets to build a finite representation of a proof tree for \( G \). During the top-down construction of the tree for \( G \), they apply in fact a generalization step that works as follows. If a goal, say \( B \), that has to be proved is subsumed by a node already visited, say \( A \), (i.e., \( B = A + A' \)), then the part of proof tree between the two goals is replaced by a proof tree for \( A + (A')^* \), where \( A + (A')^* \) is a finite representation of the union of \( A \) with the closure of \( A' \). Using Dickson’s Lemma, the authors show that the construction always terminates. The main difference with our
approach is that we study a goal independent bottom-up evaluation strategy, therefore in our fixpoint semantics we do not need any generalization step. In fact, goals which are subsumed can be automatically discarded during the bottom-up evaluation process. This simplifies the computation as shown in Example 7.26. The partial evaluation scheme of [APC97] is proposed as a target for analysis of first-order LO programs, via the so-called counting abstraction, which consists in abstracting an atom with the corresponding predicate symbol. Clearly, we can do the same with our bottom-up evaluation scheme. However, in this thesis we allow for a much more refined analysis, by discussing directly a non ground semantics for first-order programs, based on unification and most general unifiers (see Chapters 9 and 10).

Finally, we must mention the connection between our semantics for LO and the bottom-up semantics for Disjunctive Logic Programs [MRL91]. In a disjunctive logic program, the head of a clause is a disjunction of atomic formulas, whereas the body is a conjunction of atomic formulas. In the semantics of [MRL91] interpretations are collections of sets (i.e., classical disjunctions) of atomic formulas, whereas LO interpretations are multisets (i.e., \(-\text{disjunctions}\)) of atomic formulas. Therefore, in the propositional case in order to prove the convergence of the fixpoint iteration, we need an argument (Dickson’s lemma) stronger than the finiteness of the extended Herbrand base of [MRL91] (collection of minimal sets). For a detailed discussion about the connection between LO and DLP, we refer the reader to [BDM01b]. Here, we have proved that, from a proof theoretical perspective, DLP can be derived from LO by admitting the structural rule of contraction. Furthermore, we have shown that the fixpoint semantics for a DLP program can be obtained by making abstract interpretation over the fixpoint semantics of a related LO program (specifically, the abstraction maps any multiset into the corresponding set).

---

**Summary of the Chapter.** In this chapter we have discussed the foundations of a bottom-up semantics for LO specifications. As a first step, we have dealt with propositional programs. We have proved that the bottom-up semantics is correct and complete with respect to the operational semantics, and we have shown an algorithm for evaluating the bottom-up semantics which is always guaranteed to terminate. As an application, we have presented some classical systems, specified in the Petri net formalism, which can be translated into LO theories and validated using this approach.

Building on the ideas presented here, in the following chapters we will extend the bottom-up semantics to more complex fragments of linear logic. First of all, in the next chapter we will discuss a simple addition enriching propositional LO with a more refined resource management. However, the price to pay will be that provability in general becomes undecidable.

---

103
Chapter 8

Refining LO Resource Management: LO with 1

This chapter extends the bottom-up semantics for LO of Chapter 3 by considering a richer set of logical connectives. In fact, the decidability of propositional provability shows that LO is not as interesting as one could expect from a state-oriented extension of the logic programming paradigm. Specifically, LO does not provide a natural way to count resources. This feature can be introduced by considering the language \( \text{LO}_1 \) (see Section 3.3.1), i.e., by considering a slight extension of LO in which we add unit clauses defined via the constant \( 1 \) and the multiplicative conjunction \( \otimes \). The resulting language can be viewed as a first step towards more complex languages based on linear logic like LinLog [And92]. In this chapter we will also show that \( \text{LO}_1 \) can be used to model more sophisticated models of concurrent systems than LO, like Petri Nets with transfer arcs and broadcast protocols.

The addition of the constant \( 1 \) breaks down the decidability of provability which holds instead for propositional LO. If we think about the connection between bottom-up semantics and verification (see Section 6.3), it turns out that allowing axioms of the form \( H \circlearrowleft 1 \) introduces the possibility of specifying set of configurations (e.g. markings for Petri nets) which are not upward-closed. Otherwise said, the result stated in Proposition 3.9 (admissibility of the weakening rule) does not hold anymore for the fragment \( \text{LO}_1 \). Remembering the connection between LO and Petri nets (see Section 4.3), we have that undecidability of \( \text{LO}_1 \) provability follows by classical results which show that there is no algorithm to compute the reachability set for Petri nets [EN94]. Despite this negative result, in this chapter we will still be able to define an effective symbolic fixpoint operator for \( \text{LO}_1 \).

Technically, we proceed as in Chapter 7. In particular, we formulate the bottom-up evaluation procedure in two steps. We first present a simple, non-effective notion of (concrete) interpretations and the corresponding definition of fixpoint operator. Then, we define an
abstract domain and a new symbolic fixpoint operator, which is shown to be equivalent to the previous one. The abstract domain is a constraint system in the sense of [AJ01b], i.e., it provides assertions to symbolically represent infinite sets of states. Specifically, we will introduce an abstract domain based on a special class of linear constraints defined over integer variables that count resources. This abstract domain generalizes the domain used for propositional LO: the latter can be represented as the subclass of constraints with no equalities. We remark that the use of constraints we make in this chapter is radically different from that of Chapter 9. Here, we use constraints at the meta-level as symbolic representations of semantical objects, while in Chapter 9 constraints are introduced at the syntactical (and also semantical) level, as a means to allow for reasoning on specialized and heterogeneous domains (e.g., integers, reals, strings, and so on) inside LO specifications.

For the sake of simplicity, we will overload some notations used in Chapter 7. Namely, the notations \( O(P) \), \( F(P) \) for operational and fixpoint semantics, \( T_P, S_P \) for the concrete and symbolic fixpoint operators, and \( \models, \sqsubseteq \) for the concrete and abstract satisfiability judgments will be re-used to denote the analogous concepts for the language \( \text{LO}_1 \).

This chapter extends our previous works [BDM00, BDM02].

### 8.1 A Bottom-Up Semantics for \( \text{LO}_1 \)

In the following we will use the same notations introduced in Chapter 7. We always assume a fixed signature \( \Sigma \) including a finite set of propositional symbols \( a_1, \ldots, a_n \). For the convenience of the reader, in Figure 8.1 we recall a proof system for propositional \( \text{LO}_1 \) (see also Section 3.3.1). A sequent is now provable if all branches of its proof tree terminate with instances of the \( \top_r \) or the \( 1_r \) axiom.

The top-down operational semantics can be defined as follows.

**Definition 8.1 (Operational Semantics)** Given a propositional \( \text{LO}_1 \) program \( P \), its
operational semantics, denoted $O(P)$, is given by

$$O(P) = \{A \mid A \text{ is a fact and } P \vdash_1 A\}.$$  

We first note that, in contrast with Proposition 3.9, the weakening rule is not admissible in LO$_1$. This implies that we cannot use the same techniques we used for LO for defining the abstract domain and also for proving termination of the bottom-up evaluation. So the question is: can we still find a finite representation of $O(P)$? The following proposition gives us a negative answer.

**Proposition 8.2 (Undecidability of LO$_1$ provability)** Given a propositional LO$_1$ program $P$, there is no algorithm to compute $O(P)$.

**Proof** We prove the proposition by presenting an encoding of Petri nets into LO$_1$. Let a Petri net $N$, with a set of initial markings $I = \{M_1, \ldots, M_n\}$, be given. Consider the LO (and thus LO$_1$) propositional program $P = \mathcal{P}(R(N))$ (see Section 4.3). Furthermore, for every marking $M_i \in I$ consider the LO$_1$ clause $H_i \leftarrow 1$, where $H_i$ is a $\forall$-disjunction of propositional symbols such that $\hat{H}_i = M_i$, for $i \in \{1, \ldots, n\}$. Let $P'$ be the program consisting of $P$ and the above $n$ clauses. Now, it is easy to recognize that computing the operational semantics of $P'$ amounts to computing the reachability set of $N$ with initial markings $I$. From classical results on Petri Nets (see e.g. the survey [EN94]), there is no algorithm to compute this set. The conclusion follows by reduction to the marking equivalence problem that is known to be undecidable.

As the reader can see, the above encoding of Petri nets into LO$_1$ does not make use of the multiplicative conjunction $\otimes$, therefore the operational semantics for the fragment of LO$_1$ without $\otimes$ (i.e., LO with the constant $1$) is still not computable. Despite Proposition 8.2, it is still possible to define a *symbolic*, effective fixpoint operator for LO$_1$ programs, as shown in Section 8.2. Before going into more details, we first rephrase the semantics of Section 7.1 for LO$_1$. We omit the relevant proofs, which are analogous to those of Section 7.1. The definitions of Herbrand base and interpretations are the same as in Chapter 7. The concrete semantics is defined on the usual domain $\langle D, \subseteq \rangle$, where $D = \mathcal{P}(B_P)$.

**Definition 8.3 (Satisfiability Judgment)** Let $P$ be a propositional LO$_1$ program and $I$
an interpretation. The satisfiability judgment $\models$ is defined as follows:

- $I \models \top, \Delta \triangleright A'$ for any fact $A'$;
- $I \models 1 \triangleright \epsilon$;
- $I \models A \triangleright A'$ if $A + A' \in I$;
- $I \models G_1 \land G_2, \Delta \triangleright A$ if $I \models G_1, \Delta \triangleright A$ and $I \models G_2, \Delta \triangleright A$;
- $I \models G_1 \lor G_2, \Delta \triangleright A_1 + A_2$ if $I \models G_1, \Delta \triangleright A_1$ and $I \models G_2, \Delta \triangleright A_2$;
- $I \models G_1 \otimes G_2, \Delta \triangleright A$ if $I \models G_1, G_2, \Delta \triangleright A$;
- $I \models \bot, \Delta \triangleright A$ if $I \models \Delta \triangleright A$.

The satisfiability judgment $\models$ satisfies the following properties.

**Lemma 8.4** For every interpretation $I$, context $\Delta$ and fact $A$,

$$I \models \Delta \triangleright A \iff I \models \Delta, A \triangleright \epsilon.$$  

**Lemma 8.5** For any interpretations $I_1, I_2$, context $\Delta$, and fact $A$,

i. if $I_1 \subseteq I_2$ and $I_1 \models \Delta \triangleright A$ then $I_2 \models \Delta \triangleright A$;

ii. if $I_1 \subseteq I_2 \subseteq \ldots$ and $\bigcup_{i=1}^{\infty} I_i \models \Delta \triangleright A$ then there exists $k \in \mathbb{N}$ s.t. $I_k \models \Delta \triangleright A$.

The fixpoint operator $T_P$ is defined like the one for LO.

**Definition 8.6 (Fixpoint Operator $T_P$)** Given a propositional LO\textsubscript{1} program $P$ and an interpretation $I$, the fixpoint operator $T_P$ is defined as follows:

$$T_P(I) = \{ \hat{H} + A \mid H \circ G \in P, I \models G \triangleright A \}.$$  

The following property holds.

**Proposition 8.7 (Monotonicity and Continuity)** For every propositional LO\textsubscript{1} program $P$, the fixpoint operator $T_P$ is monotonic and continuous over the lattice $\langle D, \subseteq \rangle$.

Monotonicity and continuity of the $T_P$ operator imply, by Tarski's Theorem, that $\text{lfp}(T_P) = T_P|_{\omega}$. The fixpoint semantics of a propositional LO\textsubscript{1} program $P$ is then defined as follows.
Definition 8.8 (Fixpoint Semantics) Given a propositional LO₁ program \( P \), its fixpoint semantics, denoted \( F(P) \), is defined as follows:

\[
F(P) = \text{lfp}(T_P) = T_P^\omega.
\]

The fixpoint semantics is sound and complete with respect to the operational semantics, as stated in the following theorem.

Theorem 8.9 (Soundness and Completeness) For every propositional LO₁ program \( P \), \( F(P) = O(P) \).

8.2 Constraint Semantics for LO₁

In this section we will introduce an abstract domain (a constraint system using the terminology of [AJ01b]) which consists of constraint-based representations of provable multisets, and we will define a symbolic fixpoint operator over this domain. The application of this operator is effective. Proposition 8.2 shows however that there is no guarantee that its fixpoint can be reached after finitely many steps. Using the analogy between Petri nets and vector addition systems, we make use of a class of constraints over integer variables counting occurrences of atoms inside multisets (markings). Specifically, we use vectors of variables \( \mathbf{x} = \langle x_1, \ldots, x_n \rangle \), where variable \( x_i \) denotes the number of occurrences of \( a_i \in \Sigma \) in a given fact. We give the following definition.

Definition 8.10 (Occurrence Constraints) Let \( \mathcal{A} \) be a fact. We define the following class of constraints.

\[
\alpha_{\mathcal{A}}(\mathbf{x}, \mathbf{x}') \equiv \bigwedge_{i=1}^{n} x_i = x_i' + \mathcal{A}(a_i)
\]

\[
\rho_{\mathcal{A}}(\mathbf{x}, \mathbf{x}') \equiv \bigwedge_{i=1}^{n} x_i = x_i' - \mathcal{A}(a_i) \land x_i \geq 0
\]

\[
\phi_{\mathcal{A}} \equiv \bigwedge_{i=1}^{n} x_i = \mathcal{A}(a_i)
\]

\[
\phi_{[\mathcal{A}]} \equiv \bigwedge_{i=1}^{n} x_i \geq \mathcal{A}(a_i)
\]

\[
\text{Rem}(\varphi, \mathcal{A}) = \exists \mathbf{x}'. (\varphi[\mathbf{x}'/\mathbf{x}] \land \rho_{\mathcal{A}}(\mathbf{x}, \mathbf{x}'))
\]

\[
\text{Add}(\varphi, \mathcal{A}) = \exists \mathbf{x}'. (\varphi[\mathbf{x}'/\mathbf{x}] \land \alpha_{\mathcal{A}}(\mathbf{x}, \mathbf{x}'))
\]

\[
\varphi_1 \odot \varphi_2 = \exists \mathbf{x}'. \exists \mathbf{x}''. ((\varphi_1[\mathbf{x}'/\mathbf{x}] \land \varphi_2[\mathbf{x}''/\mathbf{x}] \land \mathbf{x} = \mathbf{x}' + \mathbf{x}'')
\]

Example 8.11 Let \( \Sigma = \{ a_1, a_2 \} \), and consider the multisets \( \mathcal{A} = \{ a_1, a_1, a_2 \} \), \( \mathcal{B} = \{ a_1, a_2 \} \). Then, \( \phi_{\mathcal{A}} \equiv x_1 = 2 \land x_2 = 1 \), \( \phi_{[\mathcal{A}]} \equiv x_1 \geq 2 \land x_2 \geq 1 \), \( \text{Rem}(\phi_{[\mathcal{A}]}, \mathcal{B}) \equiv \ldots \).
\[\exists x'_1, x'_2. (x'_1 \geq 2 \land x'_2 \geq 1 \land x_1 = x'_1 - 1 \land x_2 = x'_2 - 1 \land x_1 \geq 0 \land x_2 \geq 0).\] This latter constraint is equivalent to \(x_1 \geq 1 \land x_2 \geq 0\). We also have that \(\phi_{[A]} \circ \phi_B \equiv \exists x'_1, x'_2, x''_1, x''_2. (x'_1 \geq 2 \land x'_2 \geq 1 \land x''_1 = 1 \land x''_2 = 1 \land x_1 = x'_1 + x''_1 \land x_2 = x'_2 + x''_2).\) This constraint is equivalent to \(x_1 \geq 3 \land x_2 \geq 2\).

Using constraints of the form \(\phi_{[A]}\), for a fact \(A\), we can immediately recover the semantics of Section 7.2. All the operations on multisets involved in the definition of \(S_P\) (see Definition 7.14) can be expressed as operations over linear constraints. For instance, the ideal generated by the empty multiset \(\epsilon\) corresponds to the linear constraint \(\phi_{[\epsilon]}\); given the ideals \([A]\) and \([B]\), the ideal \([A \bullet B]\) is represented as the constraint \(\phi_{[A]} \land \phi_{[B]}\); \([B] \setminus [A]\) is represented as the constraint \(\text{Rem}(\phi_{[B]}, A)\). The constraint \(\rho_A\) models the removal of the occurrences of the propositional symbols in \(A\) from all elements of the denotation of \(B\). Similarly, \([B + A]\), for a given multiset \(A\), is represented as the constraint \(\text{Add}(\phi_{[B]}, A)\).

The introduction of the constant \(1\) breaks down Proposition 3.9. As a consequence, the abstraction based on ideals is not precise anymore. In order to give a semantics for \(\text{LO}_1\), we need to add a class of constraints for representing collections of multisets that are not upward-closed. Specifically, we need equality constraints of the form \(\phi_A\). The operations over linear constraints discussed previously extend smoothly when adding this new class of equality constraints. In particular, given two constraints \(\varphi_1\) and \(\varphi_2\), their conjunction \(\varphi_1 \land \varphi_2\) still plays the role that the operation \(\bullet\) (least upper bound of multisets) had in Definition 7.14, while \(\text{Rem}(\varphi, A)\), for a given constraint \(\varphi\) and a multiset \(A\), plays the role of multiset difference. The reader can compare Definition 7.14 with Definition 8.15.

For the class of linear integer constraints presented above, it is well-known that there are algorithms for checking satisfiability, entailment, and for variable elimination (see e.g. [BGP97]).

Based on these ideas, we can define a bottom-up evaluation procedure for \(\text{LO}_1\) programs via a symbolic operator \(S_P\). In the following we will use the notation \(\widehat{c}\), where \(c = \langle c_1, \ldots, c_n \rangle\) is a solution of a constraint \(\varphi\) (i.e., an assignment of natural numbers to the variables \(x\) which satisfies \(\varphi\)), to indicate the multiset over \(\Sigma = \{a_1, \ldots, a_n\}\) which contains \(c_i\) occurrences of every propositional symbol \(a_i\) (formally, \(\widehat{c} = c_1 \cdot \{a_1\} + \ldots + c_n \cdot \{a_n\}\)). Note that according to this notation, \(c\) is the unique solution of \(\phi_{\widehat{c}}\) (e.g. \(\langle 1, 0, 0, \ldots \rangle\) is the unique solution of \(\phi_{\{a_1\}}\)). We extend this definition to a set \(C\) of constraint solutions by \(\widehat{C} = \{\widehat{c} \mid c \in C\}\).

We then define the denotation of a given constraint \(\varphi\), written \([\varphi]\), as the set of multisets corresponding to solutions of \(\varphi\). Formally, we have the following definition.

**Definition 8.12 (Denotation of a Constraint)** Given a linear constraint \(\varphi\) over the integer variables \(\langle x_1, \ldots, x_n \rangle\), its denotation \([\varphi]\) is given by

\[\varphi = \{\widehat{c} \mid x = c \text{ satisfies } \varphi\}.

109
Two constraints \( \varphi \) and \( \psi \) are said to be equivalent, written \( \varphi \simeq \psi \), if and only if \([\varphi] = [\psi]\) (i.e., we identify constraints with the same set of solutions).

For the sake of simplicity, in the following we will identify a constraint with its equivalence class, i.e., we will simply write \( \varphi \) instead of \([\varphi] \simeq \). Let \( LC_\Sigma \) be the set of (equivalence classes of) linear constraints over the integer variables \( x = (x_1, \ldots, x_n) \) associated to the signature \( \Sigma = \{a_1, \ldots, a_n\} \). The operator \( SP \) is defined on constraint interpretations consisting of sets (disjunctions) of (equivalence classes of) linear constraints. We have the following definitions.

**Definition 8.13 (Constraint Interpretation)** We say that \( I \subseteq LC_\Sigma \) is a constraint interpretation. Constraint interpretations form a complete lattice \( \langle D, \subseteq \rangle \) with respect to set inclusion, where \( D = \mathcal{P}(LC_\Sigma) \).

**Definition 8.14 (Denotation of a Constraint Interpretation)** Given a constraint interpretation \( I \), its denotation \([I]\) is the interpretation given by

\[
[I] = \{[\varphi] \mid \varphi \in I\}.
\]

For brevity, we will define the semantics directly on interpretations consisting of the representative elements of the equivalence classes. Also, in the following definitions we assume that the conditions apply only when the constraints involved are satisfiable. First of all, we extend the definition of the satisfiability judgment using operations over constraints (see Definition 8.10) as follows.

**Definition 8.15 (Abstract Satisfiability Judgment)** Let \( P \) be a propositional \( \mathit{LO}_1 \) program and \( I \) an interpretation. The abstract satisfiability judgment \( \models \) is defined as follows:

\[
\begin{align*}
I \models \top, \Delta \models \phi_{\varphi[\varphi]}; \\
I \models 1, \Delta \models \phi_{\varphi}; \\
I \models A, \Delta \models \text{Rem}(\psi, A), \text{ for } \psi \in I; \\
I \models G_1 \& G_2, \Delta \models \varphi_1 \land \varphi_2 \text{ if } I \models G_1, \Delta \models \varphi_1, \ I \models G_2, \Delta \models \varphi_2; \\
I \models G_1 \otimes G_2, \Delta_1, \Delta_2 \models \varphi_1 \otimes \varphi_2 \text{ if } I \models G_1, \Delta_1 \models \varphi_1, \ I \models G_2, \Delta_2 \models \varphi_2; \\
I \models G_1 \boxdot G_2, \Delta \models \varphi \text{ if } I \models G_1, G_2, \Delta \models \varphi; \\
I \models \bot, \Delta \models \varphi \text{ if } I \models \Delta \models \varphi.
\end{align*}
\]

The connection between the satisfiability judgments \( \models \) and \( \models \) is clarified by the following lemma.
Lemma 8.16 For every interpretation $I$, context $\Delta$, and constraint $\varphi$,

i. if $I \models \Delta \triangleright \varphi$, then $[I] = \Delta \triangleright A$ for every $A \in [\varphi]$;

ii. if $[I] = \Delta \triangleright A$, then there exists $\varphi$ such that $I \models \Delta \triangleright \varphi$ and $A \in [\varphi]$.

Proof

i. By induction on the derivation of $I \models \Delta \triangleright \varphi$.

- If $I \models T, \Delta \triangleright \varphi$, then every $c$ (with $c_i \geq 0$) is solution of $\varphi \equiv \bigwedge_{i=1}^{n} x_i \geq 0$, and $[I] = T, \Delta \triangleright A'$ for every fact $A'$;

- If $I \models 1 \triangleright \varphi$, then $(0, \ldots, 0)$ is the only solution of $\varphi$, and $[I] = 1 \triangleright c$;

- If $I \models A \triangleright \varphi$ then there exists $\psi \in I$ s.t. $\varphi \equiv \exists x'.(\psi[x'/x] \land \rho_A(x, x'))$ is satisfiable. Then for every solution $c$ of $\varphi$ there exists a vector $c'$ s.t. $\psi[c'/x]$ is satisfiable and $c'_1 \geq A(a_1), c_1 = c'_1 - A(a_1), \ldots, c'_n \geq A(a_n), c_n = c'_n - A(a_n)$. From this we get that for $i = 1, \ldots, n$, $c_i' = c_i + A(a_i)$ is a solution for $\psi$, therefore $\hat{c} + A \in [\psi] \subseteq [I]$ so that we can conclude $[I] = A \triangleright \hat{c}$;

- If $I \models G_1 \& G_2, \Delta \triangleright \varphi$ then $\varphi \equiv \varphi_1 \& \varphi_2$ and $I \models G_1, \Delta \triangleright \varphi_1, I \models G_2, \Delta \triangleright \varphi_2$. By inductive hypothesis, $[I] = G_1, \Delta \triangleright \hat{c}_1$ and $[I] = G_2, \Delta \triangleright \hat{c}_2$ for every $\hat{c}_1$ and $\hat{c}_2$ solutions of $\varphi_1$ and $\varphi_2$, respectively. Thus $[I] = G_1 \& G_2, \Delta \triangleright \hat{c}$ for every $c$ which is solution of both $\varphi_1$ and $\varphi_2$, i.e., for every $c$ which is solution of $\varphi_1 \& \varphi_2$;

- If $I \models G_1 \otimes G_2, \Delta_1, \Delta_2 \triangleright \varphi$ then $\varphi = \varphi_1 \oplus \varphi_2$ and $I \models G_1, \Delta_1 \triangleright \varphi_1, I \models G_2, \Delta_2 \triangleright \varphi_2$. By inductive hypothesis, $[I] = G_1, \Delta_1 \triangleright \hat{c}_1$ and $[I] = G_2, \Delta_2 \triangleright \hat{c}_2$ for every $\hat{c}_1$ and $\hat{c}_2$ solutions of $\varphi_1$ and $\varphi_2$, respectively. Hence, we get that $[I] = G_1 \otimes G_2, \Delta_1, \Delta_2 \triangleright \hat{c}_1 + \hat{c}_2$ for every $\hat{c}_1$ and $\hat{c}_2$ solutions of $\varphi_1$ and $\varphi_2$, i.e.,

$[I] = G_1 \otimes G_2, \Delta_1, \Delta_2 \triangleright \hat{c}$ for every $c$ which is solution of $\varphi_1 \oplus \varphi_2$;

- the $\forall$-case and $\bot$-case follow by a straightforward application of the inductive hypothesis.

ii. By induction on the derivation of $[I] = \Delta \triangleright A$.

- $[I] = T, \Delta \triangleright \hat{c}$ for every $c$, and $I \models T, \Delta \triangleright \varphi$, where $\varphi \equiv x_1 \geq 0, \ldots, x_n \geq 0$, and every $c$ is solution of $\varphi$;

- If $[I] = 1 \triangleright \epsilon, \epsilon = (0, \ldots, 0)$, then $I \models 1 \triangleright \varphi$, where $\varphi \equiv x_1 = 0, \ldots, x_n = 0$, and $\langle 0, \ldots, 0 \rangle$ is solution of $\varphi$;

- If $[I] = A \triangleright \hat{c}$ then $\hat{c} + A \in [I]$, therefore there exists $\psi \in I$ s.t. $\hat{c} + A \in [\psi]$. Therefore, if $a$ is such that $\hat{a} = A$, we have that $\psi[c + a/x]$ is satisfiable, $c$ is solution of $\varphi \equiv \exists x'.(\psi[x'/x] \land \rho_A(x, x'))$ and $I \models A \triangleright \varphi$;
- if $[I] \models G_1 \& G_2, \Delta \triangleright \breve{c}$ then $[I] \models G_1, \Delta \triangleright \breve{c}$ and $[I] \models G_2, \Delta \triangleright \breve{c}$. By inductive hypothesis, there exist $\varphi_1$ and $\varphi_2$ such that $I \models G_1, \Delta \triangleright \varphi_1$ and $I \models G_2, \Delta \triangleright \varphi_2$, and $c$ is a solution of $\varphi_1$ and $\varphi_2$. Therefore $I \models G_1 \& G_2, \Delta \triangleright \varphi_1 \& \varphi_2$ and $c$ is a solution of $\varphi_1 \& \varphi_2$;

- if $[I] \models G_1 \otimes G_2, \Delta_1, \Delta_2 \triangleright \breve{c}$ then there exist $c_1$ and $c_2$ s.t. $c = c_1 + c_2$. $[I] \models G_1, \Delta_1 \triangleright c_1$ and $[I] \models G_2, \Delta_2 \triangleright c_2$. By inductive hypothesis, there exist $\varphi_1$ and $\varphi_2$ such that $I \models G_1, \Delta_1 \triangleright \varphi_1$ and $I \models G_2, \Delta_2 \triangleright \varphi_2$, $c_1$ is a solution of $\varphi_1$ and $c_2$ is a solution of $\varphi_2$. Therefore $I \models G_1 \otimes G_2, \Delta_1, \Delta_2 \triangleright \varphi_1 \otimes \varphi_2$ and $c$ is a solution of $\varphi_1 \otimes \varphi_2$;

- the $\exists$-case and $\bot$-case follow by a straightforward application of the inductive hypothesis.

\[ \square \]

The satisfiability judgment $\models$ also satisfies the following properties.

**Lemma 8.17** For every interpretation $I_1, I_2, \ldots$, context $\Delta$, and constraint $\varphi$,

\begin{enumerate}
  \item if $I_1 \subseteq I_2$ and $I_1 \models \Delta \triangleright \varphi$, then $I_2 \models \Delta \triangleright \varphi$;
  \item if $I_1 \subseteq I_2 \subseteq \ldots$ and $\bigcup_{i=1}^\infty I_i \models \Delta \triangleright \varphi$ then there exists $k \in \mathbb{N}$ s.t. $I_k \models \Delta \triangleright \varphi$.
\end{enumerate}

**Proof**

\begin{enumerate}
  \item By simple induction on the derivation of $I_1 \models \Delta \triangleright \varphi$.
  \item By induction on the derivation of $\bigcup_{i=1}^\infty I_i \models \Delta \triangleright \varphi$.
    \begin{enumerate}
      \item The $\top$ and $1$-cases follow by definition;
      \item if $\bigcup_{i=1}^\infty I_i \models \mathcal{A} \triangleright \varphi$, then there exists $\psi \in \bigcup_{i=1}^\infty I_i$ s.t. $\varphi \equiv \exists x'.(\psi[x'/x]\land \rho_A(x, x'))$ is satisfiable. Then there exists $k$ s.t. $\psi \in I_k$ and $I_k \models \mathcal{A} \triangleright \varphi$;
      \item if $\bigcup_{i=1}^\infty I_i \models G_1 \& G_2, \Delta \triangleright \varphi$, then $\varphi \equiv \varphi_1 \& \varphi_2$, and, by inductive hypothesis, there exist $k_1$ and $k_2$ s.t. $I_{k_1} \models G_1, \Delta \triangleright \varphi_1$ and $I_{k_2} \models G_2, \Delta \triangleright \varphi_2$. Then, for $k = \max\{k_1, k_2\}$, we have, by $i$, $I_k \models G_1, \Delta \triangleright \varphi_1$ and $I_k \models G_2, \Delta \triangleright \varphi_2$, therefore $I_k \models G_1 \& G_2, \Delta \triangleright \varphi_1 \& \varphi_2$;
      \item the $\otimes$-case is analogous to the $\&$-case above;
      \item the $\exists$-case and $\bot$-case follow by a straightforward application of the inductive hypothesis.
    \end{enumerate}
\end{enumerate}
We are now ready to define the extended operator $S_P$.

**Definition 8.18 (Abstract Fixpoint Operator $S_P$)** Given a propositional LO$_1$ program $P$ and an interpretation $I$, the abstract fixpoint operator $S_P$ is defined as follows:

$$S_P(I) = \{ \text{Add}(\psi, \hat{H}) \mid H \vdash G \in P, \ I \models G \triangleright \psi \}$$

We recall that the constraint $\text{Add}(\cdot, \cdot)$ is defined in Definition 8.10. The following property shows that $S_P$ is sound and complete w.r.t. $T_P$.

**Proposition 8.19** For every propositional LO$_1$ program $P$ and interpretation $I$, $[S_P(I)] = T_P([I])$.

**Proof** Let $\hat{c} \in [S_P(I)]$, then there exist $\varphi \in S_P(I)$ and a clause $H \vdash G \in P$, s.t. $c$ is solution of $\varphi$, $\varphi \equiv \exists x'.(\psi[x'/x] \land \alpha_H(x, x'))$ and $I \models G \triangleright \psi$. Then there exists $c'$ solution of $\psi$ s.t. $\hat{c} = \hat{c}' + \hat{H}$, and, by Lemma 8.16, $[I] \models G \triangleright \hat{c}'$. Therefore, by definition of $T_P$, $\hat{c} = \hat{c}' + \hat{H} \in T_P([I])$.

Vice versa, let $\hat{c} \in T_P([I])$, then there exists $H \vdash G \in P$ s.t. $[I] \models G \triangleright \mathcal{A}$ and $\hat{c} = \hat{H} + \mathcal{A}$. By Lemma 8.16, there exists $\psi$ s.t. $I \models G \triangleright \psi$ and $\mathcal{A} \in [\psi]$. Therefore $\varphi \equiv \exists x'.(\psi[x'/x] \land \alpha_H(x, x')) \in S_P(I)$, and $\hat{c} = \hat{H} + \mathcal{A} \in [\varphi] \subseteq [S_P(I)]$. $\Box$

The abstract fixpoint operator $S_P$ satisfies the following property.

**Proposition 8.20 (Monotonicity and Continuity)** For every propositional LO$_1$ program $P$, the abstract fixpoint operator $S_P$ is monotonic and continuous over the lattice $(\mathcal{D}, \subseteq)$.

**Proof** Monotonicity. Immediate from $S_P$ definition and Lemma 8.17 i.

**Continuity.** Let $I_1 \subseteq I_2 \subseteq \ldots$, be a chain of interpretations. We show that $S_P(\bigcup_{i=1}^{\infty} I_i) \subseteq \bigcup_{i=1}^{\infty} S_P(I_i)$. If $\varphi \in S_P(\bigcup_{i=1}^{\infty} I_i)$, by definition there exists a clause $H \vdash G \in P$ s.t. $\bigcup_{i=1}^{\infty} I_i \models G \triangleright \psi$ and $\varphi \equiv \exists x'.(\psi[x'/x] \land \alpha_H(x, x'))$ is satisfiable. By Lemma 8.17 ii, there exists $k$ s.t. $I_k \models G \triangleright \psi$. This implies that $\varphi \in S_P(I_k)$, i.e., $\varphi \in \bigcup_{i=1}^{\infty} S_P(I_i)$. $\Box$

**Corollary 8.21** $[\text{lfp}(S_P)] = \text{lfp}(T_P)$.

Now, let $\mathcal{F}_{\text{syn}}(P) = \text{lfp}(S_P)$, then we have the following main theorem, which shows that $S_P$ can be used (without termination guarantee) to compute symbolically the set of logical consequences of an LO$_1$ program.
Theorem 8.22 (Soundness and Completeness) For every propositional LO₁ program $P$, $O(P) = F(P) = [\text{syn}(P)]$.

Proof By Theorem 8.9 and Corollary 8.21. □

8.3 Bottom-Up Evaluation for LO₁

Using a constraint-based representation for LO₁ provable multisets, we have reduced the problem of computing $O(P)$ to the problem of computing the reachable states of a system with integer variables. As shown by Proposition 8.2, the termination of the algorithm is not guaranteed a priori. In this respect, Theorem 7.24 gives us sufficient conditions that ensure its termination. The abstract fixpoint operator $S_P$ of Section 8.2 is defined over the lattice $(\mathcal{P}(LC_\Sigma), \subseteq)$, with set inclusion being the partial order relation and set union the least upper bound operator. When we come to a concrete implementation of $S_P$, it is worth considering a weaker ordering relation between interpretations, namely pointwise subsumption. By analogy with the multiset inclusion relation $\subseteq$, let $\preceq^c$ be the partial order between (equivalence classes of) constraints given by $\varphi \preceq^c \psi$ if and only if $[[\psi]] \subseteq [[\varphi]]$. In accordance with the definition of pointwise subsumption given in Section 6.1.2, we say that an interpretation $I$ is subsumed by an interpretation $J$, written $I \sqsubseteq J$, if and only if for every $\varphi \in I$ there exists $\psi \in J$ such that $\psi \preceq^c \varphi$.

As we do not need to distinguish between different interpretations representing the same set of solutions, we can consider interpretations $I$ and $J$ to be equivalent in case both $I \sqsubseteq J$ and $J \sqsubseteq I$ hold. In this way, we get a lattice of interpretations ordered by $\subseteq$ and such that the least upper bound operator is still set union. This construction is the natural extension of the one of Section 7.2. Actually, when we limit ourselves to considering LO programs (i.e., without the constant 1) it turns out that we need only consider constraints of the form $x \geq c$, which can be abstracted away by considering the upward closure of $\hat{c}$, as we did in Section 7.2. The reader can note that the $\preceq^c$ relation defined above for constraints is an extension of the multiset inclusion relation we used in Section 7.2.

The construction based on $\subseteq$ can be directly incorporated into the semantic framework presented in Section 8.2, where, for the sake of simplicity, we have adopted an approach based on $\subseteq$. Of course, relation $\sqsubseteq$ is stronger than $\subseteq$, therefore a computation based on $\subseteq$ is correct and it terminates every time a computation based on $\sqsubseteq$ does. However, the converse does not always hold, and this is why a concrete algorithm for computing the least fixpoint of $S_P$ relies on subsumption. Let us see an example.

Example 8.23 We calculate the fixpoint semantics for the following LO₁ program made
we get redundant constraints (i.e., constraints subsumed).

Now, we can compute \( S_P^\uparrow_1 \) up of six clauses:

1. \( a \leftarrow 1 \)
2. \( a \not\leftarrow b \leftarrow \top \)
3. \( c \not\leftarrow c \leftarrow \top \)
4. \( b \not\leftarrow b \leftarrow a \)
5. \( a \leftarrow b \)
6. \( c \leftarrow a \& b \)

Let \( \Sigma = \{ a, b, c \} \) and consider constraints over the variables \( x = (x_a, x_b, x_c) \). We have that \( S_P^\uparrow_0 = \emptyset \models 1 \leftarrow x_a = 0 \land x_b = 0 \land x_c = 0 \), therefore, by the first clause, \( \varphi \in S_P^\uparrow_1 \), where \( \varphi = \exists \overline{x}', (x_a' = 0 \land x_b' = 0 \land x_c' = 0 \land x_a = x_a' + 1 \land x_b = x_b' \land x_c = x_c') \), which is equivalent to \( x_a = 1 \land x_b = 0 \land x_c = 0 \). From now on, we leave to the reader the details concerning equivalence of constraints. By reasoning in a similar way, using clauses 2 and 3 we calculate \( S_P^\uparrow_1 \) (see Figure 8.2).

We now compute \( S_P^\uparrow_2 \). By 4, as \( S_P^\uparrow_1 \models a \leftarrow x_a = 0 \land x_b = 0 \land x_c = 0 \), we get \( x_a = 0 \land x_b = 2 \land x_c = 0 \), and, similarly, we get \( x_a \geq 0 \land x_b \geq 3 \land x_c \geq 0 \). By 5, we have \( x_a \geq 2 \land x_b \geq 0 \land x_c \geq 0 \), while clause 6 is not (yet) applicable. Therefore, modulo redundant constraints (i.e., constraints subsumed by the already calculated ones) the value of \( S_P^\uparrow_2 \) is given in Figure 8.2.

Now, we can compute \( S_P^\uparrow_3 \). By 4 and \( x_a \geq 2 \land x_b \geq 0 \land x_c \geq 0 \in S_P^\uparrow_2 \) we get \( x_a \geq 1 \land x_b \geq 2 \land x_c \geq 0 \), which is subsumed by \( x_a \geq 1 \land x_b \geq 1 \land x_c \geq 0 \). By 5 and \( x_a = 0 \land x_b = 2 \land x_c = 0 \), we get \( x_a = 1 \land x_b = 1 \land x_c = 0 \), subsumed by \( x_a \geq 1 \land x_b \geq 1 \land x_c \geq 0 \). Similarly, by 5 and \( x_a \geq 0 \land x_b \geq 3 \land x_c \geq 0 \) we get redundant information. By 6, from \( x_a \geq 1 \land x_b \geq 1 \land x_c \geq 0 \) and \( x_a = 0 \land x_b = 2 \land x_c = 0 \) we get \( x_a = 0 \land x_b = 1 \land x_c = 1 \), from \( x_a \geq 1 \land x_b \geq 1 \land x_c \geq 0 \) and \( x_a = 0 \land x_b \geq 3 \land x_c \geq 0 \) we get \( x_a \geq 0 \land x_b \geq 2 \land x_c \geq 1 \), and finally from \( x_a \geq 2 \land x_b \geq 0 \land x_c \geq 0 \) and \( x_a \geq 1 \land x_b \geq 1 \land x_c \geq 0 \) we have \( x_a \geq 1 \land x_b \geq 0 \land x_c \geq 1 \). The reader can verify that
no additional provable multisets can be obtained. It is somewhat tedious, but in no way
difficult, to verify that clause 6 yields only redundant information when applied to every
possible couple of constraints in \( S^P_{↑3} \). We have then \( S^P_{↑4} = S^P_{↑3} = \mathcal{F}_{sym}(P) \), so that in this
particular case we achieve termination. We can reformulate the operational semantics of
\( P \) using the more suggestive multiset notation (we recall that \([ A ] = \{ B \mid A \preceq B \} \), where
\( \preceq \) is multiset inclusion):

\[
F(P) = \{ \{a\}, \{b, b\}, \{b, c\} \} \cup \{ \{a, b\}, \{c, c\}, \{b, b, b\}, \{a, a\}, \{b, b, c\}, \{a, c\} \}.
\]

\( \square \)

**Example 8.24** Let us consider the LO₁ program \( P \) of Example 3.13, which is given below.

1. \( c \not\prec c \circ \top \)
2. \( a \not\prec b \circ \bot \)
3. \( b \prec a \otimes c \)

Let \( \Sigma = \{a, b, c\} \) and consider constraints over the variables \( \mathbf{x} = (x_a, x_b, x_c) \). By axioms 1 and 2 we get that

\[
S^P_{↑1} = \{ x_a \geq 0 \land x_b \geq 0 \land x_c \geq 2, \ x_a = 1 \land x_b = 1 \land x_c = 0 \}.
\]

Now we can apply clause 3. We have that \( S^P_{↑1} \models a \bowtie x_a = 0 \land x_b = 1 \land x_c = 0 \), and also
\( S^P_{↑2} \models a \otimes c \bowtie x_a \geq 0 \land x_b \geq 0 \land x_c \geq 1 \), therefore by the \( \otimes \)-case of \( \models \) definition we have \( S^P_{↑1} \models a \otimes c \bowtie x_a \geq 0 \land x_b \geq 1 \land x_c \geq 1 \). By definition of \( S^P \), the constraint \( x_a \geq 0 \land x_b \geq 2 \land x_c \geq 1 \) belongs to \( S^P_{↑2} \). There are no other elements except for redundant ones, therefore we can assume

\[
S^P_{↑2} = S^P_{↑1} \cup \{ x_a \geq 0 \land x_b \geq 2 \land x_c \geq 1 \}.
\]

Now, we can still apply clause 3. We have that \( S^P_{↑2} \models a \bowtie x_a = 0 \land x_b = 2 \land x_c \geq 0 \). By combining this with \( S^P_{↑2} \models a \otimes c \bowtie x_a \geq 0 \land x_b \geq 2 \land x_c \geq 0 \), we get the following judgment: \( S^P_{↑2} \models a \otimes c \bowtie x_a \geq 0 \land x_b \geq 3 \land x_c \geq 0 \). The new element is therefore \( x_a \geq 0 \land x_b \geq 4 \land x_c \geq 0 \), and we can assume

\[
S^P_{↑3} = S^P_{↑2} \cup \{ x_a \geq 0 \land x_b \geq 4 \land x_c \geq 0 \}.
\]

As the reader can verify, \( S^P_{↑4} = S^P_{↑3} = \mathcal{F}_{sym}(P) \). Therefore, using the multiset notation we have that

\[
F(P) = \{ \{a, b\} \} \cup \{ \{c, c\}, \{b, b, c\}, \{b, b, b\} \}.
\]

\( \square \)
8.4 An Example: Readers/Writers

We conclude the chapter with an informal discussion about the connection between linear logic specifications and the theory of broadcast protocols presented in Section 4.4.2. In particular, we will show how the readers/writers example of Section 4.4.1 can be encoded in the fragment \( \text{LO}_1 \). The idea, explained below, is to exploit the capability of counting resources given by the constant \( 1 \), together with the operational semantics of the additive conjunction \( \& \), to implement the invalidation phase of the above protocol.

The readers/writers protocol presented in Section 4.4.1 is an abstraction of consistency protocols used for maintaining coherence in distributed database systems. We have a set of identical processes which compete for a shared resource. Processes can be reading, writing, or otherwise be idle. Processes are allowed to concurrently read the given resource, while writing must be an exclusive operation. This protocol can be encoded by introducing a central monitor which serializes the accesses to the shared resource. The monitor must guarantee mutual exclusion between agents with read and write access, and between processes with write access.

The linear logic encoding of the above protocol is as follows. First of all, let us discuss the initialization phase. As usual, we introduce a propositional symbol \( \text{init} \) to load the initial configuration, and the following LO clauses:

1. \( \text{init} \rightarrow \text{init} \& \text{idle} \)
2. \( \text{init} \rightarrow \text{monitor} \)

The first clause creates an arbitrary number of idle processes, which can compete for the given shared resource. The second clause terminates the initialization phase by creating the central monitor. Note that the number of initial processes is a parameter of the protocol.

Idle processes can non-deterministically request to upgrade their rights by sending a request to the monitor. In response to such a request, the monitor (whose task is to serialize the requests) enters a state in which it starts sending invalidation messages to the other agents. The encoding is as follows.

3. \( \text{idle} \& \text{monitor} \rightarrow \text{read req} \& \text{read inv} \)
4. \( \text{idle} \& \text{monitor} \rightarrow \text{write req} \& \text{write inv} \)

Clause 3 and 4 deal with the case of idle processes willing to get, respectively, read access or write access. In response to the corresponding requests, named \( \text{read req} \) and \( \text{write req} \),
the monitor enters an invalidation state called read\_inv or write\_inv. Note that after a given request has been received by the monitor, no other requests can be accepted until the first one has been processed to completion. This is enforced by requiring processes to explicitly synchronize with the atom monitor, and removing it from the current state after backchaining on the above clauses.

In the following we use the propositional symbols reading and writing to denote processes in the corresponding state. A simple refinement of the above specification is given by the following clause.

\[5. \text{reading} \otimes \text{idle} \leftarrow \text{write}\_\text{req} \otimes \text{write}\_\text{inv}\]

Namely, we allow a reading process to ask the monitor for an upgrade to state writing. As for any write request, the monitor must enter a state of write invalidation.

Let us now discuss the core of the protocol, i.e., the invalidation phase. The coherence protocol works as follows. Every time a process requests a write access, a broadcast message must be sent to all reading and writing processes, requiring they downgrade their state to idle. The management of read requests is similar, with the difference that only writing processes must be invalidated. The broadcast can be simulated using an invalidation cycle, as follows.

\[6. \text{read}\_\text{inv} \otimes \text{writing} \leftarrow \text{read}\_\text{inv} \otimes \text{idle}\]
\[7. \text{write}\_\text{inv} \otimes \text{writing} \leftarrow \text{write}\_\text{inv} \otimes \text{idle}\]
\[8. \text{write}\_\text{inv} \otimes \text{reading} \leftarrow \text{write}\_\text{inv} \otimes \text{idle}\]

The following clauses can be used to test that all processes which are required to be invalidated have moved to state idle. Note the use of the & connective.

\[9. \text{read}\_\text{inv} \otimes \text{read}\_\text{req} \leftarrow \text{test}\_\text{read} \& (\text{monitor} \otimes \text{reading})\]
\[10. \text{write}\_\text{inv} \otimes \text{write}\_\text{req} \leftarrow \text{test}\_\text{write} \& (\text{monitor} \otimes \text{writing})\]

Let us discuss for instance clause 9. Using the connective &, we split the computation into two branches. In the first one, we use the atom test\_read to test that all writing processes have been invalidated. If this check is successful, in the other branch we grant read access to the requesting process, re-initializing the monitor, which will be ready to accept new requests afterwards. Clause 10 is similar. The test phase is implemented by the following clauses. Note the use of the constant 1 to count resources.

\[11. \text{test}\_\text{read} \otimes \text{idle} \leftarrow \text{test}\_\text{read}\]
\[12. \text{test}\_\text{write} \otimes \text{idle} \leftarrow \text{test}\_\text{write}\]
\[13. \text{test}\_\text{read} \otimes \text{reading} \leftarrow \text{test}\_\text{read}\]
\[14. \text{test}\_\text{read} \leftarrow 1\]
\[15. \text{test}\_\text{write} \leftarrow 1\]

Intuitively, the test succeeds only when the resources at the moment of its first invocation do not contain occurrences of atoms representing a forbidden access (i.e., reading or writing
atoms for a write request and writing atoms for a read request). This check on the global state is performed using two important aspects of LO$^1$: the use of the additive conjunction $\&$, which allows to create an auxiliary branch of the current computation in which the global context is copied to (clauses 9 and 10), and the capability of counting resources introduced with the constant 1 (clauses 14 and 15).

**Example 8.25** Let $P$ the LO$^1$ program consisting of clauses 1 through 15 above. Figure 8.3 presents an LO proof showing how the invalidation phase works. We have a configuration consisting of the central monitor, two reading processes and two idle processes. One of the two idle processes asks the monitor for being granted a write access. The central monitor accepts the request and starts the invalidation phase. As a result, both reading processes are downgraded to state idle. Once invalidation has been performed, a test is made to ensure that all processes have been invalidated. The test is performed in the left branch of the proof in Figure 8.3: idle processes are removed one by one and a check is made to ensure that there are no other processes left. The right branch of the proof in Figure 8.3 represents the main branch of the computation: the requesting process is finally granted the write access and the monitor is re-initialized. As a result, the final configuration consists of the central monitor, one writing process and three idle processes.

\[\begin{array}{l}
P \vdash 1_r \\
P \vdash \text{test\_write} \quad \text{bc}(15) \\
P \vdash \text{idle, test\_write} \quad \text{bc}(12) \\
P \vdash \text{idle, idle, test\_write} \quad \text{bc}(12) \\
P \vdash \text{idle, idle, idle, test\_write} \quad \text{bc}(12) \\
P \vdash \text{monitor, writing, idle, idle, idle} \\
P \vdash \text{monitor, writing, idle, idle, idle} \quad \text{bc}(10) \\
P \vdash \text{idle, idle, idle, test\_write \& (monitor \& writing)} \\
P \vdash \text{idle, idle, idle, write\_req, write\_inv} \quad \text{bc}(8) \\
P \vdash \text{reading, idle, idle, write\_req, write\_inv} \quad \text{bc}(8) \\
P \vdash \text{reading, reading, idle, write\_req, write\_inv} \quad \text{bc}(4) \\
P \vdash \text{monitor, reading, reading, idle, idle} \quad \text{bc}(4) \\
\end{array}\]
8.4.1 Verification of the Readers/Writers Protocol

The readers/writers protocol presented in Section 8.4 must satisfy a mutual exclusion property. Specifically, we wish to ensure that no read and write accesses are granted simultaneously, and that write operations are exclusive. In this section we show how the constraint bottom-up semantics for LO\textsubscript{1} discussed in Section 8.2 can be exploited to formally validate the readers/writers protocol with respect to the mutual exclusion property. As LO\textsubscript{1} specifications are currently not supported by our automatic verification tool (see Appendix A), we have evaluated the semantics for the above example by hand.

We fix the following conventions. Let \( \Sigma \) be the signature comprising the propositional symbols used in the specification of the readers/writers protocol of Section 8.4. In order to manage occurrence constraints, we associate to every symbol \( a \in \Sigma \) a variable \( x_a \) (e.g. we have \( x_{\text{idle}}, x_{\text{monitor}}, \) and so on). Let \( V \) the resulting (finite) set of variables. We introduce the following compact notation. Given a constraint \( \varphi \), we denote by \( \varphi \land \text{REST} \land 0 \), where \( \land \in \{\text{=}, \geq\} \), the constraint \( \varphi \land x_1 \land 0 \land \ldots \land x_k \land 0 \), where \( \{x_1, \ldots, x_k\} = V \setminus \text{FV}(\varphi) \).

First of all, we must specify the set of unsafe states of the readers/writers protocol. As usual, we introduce LO axioms to specify the minimality violations of the mutual exclusion property, as follows.

16. \( \text{reading} \not\rightarrow \text{writing} \lor \top \)
17. \( \text{writing} \not\rightarrow \text{writing} \lor \top \)

A configuration is unsafe if there are at least two processes whose access rights are in conflict.

We are now ready to compute the constraint semantics for the resulting specification, i.e., the LO\textsubscript{1} program consisting of clauses 1 through 17. Starting from the axioms (i.e., clauses 14 through 17), we compute the interpretation made up of the following constraints:

\[
\begin{align*}
\varphi_1 & \cdot \ x_{\text{test}_\text{read}} = 1 \land \text{REST} = 0 \\
\varphi_2 & \cdot \ x_{\text{test}_\text{write}} = 1 \land \text{REST} = 0 \\
\varphi_3 & \cdot \ x_{\text{reading}} \geq 1 \land x_{\text{writing}} \geq 1 \land \text{REST} \geq 0 \\
\varphi_4 & \cdot \ x_{\text{writing}} \geq 2 \land \text{REST} \geq 0
\end{align*}
\]

The remaining program clauses can now be applied to the above constraint interpretation. However, we can immediately notice that the fixpoint computation is not terminating. In fact, consider for instance clause 11, i.e., \( \text{test}_\text{read} \not\rightarrow \text{idle} \lor \top \) - \( \text{test}_\text{read} \). By repeatedly applying this clause to the constraint \( x_{\text{test}_\text{read}} = 1 \land \text{REST} = 0 \), we get the new constraints \( x_{\text{test}_\text{read}} = 1 \land x_{\text{idle}} = 1 \land \text{REST} = 0, x_{\text{test}_\text{read}} = 1 \land x_{\text{idle}} = 2 \land \text{REST} = 0 \), and so on. In other words, clause 11 can introduce an arbitrary number of \( \text{idle} \) atoms into any configuration containing a \( \text{test}_\text{read} \) atom. Similarly, clause 12 can introduce \( \text{idle} \) atoms into a configuration containing a \( \text{test}_\text{write} \) atom, and, finally, clause 13 can introduce \( \text{reading} \) atoms into a configuration containing a \( \text{test}_\text{read} \) atom.
\[ \begin{align*}
\varphi'_1 & : x_{\text{test} \text{read}} = 1 \land x_{\text{reading}} \geq 0 \land x_{\text{idle}} \geq 0 \land \text{REST} = 0 \\
\varphi'_2 & : x_{\text{test} \text{write}} = 1 \land x_{\text{idle}} \geq 0 \land \text{REST} = 0 \\
\varphi_3 & : x_{\text{reading}} \geq 1 \land x_{\text{writing}} \geq 1 \land \text{REST} \geq 0 \\
\varphi_4 & : x_{\text{writing}} \geq 2 \land \text{REST} \geq 0
\end{align*} \]

Figure 8.4: Fixpoint computed using invariant strengthening for the readers/writers protocol

A way to avoid non-termination of the fixpoint computation is to exploit the methodology of invariant strengthening described in Section 6.1.2.1. To this aim, consider the following constraints:

\[ \begin{align*}
\varphi'_1 & : x_{\text{test} \text{read}} = 1 \land x_{\text{reading}} \geq 0 \land x_{\text{idle}} \geq 0 \land \text{REST} = 0 \\
\varphi'_2 & : x_{\text{test} \text{write}} = 1 \land x_{\text{idle}} \geq 0 \land \text{REST} = 0 \\
\varphi_3 & : x_{\text{reading}} \geq 1 \land x_{\text{writing}} \geq 1 \land \text{REST} \geq 0 \\
\varphi_4 & : x_{\text{writing}} \geq 2 \land \text{REST} \geq 0
\end{align*} \]

These constraints formalize the intuitive idea explained above. For instance, the first one states that any configuration with one \textit{test read} atom and any number of \textit{reading} and \textit{idle} atoms (and nothing else) is reachable. Similarly for the second constraint. In other words, we have performed a widening operation by transforming a chain of constraints like \( x = 0, \) \( x = 1, \) \( x = 2, \) and so on, into a constraint like \( x \geq 0. \)

Consider the interpretation shown in Figure 8.4. This interpretation strictly includes the interpretation computed at the first step (i.e., without invariant strengthening). By Proposition 6.15, we are allowed to start the evaluation of the bottom-up semantics from the set of constraints in Figure 8.4. If we can prove that the modified problem is unsatisfiable (i.e., the atom \textit{init} is not backward reachable) then Proposition 6.15 guarantees that also the original problem is unsatisfiable. Note that the intuition we used to compute the modified constraints \( \varphi'_1 \) and \( \varphi'_2 \) does not matter at this level: in order for the methodology of invariant strengthening to be sound, we only need to ensure that the set of unsafe states we start the computation from is larger than the original one (and this is evident for the interpretation of Figure 8.4).

It turns out that the interpretation shown in Figure 8.4 is actually the fixpoint yielded by the bottom-up computation. In order to verify this, we need to ensure that nothing else can be produced by applying clauses 1 through 13. We have the following:

- clauses 1 through 8 are not applicable to constraints \( \varphi'_1 \) and \( \varphi'_2 \) (e.g. the body of clause 1 contains an atom \textit{init}, whereas \( x_{\text{init}} = 0 \) in \( \varphi'_1 \) and \( \varphi'_2 \)), whereas \( \varphi_3 \) and \( \varphi_4 \) are closed w.r.t. applications of clauses 1 through 8;

- clause 9 is not applicable. In fact, the semantics of \& requires to find a common context in which the two conjuncts are both provable. However, this is not possible for the following reasons. The second conjunct, \textit{monitor} \( \notin \text{reading}, \) is only satisfiable
using \( \varphi_3 \) and \( \varphi_4 \), with an output context containing at least one writing process (i.e., \( x_{\text{writing}} \geq 1 \)). The first conjunct, \( \text{test}_r \), is satisfiable with \( \varphi'_1 \), but with an output context containing zero writing processes (i.e., \( x_{\text{writing}} = 0 \)); also, the first conjunct is not satisfiable using \( \varphi'_2 \); finally, using \( \varphi_3 \) and \( \varphi_4 \) to satisfy the first conjunct we get \( \varphi_3 \) and \( \varphi_4 \), therefore nothing new is computed;

- clause 10 is not applicable (we can use a similar argument as for clause 9; note that the second conjunct, \( \text{monitor} \land \varphi_{\text{writing}} \), is only satisfiable with an output context such that either \( x_{\text{reading}} \geq 1 \) or \( x_{\text{writing}} \geq 1 \), whereas the first conjunct, \( \text{test}_w \) is satisfiable using \( \varphi'_2 \) with an output context such that \( x_{\text{reading}} = 0 \) and \( x_{\text{writing}} = 0 \);

- every constraint in Figure 8.4 is clearly closed with respect to applications of clauses 11 through 13.

We can conclude that the mutual exclusion property holds for the readers/writers protocol presented in Section 8.4, independently of the number of initial processes.

### 8.5 Related Work

In this section we have extended the bottom-up semantics for propositional LO given in Chapter 7, by considering an extension admitting the constant 1 (and the multiplicative conjunction) in goals. Evaluating the resulting semantics amounts to computing the reachability set for Petri nets, and is therefore undecidable [EN94].

The model of parameterized broadcast protocols has been introduced in [EN98], where a technique extending the Karp and Miller’s coverability graph construction for Petri nets [KM69] is proposed, and used to verify some safety properties for an invalidation-based cache coherence problem. However, as shown in [EFM99], the procedure proposed in [EN98] may not terminate (even for protocols with only broadcast moves), whereas the model-checking problem for safety properties for broadcast protocols is decidable. In particular, in [EFM99] it is shown that a backward verification procedure, along the lines of [ACJT96], can be used to check safety properties for broadcast protocols with termination guarantee. A similar result, given in [FS01], shows that the covering problem for Petri nets with transfer arcs is decidable. In [EFM99], it is also proved that the model-checking problem for liveness properties for broadcast protocols is undecidable. Finally, we mention that in [DEP99] efficient data structures for implementing constraint systems for broadcast protocols are studied.

Concerning our work, and in particular the broadcast protocol encoding informally presented in Section 8.4, we note that in general our verification algorithm in not guaranteed
to terminate (as stated in Proposition 8.2. Clearly, the constant 1 can be used to implement more powerful computational mechanisms than transfer arcs. However, as informally shown in Section 8.4.1, termination can be enforced by using suitable acceleration operators (see e.g. [JN00, PS00]). We will address this point as part of our future work.

Finally, we mention that in [AP91a] a different computational mechanism for LO programs, involving broadcast communication, is considered. This mechanism operates at the meta-level and is an alternative to the classical interpretation of logic programs involving the concepts of shared logic variable, unification and computed answer substitution. The broadcast mechanism operates as follows. Every branch of a proof tree is seen as a different object. Proof construction can be seen as a bidirectional process, which, starting from a partially defined initial node, either extends a branch of the proof tree, or furtherly specifies the initial node. Partially specified nodes can be instantiated every time the backchaining rule (method activation) is fired, and the relevant bindings are propagated backwards to the root of the proof tree, and, consequently, to the other branches.

---

**Summary of the Chapter.** In this chapter we have discussed a fragment of propositional linear logic consisting of the language LO presented in the previous chapter enriched with the multiplicative conjunction and the constant 1. We have extended the evaluation algorithm for computing the bottom-up semantics to this fragment. The semantics is based on a class of meta-constraints which symbolically represent provable multisets by counting resource occurrences. Though evaluation is in general non-terminating, the semantics is still effective and complete with respect to the operational semantics. We have shown that the greater expressive power of this logic can be used to simulate broadcast primitives.

In the next chapter we will extend the language LO in another direction, namely we will discuss a first-order formulation of LO with constraints. Constraints are an elegant and convenient way to enrich logic languages with capabilities for reasoning on specialized domains. We will isolate interesting fragments for which evaluation of the bottom-up semantics is guaranteed to terminate.
Chapter 9

Reasoning on Specialized Domains: LO with Constraints

In this chapter we will extend the semantics presented in Chapter 7 by considering a first-order formulation of LO. In particular, we will consider program clauses enriched with constraints, as a means to reason on specialized domains. As in traditional constraint programming [JM94], reasoning on heterogeneous domains (e.g. integer or real numbers, strings and so on) is made easier by keeping the core specification logic separate from the logic dealing with the particular domain under consideration. This solution is typically more flexible because reasoning on different domains can be delegated to specialized constraint solvers.

In this chapter we will extend the construction of Chapter 7, which relates bottom-up evaluation for LO programs with verification of concurrent systems specified as Petri nets. In particular, using the concept of constraint we will be able to address the connection between linear logic and coloured Petri nets (see Section 5.3). Our construction will provide an assertional language (in the sense of [KMM$^+$97b]) to symbolically represent infinite collections of states for systems parametric in several dimensions. The notion of constraint is central to this construction. In fact, our approach is based on a combination of constraints and multiset rewriting (provided by the underlying fragment of linear logic). On the one hand, multiset rewriting allows us to locally specify the behaviour of most concurrent systems in a natural way. On the other hand, annotating multiset rewrite rules with constraints allows us to finitely and concisely (e.g. without the need of explicit axioms for arithmetic operations) represent transition relations. Building upon these ideas, we will introduce the notion of constrained multiset, which can be seen as a symbolic representation for upward-closed sets of markings of coloured Petri nets. Specifically, constraints will be the technical device used to represent data attached to processes. Similarly to Chapter 7, our construction will make use of a fixpoint operator to symbolically compute
the predecessor operator on interpretations consisting of sets of constrained multisets. The resulting bottom-up evaluation algorithm, according to the connection between provability and reachability (see Section 6.3), can be viewed as an alternative to the so-called occurrence-graph construction for coloured Petri nets [Jen97].

Technically, as in Chapter 7, we will define a bottom-up procedure to compute all goal formulas which are provable from a given program. Thanks to the use of constraint solvers, this semantics will be effective and complete with respect to the operational semantics. Furthermore, we will isolate a fragment of first-order LO, enriched with constraints, for which evaluation of the bottom-up semantics is guaranteed to terminate. We will exploit this result to prove mutual exclusion for a parameterized formulation of the so-called ticket protocol. As usual, for the sake of simplicity we re-use some notations like, e.g., the ones for fixpoint operators and judgments.

Some preliminary results concerning the contents of this chapter appeared in [BDM01a, BD02].

## 9.1 Enriching LO With Constraints

In this section we define the language LO enriched with constraints. First of all, we need to define the notion of constraint system.

### 9.1.1 Constraint Systems

**Definition 9.1 (Constraint System)** A constraint system is a tuple $\mathcal{C} = \langle \Sigma, \mathcal{V}, \mathcal{L}, \mathcal{D}, \text{Sol}, \sqsubseteq \rangle$ where:

- $\Sigma$ is a signature with predicates;
- $\mathcal{V}$ is a denumerable set of variables;
- $\mathcal{L}$ is a first-order language over $\Sigma$ and $\mathcal{V}$ (the assertional language) defining a set of formulas (the constraints), closed with respect to variable renaming, existential quantification and conjunction, and allowing equalities between variables;
- $\mathcal{D}$ is a possibly infinite set (the interpretation domain);
- $\text{Sol}(\varphi)$ is a set of mappings $\mathcal{V} \rightarrow \mathcal{D}$ (the set of solutions of a constraint $\varphi \in \mathcal{L}$) that preserves the usual semantics of equalities, $\wedge$ and $\exists$ (intersection and projection of the solutions);
vi. $\sqsubseteq^c$ is a relation such that $\varphi \sqsubseteq^c \psi$ implies $\text{Sol}(\varphi) \subseteq \text{Sol}(\psi)$ (the entailment relation: we say that $\varphi$ entails $\psi$).

We assume that $\mathcal{L}$ contains constraints, denoted true and false, which are identically true and identically false in $\mathcal{D}$.

By analogy with constraint programming, further requirements on constraint systems, like solution compactness [JL87], can be imposed. We refer to [JL87, Mah92] for a discussion.

In the following we will refer to a generic mapping $\mathcal{V} \rightarrow \mathcal{D}$ as an evaluation for the variables in $\mathcal{V}$ into $\mathcal{D}$. We use the notation $\langle x_1 \mapsto d_1, x_2 \mapsto d_2, \ldots \rangle$ to denote an evaluation mapping $x_1$ to $d_1$, $x_2$ to $d_2$, and so on, and the notation $\sigma|_x$ to denote the restriction of the evaluation $\sigma$ to the variables $x$. We also say that a constraint $\varphi$ is satisfiable if $\text{Sol}(\varphi) \neq \emptyset$.

We conclude this section showing some examples of constraint systems which we will need later on in this chapter.

Definition 9.2 (The Herbrand Constraint System HC) Let $\Sigma$ be a signature and $\mathcal{V}$ a denumerable set of variables. We define the Herbrand constraint system $HC = \langle \Sigma, \mathcal{V}, \mathcal{L}, \mathcal{D}, \text{Sol}, \sqsubseteq^c \rangle$, where: $\mathcal{D}$ is the set of non-ground terms in $T^\mathcal{V}_\Sigma$, $\mathcal{L}$ contains equalities between non-ground terms in $T^\mathcal{V}_\Sigma$, equality is interpreted as unification between terms, $\text{Sol}$ maps constraints to substitutions (term unifiers) and $\tau \sqsubseteq^c \theta$ if $\theta \leq \tau$ (i.e., $\theta$ is more general than $\tau$).

Definition 9.3 (The Constraint System LC) The class of linear integer constraints (LC-constraints) consists of integer constraints of the form

$$\varphi ::= \varphi \land \varphi \mid a_1x_1 + \ldots + a_nx_n = a_{n+1} \mid a_1x_1 + \ldots + a_nx_n > a_{n+1} \mid \text{true} \mid \text{false}$$

where $a_i \in \mathbb{Z}$ for $i : 1, \ldots, n+1$. Given $\mathcal{D} = \mathbb{Z}$, the interpretation $\text{Sol}$ maps constraints into sets of variable evaluations from $\mathcal{V}$ to $\mathbb{Z}$, and $\sqsubseteq^c$ is the usual entailment relation between integer constraints.

Definition 9.4 (The Constraint System DC) The class of difference constraints (DC-constraints) is the subclass of linear integer constraints having the form

$$\varphi ::= \varphi \land \varphi \mid x = y + c \mid x > y + c \mid \text{true} \mid \text{false}$$

where $c \in \mathbb{Z}$, Given $\mathcal{D} = \mathbb{Z}$, the interpretation $\text{Sol}$ maps constraints into sets of variable evaluations from $\mathcal{V}$ to $\mathbb{Z}$, and $\sqsubseteq^c$ is the usual entailment relation for linear integer constraints.
Definition 9.5 (The Constraint System NC) The class of name constraints (NC-constraints) is the subclass of difference constraints having the form
\[ \varphi ::= \varphi \land \varphi \mid x = y \mid x > y \mid true \mid false \]
interpreted over \( \mathbb{Z} \) and ordered with respect to the entailment relation \( \sqsubseteq_c \) of linear integer constraints.

Definition 9.6 (The Constraint System EC) The class of equality constraints (EC-constraints) is the subclass of name constraints having the form
\[ \varphi ::= \varphi \land \varphi \mid x = y \mid true \mid false \]
interpreted over \( \mathbb{Z} \) and ordered with respect to the entailment relation \( \sqsubseteq_c \) of linear integer constraints.

For linear integer constraints (and thus DC-, NC- and EC-constraints) it is well-known that there are algorithms for checking satisfiability, entailment, and for variable elimination (see e.g. [BGP97]).

Example 9.7 Let \( \varphi \) be \( x > y \land x > z \), then \( \sigma = \langle x \mapsto 2, y \mapsto 1, z \mapsto 0, \ldots \rangle \in \text{Sol}(\varphi) \). Furthermore, \( \varphi \) is satisfiable, \( \varphi \sqsubseteq_c (x > y) \), and \( \exists y. \varphi \equiv (x > z) \). \( \square \)

We are now ready to define the language \( \text{LO}(\mathcal{C}) \).

9.1.2 The Language \( \text{LO}(\mathcal{C}) \)

In order to define the language \( \text{LO}(\mathcal{C}) \), we will make a simplifying assumption. Namely, we assume that \( \text{LO}(\mathcal{C}) \) programs are built over atomic formulas consisting of a predicate symbol applied to a list of distinct variables, instead of arbitrary terms. This assumption simplifies a lot the following presentation, without causing any loss of generality. In fact, arbitrary atomic formulas can be recovered by considering a constraint system in which equality is interpreted as unification on a first-order term language (see Definition 9.2). We give the following definitions.

Definition 9.8 (Atomic formulas) Let \( \Pi \) be a finite set of predicate symbols, and \( \mathcal{V} \) a denumerable set of variables. An atomic formula over \( \Pi \) and \( \mathcal{V} \) has the form \( p(x_1, \ldots, x_n) \) (with \( n \geq 0 \)), where \( p \in \Pi \), and \( x_1, \ldots, x_n \) are distinct variables in \( \mathcal{V} \).
Definition 9.9 (LO(\(C\)) programs) Let \(\Pi\) be a finite set of predicate symbols, and \(C = \langle \Sigma, \mathcal{V}, \mathcal{L}, \mathcal{D}, \text{Sol}, \sqsubseteq^c \rangle\) a constraint system. The sets of LO(\(C\)) goal formulas, head formulas, and clauses, over \(\Pi\) and \(C\), are defined by the following grammar

\[
\begin{align*}
G & ::= G \otimes G \mid G \& G \mid A \mid T \mid \bot \\
H & ::= A \otimes \ldots \otimes A \mid \bot \\
D & ::= \forall (H \leftarrow G \sqsubseteq \varphi) \mid D \& D
\end{align*}
\]

where \(A\) is an atomic formula over \(\Pi\) and \(\mathcal{V}\), and \(\varphi\) is a satisfiable constraint in \(\mathcal{L}\). We assume that all variables appearing in a \(G\)-formula, \(H\)-formula or \(D\)-formula are distinct from each other. An LO(\(C\)) program is a \(D\)-formula over \(\Pi\) and \(C\).

Constraints can be viewed as a convenient way to represent (infinite) sets of ground clauses. Therefore, a simple way to define the operational semantics of the language LO(\(C\)) is to re-formulate the notion of ground instance of a program given in Definition 3.7. In the rest of the chapter, we will use the notation \(\sigma(F)\), where \(F\) is any expression (e.g. an LO(\(C\)) clause) with variables in \(\mathcal{V}\) and \(\sigma\) is a mapping \(\mathcal{V} \rightarrow \mathcal{D}\), to denote the application of \(\sigma\) to \(F\).

Definition 9.10 (Ground Instances) Let \(\Pi\) be a finite set of predicate symbols, and \(C = \langle \Sigma, \mathcal{V}, \mathcal{L}, \mathcal{D}, \text{Sol}, \sqsubseteq^c \rangle\) a constraint system. Given an LO(\(C\)) program \(P\), the set of ground instances of \(P\), denoted \(\text{Gnd}(P)\), is defined as follows:

\[
\text{Gnd}(P) = \{\sigma(H \leftarrow G) \mid (H \leftarrow G \sqsubseteq \varphi) \in P \text{ and } \sigma \in \text{Sol}(\varphi)\}
\]

Example 9.11 Let \(\Pi = \{p, q, r, s\}\), and consider unification constraints over the term language generated by a constant symbol \(a\) and a function symbol \(f\). Let \(\mathcal{V}\) be a denumerable set of variables and \(x, y, \ldots \in \mathcal{V}\). Let \(C\) be the clause

\[
p(x) \otimes p(z) \leftarrow (q(x') \otimes r(y)) \& s(z') \sqsubseteq z = a \land x' = x \land z' = a.
\]

Then

\[
p(f(f(a))) \otimes p(a) \leftarrow (q(f(f(a))) \otimes r(f(a))) \& s(a) \in \text{Gnd}(C).
\]

Example 9.12 Let \(\Pi = \{p, q, r, s\}\), let \(\mathcal{V}\) be a denumerable set of variables and \(x, y, \ldots \in \mathcal{V}\). Let \(C\) be the LO(DC) clause

\[
p(x) \otimes q(y) \leftarrow q(z) \otimes r(w) \otimes s(w') \sqsubseteq z = x \land w > y \land w' = w + 1
\]

Then

\[
p(1) \otimes q(2) \leftarrow q(1) \otimes r(5) \otimes s(6) \in \text{Gnd}(C).
\]
Using definition 9.10, the notion of provability can be defined in the same way as for standard LO. The corresponding proof system is shown in Figure 9.1 for convenience. As usual, an LO(\mathcal{C}) sequent has the form \( P \vdash G_1, \ldots, G_k \), where \( P = D_1 \land \cdots \land D_n \) is an LO program (the set of clauses \( D_1, \ldots, D_n \)) and \( G_1, \ldots, G_k \) is a multiset of goals. A sequent is provable if all branches of its proof tree terminate with an instance of the \( \top \) axiom. Rule bc is applicable only if the right-hand side of the current sequent consists of atomic formulas.

We can formulate the following proposition, which is analogous to Proposition 3.9.

**Proposition 9.13 (Admissibility of the Weakening Rule)** Given an LO(\mathcal{C}) program \( P \) and two multisets of goals \( \Delta, \Delta' \) such that \( \Delta \preceq \Delta' \), if \( P \vdash \Delta \) then \( P \vdash \Delta' \).

### 9.2 An Effective Semantics for LO(\mathcal{C})

In this section we will discuss the definition of an effective bottom-up semantics for LO(\mathcal{C}). First of all, we reformulate the definition of the operational semantics as follows.

**Definition 9.14 (Operational Semantics)** Given an LO(\mathcal{C}) program \( P \), its operational semantics, denoted \( O(P) \), is given by

\[
O(P) = \{ A \mid A \text{ is a multiset of ground atomic formulas and } P \vdash A \}
\]

Given that LO(\mathcal{C}) provability can be reduced to LO provability using the concept of ground instances of a clause, as explained in Section 9.1, we can re-use the results of Section 7.1 in order to define a bottom-up semantics for LO(\mathcal{C}).

An LO(\mathcal{C}) program can be seen as a (potentially infinite) set of ground clauses. We can therefore reformulate the bottom-up semantics of Section 7.1 with the following slight modifications: we consider a (generally infinite) set of ground atomic formulas instead of a finite set of propositional symbols \( \Sigma \); we substitute \( Gnd(P) \) (for an LO(\mathcal{C}) program \( P \)).
in place of $P$ in all the definitions of Section 7.1. For instance, the Herbrand base will be the set of all multisets of ground atomic formulas. With these modifications, the results of Section 7.1 still hold, the proofs being exactly analogous. Infiniteness of the Herbrand base and of the set of ground clauses of a program do not cause any harm. The resulting fixpoint semantics is therefore sound and complete with respect to the operational semantics defined above. Clearly, this bottom-up semantics is not effective.

In the rest of this section, we will refer to the definitions and notations used in Section 7.1. In particular, the (concrete) satisfiability judgment $|=\quad$ and the $T_P$ fixpoint operator are defined, respectively, in Definition 7.4 and Definition 7.7.

We are now ready to present the symbolic version of the bottom-up semantics for LO($C$).

First of all, we introduce the notion of constrained multiset, which is central to the semantics definition.

**Definition 9.15 (Constrained Multiset)** Let $\Pi$ be a finite set of predicate symbols, and $C = \langle \Sigma, \mathcal{V}, \mathcal{L}, \mathcal{D}, \text{Sol}, \sqsubseteq \rangle$ a constraint system. A constrained multiset over $\Pi$ and $C$ is a multiset of atomic formulas, annotated with a constraint, of the form

$$\{p_1(x_{11}, \ldots, x_{1k_1}), \ldots, p_n(x_{n1}, \ldots, x_{nk_n})\}: \varphi$$

where $p_1, \ldots, p_n \in \Pi$, $\varphi \in \mathcal{L}$ is a satisfiable constraint, and $x_{11}, \ldots, x_{nk_n}$ are distinct variables in $\mathcal{V}$.

We often omit brackets in constrained multiset notation.

**Notation.** By analogy with multiset unifiers (see Section 2.4), we define the following notation. Given two constrained multisets $N = N_1, \ldots, N_n : \psi$ and $M = M_1, \ldots, M_n : \varphi$ with disjoint variables (note that $|N| = |M|$), where $N_i = p_i(y_{i1}, \ldots, y_{ik_i})$ and $M_i = q_i(x_{i1}, \ldots, x_{ik_i})$ for $i : 1, \ldots, n$, we define the constraint $N = M$ as $\bigwedge_{i : 1, \ldots, n} (y_{i1} = x_{i1} \land \ldots \land y_{ik_i} = x_{ik_i})$, provided $p_i = q_i$ for every $i : 1, \ldots, n$ and $\{l_1, \ldots, l_n\}$ is a permutation of $\{1, \ldots, n\}$. We will use the notation $N = M$ to denote a constraint which is non-deterministically picked from the set of constraints which can obtained in the above manner (in general more alternatives are possible, depending on the choice of the permutation).

Constrained multisets will be used as symbolic representations for sets of elements of the (concrete) Herbrand base. With this in mind, we give the following definitions.

**Definition 9.16 (Abstract Herbrand Base)** Let $\Pi$ be a finite set of predicate symbols, and $C$ a constraint system. Given an LO($C$) program $P$, the Herbrand base of $P$, denoted $HB(P)$, is given by

$$HB(P) = \{M : \varphi \mid M : \varphi \text{ is a constrained multiset over } \Pi \text{ and } C\}.$$
Definition 9.17 (Abstract Interpretations) Let \( \Pi \) be a finite set of predicate symbols, and \( \mathcal{C} \) a constraint system. Given an LO(\( \mathcal{C} \)) program \( P \), an interpretation \( I \) is any subset of \( HB(P) \), i.e., \( I \in \mathcal{P}(HB(P)) \).

By analogy with definition 9.10, we give the following.

Definition 9.18 (Ground Instance Operator) Let \( \Pi \) be a finite set of predicate symbols, and \( \mathcal{C} = \langle \Sigma, \mathcal{V}, \mathcal{L}, \mathcal{D}, \text{Sol}, \sqsubseteq \rangle \) a constraint system. Given an interpretation \( I \), we define the operator \( \text{Inst} \) as follows:

\[
\text{Inst}(I) = \{ \sigma(M) \mid M : \varphi \in I, \sigma \in \text{Sol}(\varphi) \}.
\]

Let \( Up \) be the following operator on multisets of ground atomic formulas: \( Up(I) = \{ A + C \mid A \in I \} \). The following definition provides the connection between concrete interpretations and abstract interpretations.

Definition 9.19 (Denotation of an Interpretation) Let \( \Pi \) be a finite set of predicate symbols, and \( \mathcal{C} \) a constraint system. Given an (abstract) interpretation \( I \), its denotation \( [I] \) is the (concrete) interpretation defined as follows:

\[
[I] = Up(\text{Inst}(I)).
\]

Two interpretations \( I \) and \( J \) are said to be equivalent, written \( I \simeq J \), if and only if \( [I] = [J] \).

We are now ready to define the abstract interpretation domain. As usual, we can identify interpretations having the same denotation using equivalence classes with respect to the corresponding equivalence relation \( \simeq \).

Definition 9.20 (Abstract Interpretation Domain) Abstract interpretations form a complete lattice \( \langle \mathcal{I}, \sqsubseteq \rangle \), where

- \( \mathcal{I} = \{ [I] \simeq \mid I \text{ is an interpretation} \} \);
- \( [I] \simeq \sqsubseteq [J] \simeq \text{iff} [I] \sqsubseteq [J] \);
- the least upper bound of \( [I] \simeq \) and \( [J] \simeq \), written \( [I] \simeq \cup [J] \simeq \), is \( [I \cup J] \simeq \);
- the bottom and top elements are \( [\emptyset] \simeq \) and \( [\varepsilon : \text{true}] \simeq \), respectively.

Before going on with the definition of the satisfiability judgment, we introduce the following notion of entailment between constrained multisets.
Definition 9.21 (Entailment of Constrained Multisets) Let \( \Pi \) be a finite set of predicate symbols, and \( C \) a constraint system. We call a relation \( \sqsubseteq^M \) between constrained multisets an entailment whenever \( N : \psi \sqsubseteq^M M : \varphi \) implies \( [N : \psi] \subseteq [M : \varphi] \), for any constrained multisets \( N : \psi \) and \( M : \varphi \).

An example of entailment relation is given below. As we will prove in the following, it amounts to an effective and sufficient condition for testing the \( \sqsubseteq \) relation over interpretations. We first give the following definition.

Definition 9.22 (Entailment \( \sqsubseteq^m \)) Let \( \Pi \) be a finite set of predicate symbols, \( C = \langle \Sigma, V, \mathcal{L}, \mathcal{D}, \text{Sol}, \sqsubseteq^c \rangle \) a constraint system, and \( N : \psi, M : \varphi \) two constrained multisets over \( \Pi \) and \( C \) with disjoint variables. We say that \( N : \psi \) entails \( M : \varphi \), written \( N : \psi \sqsubseteq^m M : \varphi \), if there exists a multiset \( N' \not\prec N \) such that

\[
\exists x. (N' = M \land \psi) \sqsubseteq^c \varphi
\]

where \( x = \text{FV}(N : \psi) \).

Example 9.23 Let \( N : \psi \equiv p(y), q(z) : y > 0 \land z > 0 \), and \( M : \varphi \equiv p(x) : x > 0 \). Then \( N : \psi \sqsubseteq^m M : \varphi \). In fact, if we take \( N' = p(y) \) we have that \( \exists y, z. (x = y \land y > 0 \land z > 0) \sqsubseteq^c x > 0 \).

The following result states that \( \sqsubseteq^m \) is an entailment.

Proposition 9.24 Let \( \Pi \) be a finite set of predicate symbols, \( C \) a constraint system, and \( N : \psi, M : \varphi \) two constrained multisets over \( \Pi \) and \( C \) with disjoint variables. If \( N : \psi \sqsubseteq^m M : \varphi \) then \( [N : \psi] \subseteq [M : \varphi] \).

Proof It is sufficient to prove that for every \( \sigma \in \text{Sol}(\psi) \) there exists \( \sigma' \in \text{Sol}(\varphi) \) such that \( \sigma'(M) \not\prec \sigma(N) \).

By hypothesis, there exists a multiset \( N' \not\prec N \) s.t. \( x = \text{FV}(N : \psi) \) and \( \exists x. (N' = M \land \psi) \sqsubseteq^c \varphi \), i.e., \( \text{Sol}(\exists x.(N' = M \land \psi)) \subseteq \text{Sol}(\varphi) \).

Now, let \( M = \{M_1, \ldots, M_n\} \), \( N' = N_1, \ldots, N_n \) and \( \{i_1, \ldots, i_n\} \) the permutation of \( \{1, \ldots, n\} \) (satisfying the above relation) such that \( N_{ik} = M_k \) for \( k = 1, \ldots, n \). Assuming \( N_{ik} = p(y_{ik}) \) and \( M_k = p(x_k) \), where \( y_{ik} \) and \( x_k \) are vector of variables and \( p \in \Pi \), the above constraint is equivalent to \( y_{ik} = x_k \).

Now, let \( \sigma' \) be the evaluation such that \( \sigma'|_{x_k} = \sigma|_{y_{ik}} \) for every \( k = 1, \ldots, n \), and \( \sigma' = \sigma \) otherwise. It easy to see that \( \sigma' \in \text{Sol}(\exists x.(N' = M \land \psi)) \) (note that the constraint \( N' = M \) is simply a renaming of variables: variables inside constrained multisets are distinct and \( N : \psi, M : \varphi \) have disjoint variables). Therefore \( \sigma' \in \text{Sol}(\varphi) \) using the fact that \( \text{Sol}(\exists x.(N' = M \land \psi)) \subseteq \text{Sol}(\varphi) \). Besides, \( \sigma'(M) = \sigma(N') \) (by definition of \( \sigma' \)) and \( \sigma(N') \not\prec \sigma(N) \), from which the conclusion. \( \square \)
We extend the entailment relation to interpretations as follows.

**Definition 9.25** Let \( I \) and \( J \) be two interpretations. We say that \( I \) entails \( J \), written \( I \sqsubseteq J \), if for every \((N : \psi) \in I\) there exists \((M : \varphi) \in J\) s.t. \( N : \psi \sqsubseteq^m M : \varphi \).

**Corollary 9.26** Given two interpretations \( I \) and \( J \), if \( I \sqsubseteq J \) then \([I] \subseteq [J]\).

**Proof** By Proposition 9.24, observing that \([I] = \{[M : \varphi] \mid M : \varphi \in I\}\).

**Remark 9.27** We note that the reverse implication in Proposition 9.24 does not hold. In fact, consider the following counterexample over linear integer constraints. Let

\[
\begin{align*}
\mathcal{N} : \psi &\equiv p(x, y), p(w, z) : even(x) \land odd(z) \land y = w \\
\mathcal{M} : \varphi &\equiv p(x', y') : even(x') \land odd(y')
\end{align*}
\]

where \(even(v)\) is a shorthand for \(\exists u.(v = 2u)\) and \(odd(v)\) is a shorthand for \(\exists u.(v = 2u + 1)\). Then it is easy to see that \([N : \psi] \subseteq [M : \varphi]\), while it is not the case that \(N : \psi \sqsubseteq^m M : \varphi\). We also note that the reverse implication in Corollary 9.26 does not hold for a further reason (besides the one above). Consider in fact the following counterexample (over linear integer constraints): \(I = \{p(x) : 1 < x < 5\}\) and \(J = \{p(y) : 1 < y < 3, p(z) : 2 < z < 6\}\).

We are now ready to define the abstract satisfiability judgment. The judgment has the form \(I \models \Delta : \varphi \bowtie \mathcal{C} \bowtie \varphi'\), where \(I\) is an interpretation, \(\Delta : \varphi\) is a constrained context (i.e., a constrained multiset of goals) with \(\varphi\) satisfiable, \(\mathcal{C}\) is an output fact (i.e., a multiset of atomic formulas), and \(\varphi'\) is an output constraint.

**Definition 9.28 (Abstract Satisfiability Judgment)** Let \(P\) be an LO(\(\mathcal{C}\)) program and \(I\) an interpretation. The abstract satisfiability judgment \(\models\) is defined as follows:

\[
\begin{align*}
I \models T, \Delta : \varphi \bowtie \epsilon \bowtie \varphi; \\
I \models A : \varphi \bowtie \mathcal{C} \bowtie \varphi' &\text{ if there exist } B : \psi \in I \text{ (variant), } B' \not\equiv B, |A'| = |A|, |B'| = |B|, C = B \setminus B', \text{ and } \varphi' \equiv B' \bowtie \mathcal{C} \bowtie \mathcal{C}' \bowtie \varphi; \\
I \models G_1 \& G_2, \Delta : \varphi \bowtie \mathcal{C} \bowtie \varphi' &\text{ if } I \models G_1, \Delta : \varphi \bowtie \mathcal{C}_1 \bowtie \varphi_1, \text{ and } \Delta : \varphi \bowtie \mathcal{C}_2 \bowtie \varphi_2, \\
\text{and } D_1 \not\subseteq C_1, D_2 \not\subseteq C_2, |D_1| = |D_2|, C = C_1 + (C_2 \setminus D_2), \text{ and } \varphi' \equiv D_1 \bowtie D_2 \bowtie \varphi_1 \bowtie \varphi_2 \text{ is satisfiable}; \\
I \models G_1 \not\& G_2, \Delta : \varphi \bowtie \mathcal{C} \bowtie \varphi' &\text{ if } I \models G_1, \Delta : \varphi \bowtie \mathcal{C} \bowtie \varphi'; \\
I \models \bot, \Delta : \varphi \bowtie \mathcal{C} \bowtie \varphi' &\text{ if } I \models \Delta : \varphi \bowtie \mathcal{C} \bowtie \varphi'.
\end{align*}
\]
Example 9.29 Let $\Pi = \{p, q, r, s, t\}$, and consider unification constraints over the term language generated by a constant symbol $a$ and a function symbol $f$. Let $\mathcal{V}$ be a denumerable set of variables and $x, y, z, \ldots \in \mathcal{V}$. Consider the following clause

$$p(x) \mathcal{B} p(z) \leftarrow (q(x') \mathcal{B} r(y)) \& s(z') \quad \square z = a \wedge x' = x \wedge z' = a.$$ 

and the interpretation $I$ consisting of the following constrained multisets:

$$\mathcal{M}_1 : \psi_1 \equiv t(u_1, v_1), q(w_1) : u_1 = f(q_1) \wedge w_1 = a$$
$$\mathcal{M}_2 : \psi_2 \equiv t(u_2, v_2), s(w_2) : v_2 = f(q_2) \wedge w_2 = a$$

Let $G : \varphi$ be $(q(x') \mathcal{B} r(y)) \& s(z') : z = a \wedge x' = x \wedge z' = a$. First of all, we use the $\&$-rule for $\vdash$. We have to compute $C_1, C_2, \varphi_1$ and $\varphi_2$ such that

$$I \vdash q(x') \mathcal{B} r(y) : \varphi \bullet C_1 \bullet \varphi_1 \quad \text{and} \quad I \vdash s(z') : \varphi \bullet C_2 \bullet \varphi_2.$$ 

For the first conjunct and using the $\mathcal{B}$-rule, we have that $I \vdash q(x') \mathcal{B} r(y) : \varphi \bullet C_1 \bullet \varphi_1$ iff $I \vdash q(x'), r(y) : \varphi \bullet C_1 \bullet \varphi_1$. By the second rule for $\vdash$, applied to $\mathcal{M}_1 : \psi_1 \in I$, we have that $C_1 = t(u_1, v_1)$ and $\varphi_1 \equiv x' = w_1 \wedge v_1 = a \wedge z = a \wedge x' = x \wedge z' = a \wedge u_1 = f(q_1)$. For the second conjunct and using the second rule for $\vdash$ applied to $\mathcal{M}_2 : \psi_2 \in I$, we have that $C_2 = t(u_2, v_2)$ and $\varphi_2 \equiv z' = w_2 \wedge v_2 = a \wedge z = a \wedge x' = x \wedge z' = a \wedge v_2 = f(q_2)$. Therefore by definition of the $\&$-rule, if we unify $t(u_1, v_1)$ and $t(u_2, v_2)$, we have that

$$I \vdash G : \varphi \bullet t(u_1, v_1) \bullet u_1 = u_2 \wedge v_1 = v_2 \wedge \varphi_1 \land \varphi_2.$$ 

We also have, again by the $\&$-rule (by choosing empty sub-multisets), that

$$I \vdash G : \varphi \bullet t(u_1, v_1), t(u_2, v_2) \bullet \varphi_1 \land \varphi_2.$$ 

The following lemma states a simple property which we will need later.

Lemma 9.30 For every interpretation $I$, constrained multiset of goals $\Delta : \varphi$, fact $C$, and constraint $\varphi'$, if $I \vdash \Delta : \varphi \bullet C \bullet \varphi'$ then there exists a constraint $\psi$ s.t. $\varphi' \equiv \varphi \land \psi$, $\varphi'$ is satisfiable, $FV(\psi) \cap FV(\varphi) \subseteq FV(\Delta)$, $FV(C) \cap FV(\varphi) = \emptyset$, and $FV(C) \cap FV(\Delta) = \emptyset$.

Proof By simple induction on the $\vdash$ definition.

The connection between the satisfiability judgments $|= \vdash$ is clarified by the following lemma.

134
Lemma 9.31  For every interpretation $I$, constrained multiset of goals $\Delta : \varphi$, facts $C$ and $S$, constraint $\varphi'$ and evaluation $\sigma$,

i. if $I \models \Delta : \varphi \triangleright C \triangleright \varphi'$ and $\sigma \in \text{Sol}(\varphi')$, then $[I] \models \sigma(\Delta) \triangleright S$ for all $S$ s.t $\sigma(C) \not\ll S$;

ii. if $[I] \models \sigma(\Delta) \triangleright S$ and $\sigma \in \text{Sol}(\varphi)$, then there exist a multiset $C$, a satisfiable constraint $\varphi'$, and an evaluation $\sigma' \in \text{Sol}(\varphi')$ s.t. $I \models \Delta : \varphi \triangleright C \triangleright \varphi'$, $\sigma'(C) \not\ll S$, and $\sigma'|_{FV(\Delta)} = \sigma|_{FV(\Delta)}$.

Proof

i. By induction on the derivation of $I \models \Delta : \varphi \triangleright C \triangleright \varphi'$.

- If $I \models \top, \Delta : \varphi \triangleright \varphi$ and $\sigma \in \text{Sol}(\varphi)$ then $[I] \models \sigma(\top, \Delta) \triangleright S$ for every $S$;

- if $I \models A : \varphi \triangleright C \triangleright \varphi'$ and $\sigma \in \text{Sol}(\varphi')$, then there exists a variant $B : \psi$ of an element in $I$ s.t. $B' \ll B$, $A' \ll A$, $|B'| = |A'|$, $\varphi' \equiv B' = A' \land \varphi \land \psi$ is satisfiable, and $C = B \setminus B'$.

We have that $[I] \models \sigma(A) \triangleright S$ if $\sigma(A) + S \in [I]$. This holds if $\sigma(B) \ll \sigma(A) + S$ iff (remember that $B' \ll B$ and $A' \ll A$) $\sigma(B \setminus B') + \sigma(B') \not\ll \sigma(A \setminus A') + \sigma(A') + S$ iff $\sigma(B \setminus B') \not\ll \sigma(A \setminus A') + S$ if $\sigma(C) \not\ll \sigma(A \setminus A') + S$. The latter relation holds if $\sigma(C) \not\ll S$;

- if $I \models G_1 \& G_2, \Delta : \varphi \triangleright A \triangleright \varphi'$ and $\sigma \in \text{Sol}(\varphi')$, then

$$I \models G_1, \Delta : \varphi \triangleright A_1 \triangleright \varphi_1 \text{ and } I \models G_2, \Delta : \varphi \triangleright A_2 \triangleright \varphi_2,$$

with $A_1' \ll A_1, A_2' \ll A_2$, $|A_1'| = |A_2'|$, $\varphi' \equiv A_1' = A_2' \land \varphi_1 \land \varphi_2$ is satisfiable, and $A = A_1 + (A_2 \setminus A_2')$.

Clearly, $\sigma \in \text{Sol}(\varphi')$ implies $\sigma \in \text{Sol}(\varphi_1)$ and $\sigma \in \text{Sol}(\varphi_2)$. Therefore, by inductive hypothesis, we have that $[I] \models \sigma(G_1, \Delta) \triangleright S$ for all $S$ s.t. $\sigma(A_1) \not\ll S$, and $[I] \models \sigma(G_2, \Delta) \triangleright S$ for all $S$ s.t. $\sigma(A_2) \not\ll S$.

Clearly, $\sigma(G_1, \Delta) = \sigma(G_1), \sigma(\Delta)$, and similarly for $G_2$. Therefore we have that $[I] \models \sigma(G_1) \& \sigma(G_2), \sigma(\Delta) \triangleright S$ for all $S$ s.t. $\sigma(A_1) \not\ll S$ and $\sigma(A_2) \not\ll S$, i.e.,

$[I] \models \sigma(G_1 \& G_2, \Delta) \triangleright S$ for all $S$ s.t. $\sigma(A_1) \not\ll S$ and $\sigma(A_2) \not\ll S$.

In order to conclude, it is sufficient to observe that $\sigma(A_1) \not\ll \sigma(A)$ and $\sigma(A_2) \not\ll \sigma(A)$. In fact, $A = A_1 + (A_2 \setminus A_2')$, so that $\sigma(A_1) \not\ll \sigma(A)$. Besides, $\sigma(A_2) \not\ll \sigma(A)$ because $\sigma(A_2') = \sigma(A_1')$ and $\sigma(A_1') \not\ll \sigma(A_1)$;

- if $\Delta = G_1 \not\ll G_2, \Delta'$ of $\Delta = \top, \Delta'$, the conclusion follows by a straightforward application of the inductive hypothesis.

135
ii. By induction on the derivation of \([I]\models\sigma(\Delta)\rhd S\).

- If \([I]\models T, \Delta\rhd S\), \(\sigma\in\text{Sol}(\varphi)\) and \(\Delta = \sigma(\Delta_1)\), then we take \(C = \epsilon\) and \(\sigma' = \sigma\), and we have \(I\vDash T, \Delta_1: \varphi\rhd \varphi\).

- if \([I]\models A\rhd S\), \(\sigma\in\text{Sol}(\varphi)\), and \(A = \sigma(A_1)\), then \(A + S\in[I]\). Therefore there exists \(B\in[I]\) s.t. \(B = A + S\), i.e., there exists \(B_1: \psi\), variant of an element in \(I\), and an evaluation \(\sigma'\in\text{Sol}(\psi)\) s.t. \(\sigma'(B_1)\not\succeq B\).

Let \(B' = \sigma'(B_1)\setminus S\), and \(B_1 \not\succeq B_1\) s.t. \(\sigma'(B_1) = B'\) (it is easy to show that such \(B_1\) exists). By definitions and the relations \(\sigma'(B_1)\not\succeq B\) and \(B = A + S\), it also follows that \(B' \not\succeq A\). Therefore, let \(A_1 \not\succeq A_1\) s.t. \(\sigma(A_1) = B'\).

Now, let \(\sigma''\) be the evaluation such that

\[
\sigma''_{\text{FV}(B_1,\psi)} = \sigma'_{\text{FV}(B_1,\psi)} \quad \text{and} \quad \sigma'' = \sigma \quad \text{otherwise}
\]

(note that \(B_1: \psi\) is a variant, therefore \(B_1: \psi\) and \(A_1: \varphi\) have no variables in common). Let \(\varphi' = B_1 = A_1 \land \varphi \land \psi\).

We have that \(\sigma''\in\text{Sol}(\varphi')\), in fact \(\sigma''(B_1) = \sigma(B_1) = \sigma(A_1) = \sigma''(A_1)\), besides it can be easily shown that \(\sigma''\) is a solution for both \(\varphi\) and \(\psi\).

It follows that \(I\vDash A_1: \varphi\rhd C \rhd \varphi'\) and \(C = B_1\setminus B_1\). Therefore \(\sigma''(C) = \sigma''(B_1\setminus B_1) = (\text{remember that } B_1 \not\succeq B_1 \land \sigma''(B_1) \not\succeq B\setminus B_1 = (A + S)\setminus B_1 \not\succeq S\) (because \(B' \not\succeq A\)). Finally, \(\sigma''_{\text{FV}(A_1)} = \sigma'_{\text{FV}(A_1)}\) by construction;

- if \([I]\models F & G, \Delta\rhd S\), \(\sigma\in\text{Sol}(\varphi)\), and \(F & G, \Delta = \sigma(F_1 & G_1, \Delta_1)\), then \([I]\models F, \Delta\rhd S\), \([I]\models G, \Delta\rhd S\). By inductive hypothesis, there exist multisets \(A_1\) and \(A_2\), satisfiable constraints \(\varphi_1, \varphi_2\), and evaluations \(\sigma_1'\in\text{Sol}(\varphi_1)\), \(\sigma_2'\in\text{Sol}(\varphi_2)\) s.t

\[
I\vDash F, \Delta_1: \varphi_1\rhd A_1\rhd \varphi_1 \quad \text{and} \quad I\vDash G, \Delta_1: \varphi_2\rhd A_2\rhd \varphi_2,
\]

with \(\sigma_1'(A_1) \not\succeq S\), \(\sigma_2'(A_2) \not\succeq S\), \(\sigma_1'_{\text{FV}(F_1, \Delta_1)} = \sigma_{\text{FV}(F_1, \Delta_1)}\), and \(\sigma_2'_{\text{FV}(G_1, \Delta_1)} = \sigma_{\text{FV}(G_1, \Delta_1)}\). By Lemma 9.30, we also have that there exist \(\psi_1\) and \(\psi_2\) such that \(\varphi_1 \equiv \varphi \land \psi_1\) and \(\varphi_2 \equiv \varphi \land \psi_2\).

Now, let \(\sigma''\) be the evaluation s.t.

\[
\sigma''_{\text{FV}(F_1, \Delta_1, A_1; \varphi_1)} = \sigma_1'_{\text{FV}(F_1, \Delta_1, A_1; \varphi_1)} \quad \text{and} \quad \sigma'' = \sigma_2' \quad \text{otherwise}.
\]

We have that \(\sigma''\) is a solution for \(\varphi_1\) (by construction) and \(\sigma''\) is a solution for \(\varphi_2\). In fact \(\varphi_2 \equiv \varphi \land \psi_2\), \(\sigma''\) is a solution for \(\varphi\) because \(\varphi_1 \equiv \varphi \land \psi_1\), and \(\sigma''\) is a solution for \(\psi_2\) because \(\sigma''_{\text{FV}(\psi_2)} = \sigma_2'_{\text{FV}(\psi_2)}\) (note that \(\text{FV}(\psi_2) \cap \text{FV}(\varphi) \not\succeq \text{FV}(G_1, \Delta_1)\) by Lemma 9.30 and that \(\sigma_1'\) and \(\sigma_2'\) coincide on variables in \(\text{FV}(G_1, \Delta_1) \cap \text{FV}(F_1, \Delta_1)\)).

Now, let \(B = \sigma''(A_1)\bullet \sigma''(A_2)\) (we recall that \(\bullet\) denotes the merge of multisets,
see Section 2.2), and \( \mathcal{A}'_1, \mathcal{A}'_2 \) s.t. \( \mathcal{A}'_1 \preceq \mathcal{A}_1, \mathcal{A}'_2 \preceq \mathcal{A}_2 \), \( \sigma''(\mathcal{A}'_1) = \sigma''(\mathcal{A}'_2) \), and \( \sigma''(\mathcal{A}_1 \setminus \mathcal{A}'_2) = \mathcal{B} \) (it is easy to show that such \( \mathcal{A}'_1 \) and \( \mathcal{A}'_2 \) exist). We have that

\[
I \models F_1 \& G_1, \Delta : \varphi \rightarrow A \rightarrow \varphi',
\]

where \( \varphi' \equiv \mathcal{A}'_1 = \mathcal{A}'_2 \land \varphi_1 \land \varphi_2 \), \( \sigma'' \in \text{Sol}(\varphi') \), and \( A = \mathcal{A}_1 + (\mathcal{A}_2 \setminus \mathcal{A}'_2) \). We also have that \( \sigma''(A) = \sigma''(A_1 + (A_2 \setminus A'_2)) = (\text{remember that } \mathcal{A}'_2 \preceq \mathcal{A}_2) \sigma''(A_1) + \sigma''((A_2 \setminus A'_2)) = \mathcal{B} = \sigma''(A_1) \bullet \sigma''(A_2) \preceq \mathcal{S} \) because \( \sigma''(A_1) = \sigma'_1(A_1) \preceq \mathcal{S} \) and \( \sigma''(A_2) = \sigma'_2(A_2) \preceq \mathcal{S} \). Finally, \( \sigma''_{\mathcal{F}(F_1 \& G_1)} = \sigma_{\mathcal{F}(F_1 \& G_1)} \) by construction;

- if \( \Delta = G_1 \& G_2, \Delta' \) of \( \Delta = \bot, \Delta' \), the conclusion follows by a straightforward application of the inductive hypothesis.

\( \square \)

The satisfiability judgment \( \models \) also satisfies the following properties.

**Lemma 9.32** For any interpretations \( I_1, I_2, \ldots \), constrained multiset of goals \( \Delta : \varphi, \text{ fact } C', \text{ constraint } \varphi' \) and evaluation \( \sigma \),

i. if \( I_1 \subseteq I_2 \) and \( I_1 \models \Delta : \varphi \rightarrow C' \rightarrow \varphi' \) and \( \sigma \in \text{Sol}(\varphi') \), then there exist a fact \( C'' \), a constraint \( \varphi'' \) and an evaluation \( \sigma' \in \text{Sol}(\varphi'') \) s.t. \( I_2 \models \Delta : \varphi \rightarrow C'' \rightarrow \varphi'' \), \( \sigma'(C'') \preceq \sigma(C') \) and \( \sigma'_{\mathcal{F}(\Delta)} = \sigma_{\mathcal{F}(\Delta)} \);

ii. if \( I_1 \subseteq I_2 \subseteq \ldots \) \( \bigcup_{i=1}^{\infty} I_i \models \Delta : \varphi \rightarrow C' \rightarrow \varphi' \) and \( \sigma \in \text{Sol}(\varphi') \), then there exist \( k \in \mathbb{N} \), a fact \( C'' \), a constraint \( \varphi'' \) and an evaluation \( \sigma' \in \text{Sol}(\varphi'') \) s.t. \( I_k \models \Delta : \varphi \rightarrow C'' \rightarrow \varphi'' \), \( \sigma'(C'') \preceq \sigma(C') \) and \( \sigma'_{\mathcal{F}(\Delta)} = \sigma_{\mathcal{F}(\Delta)} \).

**Proof**

i. Suppose \( I_1 \subseteq I_2 \) and \( I_1 \models \Delta : \varphi \rightarrow C' \rightarrow \varphi' \). Then by Lemma 9.31 i we have that for every \( \sigma \in \text{Sol}(\varphi') \), \( [I_1] \models \sigma(\Delta) \bullet \sigma(C') \). By hypothesis, \( [I_1] \subseteq [I_2] \), therefore by Lemma 7.6 i we have \( [I_2] \models \sigma(\Delta) \bullet \sigma(C') \). The conclusion then follows from Lemma 9.31 ii;

ii. suppose \( I_1 \subseteq I_2 \subseteq \ldots \) and \( \bigcup_{i=1}^{\infty} I_i \models \Delta : \varphi \rightarrow C' \rightarrow \varphi' \). Then by Lemma 9.31 i we have that for every \( \sigma \in \text{Sol}(\varphi') \), \( \bigcup_{i=1}^{\infty} [I_i] \models \sigma(\Delta) \bullet \sigma(C') \), i.e., as it can be readily verified from Definition 9.19 and Definition 9.20, \( \bigcup_{i=1}^{\infty} [I_i] \models \sigma(\Delta) \bullet \sigma(C') \). By Lemma 7.6 ii, we have that there exists \( k \in \mathbb{N} \) s.t. \( [I_k] \models \sigma(\Delta) \bullet \sigma(C') \). The conclusion then follows from Lemma 9.31 ii.
We are now ready to define the abstract fixpoint operator \( S_P : \mathcal{I} \rightarrow \mathcal{I} \). As in Chapter 7, we will proceed in two steps. We will first define an operator working over interpretations (i.e., elements of \( \mathcal{P}(\mathcal{H}(P)) \)) and then we will lift it to our abstract domain \( \mathcal{I} \) consisting of the equivalence classes of elements of \( \mathcal{P}(\mathcal{H}(P)) \) w.r.t. the relation \( \simeq \) defined in Definition 9.19. Formally, we first introduce the following definitions.

**Definition 9.33 (Clause Variants)** Let \( \Pi \) be a finite set of predicate symbols, and \( \mathcal{C} = \langle \Sigma, \mathcal{V}, \mathcal{L}, \mathcal{D}, \mathcal{Sol}, \subseteq \rangle \) a constraint system. Given an LO(\( \mathcal{C} \)) program \( P \), the set of variants of clauses in \( P \), denoted \( \text{Vrn}(P) \), is defined as follows:

\[
\text{Vrn}(P) = \{ (H \circ G \varphi \theta) | \forall (H \circ G \varphi) \in P \text{ and } \theta \text{ is a renaming of the variables in } \text{FV}(H \circ G \varphi) \text{ with new variables} \}
\]

**Definition 9.34 (Symbolic Fixpoint Operator \( S_P \))** Given an LO(\( \mathcal{C} \)) program \( P \) and an interpretation \( I \), the symbolic fixpoint operator \( S_P \) is defined as follows:

\[
S_P(I) = \{ \hat{H} + C : \exists x \varphi' | (H \circ G \varphi) \in \text{Vrn}(P), I \models G : \varphi \triangleright C \triangleright \varphi' \text{ and } x = \text{FV}(\varphi') \setminus \text{FV}(\hat{H} + C) \}
\]

The following property shows that \( S_P \) is sound and complete w.r.t \( T_P \).

**Proposition 9.35** Let \( P \) be an LO(\( \mathcal{C} \)) program and \( I \) and interpretation. Then \([S_P(I)] = T_{\text{Gnd}(P)}([I])\).

**Proof**

- \([S_P(I)] \subseteq T_{\text{Gnd}(P)}([I])\).

Let \( B : \exists x \varphi' = \hat{H} + C : \exists x \varphi' \in S_P(I) \), where \( H \circ G \varphi \) is a variant of a clause in \( P \), \( I \models G : \varphi \triangleright C \triangleright \varphi' \), and \( x = \text{FV}(\varphi') \setminus \text{FV}(B) \).

Let \( \sigma \in \text{Sol}(\exists x \varphi') \). Then, there exists \( \sigma' \in \text{Sol}(\varphi') \) s.t. \( \sigma'|_{\text{FV}(B)} = \sigma|_{\text{FV}(B)} \). By Lemma 9.30, there exists \( \psi \) s.t. \( \varphi' \equiv \varphi \land \psi \), therefore \( \sigma' \in \text{Sol}(\varphi) \).

Let \( H_1 \circ G_1 = \sigma'(H \circ G) \in \text{Gnd}(P) \). By Lemma 9.31 \( i \), we also have that \([I] \models \sigma'(G) \triangleright A \) for all \( A \) s.t. \( \sigma'(C) \not\leq A \).

By \( T_P \) definition, \( \hat{H}_1 + A \in T_{\text{Gnd}(P)}([I]) \), for all \( A \) s.t. \( \sigma'(C) \not\leq A \). That is, \( D \in T_{\text{Gnd}(P)}([I]) \), for all \( D \) s.t. \( \sigma'(B) \not\leq D \). Note that \( \sigma'(B) = \sigma(B) \), therefore it follows that \( D \in T_{\text{Gnd}(P)}([I]) \) for all \( D \in [B : \exists x \varphi'] \) and hence the conclusion.

138
Let $B = \widehat{H} + A \in T_{Gnd(P)}([I])$, where $H \vdash G \in T_{Gnd(P)}$ and $[I] \models G \bowtie A$.

Then, there exist $H_1 \vdash G_1 \square \varphi$, a variant of a clause in $P$ and $\sigma \in \text{Sol}(\varphi)$ s.t. $H \vdash G = \sigma(H_1 \vdash G_1)$. By Lemma 9.31 ii and Lemma 9.30, there exist a multiset $\mathcal{C}$, satisfiable constraints $\varphi'$ and $\psi$, and evaluation $\sigma'$ s.t. $\sigma' \in \text{Sol}(\varphi')$, $\sigma'|_{\text{FV}(G_1)} = \sigma|_{\text{FV}(G_1)}$, $\varphi' \equiv \varphi \land \psi$, $I \models G_1 : \varphi \bowtie \mathcal{C} \bowtie \varphi'$, and $\sigma'(\mathcal{C}) \preceq A$.

Let $\sigma''$ be the evaluation such that

$$\sigma''|_{\text{FV}(H_1, \varphi)} = \sigma|_{\text{FV}(H_1, \varphi)} \quad \text{and} \quad \sigma'' = \sigma' \quad \text{otherwise.}$$

We have that $\sigma'' \in \text{Sol}(\varphi')$. In fact, first of all $\varphi' \equiv \varphi \land \psi$; $\sigma'' \in \text{Sol}(\varphi)$ because $\sigma \in \text{Sol}(\varphi)$ and $\sigma''|_{\text{FV}(\varphi)} = \sigma|_{\text{FV}(\varphi)}$ by definition; finally, $\sigma'' \in \text{Sol}(\psi)$ because $\sigma' \in \text{Sol}(\psi)$ and $\sigma''|_{\text{FV}(\psi)} = \sigma'|_{\text{FV}(\psi)}$ (in fact, note that: $\text{FV}(\psi) \cap \text{FV}(H_1) \subseteq \text{FV}(G_1) \cup \text{FV}(\varphi)$ because $H_1$ does not appear in the judgment $I \models G_1 : \varphi \bowtie \mathcal{C} \bowtie \varphi'$; $\text{FV}(\psi) \cap \text{FV}(\varphi) \subseteq \text{FV}(G_1)$ by Lemma 9.30; $\sigma|_{\text{FV}(G_1)} = \sigma'|_{\text{FV}(G_1)}$). We also have that $\sigma''(\mathcal{C}) = \sigma'(\mathcal{C})$ by Lemma 9.30 and $\sigma''$ definition.

Clearly, $\sigma'' \in \text{Sol}(\varphi')$ implies $\sigma'' \in \text{Sol}(\exists x. \varphi')$. By $S_P$ definition, $\widehat{H_1} + \mathcal{C} : \exists x. \varphi' \in S_P(I)$. It follows that $\sigma''(\widehat{H_1} + \mathcal{C}) = \sigma''(\widehat{H_1}) + \sigma''(\mathcal{C}) = \sigma(\widehat{H_1}) + \sigma'(\mathcal{C}) = \widehat{H} + \sigma'(\mathcal{C}) \preceq \widehat{H} + A = B$, hence the conclusion. \hfill \Box

Furthermore, the following corollary holds.

**Corollary 9.36** For every LO(C) program $P$ and interpretations $I$ and $J$, if $I \simeq J$ then $S_P(I) \simeq S_P(J)$.

**Proof** If $I \simeq J$, i.e., $[I] = [J]$, we have that $T_{Gnd(P)}([I]) = T_{Gnd(P)}([J])$, and, by Proposition 9.35, $[S_P(I)] = [S_P(J)]$, i.e., $S_P(I) \simeq S_P(J)$. \hfill \Box

The previous Corollary allows us to safely lift $S_P$ definition to the lattice $\langle I, \sqsubseteq \rangle$. Formally, we define the abstract fixpoint operator as follows.

**Definition 9.37 (Abstract Fixpoint Operator $S_P$)** Given an LO(C) program $P$ and an equivalence class $[I]_\simeq$ of $I$, the abstract fixpoint operator $S_P$ is defined as follows:

$$S_P([I]_\simeq) = [S_P(I)]_\simeq$$

where $S_P(I)$ is defined in Definition 9.34.
For the sake of simplicity, in the following we will often use \( I \) to denote its class \([I]_\approx\), and we will simply use the term (abstract) interpretation to refer to an equivalence class, i.e., an element of \( \mathcal{I} \). The abstract fixpoint operator \( S_P \) satisfies the following property.

**Proposition 9.38 (Monotonicity and Continuity)** For every \( LO(C) \) program \( P \), the abstract fixpoint operator \( S_P \) is monotonic and continuous over the lattice \( \langle \mathcal{I}, \sqsubseteq \rangle \).

**Proof**

**Monotonicity.**

Assume that \( I \sqsubseteq J \) and \( \hat{H} + C' : \exists x . \varphi' \in S_P(I) \), where \( H \circ G \varphi \) is a variant of a clause in \( P \), \( x = FV(\varphi') \setminus FV(\hat{H} + C') \), and \( I \vDash G : \varphi \vartriangleright C' \vartriangleright \varphi' \).

Let \( \sigma \in Sol(\exists x . \varphi') \). Then, there exists \( \tau \in Sol(\varphi') \) s.t. \( \sigma|_{FV(H,C')} = \tau|_{FV(H,C')} \).

By Lemma 9.31, we have that there exist \( C'' \), \( \varphi'' \) and \( \tau' \in Sol(\varphi'') \) s.t. \( J \vDash G : \varphi \vartriangleright C'' \vartriangleright \varphi'' \), \( \tau'(C'') \sqsubseteq \tau(C') \) and \( \tau'|_{FV(G)} = \tau|_{FV(G)} \). By Lemma 9.30, we also have that \( \tau \in Sol(\varphi) \) and there exists \( \psi \) s.t. \( \varphi'' = \varphi \land \psi \).

By \( S_P \) definition, we have that \( \hat{H} + C' : \exists y . \varphi'' \in S_P(J) \), with \( y = FV(\varphi'') \setminus FV(\hat{H} + C'') \).

Now, let \( \sigma' \) be the evaluation s.t.

\[
\sigma'|_{FV(H,\varphi')} = \tau|_{FV(H,\varphi')} \quad \text{and} \quad \sigma' = \tau' \quad \text{otherwise}.
\]

We have that \( \sigma' \in Sol(\varphi'') \). In fact, first of all \( \varphi'' = \varphi \land \psi ; \sigma' \in Sol(\varphi) \) because \( \tau \in Sol(\varphi) \) and \( \sigma'|_{FV(\varphi)} = \tau|_{FV(\varphi)} \) by definition; finally, \( \sigma' \in Sol(\psi) \) because \( \tau \in Sol(\psi) \) and \( \sigma'|_{FV(\psi)} = \tau'|_{FV(\psi)} \) (in fact, note that: \( FV(\psi) \cap FV(H) \subseteq FV(G) \cup FV(\varphi) \) because \( H \) does not appear in the judgment \( J \vDash G : \varphi \vartriangleright C'' \vartriangleright \varphi'' \); \( FV(\psi) \cap FV(\varphi) \subseteq FV(G) \) by Lemma 9.30; \( \tau|_{FV(G)} = \tau'|_{FV(G)} \)). We also have that \( \sigma'(C'') = \tau'(C'') \) by Lemma 9.30 and \( \sigma' \) definition.

Clearly, \( \sigma' \in Sol(\varphi'') \) implies \( \sigma' \in Sol(\exists y . \varphi'') \). Therefore we have proved that for every \( \sigma \in Sol(\exists x . \varphi') \) there exists \( \sigma' \in Sol(\exists y . \varphi'') \) s.t. \( \sigma'(\hat{H} + C'') = \sigma'(\hat{H}) + \sigma'(C'') = \tau(\hat{H}) + \tau'(C'') \leq \tau(\hat{H}) + \tau(C') = \tau(\hat{H} + C') = \sigma(\hat{H} + C') \).

Using the fact that \( \hat{H} + C' : \exists y . \varphi'' \in S_P(J) \) and \( \hat{H} + C' : \exists x . \varphi' \in S_P(I) \), we can conclude that \( [S_P(I)] \subseteq [S_P(J)], \) i.e., \( S_P(I) \sqsubseteq S_P(J) \).

**Continuity.**

We show that \( S_P \) is finitary. Suppose \( I_1 \sqsubseteq I_2 \sqsubseteq \ldots \), and \( \hat{H} + C' : \exists x . \varphi' \in S_P(\bigsqcup_{i=1}^\infty I_i) \), with \( H \circ G \varphi \) variant of a clause in \( P \), \( x = FV(\varphi') \setminus FV(\hat{H} + C') \), and \( \bigsqcup_{i=1}^\infty I_i \vDash G : \varphi \vartriangleright C' \vartriangleright \varphi' \).

Let \( \sigma \in Sol(\exists x . \varphi') \). Then, there exists \( \tau \in Sol(\varphi') \) s.t. \( \sigma|_{FV(H,C')} = \tau|_{FV(H,C')} \).
By Lemma 9.32 ii we have that there exist \( k \in \mathbb{N} \), \( C'' \), \( \varphi'' \) and \( \tau' \in \text{Sol}(\varphi'') \) s.t. \( I_k \models G : \varphi \triangleleft C'' \triangleleft \varphi'' \), and \( \tau' |_{FV(G)} = \gamma_{FV(G)} \).

By \( S_P \) definition, we have that \( \hat{H} + C'' : \exists y. \varphi'' \in S_P(I_k) \), with \( y = FV(\varphi'') \setminus FV(\hat{H} + C'') \).

At this point, we proceed exactly as above (proof of monotonicity) to build an evaluation \( \sigma' \in \text{Sol}(\exists y. \varphi'') \) s.t. \( \sigma'(\hat{H} + C'') \preceq \sigma(\hat{H} + C') \).

It follows that \( [S_P(\bigcup_{i=1}^{\infty} I_i)] \subseteq [S_P(I_k)] \). Using the fact that \( [S_P(I_k)] \subseteq [\bigcup_{i=1}^{\infty} S_P(I_i)] \), we can conclude that \( [S_P(\bigcup_{i=1}^{\infty} I_i)] \subseteq [\bigcup_{i=1}^{\infty} S_P(I_i)] \), i.e., \( S_P(\bigcup_{i=1}^{\infty} I_i) \subseteq \bigcup_{i=1}^{\infty} S_P(I_i) \). \( \square \)

**Corollary 9.39** For every \( \text{LO}(C) \) program \( P \), \( [\text{lfp}(S_P)] = \text{lfp}(T_{Gnd(P)}) \).

Let \( \mathcal{F}_{sym}(P) = \text{lfp}(S_P) \), then we have the following main theorem.

**Theorem 9.40 (Soundness and Completeness)** For every \( \text{LO}(C) \) program \( P \), \( O(P) = [\mathcal{F}_{sym}(P)] \).

**Proof** From Definition 9.14, Theorem 7.10 and Corollary 9.39. \( \square \)

The previous results give us an algorithm to compute the operational and fixpoint semantics of an \( \text{LO}(C) \) program \( P \) via the fixpoint operator \( S_P \).

**Example 9.41** Let \( \Pi = \{p, q, r, s, t\} \), and consider unification constraints over the term language generated by a constant symbol \( a \) and a function symbol \( f \). Let \( \mathcal{V} \) be a denumerable set of variables and \( x, y, z, \ldots \in \mathcal{V} \). Let \( H \models G \square \varphi \) be the following clause

\[
p(x) \not\models p(z) \models (q(x') \not\models r(y)) \& s(z') \square z = a \land x' = x \land z' = a
\]

and \( I \) the interpretation of Example 9.29, i.e., consisting of the following constrained multisets:

\[
\mathcal{M}_1 : \psi_1 \equiv t(u_1, v_1), q(w_1) : u_1 = f(q_1) \land w_1 = a \\
\mathcal{M}_2 : \psi_2 \equiv t(u_2, v_2), s(w_2) : v_2 = f(q_2) \land w_2 = a
\]

From Example 9.29, we know that

\[
I \models G : \varphi \triangleright t(u_1, v_1) \triangleright \varphi' \quad \text{and} \quad I \models G : \varphi \triangleright t(u_1, v_1), t(u_2, v_2) \triangleright \varphi'',
\]

with \( \varphi' \equiv u_1 = u_2 \land v_1 = v_2 \land \varphi_1 \land \varphi_2 \), \( \varphi'' \equiv \varphi_1 \land \varphi_2 \),

\[
\varphi_1 \equiv x' = w_1 \land w_1 = a \land z = a \land x' = x \land z' = a \land u_1 = f(q_1), \\
\varphi_2 \equiv z' = w_2 \land w_2 = a \land z = a \land x' = x \land z' = a \land v_2 = f(q_2).
\]

141
Therefore, we get that $p(x), p(z), t(u_1, v_1) : \psi' \in S_P(I)$ and $p(x), p(z), t(u_1, v_1), t(u_2, v_2) : \psi'' \in S_P(I)$, where $\psi' \equiv \exists x', z', w_1, w_2, q_1, q_2.(\phi') \equiv x = a \land z = a \land u_1 = f(q_1) \land v_1 = f(q_2)$ and $\psi'' \equiv \exists x', z', w_1, w_2, q_1, q_2.(\phi'') \equiv x = a \land z = a \land u_1 = f(q_1) \land v_2 = f(q_2)$. Summarizing, $S_P(I)$ contains the two elements

$$p(x), p(z), t(u_1, v_1) : x = a \land z = a \land u_1 = f(q_1) \land v_1 = f(q_2)$$
$$p(x), p(z), t(u_1, v_1), t(u_2, v_2) : x = a \land z = a \land u_1 = f(q_1) \land v_2 = f(q_2)$$

By considering the ground instances of the above constrained multisets, we have that $[S_P(I)]$ contains, e.g., $p(a), p(a), t(f(a), f(a))$ and $p(a), p(a), t(f(a), a), t(a, f(a))$. Note that different ways of choosing the sub-multisets $D_1$ and $D_2$ in Definition 9.28 yield different (and not redundant) elements.

\[\square\]

### 9.3 Ensuring Termination

In this section we present a fragment of LO, enriched with linear constraints over integer variables, for which termination of the bottom-up evaluation algorithm presented in Section 9.2 is guaranteed. We will exploit this result in Section 9.4. Specifically, we introduce the class of programs with monadic predicate symbols, over the constraint system NC of definition 9.5, and the corresponding class of constrained multisets. The results of this section appeared in [BD02]. We note, however, that some results presented in [BD02], specifically the ones concerning programs with predicate symbols with arity greater than one, were incorrect. Suitable further restrictions, which make the results of [BD02] sound, are currently under investigation.

**Definition 9.42 (The $\mathcal{P}_1(\text{NC})$ class)** The class $\mathcal{P}_1(\text{NC})$ consists of LO(\text{NC}) programs such that every predicate symbol has arity at most one.

**Definition 9.43 (Constrained Multisets $\mathcal{M}_1(\text{NC})$)** The class $\mathcal{M}_1(\text{NC})$ consists of constrained multisets with predicate symbols with arity at most one, annotated with NC constraints.

**Example 9.44** Let $\Pi = \{p, q, r, s\}$, with $p, q$ having arity one and $r, s$ arity zero. Let $\mathcal{V}$ be a denumerable set of variables, and $x, y, \ldots \in \mathcal{V}$, Then the clause

$$r \, \exists \, p(x) \, \exists \, q(y) \, \leftarrow \, s \, \exists \, p(x') \, \exists \, q(y') \, \square \, y = x \land x' = x \land y' > y$$

is in the class $\mathcal{P}_1(\text{NC})$, and the constrained multiset $r, p(x), q(y), q(z) : x = y \land z > y$ is in the class $\mathcal{M}_1(\text{NC})$.

\[\square\]
The following result holds.

**Proposition 9.45** The class \( M_1(\text{NC}) \) is closed under applications of \( S_P \), i.e., if \( I \subseteq M_1(\text{NC}) \) then \( S_P(I) \subseteq M_1(\text{NC}) \).

**Proof** Immediate by Definition 9.34 and Definition 9.28. Note that NC constraints are closed with respect to existential quantification. \( \square \)

The other property we shall prove is that there exists an entailment relation for constrained multisets that ensures the termination of the fixpoint computation for the bottom-up semantics. To improve the presentation, we will prove this property in two steps.

**Entailment for the class \( M_1(\text{EC}) \)**

We first consider the subclass \( M_1(\text{EC}) \) of constrained multisets annotated with equality constraints only (see Definition 9.6), i.e. without >-constraints. Without loss of generality, we assume hereafter to deal with a set of predicate symbols with arity one. If it is not the case, we can complete predicates with arity less than one with dummy variables.

**Definition 9.46** Let \( M : \varphi \) be a constrained multiset in \( M_1(\text{EC}) \). The symmetric and transitive closure of the relation \( = \) induces an equivalence relation \( \equiv \) on the variables in \( M : \varphi \). Let \( C_1, \ldots, C_r \) be the \( \equiv \)-equivalence classes for these variables. We define \( M_i \) as the multiset of predicate symbols having a variable \( x \in C_i \) as argument in \( M \). Furthermore, we define \( S(M : \varphi) \) as the multiset \( \{M_1, \ldots, M_r\} \).

**Example 9.47** Let \( M : \varphi \) be the constrained multiset

\[
p(x), q(y), p(z), q(t), r(u), q(v), r(w) : x = y \land x = z \land t = u \land v = w.
\]

The variables \( x, y, z, t, u, v, w \) can be partitioned into three clusters: \( C_1 = \{x, y, z\} \), \( C_2 = \{t, u\} \), and \( C_3 = \{v, w\} \). \( S(M : \varphi) \) is the multiset consisting of the elements \( M_1 = \{p, p, q\} \), \( M_2 = \{q, r\} \), and \( M_3 = \{q, r\} \), i.e. \( S(M : \varphi) = \{\{p, p, q\}, \{q, r\}, \{q, r\}\} \). \( \square \)

We introduce now the following ordering \( \sqsubseteq^S \) between multisets of clusters of predicate symbols.

**Definition 9.48** Let \( N : \psi \) and \( M : \varphi \) be two constrained multisets in \( M_1(\text{EC}) \), and let \( S(N : \psi) = \{N_1, N_2, \ldots, N_r\} \) and \( S(M : \varphi) = \{M_1, M_2, \ldots, M_k\} \). We write \( S(N : \psi) \sqsubseteq^S S(M : \varphi) \) if there exists an injective mapping \( h \) from \( \{1, \ldots, k\} \) to \( \{1, \ldots, r\} \) such that \( M_i \sqsubseteq N_{h(i)} \) for \( i : 1, \ldots, k \).
Example 9.49 As an example, \{\{p, p, p\}, \{t, t\}, \{q, q\}, \{r, r, r\}\} \subseteq S \{\{p, p\}, \{q\}, \{r, r\}\} by mapping \{p, p\} into \{p, p, p\} (in fact, \{p, p\} \not\subseteq \{p, p, p\}\), \{q\} into \{q, q\}, and \{r, r\} into \{r, r, r\}. \hfill \Box

The following property holds.

Proposition 9.50 Let \(\mathcal{N} : \psi\) and \(\mathcal{M} : \varphi\) be two constrained multiset in \(\mathcal{M}_1(\text{EC})\). Then \(S(\mathcal{N} : \psi) \sqsubseteq S(\mathcal{M} : \varphi)\) implies \([\mathcal{N} : \psi]\subseteq [\mathcal{M} : \varphi]\).

Proof Let \(S(\mathcal{N} : \psi) = \{N_1, N_2, \ldots, N_t\}\) and \(S(\mathcal{M} : \varphi) = \{M_1, M_2, \ldots, M_k\}\), and suppose \(S(\mathcal{N} : \psi) \sqsubseteq S(\mathcal{M} : \varphi)\). Thus, for every \(M_i\) there exists a distinct \(N_{h(i)}\) such that \(M_i \not\preceq N_{h(i)}\). If \(C \in [\mathcal{N} : \psi]\), then there exist \(\sigma \in \text{Sol}(\psi)\) and \(D\) s.t. then \(C = \sigma(\mathcal{N}) + D\). Since \(\sigma \in \text{Sol}(\psi)\), it follows that for \(i = 1, \ldots, r\) there exists \(v_i\) s.t. \(\sigma(x) = \sigma(y) = v_i\) for every pair of variables \(x\) and \(y\) in the \(i\)-th cluster of \(\mathcal{N} : \psi\), i.e., \(C = C_1 + \ldots + C_r + D\), where \(C_i = \{p_{i1}(v_i), p_{i2}(v_i), \ldots, p_{im}(v_i)\}\) and \(N_i = \{p_{i1}, p_{i2}, \ldots, p_{im}\}\) for \(i = 1, \ldots, r\).

Let us now consider the multiset \(E = E_1 + \ldots + E_k\) where \(E_i = \{q_{i1}(v_{h(i)}), q_{i2}(v_{h(i)}), \ldots, q_{iz_i}(v_{h(i)})\}\), where \(M_i = \{q_{i1}, q_{i2}, \ldots, q_{iz_i}\}\) for \(i = 1, \ldots, k\). Then, by definition of \(S(\mathcal{M} : \varphi)\), \(E \in [\mathcal{M} : \varphi]\). Furthermore, since, by hypothesis, there exist \(N_{h(1)}, \ldots, N_{h(k)}\) s.t. \(M_i \not\preceq N_{h(i)}\) for \(i = 1, \ldots, k\), and from the injectivity of \(h\), it follows that \(E \not\preceq \sigma(\mathcal{N}) \not\preceq C\), hence \(C \in [\mathcal{M} : \varphi]\). \hfill \Box

Remark 9.51 Note that the condition stated in Proposition 9.50 is not necessary. For instance, consider the following counterexample: \(\mathcal{M} : \varphi \equiv p(x), q(y) : \text{true}\) and \(\mathcal{N} : \psi \equiv p(x'), q(y') : x' = y';\) clearly \([\mathcal{N} : \psi] \subseteq [\mathcal{M} : \varphi]\), but \(\mathcal{N} : \psi \sqsubseteq S \mathcal{M} : \varphi\) does not hold. In fact, we cannot find an injective mapping from \(\{\{p\}, \{q\}\}\), which contains two elements, into \(\{\{p, q\}\}\), which contains only one element.

Furthermore, we have the following property. In the following we use the notation \(\sqsupseteq S\) for the reverse of the relation \(\sqsubseteq S\), i.e. \(S \sqsupseteq S\) stands for \(T \sqsupseteq S\).

Lemma 9.52 The entailment relation \(\sqsupseteq S\) between constrained multisets in \(\mathcal{M}_1(\text{EC})\) is a well-quasi-ordering.

Proof The conclusion follows from Dickson’s Lemma (see Proposition 6.25), stating that the submultiset relation for multisets over a finite set is a wqo, and from Proposition 6.24 (iii). \hfill \Box
Entailment for the class $\mathcal{M}_1(\text{NC})$

Let us now deal with the case of $\triangleright$-constraints. First of all, we note that, given a set $S$ of constrained multisets in the class $\mathcal{M}_1(\text{NC})$, we can always construct a set $S'$ of elements of the same class, such that:

- for every $\mathcal{M} : \varphi$ in $S'$, every pair of variables $x, y$ occurring in $\varphi$ are related by an atomic constraint ($=\text{ or } \triangleright$) in $\varphi$;
- $[S] = [S']$.

The transformation of $S$ into $S'$ amounts to the completion of the constraints occurring in a constrained configuration with all the possible missing ones. Specifically, given a (satisfiable) constrained multiset $\mathcal{M} : \varphi$, we can transform it into the set of (satisfiable) constrained configurations obtained by completing the partial ordering induced by $\varphi$ on the variables in $\mathcal{M}$ into a total ordering (if $x$ is unrelated w.r.t. $y$ in $\varphi$, we consider all the possible completions given by the alternatives $x = y$, $x > y$, or $y > x$).

Given a set $S$ of constrained multisets in $\mathcal{M}_1(\text{NC})$, let us call $\text{Comp}(S)$ the operator that returns the (union of the sets of) completions of the elements in $S$. From now on, we will work using completed elements in the class $\mathcal{M}_1(\text{NC})$.

Let $\mathcal{M} : \varphi$ be a completed constrained multiset in $\mathcal{M}_1(\text{NC})$. As usual, the symmetric and transitive closure of the relation $=$ induces an equivalence relation $\equiv$ on the corresponding variables. Let $C_1, \ldots, C_r$ be the $\equiv$-equivalence classes for the variables of $\mathcal{M} : \varphi$.

We define $M_i$ as the multiset of predicate symbols having a variable $x \in C_i$ as argument in $\mathcal{M}$. Since variables are totally ordered w.r.t. $\triangleright$ in $\varphi$ (as we work with completed constrained multisets) and the constraint $\varphi$ is satisfiable, we can order the resulting clusters $M_1, \ldots, M_r$ into a string $M_i \cdot \ldots \cdot M_r$, such that

- if $p$ and $q$ occur in $M_i$, then for some variables $x$ and $y$, $p(x)$ and $q(y)$ occur in $\mathcal{M}$ and $x = y$ follows from $\varphi$;
- if $p$ occurs in $M_{ij}$ and $q$ occurs in $M_{ik}$, with $i_j > i_k$, then for some variables $x$ and $y$, $p(x)$ and $q(y)$ occur in $\mathcal{M}$ and $x > y$ follows from $\varphi$.

Based on this idea, we define $\text{Str}(\mathcal{M})$ as the string $M_{i_1} \cdot \ldots \cdot M_{i_r}$.

**Example 9.53** Let $\mathcal{M} : \varphi$ be the constrained multiset $p(x), q(y), p(z), q(t), r(u) : x > y \land y = z \land z > t \land t = u$.

The variables $x, y, z, t, u$ can be partitioned into three clusters $C_1 = \{t, u\}$, $C_2 = \{y, z\}$, and $C_3 = \{x\}$. $\text{Str}(\mathcal{M} : \varphi)$ is the string $\{q, r\} \cdot \{p, q\} \cdot \{p\}$.

\[\square\]
We introduce now the following ordering between multisets of clusters of predicate symbols.

**Definition 9.54** Let $N : \psi$ and $M : \varphi$ be two constrained multisets in $\mathcal{M}_1(\text{NC})$, and let $\text{Str}(N : \psi) = N_1 \cdot N_2 \cdot \ldots \cdot N_r$ and $\text{Str}(M : \varphi) = M_1 \cdot M_2 \cdot \ldots \cdot M_k$. We write $\text{Str}(N : \psi) \sqsubseteq^* \text{Str}(M : \varphi)$ iff there exists an injective mapping $h$ from $1, \ldots, k$ to $1, \ldots, r$ such that if $i < j$ then $h(i) < h(j)$ ($h$ is monotone), and $M_i \not\leq N_{h(i)}$ for $i : 1, \ldots, k$.

**Example 9.55** As an example, $\{p, p, p\} \cdot \{t, t\} \cdot \{q, q\} \cdot \{r, r, r\} \sqsubseteq^S \{p, p\} \cdot \{q\} \cdot \{r, r\}$ by mapping $\{p, p, p\}$ into $\{p, p, p\}$ (in fact, $\{p, p\} \not\leq \{p, p, p\}$), $\{q\}$ into $\{q, q\}$, and $\{r, r\}$ into $\{r, r, r\}$. On the contrary, $\{p, p, p\} \cdot \{t, t\} \cdot \{r, r, r\} \cdot \{q, q\} \sqsubseteq^S \{p, p\} \cdot \{q\} \cdot \{r, r\}$ does not hold (note that the previous mapping would not respect the monotonicity property).

The following property holds.

**Proposition 9.56** Let $N : \psi$ and $M : \varphi$ be two constrained multisets in $\mathcal{M}_1(\text{NC})$. Then $\text{Str}(N : \psi) \sqsubseteq^* \text{Str}(M : \varphi)$ implies $[N : \psi] \subseteq [M : \varphi]$.

**Proof** Let $\text{Str}(N : \psi) = N_1 \cdot N_2 \cdot \ldots \cdot N_r$ and $\text{Str}(M : \varphi) = M_1 \cdot M_2 \cdot \ldots \cdot M_k$, and suppose $\text{Str}(N : \psi) \sqsubseteq^* \text{Str}(M : \varphi)$. Thus, for every $M_i$ there exists a distinct $N_{h(i)}$ (with $h(i) < h(j)$ for $i < j$) such that $M_i \not\leq N_{h(i)}$. If $C \in [N : \psi]$, then there exist $\sigma \in \text{Sol}(\psi)$ and $D$ s.t. $C = \sigma(N) + D$. Since $\sigma \in \text{Sol}(\psi)$, it follows that for $i : 1, \ldots, r$ there exists $v_i$ s.t. $\sigma(x) = \sigma(y) = v_i$ for every pair of variables $x$ and $y$ in the $i$-th cluster of $N : \psi$, and $v_i > v_j$ for $i > j$. Thus, $C = C_1 + \ldots + C_r + D$, where $C_i = \{p_{i_1}(v_i), p_{i_2}(v_i), \ldots, p_{i_m}(v_i)\}$ and $N_i = \{p_{i_1}, p_{i_2}, \ldots, p_{i_m}\}$ for $i : 1, \ldots, r$, with $v_i > v_j$ if $i > j$.

Let us now consider the multiset $E = E_1 + \ldots + E_k$ where $E_i = \{q_{i_1}(v_{h(i)}), q_{i_2}(v_{h(i)}), \ldots, q_{i_z}(v_{h(i)})\}$ and $M_i = \{q_{i_1}, q_{i_2}, \ldots, q_{i_z}\}$ for $i : 1, \ldots, k$, with $v_i > v_j$ if $i > j$. Then, by definition of $\text{Str}(M : \varphi)$ and from the monotonicity property of $h$, we have that $E \in [M : \varphi]$. Furthermore, since, by hypothesis, there exists $N_{h(1)}, \ldots, N_{h(k)}$ such that $M_i \not\leq N_{h(i)}$ for $i : 1, \ldots, k$, and from the injectivity of $h$, it follows that $E \not\leq \sigma(N) \not\leq C$, hence $C \in [M : \varphi]$.

Furthermore, we have the following property. As usual, we denote by $\sqsupset^*$ the reverse of the relation $\sqsubseteq^*$.

**Lemma 9.57** The entailment relation $\sqsupset^*$ between completed constrained multisets in $\mathcal{M}_1(\text{NC})$ is a well-quasi-ordering.

**Proof** The conclusion follows from Dickson’s Lemma (see Proposition 6.25) and Proposition 6.24 iv.

146
We can therefore state the following result, which proves termination of the bottom-up evaluation for the class \( P_1(\text{NC}) \).

**Corollary 9.58** Let \( P \) be a \( P_1(\text{NC}) \) program. Then there exists \( k \in \mathbb{N} \) such that \( \mathcal{F}_{\text{sym}}(P) = \bigsqcup_{i=0}^{k} S_{P}^{\uparrow k}(\emptyset) \).

**Proof** By Propositions 9.45, 9.57, and 6.23.

**Remark 9.59** Concerning Corollary 9.58, note that the termination of bottom-up evaluation has been proved as soon as the backward reachability algorithm discussed in Section 9.2 is extended so as to maintain intermediate constrained multisets in completed form. However, as completion of constrained multisets can be computationally expensive, efficiency reasons suggest to use entailment tests like, e.g., the \( \sqsubseteq_m \) relation of Definition 9.22, which has been proved to be sound (though we have not directly proved termination with this ordering).

### 9.3.1 A Dynamic Abstraction from DC to NC

We conclude this section by discussing a *dynamic abstraction* (see Section 6.1.2.1) which can be used, in conjunction with the result given by Corollary 9.58, to automatically verify systems specified over the DC constraint system. An application of this abstraction will be presented in Section 9.4.2. The abstraction is as follows.

**Definition 9.60 (Abstraction from DC to NC)** Let \( \odot \in \{=,>\} \). The abstraction \( \alpha \) from DC-constraints to NC-constraints is defined as follows: \( \alpha(\text{true}) = \text{true} \), \( \alpha(\text{false}) = \text{false} \), \( \alpha(\varphi_1 \land \varphi_2) = \alpha(\varphi_1) \land \alpha(\varphi_2) \), and

\[
\alpha(x \odot y + c) = \begin{cases} 
  x \odot y & \text{if } c = 0 \\
  x > y & \text{if } c > 0 \\
  y > x & \text{if } c < 0 \text{ and } \odot \text{ is } \geq \\
  \text{true} & \text{otherwise}
\end{cases}
\]

The abstraction \( \alpha \) can be lifted to constrained multisets and interpretations as follows: \( \alpha(\mathcal{M} : \varphi) = \mathcal{M} : \alpha(\varphi) \), and \( \alpha(I) = \{ \alpha(\mathcal{M} : \varphi) \mid \mathcal{M} : \varphi \in I \} \).

Clearly, we have that \( \text{Sol}(\varphi) \subseteq \text{Sol}(\alpha(\varphi)) \) for every DC-constraint \( \varphi \), and, consequently, \( \text{Sol}(I) \subseteq \text{Sol}(\alpha(I)) \) for every interpretation \( I \).

We will see an example of application of this technique in Section 9.4.2.
We conclude this chapter with an example consisting in a multi-client, multi-server parameterized formulation of the classical ticket protocol. As usual, we have automatically validated the proposed examples using the verification tool of Appendix A.

The ticket protocol is a mutual exclusion protocol designed for multi-client systems operating on shared resources (e.g. memory). In order to access the critical section, every client executes the protocol, which is based on a first-in first-served access policy. A description of the protocol in an idealized high-level language is presented in Figure 9.2. We use $P \parallel Q$ to denote the interleaving parallel execution of the sub-programs $P$ and $Q$, and $\langle \cdot \rangle$ to denote atomic fragments of code. The initial number of clients is a parameter $n$ of the protocol. The protocol works as follows. Initially, all clients are thinking, while $t$ and $s$ store the same initial value. When requesting the access to the critical section, a client stores the value of the current ticket $t$ in its local variable $a$. A new ticket is then emitted by incrementing $t$. Clients wait for their turn until the value of their local variable $a$ equals the value of $s$. After the elaboration inside the critical section, a process releases it and the current turn is updated by incrementing $s$. During the execution, the global state of the protocol consists of the internal state (current value of the local variable) of $n$ processes together with the current value of $s$ and $t$. As remarked in [BGP97], even with a finite number of clients (e.g. for $n = 2$) the values of the local variables of individual processes as well as $s$ and $t$ may get unbounded. This implies that any instance (for fixed values of $n$) of the scheme of Figure 9.2 gives rise to an infinite-state system. Obviously, the algorithm is supposed to work for any value of $n$.

In Section 9.4.1 we will discuss different formulations of the ticket protocol, in particular a multi-client single-server version and a multi-client multi-server version. We model it using LO clauses enriched with difference constraints (see 9.4) allowing arithmetic operations like increment and decrement of data variables. This way, we can give a model which is faithful to the original formulation of Figure 9.2: we do not abstract away global and local integer variables attached to individual clients, that in fact can still grow unboundedly in our specification.

Validation of the ticket protocol will be discussed in Section 9.4.2. Specifically, we have defined an automated and conservative abstraction that can be used to attack verification problems for systems defined over linear integer constraints (like the DC-constraints above) that lay outside the NC class for which we have proved that termination of backward reach-
The system with \( n \) processes

Program

local var \( s, t : \text{integer} \)

begin

\( t := 0 \)
\( s := 0 \)
\( P_1 \mid \ldots \mid P_n \)

end

The \( i \)-th component

Process \( P_i := \)

local var \( a : \text{integer} \)

repeat

\( \text{think} : \langle a := t; t := t + 1 \rangle \)
\( \text{wait} : \text{when} \langle a = s \rangle \text{do} \)
\( \text{use} : \text{begin} \)

\( \text{critical section} \)
\( \langle s := s + 1 \rangle \)

end

forever

Figure 9.2: High-level description of the ticket protocol

ability is guaranteed. The abstraction maps linear integer constraints into NC-constraints, and it can be computed automatically on sets of constrained multisets. We will also discuss optimization of the fixpoint computation using pruning invariants.

9.4.1 Specifying the Ticket Protocol

In this section we present different formulations of the ticket protocol. First of all, we have encoded a multi-client, single-server version of the protocol, i.e., with an arbitrary but finite number of clients, fixed after an initialization phase. Translation into LO(DC) is shown in Figure 9.3. The (infinite) collection of admissible initial states consists of all configurations with an arbitrary but finite number of thinking processes and two counters having the same initial value (\( t = s \)). This set can be specified via rules 1 and 2 in Figure 9.3: as usual we define a predicate called \( \text{init} \) acting as the seed of all possible runs of the protocol. The counters are represented via the atoms \( \text{count}(t) \) and \( \text{turn}(s) \). Thinking clients are represented via the propositional symbol \( \text{think} \) (the value of the local variable of a client does not matter at this point). The behaviour of an individual client can be described via rules 3 through 5, which faithfully correspond to the high-level description of Figure 9.2. The relations between the local variable and the global counters are represented via DC-constraints. Finally, we allow thinking processes to terminate their execution as specified by rule 6. Note that the specification is independent of the number of clients, and that we keep an explicit representation of the data variables without putting any restrictions on their values.

The second version of the ticket protocol is a slight variation of the one presented in Figure 9.3. Namely, we allow dynamic creation of clients. The extended model is obtained by introducing a demon process whose only goal is to non deterministically generate new
Initial States

1. init ← think init □ true
2. init ← count(t) turn(s) □ t = s

Individual Behaviour

3. think count(t) ← wait(a) count(t') □ a = t ∧ t' = t + 1
4. wait(a) turn(s) ← use(a') turn(s') □ a = s ∧ a' = a ∧ s' = s
5. use(a) turn(s) ← think turn(s') □ s' = s + 1

Termination

6. think ← ⊥ □ true

Figure 9.3: A multi-client, single-server version of the ticket protocol

clients. The encoding is obtained by changing the first two rules in Figure 9.3 with the rules below (whereas the rules for individual processes remain unchanged):

Initial States and Dynamic Process Generation

1'. init ← count(t) turn(s) demon □ t = s
2'. demon ← demon think □ true

Finally, let us consider now a system with an arbitrary but finite number of shared resources, each one controlled by two local counters s and t. The idea is to associate a unique identifier to each resource and use it to stamp the corresponding pair of counters. The resulting specification is shown in Fig 9.4. We consider only the more interesting case in which clients and servers are generated dynamically. The process demon(n) maintains a local counter n used to generate a new identifier, say id, to associate to newly created resources. Resources are in turn represented via pairs of the form count(id, t) and turn(id, s). A thinking process can non-deterministically chooses which resource to wait for by synchronizing with one of the counters in the system (rule 4 in Figure 9.4). After this choice, the algorithm behaves as usual w.r.t. to the chosen resource. The termination rules can be specified as natural extensions of the single-server case. We show an example trace in Figure 9.5, where P is the program in Figure 9.4. Note that in this specification the sources of infiniteness are: the number of clients, the number of shared resources, the values of resource identifiers and the values of tickets.
Initial States

1. \( \text{init} \leftarrow \text{demon}(n) \not\square \text{true} \)

Dynamic Server and Process Generation

2. \( \text{demon}(n) \leftarrow \text{demon}(n') \not\square \text{count}(id, t) \not\square \text{turn}(id', s) \not\square n' = n + 1 \land t = s \land id = n \land id' = id \)

3. \( \text{demon}(n) \leftarrow \text{demon}(n') \not\square \text{think} \not\square n' = n \)

Individual Behaviour

4. \( \text{think} \not\square \text{count}(id, t) \leftarrow \text{think}(r) \not\square \text{count}(id', t') \not\square r = id \land id' = id \land t' = t \)

5. \( \text{think}(r) \not\square \text{count}(id, t) \leftarrow \text{wait}(r', a) \not\square \text{count}(id', t') \not\square r' = r \land id' = id \)

6. \( \text{wait}(r, a) \not\square \text{turn}(id, s) \leftarrow \text{use}(r', a') \not\square \text{turn}(id', s') \not\square r = id \land a = s \land a' = a \land s' = s \land r' = r \land id' = id \)

7. \( \text{use}(r, a) \not\square \text{turn}(id, s) \leftarrow \text{think} \not\square \text{turn}(id', s') \not\square r = id \land s' = s + 1 \land id' = id \)

Termination

8. \( \text{think}(r) \leftarrow \perp \not\square \text{true} \)

9. \( \text{think} \leftarrow \perp \not\square \text{true} \)

Figure 9.4: A multi-client, multi-server version of the ticket protocol

9.4.2 Verifying the Ticket Protocol

We have validated the different versions of the ticket protocol analyzed in Section 9.4.1, with respect to the mutual exclusion property, using the verification tool presented in Appendix A. Let us describe the verification process in detail.

Safety Properties. The set of violations to mutual exclusion can be represented through the following LO axioms:

\[ \text{use}(x) \not\square \text{use}(y) \leftarrow \top \not\square \text{true} \]

for the single-server formulation, and

\[ \text{use}(id, x) \not\square \text{use}(id', y) \leftarrow \top \not\square id = id' \]

for the multi-server formulation. In both cases, they denote all configurations with at least two clients in the critical section at the same time.
Figure 9.5: Multi-client ticket protocol: example trace

Relaxing Constraints. The bottom-up evaluation algorithm, when applied to the specifications in Figure 9.3 and Figure 9.4 and the above safety axioms, is not guaranteed to terminate. However, we can apply the abstraction technique described in Section 6.1.2.1 using the function $\alpha$ discussed in Section 9.3 (see Definition 9.60). In this way, termination is guaranteed for the specification given in Figure 9.3 and its variation based on dynamic process generation. In fact, the single-server specification of the ticket protocol of Figure 9.3 makes use of monadic predicate symbols, therefore using the abstraction $\alpha$ we have a verification problem covered by Corollary 9.58. Although the verification problem for the multi-server specification of Figure 9.4 is not covered by the results of Section 9.3, the symbolic backward reachability algorithm still terminates (see Table 9.1).

Pruning and Invariant Strengthening. Using the so-called counting abstraction, which simply forgets local data, while keeping track of the number of processes in a given state, we obtain models as expressive as Petri nets. Computing the structural invariants for the related Petri net model is not useful to prove properties that depend on the values attached to the tokens like mutual exclusion for the ticket protocol. However, these invariants can still be used in combination with the pruning technique of Section 6.1.2.2 in order to relieve the state explosion problem. Using a package for Petri net analysis, we can automatically obtain the invariants $x_{\text{init}} + x_{\text{count}} = 1$ and $x_{\text{init}} + x_{\text{turn}} = 1$, which imply that the number of tokens in places $\text{turn}$ and $\text{count}$ are always bounded by one. A similar reasoning can be applied to the multi-server protocol concerning the counters associated
Table 9.1: Validating the ticket protocol: experimental results

to a given identifier (at most one copy of each counter) and the demon process (at most one copy). We have used these information to prune the backward search space.

In an additional series of experiments, we have tested again the three models using the structural invariants to perform invariant strengthening (i.e., adding them to the set of unsafe states, see Section 6.1.2.1) instead of using them for pruning. This technique, so to say, has an effect which is similar to dynamic pruning (in fact, all constrained multisets that entail an invariant are discarded during the fixpoint computation, instead of being pruned).

Experimental Results. In Table 9.1, we show some experimental results. In column ‘α’, \(\sqrt{}\) indicates that the abstraction \(\alpha\) has been applied after each application of the symbolic predecessor operator; in column ‘Static Opt’ we have denoted by ‘Pruning’ and ‘Invar’ the use of structural invariants, for, respectively, pruning the search space or performing invariant strengthening (‘Pr/Invar’ denotes use of either technique); column ‘Steps’ contains the number of iterations needed to reach a fixpoint or before stopping the program (↑ indicates that the procedure was still computing after several hours); column ‘Size’ denotes the number of constrained multisets contained in the fixpoint (or when the program was stopped); finally, ‘Time’ is the execution time (in seconds). We refer to Appendix A.4 for details on the experimental environment. As shown in Table 9.1, using the abstract (theoretically always terminating) backward reachability algorithm we managed to prove all
safety properties we were interested in. Furthermore, we managed to prove mutual exclusion without using structural invariants for the first two models, whereas it was necessary to use them in the third example, in order to avoid the state explosion problem.

9.5 Related Work

In this chapter we have defined a bottom-up semantics for the language LO enriched with constraints, from which we have derived a fully automated and sound method to attack verification of parameterized systems with unbounded local data. We have exemplified our approach by proving mutual exclusion for a parameterized, multi-client and multi-server formulation of the ticket protocol. The method is powered by using static analysis techniques coming from the structural theory of Petri Nets and by automatic abstractions working on constraints.

This work is inspired to the approach of [AJ98, AN00]. In [AJ98], Abdulla and Jonsson proposed an assertional language for Timed Petri Nets in which they use dedicated data structures to symbolically represent markings parametric in the number of tokens and in the age (represented as a real number) associated to tokens. In [AN00], Abdulla and Nylén formulate a symbolic algorithm using existential regions to represent the state-space of Timed Petri Nets. Our approach is an attempt to generalize the ideas of [AJ98, AN00] to problems and constraint systems that do not depend on the notion of time. In [AJ01a], Abdulla and Jonsson have used similar techniques to prove termination for backward reachability of lossy/non lossy unordered channel systems (i.e., finite-state control, unbounded channels) in which messages can vary over an infinite name domain.

For networks of finite-state processes, it is important to mention the automata theoretic approach to parameterized verification followed, e.g., in [KMM+97b, ABJN99, BBLS00, JN00, Nil00, PS00]. In this setting, the set of possible local states of individual processes are abstracted into a finite alphabet, whereas sets of global states are represented as networks parameterized by regular languages. Finite-state transducers are then applied to compute sets of predecessors. Manipulations on regular sets can be performed automatically by using tools like Mona [HJJ+95] or MoSel [KMM+97a]. In [KMM+97b], these ideas are applied to networks of processes arranged into an array topology, and specified in a second-order logic derived from WS1S (weak second order logic of one successor [Tho90]). An extension to tree topologies is considered. A similar approach is followed in [BBLS00], where symbolic exploration is performed by means of automated abstractions techniques. Specifically, networks of processes are specified in the logic WS1S and then abstracted into a finite-state system which can be model-checked. The approach can be used to verify a class of liveness properties. In [ABJN99], the focus is on model-checking of parameterized systems whose actions are specified by means of local transitions and global transitions.
Symbolic exploration is made effective by using operations over automata with ad hoc accelerations. Specifically, a method is devised to compute the result of applying a given action an arbitrary number of times. A general method for deriving the transitive closure for a large class of actions, improving the approach of [ABJN99], is presented in [JN00]. As a difference with [BBLS00], this method only works for safety properties. The construction of [ABJN99, JN00] is generalized in [PS00], where a model-checking procedure is presented, which can be used to verify liveness properties as well. Finally, a similar automata theoretic approach, called regular model checking, is presented in [Nil00] (see also [BJNT00]). It makes use of regular sets to represent both sets of states and transition relations, and it is usable for a class of liveness properties.

Though limited to studying safety properties, our approach has some advantages over the automata theoretic approaches described above. In particular, differently from the automata theoretic approach, in our setting we handle parameterized systems in which individual components have local variables that range over unbounded values. As an example, in our model of the ticket protocol local variables and tickets range over unbounded integers. Furthermore, note that the abstraction from DC to NC-constraints does not “finitize” the resulting symbolic representation, though termination can be guaranteed by applying the theory of well-quasi-orderings. This way, we do not have to apply manual abstractions to describe individual processes. This is an important aspect to take into account when comparing our practical results for the single-server system (the first experiment of Section 9.4.1) with those obtained in [JN00, Nil00], in which an idealized version of the ticket algorithm has been automatically verified using the regular model checking method (actually, a precise comparison is difficult here because the verified model is not described in [JN00, Nil00]).

The previous features also distinguish our approach from the verification with invisible invariants method of [PRZ01, APR+01]. The method of [PRZ01, APR+01] uses heuristics, based on model-checking finite instances of a parameterized system, to automatically guess induction invariants. Invariants computed in this way are then verified by checking they hold for a problem-dependent finite dimension of the parameterized system (a theorem is provided, working for some classes of invariants, which ensures that the invariants hold for every possible dimension). As an example, invisible invariants have been applied to automatically verify a restricted version of the parameterized bakery algorithm in which a special reducing process is needed to force the value of the tickets to stay within a given range (finite, though with parametric bound). A related approach is the one given in [FPP01], where the authors use constraint logic programming over the WSkS logic (weak second order logic of k successors) and folding/unfolding program transformations, to prove mutual exclusion for a parameterized version of the bakery algorithm.

As mentioned in the previous sections, a formulation of the single-server ticket protocol with two processes, but unbounded global and local variables, has been automatically
verified using constraint-based model checkers equipped with Presburger constraint solvers [BGP97], or real arithmetic constraint solvers [DP99]. We are not aware, however, of methods that can automatically handle the parameterized, multi-server and multi-client model of the ticket protocol presented in Section 9.4.1.

Our ideas are related to previous works connecting Constraint Logic Programming and verification, see e.g. [DP99, Fri99]. In this setting, transition systems are encoded via CLP programs used to encode the global state of a system and its updates. In our approach, we refine this idea by using multiset rewriting and constraints to locally specify updates to the global state. The notion of constrained multiset naturally extends the notion of constrained atom of [DP99]. The locality in the representation of rules allows us to consider rich denotations (upward-closures) instead of flat ones (instances) like, e.g., in [DP99]. This way, we can lift the approach to the parameterized case.

Finally, an alternative approach to verification is based on theorem proving, e.g. by using induction and invariant generation. We refer to Section 10.6 for a detailed discussion.

---

**Summary of the Chapter.** In this chapter we have presented an extension of LO with constraints. This extension is inspired by traditional constraint programming languages, and is suitable to specify systems which use data structures with values ranging over heterogeneous domains. Validation can then be performed with the help of specialized constraint solvers. We have discussed the monadic fragment of LO, enriched with a subclass of linear integer constraints, for which we have proved termination of the bottom-up evaluation algorithm. We have also proved mutual exclusion for a parameterized, multi-client and multi-server formulation of the well-known ticket protocol.

In the next chapter we will extend the language LO in another direction, namely we will consider a first-order formulation of the logic, with universal quantification in goals providing a way to generate new values.
Chapter 10

Bottom-Up Evaluation of First-Order LO Programs

In this chapter we will discuss a first-order formulation of LO which can be seen as a particular case of the construction shown in Chapter 9, where the constraint system at hand is the Herbrand constraint system \(HC\) of Definition 9.2. The first reason why we chose to deal with this fragment separately, is that the bottom-up evaluation construction of Chapter 9 can be specialized to the case of the Herbrand constraint system. Similarly to traditional logic programming, the specialized construction is based on notions like Herbrand interpretations, substitutions, and most general unifiers. The second reason for which we are discussing this fragment in a separate chapter, is that we want to allow an interesting extension, namely we want to extend LO programs with universal quantification in goals.

Universal quantification can be thought of as a primitive for dynamically introducing new names during the computation. To understand the point, consider the specification of the ticket protocol given in Figure 9.4, and in particular clause 2, given below.

2. \(\text{demon}(n) \leftarrow \text{demon}(n') \otimes \text{count}(id, t) \otimes \text{turn}(id', s)\) \(\square n' = n + 1 \land t = s \land id = n \land id' = id\)

Here, we have used a demon\((n)\) process, where \(n\) works as a counter, to generate new identifiers for stamping pairs of count/turn processes. Given that we do not care about the actual value count/turn processes are stamped with (we only require that different pairs are associated with different values) we could achieve the same effect by exploiting the operational semantics of the universal quantifier (see Section 3.3.2). In the specific case, the only thing we have to do is to quantify over the variable \(id\) and use it to stamp both the count and the turn atoms (the counter \(n\) is not needed anymore). An example of specification showing the use of the universal quantifier will be presented in Section
10.5. Clearly, the connection between the bottom-up semantics for the fragment $\text{LO}_\forall$ and verification of parameterized systems is the same as in Chapter 9. In particular, we will show that the fragment $\text{LO}_\forall$ is suitable to reason about application domains like security protocols (this topic will be discussed in Chapter 11). In fact, the logical fragment considered in this chapter has been inspired by multiset rewriting with universal quantification (see Section 5.4) and [CDL+99], which discusses an equivalent fragment where existential quantification is used as a way to generate nonces in security protocols.

Technically, as in the previous chapters our aim is to define a bottom-up procedure to compute all goal formulas which are provable from a given program. Needless to say, the decidability property which we were able to prove for propositional $\text{LO}$ is now lost. Nevertheless, we will still be able to define an effective, symbolic fixpoint operator for which every single step can be finitely computed, and we will able to prove soundness and completeness of this operator with respect to the operational semantics. Furthermore, re-using the results of Section 9.3, we will state termination of the bottom-up fixpoint computation algorithm for the class of monadic $\text{LO}$ programs with universal quantification. In this chapter we will deal with the fragment $\text{LO}_\forall$ (see Section 3.3.2), i.e., a first-order formula comprising the logical connectives $\forall, \land, \lor, \top, \bot$, and universal quantification over goals.

As usual, in order to ease the proof of soundness and completeness, the definition of the bottom-up semantics will be presented in two steps. More precisely, we first present a simple, non-effective notion of (concrete) interpretation and the corresponding definition of fixpoint operator, which we call $T_P$. We then present an extended notion of (abstract) interpretation, and we define a symbolic and effective version of the fixpoint operator, called $S_P$. Owing to the presence of universal quantification, the semantic definition must carefully take into account the fact that program signatures can dynamically grow. As usual, for the sake of simplicity, we re-use some notations like, e.g., the ones for fixpoint operators and judgments.

### 10.1 A Proof-system for $\text{LO}_\forall$

**Some notations.** Given an $\text{LO}_\forall$ program $P$, we denote by $\Sigma_P$ the signature comprising the set of constant, function, and predicate symbols in $P$. We assume to have an infinite set $\mathcal{V}$ of variable symbols (usually noted $x, y, z, \ldots$). In order to deal with signature augmentation (due to the presence of universal quantification over goals) we also need an infinite set $E$ of new constants (called eigenvariables). We denote by $\text{Sig}_P$ the set of signatures which comprise at least the symbols in $\Sigma_P$ (and possibly some eigenvariables). $T^\mathcal{V}_E$ denotes the set of non ground terms over $\Sigma$ and $A^\mathcal{V}_E$ the set of non ground atoms over $\Sigma$. Similarly to Chapter 7, multisets of atoms over $A^\mathcal{V}_E$ will be called facts, and usually noted
We will also overload the usual notation for sets to indicate multisets. The notation $B \succ A$ will stand for $A \preceq B$. The notations for multisets of formulas (contexts) are the same as in Chapter 7. Some notions concerning substitutions and the definition of most general unifiers of multisets have been given in Section 2.4.

Before discussing the bottom-up semantics, we need to lift the definition of operational semantics to LO$_\forall$ programs. One possible solution is to resort to Definition 7.1 by considering the so-called ground instances of first-order programs. However, in presence of universal quantification in goals, this solution is not completely satisfactory. Consider, in fact, the following example. Take a signature with a predicate symbol $p$ and two constants $a$ and $b$, and consider the LO$_\forall$ program consisting of the axiom $\forall x.p(x) \leftarrow \top$ and the program consisting of the two axioms $p(a) \leftarrow \top$ and $p(b) \leftarrow \top$. The two programs have the same ground semantics according to Definition 7.1, i.e., $\{p(a), p(b)\}$. However, the LO$_\forall$ goal $\forall x.p(x)$ succeeds only in the first one, as the reader can verify. In order to distinguish the two programs, one possibility could be to consider the eigenvariables as part of the signature on which the ground semantics is computed. However, in this thesis we take an alternative approach, namely we consider the so-called non ground semantics. In particular, our aim in this chapter will be to extend the so-called C-semantics of [FLMP93, BGLM94] (see also Section 2.5) to first-order LO.

First of all, by analogy with Definition 9.33 we give the following definition.

**Definition 10.1 (Clause Variants)** Given an LO$_\forall$ program $P$, the set of variants of clauses in $P$, denoted $\text{Vrn}(P)$, is defined as follows:

$$\text{Vrn}(P) = \{(H \leftarrow G) \theta \mid \forall (H \leftarrow G) \in P \text{ and } \theta \text{ is a renaming of the variables in } \text{FV}(H \leftarrow G) \text{ with new variables}\}$$

Now, we need to reformulate the proof-theoretical semantics of Section 3.3.2 (see Figure 3.6). According to the C-semantics of [FLMP93, BGLM94], our goal is to define the set of non ground goals which are provable from a given program $P$ with an empty answer substitution. Slightly departing from [FLMP93, BGLM94], we define the proof system presented in Figure 10.1. This proof system is based on the idea of considering a first order program as the (generally infinite) collection of (non ground) instances of its clauses. By instance of a clause $H \leftarrow G$, we mean a clause $H\theta \leftarrow G\theta$, where $\theta$ is any substitution. The reader can see that, with this intuition, the set of goals provable from the system presented in Figure 10.1 correspond to the set of non ground goals which are provable with an empty answer substitution according to [FLMP93, BGLM94]. We remark that in this proof system there is no notion of unification. This formulation of the proof system is the proof-theoretical counterpart of the bottom-up semantics we will define in Section 10.2.

All formulas (and also substitutions) on the right-hand side of sequents in Figure 10.1 are implicitly assumed to range over the set of non ground terms over $\Sigma$. Rule $\forall_r$ is responsible
for signature augmentation. Every time rule $\forall_r$ is fired, a new constant $c$ is added to the current signature, and the resulting goal is proved in the new signature. This behaviour is standard in logic programming languages [MNPS91]. Rule $bc$ denotes a backchaining (resolution) step, where $\theta$ indicates any substitution. For our purposes, we can assume $\text{Dom}(\theta) \subseteq \text{FV}(H) \cup \text{FV}(G)$ (we remind that $\text{FV}(F)$ denotes the free variables of $F$). Note that $H \vdash G$ is assumed to be a variant, therefore it has no variables in common with $A$. According to the usual concept of uniformity, $bc$ can be executed only if the right-hand side of the current sequent consists of atomic formulas. Rules $\top_r$, $\forall_r$, $\&_r$ and $\bot_r$ are the same as in propositional LO. A sequent is provable if all branches of its proof tree terminate with instances of the $\top_r$ axiom.

Clearly, the proof system of Figure 10.1 is not effective, however it will be sufficient for our purposes. An effective way to compute the set of goals which are provable from the above proof system will be discussed in Section 10.3.

We give the following definition, where $\vdash_{\Sigma}$ is the provability relation defined by the proof system in Figure 10.1.

**Definition 10.2 (Operational Semantics)** Given an LO$_\forall$ program $P$, its operational semantics, denoted $O(P)$, is given by

$$O(P) = \{ A \mid A \text{ is a multiset of (non ground) atoms in } A^V_{\Sigma_p} \text{ and } P \vdash_{\Sigma_p} A \}.$$  

Intuitively, the set $O(P)$ is closed by instantiation, i.e., $A\theta \in O(P)$ for any substitution $\theta$, provided $A \in O(P)$. Note that the operational semantics only include multisets of (non ground) atoms, therefore no connective (including the universal quantifier) can appear in the set $O(P)$. However, the intuition will be that the variables appearing in a multiset in $O(P)$ must be implicitly considered universally quantified (e.g., $\{p(x), q(x)\} \in O(P)$ implies that the goal $\forall x.(p(x) \not\sqsubseteq q(x))$ is provable from $P$). As usual, the idea is that provability of a compound goal can always be reduced to provability of a finite set of atomic multisets.
10.2 A Bottom-Up Semantics for LO∀

We will now discuss the bottom-up semantics. In presence of universal quantification and therefore signature augmentation, we need to extend the definition of Herbrand base and (concrete) interpretations as follows. Namely, the definition of Herbrand base now depends explicitly on the signature, and interpretations can be thought of as infinite tuples, with one element for every signature Σ ∈ SigP. We give the following definitions.

**Definition 10.3 (Herbrand Base)** Given an LO∀ program P and a signature Σ ∈ SigP, the Herbrand base of P over Σ, denoted HBΣ(P), is given by
\[ HBΣ(P) = MS(\mathcal{A}^Σ_P) = \{ \mathcal{A} | \mathcal{A} \text{ is a multiset of (non ground) atoms in } \mathcal{A}^Σ_P \}. \]

**Definition 10.4 (Interpretations)** Given an LO∀ program P, a (concrete) interpretation is a family of sets \( \{I_Σ\}_{Σ ∈ SigP} \), where \( I_Σ ∈ \mathcal{P}(HBΣ(P)) \) for every \( Σ ∈ SigP \).

In the following we often use the notation I for an interpretation to denote the family \( \{I_Σ\}_{Σ ∈ SigP} \).

Interpretations form a complete lattice where inclusion and least upper bound are defined like (component-wise) set inclusion and union. In the following definition we therefore overload the symbols \( ⊆ \) and \( ∪ \) for sets.

**Definition 10.5 (Interpretation Domain)** Interpretations form a complete lattice \( ⟨D, ⊆⟩ \), where:

- \( D = \{ I | I \text{ is an interpretation} \} \);
- \( I ⊆ J \) if \( I_Σ ⊆ J_Σ \) for every \( Σ ∈ SigP \);
- the least upper bound of \( I \) and \( J \) is \( \{I_Σ ∪ J_Σ\}_{Σ ∈ SigP} \);
- the bottom and top elements are \( ∅ = \{∅_Σ\}_{Σ ∈ SigP} \) and \( \{HBΣ(P)\}_{Σ ∈ SigP} \), respectively.

Before introducing the definition of fixpoint operator, we need to define the notion of satisfiability of a context ∆ in a given interpretation I. For this purpose, we introduce the judgment \( I \models_Σ Δ ▷ C \), where I is an interpretation, ∆ is a context, and C is an output fact. The judgment is also parametric with respect to a given signature Σ. The parameter C must be thought of as an output fact such that \( C + Δ \) is valid in I. The notion of output fact will simplify the presentation of the algorithmic version of the judgment which we will present in Section 10.3.
Remark 10.6 In what follows, using the notation $I \vdash_{\Sigma} \Delta \triangleright C$ we always make the implicit assumption that $\Delta$ is a context defined over $\Sigma$ (i.e., terms constructors in $\Delta$ must belong to $\Sigma$). As a result, also the output fact $C$ must be defined over $\Sigma$. This assumption, which is the counterpart of an analogous assumption for proof systems like the one in Figure 3.6, i.e., with explicit signature notation, will always and tacitly hold in the following. For example, note that in the $\forall$-case of the $\vdash_{\Sigma}$ definition below, the newly introduced constant $c$ cannot be exported through the output fact $C$. This is crucial to capture the operational semantics of the universal quantifier.

As usual, the notion of satisfiability is modeled according to the right-introduction (decomposition) rules of the proof system, as follows.

Definition 10.7 (Satisfiability Judgment) Let $P$ be an LO$_\forall$ program, $\Sigma \in \text{Sig}_{p}$, and $I = \{I_{\Sigma}\}_{\Sigma \in \text{Sig}_{p}}$ an interpretation. The satisfiability judgment $\vdash_{\Sigma}$ is defined as follows:

$I \vdash_{\Sigma} \top, \Delta \triangleright C$ for any fact $C$ in $A_{\Sigma}^{\text{V}}$;
$I \vdash_{\Sigma} A \triangleright C$ if $A + C \in I_{\Sigma}$;
$I \vdash_{\Sigma} \forall x.G, \Delta \triangleright C$ if $I \vdash_{\Sigma,c} G[c/x], \Delta \triangleright C$, with $c \notin \Sigma$ (see remark 10.6);
$I \vdash_{\Sigma} G_{1} \& G_{2}, \Delta \triangleright C$ if $I \vdash_{\Sigma} G_{1}, \Delta \triangleright C$, $I \vdash_{\Sigma} G_{2}, \Delta \triangleright C$;
$I \vdash_{\Sigma} G_{1} \&\& G_{2}, \Delta \triangleright C$ if $I \vdash_{\Sigma} G_{1}, G_{2}, \Delta \triangleright C$;
$I \vdash_{\Sigma} \bot, \Delta \triangleright C$ if $I \vdash_{\Sigma} \Delta \triangleright C$.

The satisfiability judgment $\vdash_{\Sigma}$ satisfies the following properties.

Lemma 10.8 For every interpretation $I = \{I_{\Sigma}\}_{\Sigma \in \text{Sig}_{p}}$, context $\Delta$, and fact $C$,

$I \vdash_{\Sigma} \Delta \triangleright C$ iff $I \vdash_{\Sigma} \Delta, C \triangleright \epsilon$.

Proof If part. By induction on the derivation of $I \vdash_{\Sigma} \Delta, C \triangleright \epsilon$.

- If $\Delta = \top$, $\Delta'$, obvious;

- if $\Delta = A$ and $A + C \in I_{\Sigma}$, then also $I \vdash_{\Sigma} A \triangleright C$ holds;

- if $\Delta = \forall x.G, \Delta'$ and $I \vdash_{\Sigma,c} G[c/x], \Delta', C \triangleright \epsilon$, with $c \notin \Sigma$, then by inductive hypothesis
$I \vdash_{\Sigma,c} G[c/x], \Delta' \triangleright C$, which implies $I \vdash_{\Sigma} \forall x.G, \Delta' \triangleright C$;

- if $\Delta = G_{1} \& G_{2}, \Delta'$, $I \vdash_{\Sigma} G_{1}, \Delta', C \triangleright \epsilon$ and $I \vdash_{\Sigma} G_{2}, \Delta', C \triangleright \epsilon$, by inductive hypothesis
$I \vdash_{\Sigma} G_{1}, \Delta' \triangleright C$ and $I \vdash_{\Sigma} G_{2}, \Delta' \triangleright C$, which implies $I \vdash_{\Sigma} G_{1} \& G_{2}, \Delta' \triangleright C$;
Only if part. By induction on the derivation of $I \models_\Sigma \Delta \triangleright C$.

- If $\Delta = \top, \Delta'$, obvious;
- if $\Delta = A$ and $A + C \in I_\Sigma$, then also $I \models_\Sigma A, C \triangleright \epsilon$ holds;
- if $\Delta = \forall x.G, \Delta'$ and $I \models_\Sigma \forall c.G[c/x], \Delta' \triangleright C$, with $c \notin \Sigma$, then by inductive hypothesis $I \models_\Sigma \forall c.G[c/x], \Delta', C \triangleright \epsilon$, which implies $I \models_\Sigma \forall x.G, \Delta', C \triangleright \epsilon$;
- if $\Delta = G_1 \& G_2, \Delta'$, $I \models_\Sigma G_1, \Delta' \triangleright C$ and $I \models_\Sigma G_2, \Delta' \triangleright C$, by inductive hypothesis $I \models_\Sigma G_1, \Delta', C \triangleright \epsilon$ and $I \models_\Sigma G_2, \Delta', C \triangleright \epsilon$, which implies $I \models_\Sigma G_1 \& G_2, \Delta', C \triangleright \epsilon$;
- if $\Delta = G_1 \& G_2, \Delta'$ or $\Delta = \bot, \Delta'$, the conclusion follows by a straightforward application of the inductive hypothesis.

\[ \square \]

Lemma 10.9  For any interpretations $I_1 = \{(I_1)_\Sigma\}_{\Sigma \in Sig}, I_2 = \{(I_2)_\Sigma\}_{\Sigma \in Sig}, \ldots$, context $\Delta$, and fact $C$,

i. if $I_1 \subseteq I_2$ and $I_1 \models_\Sigma \Delta \triangleright C$ then $I_2 \models_\Sigma \Delta \triangleright C$;

ii. if $I_1 \subseteq I_2 \subseteq \ldots$ and $\bigcup_{i=1}^{\infty} I_i \models_\Sigma \Delta \triangleright C$ then there exists $k \in \mathbb{N}$ s.t. $I_k \models_\Sigma \Delta \triangleright C$.

Proof

i. By induction on the derivation of $I_1 \models_\Sigma \Delta \triangleright C$.

- If $\Delta = \top, \Delta'$, obvious;
- if $\Delta = A$ and $A + C \in (I_1)_\Sigma$, then $A + C \in (I_2)_\Sigma$, because $I_1 \subseteq I_2$, therefore $I_2 \models_\Sigma A \triangleright C$;
- if $\Delta = \forall x.G, \Delta'$ and $I_1 \models_\Sigma \forall c.G[c/x], \Delta' \triangleright C$, with $c \notin \Sigma$, then by inductive hypothesis $I_2 \models_\Sigma \forall c.G[c/x], \Delta' \triangleright C$, which implies $I_2 \models_\Sigma \forall x.G, \Delta' \triangleright C$;
- if $\Delta = G_1 \& G_2, \Delta'$, $I_1 \models_\Sigma G_1, \Delta' \triangleright C$ and $I_1 \models_\Sigma G_2, \Delta' \triangleright C$, by inductive hypotheses $I_2 \models_\Sigma G_1, \Delta' \triangleright C$ and $I_2 \models_\Sigma G_2, \Delta' \triangleright C$, which implies $I_2 \models_\Sigma G_1 \& G_2, \Delta' \triangleright C$;
- if $\Delta = G_1 \& G_2, \Delta'$ or $\Delta = \bot, \Delta'$, the conclusion follows by a straightforward application of the inductive hypothesis.
ii. By induction on the derivation of $\bigcup_{i=1}^{\infty} I_i \models_\Sigma \Delta \triangleright C$.

- If $\Delta = \top, \Delta'$, then for every $k \in \mathbb{N}$, $I_k \models_\Sigma \Delta \triangleright C$;
- if $\Delta = \forall x. G$ and $A + C \in (\bigcup_{i=1}^{\infty} I_i)_\Sigma$, there exists $k \in \mathbb{N}$ s.t. $A + C \in (I_k)_\Sigma$, i.e., $I_k \models_\Sigma A \triangleright C$;
- if $\Delta = \forall x. G', \Delta'$ and $\bigcup_{i=1}^{\infty} I_i \models_\Sigma, c G[c/x], \Delta' \triangleright C$, with $c \notin \Sigma$, then by inductive hypothesis there exists $k \in \mathbb{N}$ s.t. $I_k \models_\Sigma, c G[c/x], \Delta' \triangleright C$, therefore $I_k \models_\Sigma \forall x. G, \Delta' \triangleright C$;
- if $\Delta = G_1 \& G_2, \Delta'$, $\bigcup_{i=1}^{\infty} I_i \models_\Sigma G_1, \Delta' \triangleright C$ and $\bigcup_{i=1}^{\infty} I_i \models_\Sigma G_2, \Delta' \triangleright C$, by inductive hypothesis there exist $k_1, k_2 \in \mathbb{N}$ s.t. $I_{k_1} \models_\Sigma G_1, \Delta' \triangleright C$ and $I_{k_2} \models_\Sigma G_2, \Delta' \triangleright C$. By taking $k = \text{max}\{k_1, k_2\}$, by $i$ we get $I_k \models_\Sigma G_1, \Delta' \triangleright C$ and $I_k \models_\Sigma G_2, \Delta' \triangleright C$, which implies $I_k \models_\Sigma G_1 \& G_2, \Delta' \triangleright C$;
- if $\Delta = G_1 \not\supset G_2, \Delta'$ or $\Delta = \bot, \Delta'$, the conclusion follows by a straightforward application of the inductive hypothesis.

\[ \square \]

We are now ready to define the fixpoint operator $T_P$.

**Definition 10.10 (Fixpoint Operator $T_P$)** Given an LO$_\downarrow$ program $P$ and an interpretation $I = \{I_\Sigma\}_{\Sigma \in \text{Sig}_{\downarrow}}$, the fixpoint operator $T_P$ is defined as follows:

\[
T_P(I) = \{(T_P(I))_\Sigma\}_{\Sigma \in \text{Sig}_{\downarrow}} \\
(T_P(I))_\Sigma = \{H \theta + C \mid (H \circ G) \in \text{Vrn}(P), \theta \text{ is any substitution, and } I \models_\Sigma G \theta \triangleright C\}.
\]

**Remark 10.11** In the previous definition, $\theta$ is implicitly assumed to be defined over $\Sigma$, i.e., $\theta$ can only map variables in $\text{Dom}(\theta)$ to terms in $T_P(I)$.

The following property holds.

**Proposition 10.12 (Monotonicity and Continuity)** For every LO$_\downarrow$ program $P$, the fixpoint operator $T_P$ is monotonic and continuous over the lattice $\langle \mathcal{D}, \subseteq \rangle$.

**Proof** Monotonicity. Immediate from $T_P$ definition and Lemma 10.9 i.

Continuity. We prove that $T_P$ is finitary, i.e., for any increasing chain of interpretations $I_1 \subseteq I_2 \subseteq \ldots$ we have that $T_P(\bigcup_{i=1}^{\infty} I_i) \subseteq \bigcup_{i=1}^{\infty} T_P(I_i)$, i.e., for every $\Sigma \in \text{Sig}_{\downarrow}$, $(T_P(\bigcup_{i=1}^{\infty} I_i))_\Sigma \subseteq (\bigcup_{i=1}^{\infty} T_P(I_i))_\Sigma$. Let $A \in (T_P(\bigcup_{i=1}^{\infty} I_i))_\Sigma$. By $T_P$ definition, there exist $H \circ G$ variant of a clause in $P$, a substitution $\theta$ and a fact $C$ s.t. $A = \hat{H} \theta + C$ and $\bigcup_{i=1}^{\infty} I_i \models_\Sigma G \theta \triangleright C$. By Lemma 10.9 i, we have that there exists $k \in \mathbb{N}$ s.t. $I_k \models_\Sigma G \theta \triangleright C$. Again by $T_P$ definition, we get $A = \hat{H} \theta + C \in (T_P(I_k))_\Sigma \subseteq (\bigcup_{i=1}^{\infty} T_P(I_i))_\Sigma$. \[ \square \]
Monotonicity and continuity of the $T_P$ operator imply, by Tarski’s Theorem, that $\text{lfp}(T_P) = T_P \uparrow \omega$. The fixpoint semantics of a program $P$ is then defined as follows.

**Definition 10.13 (Fixpoint Semantics)** Given an LO$_\Sigma$ program $P$, its fixpoint semantics, denoted $F(P)$, is defined as follows:

$$F(P) = (\text{lfp}(T_P))\Sigma_P = (T_P\uparrow \omega(\{\emptyset\Sigma\}_{\Sigma \in \text{Sig}_P}))\Sigma_P.$$  

We conclude this section by proving the following fundamental result, which states that the fixpoint semantics is sound and complete with respect to the operational semantics.

**Theorem 10.14 (Soundness and Completeness)** For every LO$_\Sigma$ program $P$, $F(P) = O(P)$.

**Proof** $F(P) \subseteq O(P)$. We prove that for every $k \in \mathbb{N}$, for every signature $\Sigma \in \text{Sig}_P$, and for every context $\Delta$, $T_P \uparrow k \models \Sigma \Delta \triangleright \epsilon$ implies $P \vdash \Sigma \Delta$. The proof is by lexicographic induction on $(k, h)$, where $h$ is the length of the derivation of $T_P \uparrow k \models \Sigma \Delta \triangleright \epsilon$.

- If $\Delta = \top, \Delta'$, obvious;

- if $\Delta = \mathcal{A}$ and $\mathcal{A} \in (T_P \uparrow k)\Sigma$, then there exist a variant $H \triangleright G$ of a clause in $P$, a fact $\mathcal{C}$ and a substitution $\theta$ s.t. $\mathcal{A} = \bar{H}\theta + \mathcal{C}$ and $T_P \uparrow k-1 \models \Sigma G\theta \triangleright \mathcal{C}$. By Lemma 10.8, this implies $T_P \uparrow k-1 \models \Sigma G\theta, \mathcal{C} \triangleright \epsilon$. Then by inductive hypothesis we have $P \vdash \Sigma G\theta, \mathcal{C}$, from which $P \vdash \Sigma \bar{H}\theta, \mathcal{C}$, i.e., $P \vdash \Sigma \mathcal{A}$ follows by $b\mathcal{C}$ rule;

- if $\Delta = \forall x.G, \Delta'$ and $T_P \uparrow k \models \Sigma, x. G[c/x], \Delta' \triangleright \epsilon$, with $c \not\in \Sigma$, then by inductive hypothesis we have $P \vdash \Sigma, x. G[c/x], \Delta'$ from which $P \vdash \Sigma \forall x.G, \Delta'$ follows by $\forall x$ rule;

- if $\Delta = G_1 \& G_2, \Delta'$ and $T_P \uparrow k \models \Sigma G_1, \Delta' \triangleright \epsilon$, and $T_P \uparrow k \models \Sigma G_2, \Delta' \triangleright \epsilon$, then by inductive hypothesis we have $P \vdash \Sigma G_1, \Delta'$ and $P \vdash \Sigma G_2, \Delta'$, from which $P \vdash \Sigma G_1 \& G_2, \Delta'$ follows by $\&$ rule;

- if $\Delta = G_1 \not\triangleright G_2, \Delta'$ and $T_P \uparrow k \models \Sigma G_1, G_2, \Delta' \triangleright \epsilon$, then by inductive hypothesis we have $P \vdash \Sigma G_1, G_2, \Delta'$, from which $P \vdash \Sigma G_1 \not\triangleright G_2, \Delta'$ follows by $\not\triangleright$ rule;

- if $\Delta = \bot, \Delta'$ and $T_P \uparrow k \models \Sigma \Delta' \triangleright \epsilon$, then by inductive hypothesis we have $P \vdash \Sigma \Delta'$, from which $P \vdash \Sigma \bot, \Delta'$ follows by $\bot$ rule.

$O(P) \subseteq F(P)$. We prove that for every signature $\Sigma \in \text{Sig}_P$ and for every context $\Delta$, if $P \vdash \Sigma \Delta$ then there exists $k \in \mathbb{N}$ s.t. $T_P \uparrow k \models \Sigma \Delta \triangleright \epsilon$. The proof is by induction on the derivation of $P \vdash \Sigma \Delta$.  

165
- If $\Delta = \top, \Delta'$, then for every $k \in \mathbb{N}$, $T_P \uparrow_k \models_{\Sigma} \Delta \triangleright \epsilon$;

- if $\Delta = \hat{H} \theta, \mathbf{A}$, with $H \circ \neg G$ variant of a clause in $P$, $\theta$ substitution, and $P \vdash_{\Sigma} G \theta, \mathbf{A}$, then by inductive hypothesis we have that there exists $k \in \mathbb{N}$ s.t. $T_P \uparrow_k \models_{\Sigma} G \theta, \mathbf{A} \triangleright \epsilon$. Then, by Lemma 10.8, $T_P \uparrow_k \models_{\Sigma} G \theta, \mathbf{A} \triangleright \epsilon$. By $T_P$ definition, $\hat{H} \theta + \mathbf{A} \in (T_P \uparrow_{k+1})_{\Sigma}$, which implies $T_P \uparrow_{k+1} \models_{\Sigma} \hat{H} \theta + \mathbf{A} \triangleright \epsilon$;

- if $\Delta = \forall x. G, \Delta'$ and $P \vdash_{\Sigma, e} G[c/x], \Delta'$, with $e \not\in \Sigma$, then by inductive hypothesis we have that there exist $k \in \mathbb{N}$ s.t. $T_P \uparrow_k \models_{\Sigma, e} G[c/x], \Delta' \triangleright \epsilon$, from which $T_P \uparrow_k \models_{\Sigma} \forall x. G, \Delta' \triangleright \epsilon$ follows;

- if $\Delta = G_1 \& G_2, \Delta'$, $P \vdash_{\Sigma} G_1, \Delta'$ and $P \vdash_{\Sigma} G_2, \Delta'$, then by inductive hypothesis we have that there exist $k_1, k_2 \in \mathbb{N}$ s.t. $T_P \uparrow_{k_1} \models_{\Sigma} G_1, \Delta' \triangleright \epsilon$ and $T_P \uparrow_{k_2} \models_{\Sigma} G_2, \Delta' \triangleright \epsilon$. By taking $k = \max\{k_1, k_2\}$, by Lemma 10.9 i and monotonicity of $T_P$ we get $T_P \uparrow_k \models_{\Sigma} G_1, \Delta' \triangleright \epsilon$ and $T_P \uparrow_k \models_{\Sigma} G_2, \Delta' \triangleright \epsilon$, from which $T_P \uparrow_k \models_{\Sigma} G_1 \& G_2, \Delta' \triangleright \epsilon$ follows;

- if $\Delta = G_1 \nRightarrow G_2, \Delta'$ or $\Delta = \bot, \Delta'$, the conclusion follows by a straightforward application of the inductive hypothesis.

\[ \square \]

**Example 10.15** Let $\Sigma$ be a signature including the constant symbols $a$ and $b$, a function symbol $f$, and the predicate symbols $p, q, r$, let $\mathcal{V}$ be a denumerable set of variables, and let $P$ be the following following LO$_\mathcal{V}$ program:

1. $\exists f(b) \varnothing p(a) \circ \top$
2. $p(x) \circ \top$
3. $q(y) \circ (\forall x. p(x)) \& r(y)$

Let $I_0 = \{ \theta_{\Sigma} \}_{\Sigma \in Sig_p}$, and let us compute $I_1 = T_P(I_0)$. Using axioms 1 and 2, we get that $(I_1)_\Sigma$ contains the multisets of atoms of the form $\{ r(f(b)), p(a) \} + \mathbf{A}$, and $\{ p(t) \} + \mathbf{A}$, where $\mathbf{A}$ is any multiset of (possibly non-ground) atoms in $A_{\Sigma}^T$, while $t$ is any (possibly non ground) term in $T_{\Sigma}^V$. Similarly $(I_1)_{\Sigma'}$, for a generic signature $\Sigma'$ such that $\Sigma \subseteq \Sigma'$, contains all multisets of the above form where $\mathbf{A}$ and $t$ are taken from, respectively, $A_{\Sigma}^T$ and $T_{\Sigma'}^V$. For instance, let $c$ be a new constant not appearing in $\Sigma$. The set $(I_1)_{\Sigma}$ will contain, e.g., the multisets $\{ p(c) \}, \{ p(f(c)), q(b) \}$, and so on.

Now, consider the substitution $\theta = [y \mapsto f(b)]$ and the following corresponding instance of clause 3: $q(f(b)) \circ (\forall x. p(x)) \& r(f(b))$. Suppose we want to compute an output fact $C$ for the judgment 

$I_1 \models_{\Sigma} (\forall x. p(x)) \& r(f(b)) \triangleright C$. 

166
By \( \models \) definition, we have to compute \( I_1 \models (\forall x. p(x)) \uparrow \mathcal{C} \) and \( I_1 \models r(f(b)) \uparrow \mathcal{C} \). For the latter judgment we have that, e.g., \( I_1 \models r(f(b)) \uparrow p(a) \). For the first judgment, by \( \models \) definition, we must compute \( I_1 \models (\forall x. p(c)) \uparrow C \), where \( c \) is a new constant not in \( \Sigma \).

As \( \{p(c)\} \) is contained in \( (I_1)_{\Sigma,c} \), we can get that \( I_1 \models (\forall x. p(c)) \uparrow \epsilon \). We can also get \( I_1 \models (\forall x. p(c)) \uparrow p(a) \) (in fact \( \{p(c), p(a)\} \) is also contained in \( (I_1)_{\Sigma,c} \). By applying the \&-rule for \( \models \), we get that \( I_1 \models (\forall x. p(x)) \& r(f(b)) \uparrow p(a) \). Therefore, e.g., the multiset \( \{q(b), p(a)\} \) is in \( (I_2)_{\Sigma} = (TP(I_1))_{\Sigma} \). 

### 10.3 An Effective Semantics for LO\( _{\forall} \)

The fixpoint operator \( TP \) defined in the previous section does not enjoy one of the crucial properties we required for our bottom-up semantics, namely its definition is not effective. This is a result of both the definition of the satisfiability judgment (whose clause for \( \top \) is clearly not effective) and the definition of interpretations as infinite tuples. In order to solve these problems, we first define the (abstract) Herbrand base and (abstract) interpretations as follows.

**Definition 10.16 (Abstract Herbrand Base)** Given an LO\( _{\forall} \) program \( P \), the Herbrand base of \( P \), denoted \( HB(P) \), is given by 

\[
HB(P) = HB_{\Sigma_p}(P).
\]

**Definition 10.17 (Abstract Interpretations)** Given an LO\( _{\forall} \) program \( P \), an interpretation \( I \) is any subset of \( HB(P) \), i.e., \( I \in \mathcal{P}(HB(P)) \).

In order to define the abstract domain of interpretations, we need the following definitions.

**Definition 10.18 (Instance Operator)** Given an interpretation \( I \) and a signature \( \Sigma \in \text{Sig}_P \), we define the operator \( \text{Inst}_\Sigma \) as follows:

\[
\text{Inst}_\Sigma(I) = \{ A\theta \mid A \in I, \theta \text{ substitution over } \Sigma \}.
\]

**Definition 10.19 (Upward-closure Operator)** Given an interpretation \( I \) and a signature \( \Sigma \in \text{Sig}_P \), we define the operator \( \text{Up}_\Sigma \) as follows:

\[
\text{Up}_\Sigma(I) = \{ A + C \mid A \in I, \ C \text{ fact over } \Sigma \}.
\]

**Remark 10.20** Note that, as usual, in the previous definitions we assume the substitution \( \theta \) and the fact \( C \) to be defined over \( \Sigma \).
The following definition provides the connection between the (abstract) interpretations defined in Definition 10.17 and the (concrete) interpretations of Definition 10.4. The idea behind the definition is that an interpretation implicitly denotes the set of elements which can be obtained by either instantiating or closing upwards elements in the interpretation itself (where the concepts of instantiation and upward-closure are made precise by the above definitions). The operation of instantiation is related to the notion of C-semantics [FLMP93] (see Definition 10.2), while the operation of upward-closure is justified by Proposition 3.14. Note that the operations of instantiation and upward-closure are performed for every possible signature $\Sigma \in \text{Sig}_P$.

**Definition 10.21 (Denotation of an Interpretation)** Given an (abstract) interpretation $I$, its denotation $[I]$ is the (concrete) interpretation $\{[I]_\Sigma\}_{\Sigma \in \text{Sig}_P}$ defined as follows:

$$[I]_\Sigma = \text{Inst}_\Sigma(\text{Up}_\Sigma(I)) \quad \text{(or, equivalently, } [I]_\Sigma = \text{Up}_\Sigma(\text{Inst}_\Sigma(I))).$$

Two interpretations $I$ and $J$ are said to be equivalent, written $I \simeq J$, if and only if $[I] = [J]$.

The equivalence of the two different equations in Definition 10.21 is stated in the following proposition.

**Proposition 10.22** For every interpretation $I$, and signature $\Sigma \in \text{Sig}_P$,

$$\text{Inst}_\Sigma(\text{Up}_\Sigma(I)) = \text{Up}_\Sigma(\text{Inst}_\Sigma(I)).$$

**Proof** Let $(A + C)\theta \in \text{Inst}_\Sigma(\text{Up}_\Sigma(I))$, with $A \in I$. Then $(A + C)\theta = (A\theta) + C\theta \in \text{Up}_\Sigma(\text{Inst}_\Sigma(I))$. Vice versa, let $A\theta + C \in \text{Up}_\Sigma(\text{Inst}_\Sigma(I))$, with $A \in I$. Let $B$ be a variant of $C$ with new variables (not appearing in $A$, $\theta$, and $C$) and $\theta'$ be the substitution with domain $\text{Dom}(\theta) \cup \text{FV}(B)$ and s.t. $\theta'|_{\text{Dom}(\theta)} = \theta$ and $\theta'$ maps $B$ to $C$. Then $A\theta + C = A\theta' + B\theta' = (A + B)\theta' \in \text{Inst}_\Sigma(\text{Up}_\Sigma(I)).$  

We are now ready to define the abstract interpretation domain. As we do not need to distinguish between interpretations having the same denotation, we simply identify them using equivalence classes with respect to the corresponding equivalence relation $\simeq$.

**Definition 10.23 (Abstract Interpretation Domain)** Abstract interpretations form a complete lattice $(\mathcal{I}, \sqsubseteq)$, where

- $\mathcal{I} = \{[I]_\simeq \mid I \text{ is an interpretation}\}$;
- $[I]_\simeq \sqsubseteq [J]_\simeq$ iff $[I] \subseteq [J]$;
- the least upper bound of \([I]_\preceq\) and \([J]_\preceq\), written \([I]_\preceq \bigsqcup [J]_\preceq\), is \([I \cup J]_\preceq\);
- the bottom and top elements are \([\emptyset]_\preceq\) and \([\epsilon]_\preceq\), respectively.

The following proposition provides an effective and equivalent condition for testing the \(\sqsubseteq\) relation over interpretations. We will need this result later on.

**Proposition 10.24** Given two interpretations \(I\) and \(J\), \([I] \subseteq [J]\) iff for every \(A \in I\), there exist \(B \in J\), a substitution \(\theta\) and a fact \(C\) (defined over \(\Sigma_P\)) s.t. \(A = B\theta + C\).

**Proof** If part. We prove that for every \(\Sigma \in \text{Sig}_P\), \([I]_\Sigma \subseteq [J]_\Sigma\). Let \(A' = A\theta' + C' \in \text{Up}_\Sigma(\text{Inst}_\Sigma(I)) = [I]_\Sigma\), with \(A \in I\) and \(\theta', C'\) defined over \(\Sigma\). By hypothesis, there exist \(B \in J\), a substitution \(\theta\) and a fact \(C\) (defined over \(\Sigma_P\)) s.t. \(A = B\theta + C\). Therefore, \(A' = A\theta' + C' = (B\theta + C)\theta' + C' = B\theta\theta' + (C\theta' + C') \in \text{Up}_\Sigma(\text{Inst}_\Sigma(J)) = [J]_\Sigma\) (note that \(\theta\theta'\) and \(C\theta' + C'\) are both defined over \(\Sigma\) because \(\Sigma_P \subseteq \Sigma\)).

Only if part. Let \(A \in I\), then \(A \in [I]_\Sigma\) (note that \(A\) is defined over \(\Sigma_P\) by definition of interpretation). Then, by hypothesis we have that \(A \in [J]_\Sigma = \text{Up}_{\Sigma_P}(\text{Inst}_{\Sigma_P}(J))\), i.e., there exist \(B \in J\), a substitution \(\theta\) and a fact \(C\) (defined over \(\Sigma_P\)) s.t. \(A = B\theta + C\). □

We now define the abstract satisfiability judgment \(I \models_\Sigma \Delta \bullet C \bullet \theta\), where \(I\) is an interpretation, \(\Delta\) is a context, \(C\) is an output fact, and \(\theta\) is an output substitution.

**Remark 10.25** As usual, the notation \(I \models_\Sigma \Delta \bullet C \bullet \theta\) requires that \(\Delta\), \(C\), and \(\theta\) are defined over \(\Sigma\). As a consequence, the newly introduced constant \(c\) in the \(\forall\)-case of the \(\models_\Sigma\) definition below cannot be exported through the output parameters \(C\) or \(\theta\).

The judgment \(\models_\Sigma\) can be thought of as an abstract version of the judgment \(|=\Sigma\). We now need one more parameter, namely an output substitution. The idea behind the definition is that the output fact \(C\) and the output substitution \(\theta\) are minimal (in a sense to be clarified) so that they can be computed effectively given a program \(P\), an interpretation \(I\), and a signature \(\Sigma\). The output substitution \(\theta\) is needed in order to deal with clause instantiation, and its minimality is ensured by using most general unifiers in the definition. We recall that the notation \(\theta_1 \uparrow \theta_2\) denotes the least upper bound of substitutions (see Section 2.4).

**Definition 10.26 (Abstract Satisfiability Judgment)** Let \(P\) be an \(\text{LO}_\forall\) program, \(I\) an interpretation, and \(\Sigma \in \text{Sig}_P\). The abstract satisfiability judgment \(\models_\Sigma\) is defined as
follows:

\[ I \vdash_{\Sigma} T, \Delta \rightarrow \varepsilon \rightarrow \text{nil}; \]
\[ I \vdash_{\Sigma} A \rightarrow C \rightarrow \theta \text{ if there exist } B \in I \text{ (variant), } B' \preceq B, \ A' \preceq A, \ |B'| = |A'|, \ C = B' \setminus B', \text{ and } \theta = m.g.u.(B', A')_{FV(A, C)}; \]
\[ I \vdash_{\Sigma} \forall x.G, \Delta \rightarrow C \rightarrow \theta \text{ if } I \vdash_{\Sigma,c} G[c/x], \Delta \rightarrow C \rightarrow \theta, \text{ with } c \notin \Sigma \text{ (see remark 10.25)}; \]
\[ I \vdash_{\Sigma} G_1 \& G_2, \Delta \rightarrow C \rightarrow \theta \text{ if } I \vdash_{\Sigma} G_1, \Delta \rightarrow C_1 \rightarrow \theta_1, \ I \vdash_{\Sigma} G_2, \Delta \rightarrow C_2 \rightarrow \theta_2, \ D_1 \preceq C_1, \ D_2 \preceq C_2, \ |D_1| = |D_2|, \ \theta_3 = m.g.u.(D_1, D_2), \ C = C_1 + (C_2 \setminus D_2), \text{ and } \theta = (\theta_1 \triangleleft \theta_2 \triangleleft \theta_3)_{FV(G_1, G_2, \Delta, C)}; \]
\[ I \vdash_{\Sigma} G_1 \not\preceq G_2, \Delta \rightarrow C \rightarrow \theta \text{ if } I \vdash_{\Sigma} G_1, G_2, \Delta \rightarrow C \rightarrow \theta; \]
\[ I \vdash_{\Sigma} \bot, \Delta \rightarrow C \rightarrow \theta \text{ if } I \vdash_{\Sigma} \Delta \rightarrow C \rightarrow \theta. \]

We recall that two multisets in general may have more than one (not necessarily equivalent) most general unifier and that using the notation \( m.g.u.(B', A') \) we mean any unifier which is non-deterministically picked from the set of most general unifiers of \( B' \) and \( A' \) (see Section 2.4).

**Example 10.27** Let us consider a signature with a function symbol \( f \) and predicate symbols \( p, q, r, s \). Let \( \mathcal{V} \) be a denumerable set of variables, and \( u, v, w, \ldots \in \mathcal{V} \). Let \( I \) be the interpretation consisting of the two multisets \( \{ p(x), q(x) \} \) and \( \{ r(y), p(f(y)) \} \) (for simplicity, hereafter we omit brackets in multiset notation), and \( P \) the program

1. \( r(w) \vdash q(f(w)) \)
2. \( s(z) \vdash \forall x.p(f(x)) \)
3. \( \bot \vdash q(u) \& r(v) \)

Let’s consider (a renaming of) the body of the first clause, \( q(f(w)) \), and (a renaming of) the first element in \( I, p(x'), q(x') \). Using the second clause for the \( I \vdash_{\Sigma_P} \) judgment, with \( A = A' = q(f(w')) \), \( B = p(x'), q(x') \), \( B' = q(x') \), we get

\[ I \vdash_{\Sigma_P} q(f(w')) \rightarrow p(x') \rightarrow [x' \mapsto f(w')]. \]

Let’s consider now (a renaming of) the body of the second clause, \( \forall x.p(f(x)) \), and another renaming of the first element, \( p(x''), q(x'') \). From the \( \forall \)-case of \( I \vdash_{\Sigma_P} \) definition, \( I \vdash_{\Sigma_P} \forall x.p(f(x)) \rightarrow C \rightarrow \theta \text{ if } I \vdash_{\Sigma_P,c} p(f(c)) \rightarrow C \rightarrow \theta, \text{ with } c \notin \Sigma_P. \) Now, we can apply the second clause for \( I \vdash_{\Sigma_P,c} \). Unfortunately, we can’t choose \( A' \) to be \( p(f(c)) \) and \( B' \) to be \( p(x'') \). In fact, by unifying \( p(f(c)) \) with \( p(x'') \), we should get the substitution \( \theta = [x'' \mapsto f(c)] \) and the output fact \( q(x'') \) (note that \( x'' \) is a free variable in the output fact) and this is not allowed because the substitution \( \theta \) must be defined on \( \Sigma_P \), in order for \( I \vdash_{\Sigma_P} \forall x.p(f(x)) \rightarrow C \rightarrow \theta \).
to be meaningful. It turns out that the only way to use the second clause for \( \models_{\Sigma_{\mathcal{P},c}} \) is to choose \( \mathcal{A}' = \mathcal{B}' = \epsilon \), which is useless in the fixpoint computation (see Example 10.39).

Finally, let's consider (a renaming of) the body of the third clause, \( \bot \circ q(u') & r(v') \). According to the \&-rule for the \( \models_{\Sigma} \) judgment, we must first compute \( C_1, C_2, \theta_1 \) and \( \theta_2 \) such that \( I \models_{\Sigma_{\mathcal{P}}} q(u') \triangleright C_1 \triangleright \theta_1 \) and \( I \models_{\Sigma_{\mathcal{P}}} r(v') \triangleright C_2 \triangleright \theta_2 \). To this aim, take two variants of the multisets in \( I, p(x''), q(x'') \) and \( r(y'), p(f(y')) \). Proceeding as above, we get that

\[
I \models_{\Sigma_{\mathcal{P}}} q(u') \triangleright p(x'') \triangleright [u' \mapsto x''] \quad \text{and} \quad I \models_{\Sigma_{\mathcal{P}}} r(v') \triangleright p(f(y')) \triangleright [v' \mapsto y'].
\]

Now, we can apply the \&-rule for the \( \models_{\Sigma} \) judgment, with \( D_1 = p(x''), D_2 = p(f(y')) \), and \( \theta_3 = [x'' \mapsto f(y')] \). We have that \( \theta_1 \uparrow \theta_2 \uparrow \theta_3 = [u' \mapsto f(y'), v' \mapsto y', x'' \mapsto f(y')] \). Therefore, we get that

\[
I \models_{\Sigma_{\mathcal{P}}} q(u') & r(v') \triangleright p(x'') \triangleright [u' \mapsto f(y'), v' \mapsto y', x'' \mapsto f(y')].
\]

The following lemma states a simple property of the substitution domain, which we will need in the following.

**Lemma 10.28** For every interpretation \( I \), context \( \Delta \), fact \( C \), and substitution \( \theta \), if \( I \models_{\Sigma} \Delta \triangleright C \triangleright \theta \) then \( \text{Dom}(\theta) \subseteq \text{FV}(\Delta) \cup \text{FV}(C) \).

**Proof** Immediate by induction on \( \models_{\Sigma} \) definition. \( \square \)

The connection between the satisfiability judgments \( \models_{\Sigma} \) and \( \models_{\Sigma} \) is clarified by the following lemma.

**Lemma 10.29** For every interpretation \( I \), context \( \Delta \), fact \( C \), and substitution \( \theta \),

- i. if \( I \models_{\Sigma} \Delta \triangleright C \triangleright \theta \) then \( \models_{\Sigma} \Delta \theta \triangleright C \triangleright \theta' \) for every substitution \( \theta' \) and fact \( C' \preceq C \);
- ii. if \( [I] \models_{\Sigma} \Delta \triangleright C \) then there exist a fact \( C' \), and substitutions \( \theta' \) and \( \sigma \) s.t. \( I \models_{\Sigma} \Delta \triangleright C' \triangleright \theta', \theta|_{\text{FV}(\Delta)} = (\theta' \circ \sigma)|_{\text{FV}(\Delta)}, C' \theta' \sigma \preceq C \).

**Proof**

- i. By induction on the derivation of \( I \models_{\Sigma} \Delta \triangleright C \triangleright \theta \).
  - If \( \Delta = \top, \Delta' \), obvious;
- suppose $\Delta = \mathcal{A}$, with $\mathcal{B} \in I$ (variant), $\mathcal{B}' \preceq \mathcal{B}$, $\mathcal{A}' \preceq \mathcal{A}$, $\mathcal{C} = \mathcal{B} \setminus \mathcal{B}'$, and
  $\theta = m.g.u.((\mathcal{B}', \mathcal{A}'))_{FV(\Delta, \mathcal{C})}$. We want to prove that $[[I]] =_{\Sigma} \mathcal{A}\theta\theta' \Rightarrow C\theta'$ for every substitution $\theta'$ and fact $C' \succeq C\theta$, i.e., $\mathcal{A}\theta\theta' + C\theta' + \mathcal{D}\theta' \in [I]_{\Sigma}$ for every substitution $\theta'$ and fact $\mathcal{D}$.

Now, $\mathcal{A}\theta\theta' + C\theta' + \mathcal{D}\theta' = (\mathcal{A}\theta + \mathcal{C}\theta + \mathcal{D})\theta' = (\mathcal{A}'\theta + (\mathcal{A}\setminus \mathcal{A}')\theta + (\mathcal{B}\setminus \mathcal{B}')\theta + \mathcal{D})\theta' = (\mathcal{A}\setminus \mathcal{A}')\theta\theta' + (\mathcal{B}\setminus \mathcal{B}')\theta + \mathcal{D}\theta' \in [I]_{\Sigma}$;

- if $\Delta = \forall x. G, \Delta'$ and $I \vdash_{\Sigma, c} G[c/x], \Delta' \Rightarrow C \Rightarrow \theta$, with $c \not\in \Sigma$, then by inductive hypothesis we have that
  
  $[[I]] =_{\Sigma, c} G[c/x]\theta\theta', \Delta'\theta\theta' \Rightarrow C'\theta'$

  for every substitution $\theta'$ and fact $C' \succeq C\theta$ (where $\theta'$ and $C'$ are defined over $\Sigma, c$).

Assuming that the variable $x$ is not in the domain of $\theta\theta'$ (it is always possible to rename the universally quantified variable $x$ in $\forall x. G$), we have that $[[I]] =_{\Sigma, c} G\theta\theta'[c/x], \Delta'\theta\theta' \Rightarrow C'\theta'$, and, by definition of the judgment, we get $[[I]] =_{\Sigma} \forall x. (G\theta\theta'), \Delta'\theta\theta' \Rightarrow C'\theta'$, i.e., $[[I]] =_{\Sigma} (\forall x. G, \Delta')\theta\theta' \Rightarrow C'\theta'$, for every substitution $\theta'$ and fact $C'$ defined over $\Sigma, c$ (and therefore also for every substitution $\theta'$ and fact $C'$ defined over $\Sigma$), with $C' \succeq C\theta$;

- suppose $\Delta = \mathcal{G}_1 \& \mathcal{G}_2, \Delta'$ and $I \vdash_{\Sigma} \mathcal{G}_1 \& \mathcal{G}_2, \Delta' \Rightarrow C \Rightarrow \theta$. We need to prove that
  
  $[[I]] =_{\Sigma} (\mathcal{G}_1 \& \mathcal{G}_2, \Delta')\theta\theta' \Rightarrow C'\theta'$

  for every substitution $\theta'$ and fact $C' \succeq C\theta$, i.e., that $[[I]] =_{\Sigma} (\mathcal{G}_1 \& \mathcal{G}_2, \Delta')\theta\theta' \Rightarrow C\theta' + \mathcal{F}\theta'$ for every substitution $\theta'$ and fact $\mathcal{F}$.

By $\vdash_{\Sigma}$ definition, we have that there exist facts $\mathcal{C}_1 \preceq \mathcal{C}_1, \mathcal{C}_2 \preceq \mathcal{C}_2$ with $|\mathcal{C}_1| = |\mathcal{C}_2|$, and substitutions $\theta_1, \theta_2, \theta_3$ s.t.

  $\theta_3 = m.g.u.((\mathcal{C}_1', \mathcal{C}_2'), C = \mathcal{C}_1 + (\mathcal{C}_2 \setminus \mathcal{C}_2)', \theta = (\theta_1 \uparrow \theta_2 \uparrow \theta_3)_{FV(\Delta, \mathcal{C})}$,

  
  $I \vdash_{\Sigma} \mathcal{G}_1, \Delta' \Rightarrow \mathcal{C}_1 \Rightarrow \theta_1$ and $I \vdash_{\Sigma} \mathcal{G}_2, \Delta' \Rightarrow \mathcal{C}_2 \Rightarrow \theta_2$.

By inductive hypothesis, we have that

  $[[I]] =_{\Sigma} (\mathcal{G}_1, \Delta')\theta_1\theta_1' \Rightarrow \mathcal{C}_1\theta_1\theta_1' + \mathcal{D}_1\theta_1'$ and $[[I]] =_{\Sigma} (\mathcal{G}_2, \Delta')\theta_2\theta_2' \Rightarrow \mathcal{C}_2\theta_2\theta_2' + \mathcal{D}_2\theta_2'$

  for every substitutions $\theta_1', \theta_2'$ and facts $\mathcal{D}_1, \mathcal{D}_2$.

By choosing $\mathcal{D}_1 = (\mathcal{C}_2 \setminus \mathcal{C}_2')\theta_1 + \mathcal{F}_1$ and $\mathcal{D}_2 = (\mathcal{C}_1 \setminus \mathcal{C}_1')\theta_2 + \mathcal{F}_2$, we have, for every substitutions $\theta_1', \theta_2'$ and facts $\mathcal{F}_1, \mathcal{F}_2$,

  $[[I]] =_{\Sigma} (\mathcal{G}_1, \Delta')\theta_1\theta_1' \Rightarrow (\mathcal{C}_1 + (\mathcal{C}_2 \setminus \mathcal{C}_2'))\theta_1\theta_1' + \mathcal{F}_1\theta_1'$,

  $[[I]] =_{\Sigma} (\mathcal{G}_2, \Delta')\theta_2\theta_2' \Rightarrow (\mathcal{C}_2 + (\mathcal{C}_1 \setminus \mathcal{C}_1'))\theta_2\theta_2' + \mathcal{F}_2\theta_2'$.

172
By θ definition, we have that there exist substitutions γ₁, γ₂, γ₃ and τ s.t.

\[ \tau = \theta_1 \circ \gamma_1, \quad \tau = \theta_2 \circ \gamma_2, \quad \tau = \theta_3 \circ \gamma_3, \quad \text{and} \quad \theta = \gamma_{FV(\Delta, C)}. \]

Now, let \( F_1 \) be a variant of \( F\theta' \) with new variables, and define the substitution \( \theta'_1 \) s.t. \( \text{Dom}(\theta'_1) = \text{Dom}(\gamma_1 \circ \theta') \cup FV(F_1) \) (clearly these two latter sets are disjoint), \( \theta'_1|_{\text{Dom}(\gamma_1 \circ \theta')} = \gamma_1 \circ \theta' \) and \( F_1 \theta'_1 = F\theta' \). Do the same for \( F_2 \), i.e., let it be another variant of \( F\theta' \) with new variables, and define \( \theta'_2 \) in the same way, so that \( \text{Dom}(\theta'_2) = \text{Dom}(\gamma_2 \circ \theta') \cup FV(F_2), \theta'_2|_{\text{Dom}(\gamma_2 \circ \theta')} = \gamma_2 \circ \theta', \) and \( F_2 \theta'_2 = F\theta' \).

Now, from τ definition it follows \( (G_1, \Delta')\theta_1\theta'_1 = (G_1, \Delta')\gamma_1\theta' = (G_1, \Delta')\theta' \), and similarly \( (G_2, \Delta')\theta_2\theta'_2 = (G_2, \Delta')\theta' \). Besides, \( (C_1 + (C_2 \setminus C'_2))\theta_1\theta'_1 = C\theta_1\theta'_1 = C\theta' \).

We also have that \( (C_2 + (C_1 \setminus C'_1)|_{\theta_2\theta'_2} = (C_2 + (C_1 \setminus C'_1))\gamma_2\theta' = (C_2 + (C_1 \setminus C'_1))\tau\theta' = (C_2 + (C_1 \setminus C'_1))\theta_3\gamma_3\theta' = (C_2 + (C_1 \setminus C'_1))\theta_3\gamma_3\theta' = (C_2 + (C_1 \setminus C'_1))\theta_3\gamma_3\theta' = (C_2 + (C_1 \setminus C'_1))\theta_3\gamma_3\theta' \).

It follows that, by putting everything together, the inductive hypotheses become \( [I] \models_\Sigma (G_1, \Delta')\theta' \bullet C\theta' + F\theta' \) and \( [I] \models_\Sigma (G_2, \Delta')\theta' \bullet C\theta' + F\theta' \), from which the thesis follows by \( \models_\Sigma \) definition:

- if \( \Delta = G_1 \not\subseteq G_2, \Delta' \) and \( I \models_\Sigma G_1, G_2, \Delta' \bullet C \bullet \theta \), then by inductive hypothesis we have that \( [I] \models_\Sigma (G_1, G_2, \Delta')\theta' \bullet C\theta' \), for every substitution \( \theta' \) and fact \( C' \supseteq C\theta \). Therefore, \( [I] \models_\Sigma G_1, \theta' \bullet G_2, \theta' \bullet \Delta', \theta' \bullet C\theta \), and, by definition of the judgment, we get \( [I] \models_\Sigma (G_1 \not\subseteq G_2, \Delta)\theta' \bullet C\theta' \);

- if \( \Delta = \perp, \Delta' \), the conclusion follows by a straightforward application of the inductive hypothesis.

\[ \text{ii. By induction on the derivation of} \ [I] \models_\Sigma \Delta \bullet C. \]

- If \( \Delta = \top, \Delta' \), take \( C' = \epsilon, \theta' = \text{nil} \), and \( \sigma = \theta \);

- suppose \( [I] \models_\Sigma A\theta \bullet C \) and \( A\theta \supseteq C \in [I]_\Sigma = \text{Up}_\Sigma(\text{Inst}_\Sigma(I)) \). Then there exist \( B \in I \), a fact \( D \) and a substitution \( \tau \) (defined on \( \Sigma \)) s.t. \( A\theta \cup D = B\tau + D \). We can safely assume, thanks to the substitution \( \tau \), that \( B \) is a variant of an element in \( I \). Also, we can assume that \( \text{Dom}(\tau) \subseteq FV(B) \) and \( \text{Dom}(\theta) \cap \text{Dom}(\tau) = \emptyset \).

Now, take the substitution \( \gamma \) s.t. \( \text{Dom}(\gamma) = (\text{Dom}(\theta) \cap FV(A)) \cup \text{Dom}(\tau) \),

\[ \gamma|_{\text{Dom}(\theta) \cap FV(A)} = \theta|_{\text{Dom}(\theta) \cap FV(A)} \quad \text{and} \quad \gamma|_{\text{Dom}(\tau)} = \tau. \]

We have that \( A\gamma \cup D = B\gamma \cup D \). Let \( A' \not\leq A \) and \( B' \not\leq B \) be two maximal sub-multisets s.t. \( A'\gamma = B'\gamma, \rho = m.g.u.(A', B') \), and \( \theta' = \rho|_{FV(A) \cup FV(B'B')} \). By
definition of the $\models_\Sigma$ judgment, we have that $I \models_\Sigma A \triangleright C' \triangleright \theta'$, where $C' = B\setminus B'$.

As $\gamma$ is a unifier for $A', B'$, while $\rho = m.g.u. (A', B')$, we have that there exists a substitution $\sigma$ s.t. $\gamma = \rho \circ \sigma$. Therefore, $\theta |_{FV(A)} = \gamma |_{FV(A)} = (\rho \circ \sigma) |_{FV(A)} = (\rho |_{FV(A)} \circ \rho |_{FV(A)}) = (\theta' \circ \sigma) |_{FV(A)}$, as required.

Furthermore, since $A \gamma + C = B \gamma + D$ and $A' \not\subseteq A$, it follows that $A' \gamma + (A \setminus A') \gamma + C = B' \gamma + (B \setminus B') \gamma + D$, i.e., $(A \setminus A') \gamma + C = (B \setminus B') \gamma + D$. By this equality and maximality of $A'$ and $B'$, we get that necessarily $(B \setminus B') \gamma \not\subseteq C$ (otherwise, $(B \setminus B') \gamma$ and $(A \setminus A') \gamma$ would have elements in common). Therefore, $C' \theta' \sigma = (B \setminus B') \theta' \sigma = (B \setminus B') \rho \sigma = (B \setminus B') \gamma \not\subseteq C$, as required;

- If $\Delta = \forall x.G, \Delta'$ and $[I] \models_{\Sigma e} (G[c/x], \Delta') \theta \triangleright C$, with $c \not\in \Sigma$, then by inductive hypothesis there exist a fact $C'$, and substitutions $\theta'$ and $\sigma$ (defined over $\Sigma, c$) s.t.

$$ I \models_{\Sigma e} G[c/x], \Delta' \triangleright C' \triangleright \theta' $$

$$ \theta |_{FV(G[c/x], \Delta')} = (\theta' \circ \sigma) |_{FV(G[c/x], \Delta')}, \text{ and } C' \theta' \sigma \not\subseteq C. \text{ By definition of the } \models_\Sigma \text{ judgment, we get that}$$

$$ I \models_{\Sigma} \forall x.G, \Delta' \triangleright C' \triangleright \theta'. $$

The conclusion follows (remember that we must ensure that $C'$, $\theta'$ and $\sigma$ are defined over $\Sigma$) by the following crucial observations:

- $Dom(\theta') \subseteq (FV(G[c/x], \Delta') \cup FV(C'))$ by Lemma 10.28;
- $\theta'$ does not map variables in $G[c/x], \Delta'$ to the eigenvariable $c$. In fact we know that $\theta$ does not map variables in $G'[c/x], \Delta'$ to $c$ (by hypothesis) and we know that $(\theta' \circ \sigma) |_{FV(G[c/x], \Delta')} = \theta |_{FV(G[c/x], \Delta')};$
- $\theta'$ does not map variables in $C'$ to $c$ and $C'$ itself does not contain $c$. In fact we know that $C$ does not contain $c$ (by hypothesis) and also that $C' \theta' \sigma \not\subseteq C$;
- We can safely assume that $Dom(\sigma)$ does not contain variables mapped to $c$. Intuitively, these bindings are useless. Formally, we can restrict the domain of $\sigma$ to variables that are not mapped to $c$: with this restriction, the equalities $\theta |_{FV(G[c/x], \Delta')} = (\theta' \circ \sigma) |_{FV(G[c/x], \Delta')}$ and $C' \theta' \sigma \not\subseteq C$ still hold.

- Suppose $\Delta = G_1 \& G_2, \Delta'$ and $[I] \models_{\Sigma} (G_1 \& G_2 \Delta') \theta \triangleright C$. We need to prove that there exist a fact $C'$ and substitutions $\theta'$ and $\sigma$ s.t. $I \models_{\Sigma} G_1 \& G_2, \Delta' \triangleright C' \triangleright \theta'$, $\theta |_{FV(G_1, G_2, \Delta')} = (\theta' \circ \sigma) |_{FV(G_1, G_2, \Delta')}, \text{ and } C' \theta' \sigma \not\subseteq C. \text{ By } \models_{\Sigma} \text{ definition, we have that}$$

$$ I \models_{\Sigma} (G_1, \Delta') \theta \triangleright C \text{ and } I \models_{\Sigma} (G_2, \Delta') \theta \triangleright C. $$

By inductive hypothesis, we have that there exist facts $C_1, C_2$ and substitutions $\theta_1, \theta_2, \sigma_1, \sigma_2$ s.t.

$$ I \models_{\Sigma} G_1, \Delta' \triangleright C_1 \triangleright \theta_1 \text{ and } I \models_{\Sigma} G_2, \Delta' \triangleright C_2 \triangleright \theta_2, $$
\( \theta_{FV(G_1, \Delta')} = (\theta_1 \circ \sigma_1)|_{FV(G_1, \Delta')} \), \( C_1 \theta_1 \sigma_1 \preceq C \), \( \theta|_{FV(G_2, \Delta')} = (\theta_2 \circ \sigma_2)|_{FV(G_2, \Delta')} \) and \( C_2 \theta_2 \sigma_2 \preceq C \).

Now, let \( D_1 \preceq C_1 \) and \( D_2 \preceq C_2 \) s.t. \( D_1 \theta_1 \sigma_1 = D_2 \theta_2 \sigma_2 = C_1 \theta_1 \sigma_1 \cap C_2 \theta_2 \sigma_2 \). Let \( \tau \) be the substitution \( (\theta_1 \circ \sigma_1)|_{FV(G_1, \Delta'; C_1)} \cup (\theta_2 \circ \sigma_2)|_{FV(G_2, \Delta'; C_2)} \); \( \tau \) is well defined because \( \theta_1 \circ \sigma_1 \) and \( \theta_2 \circ \sigma_2 \) both behave like \( \theta \) on variables in \( FV(G_1, \Delta') \) \( \cap \) \( FV(G_2, \Delta') \), and \( C_1, C_2 \) do not have variables in common except for variables in \( G_1, G_2, \Delta' \) (note that new variants of elements in \( I \) are chosen every time the judgment \( \models_\Sigma \) is computed).

Now, \( D_1 \) and \( D_2 \) are unified by \( \tau \), because \( D_1 \tau = D_1 \theta_1 \sigma_1 = D_2 \theta_2 \sigma_2 = D_2 \tau \). Therefore, there exists \( \theta_3 = m.g.u.(D_1, D_2) \) s.t. \( \tau \geq \theta_3 \) (\( \theta_3 \) is more general than \( \tau \)). Also, \( \tau \geq \theta_1 \sigma_1 \geq \theta_1 \) and \( \tau \geq \theta_2 \sigma_2 \geq \theta_2 \). Therefore, \( \tau \) is an upper bound for \( \{\theta_1, \theta_2, \theta_3\} \), hence there exist \( \theta' = (\theta_1 \uplus \theta_2 \uplus \theta_3)|_{FV(G_1, G_2, \Delta'; C)} \), and a substitution \( \gamma \) s.t. \( \tau = \theta' \circ \gamma \). Now we can apply \( \models_\Sigma \) definition (rule for \( \& \)) and we get that

\[
i \models_\Sigma G_1 \& G_2, \Delta' \triangleright C' \triangleright \theta',
\]

where \( C' = C_1 + (C_2 \setminus D_2) \). Let \( \sigma = \gamma \), and let’s prove the thesis.

First of all, since \( \theta' \circ \sigma = \theta' \circ \gamma = \tau \), and by definition of \( \tau \), we have that \( \theta|_{FV(G_1, G_2, \Delta')} = (\theta' \circ \sigma)|_{FV(G_1, G_2, \Delta')} \). It remains to prove that \( C' \theta' \sigma \preceq C \). Now, \( C' \theta' \sigma = C' \tau = C_1 \tau + C_2 \tau \setminus D_2 \tau = C_1 \tau + C_2 \tau \setminus D_2 \theta_2 \sigma_2 = C_1 \tau + C_2 \tau \setminus (C_1 \theta_1 \sigma_1 \cap C_2 \theta_2 \sigma_2) = C_1 \tau + C_2 \tau \setminus (C_1 \tau \cap C_2 \tau) \preceq C \). The last passage holds because \( C_1 \tau \preceq C \) and \( C_2 \tau \preceq C \) (by definition of \( \tau \) and by inductive hypothesis) and relies on the following property of multisets: \( A \preceq D \) and \( B \preceq D \) implies \( A + B \setminus (A \cap B) \preceq D \);

- if \( \Delta = G_1 \not\models G_2, \Delta' \) of \( \Delta = \bot, \Delta' \), the conclusion follows by a straightforward application of the inductive hypothesis.

\[\square\]

The satisfiability judgment \( \models_\Sigma \) also satisfies the following properties.

**Lemma 10.30** For any interpretations \( I_1, I_2, \ldots, \) context \( \Delta \), fact \( C \), and substitution \( \theta \),

i. if \( I_1 \sqsubseteq I_2 \) and \( I_2 \models_\Sigma \Delta \triangleright C \triangleright \theta \) then there exist a fact \( C' \), and substitutions \( \theta' \) and \( \sigma \) s.t. \( I_2 \models_\Sigma \Delta \triangleright C' \triangleright \theta', \theta|_{FV(\Delta)} = (\theta' \circ \sigma)|_{FV(\Delta)} \), \( C' \theta' \sigma \preceq C \theta \);

ii. if \( I_1 \sqsubseteq I_2 \sqsubseteq \ldots \) and \( \bigcup_{i=1}^{\infty} I_i \models_\Sigma \Delta \triangleright C \triangleright \theta \) then there exist \( k \in \mathbb{N} \), a fact \( C' \), and substitutions \( \theta' \) and \( \sigma \) s.t. \( I_k \models_\Sigma \Delta \triangleright C' \triangleright \theta', \theta|_{FV(\Delta)} = (\theta' \circ \sigma)|_{FV(\Delta)} \), \( C' \theta' \sigma \preceq C \theta \).

**Proof**
i. Suppose \( I_1 \models_\Sigma \Delta \triangleright C \triangleright \theta \) and \( I_1 \sqsubseteq I_2 \). By Lemma 10.29 i, \([I_1] \models_\Sigma \Delta \triangleright C \triangleright \theta\). By Lemma 10.9 i, \([I_2] \models_\Sigma \Delta \triangleright C \triangleright \theta\). The conclusion then follows from Lemma 10.29 ii.

ii. Suppose \( \bigcup_{i=1}^{\infty} I_i \models_\Sigma \Delta \triangleright C \triangleright \theta \) and \( I_1 \sqsubseteq I_2 \sqsubseteq \ldots \). By Lemma 10.29 i, \([\bigcup_{i=1}^{\infty} I_i] \models_\Sigma \Delta \triangleright C \triangleright \theta\), i.e., as it can be readily verified from Definition 10.21 and Definition 10.23, \( \bigcup_{i=1}^{\infty} [I_i] \models_\Sigma \Delta \triangleright C \triangleright \theta\). By Lemma 10.9 ii, there exists \( k \in \mathbb{N} \) s.t. \([I_k] \models_\Sigma \Delta \triangleright C \triangleright \theta\). The conclusion then follows from Lemma 10.29 ii.

We are now ready to define the abstract fixpoint operator \( S_P : \mathcal{I} \to \mathcal{I} \). As usual, we will proceed in two steps. We will first define an operator working over interpretations (i.e., elements of \( \mathcal{P}(\text{HB}(P)) \)). With a little bit of overloading, we will call the operator with the same name, i.e., \( S_P \). This operator should satisfy the equation \([S_P(I)] = T_P([I])\) for every interpretation \( I \). This property ensures soundness and completeness of the symbolic representation.

After defining the operator over \( \mathcal{P}(\text{HB}(P)) \), we will lift it to our abstract domain \( \mathcal{I} \) consisting of the equivalence classes of elements of \( \mathcal{P}(\text{HB}(P)) \) w.r.t. the relation \( \simeq \) defined in Definition 10.21. Formally, we first introduce the following definition.

**Definition 10.31 (Symbolic Fixpoint Operator \( S_P \))** Given an LO\(\gamma\) program \( P \) and an interpretation \( I \), the symbolic fixpoint operator \( S_P \) is defined as follows:

\[
S_P(I) = \{ (\hat{H} + C) \circ \theta \mid (H \leftarrow G) \in \text{Vrn}(P), \ I \models_{\Sigma_p} G \triangleright C \triangleright \theta \}.
\]

Note that the \( S_P \) operator is defined using the judgment \( \models_{\Sigma_p} \).

Proposition 10.33 states that \( S_P \) is sound and complete w.r.t \( T_P \). In order to prove it, we need to formulate Lemma 10.32 below.

**Notation.** Let \( P \) be an LO\(\gamma\) program, and \( \Sigma, \Sigma_1 \in \text{Sig}_P \) be two signatures such that \( \Sigma_1 \sqsubseteq \Sigma \). Given a fact \( C \), defined on \( \Sigma \), we use \( [C]_{\Sigma \rightarrow \Sigma_1} \) to denote any fact which is obtained in the following way. For every constant (eigenvariable) \( c \in (\Sigma \setminus \Sigma_1) \), pick a new variable in \( \mathcal{V} \) (not appearing in \( C \)), let it be \( x_c \) (distinct variables must be chosen for distinct eigenvariables). Now, \( [C]_{\Sigma \rightarrow \Sigma_1} \) is obtained by \( C \) by replacing every \( c \in (\Sigma \setminus \Sigma_1) \) with \( x_c \). For instance, if \( C = \{ p(x, f(c)), q(y, d) \} \), with \( c \in (\Sigma \setminus \Sigma_1) \) and \( d \in \Sigma_1 \), we have that \( [C]_{\Sigma \rightarrow \Sigma_1} = \{ p(x, f(x_c)), q(y, d) \} \).

Given a context (multiset of goals) \( \Delta \), defined on \( \Sigma \), we define \( [\Delta]_{\Sigma \rightarrow \Sigma_1} \) in the same way. Similarly, given a substitution \( \theta \), defined on \( \Sigma \), we use the notation \( [\theta]_{\Sigma \rightarrow \Sigma_1} \) to denote the substitution obtained from \( \theta \) by replacing every \( c \in (\Sigma \setminus \Sigma_1) \) with a new variable \( x_c \) in
every binding of $\theta$. For instance, if $\theta = [u \mapsto p(x, f(c)), v \mapsto q(y, d)]$, with $c \in (\Sigma \setminus \Sigma_1)$ and $d \in \Sigma_1$, we have that $[\theta]_{\Sigma_1} = [u \mapsto p(x, f(x_c)), v \mapsto q(y, d)]$.

Using the notation $[I] \models_{\Sigma_1} [\Delta] \triangleright [C] \triangleright [\theta]$ we mean the judgment obtained by replacing every $c \in (\Sigma \setminus \Sigma_1)$ with $x_c$ simultaneously in $D$ and $C$. Newly introduced variables must not appear in $\Delta$, $C$, or $I$.

When $\Sigma$ and $\Sigma_1$ are clear from the context, we simply write $[\Delta]$, $[\theta]$ for $[\Delta]_{\Sigma_\rightarrow \Sigma_1}$, $[\theta]_{\Sigma_\rightarrow \Sigma_1}$.

Finally, we use $\xi_{\Sigma_1 \rightarrow \Sigma}$ (or simply $\xi$ if it is not ambiguous) to denote the substitution which maps every variable $x_c$ back to $c$ (for every $c \in (\Sigma \setminus \Sigma_1)$), i.e., consisting of all bindings of the form $x_c \mapsto c$ for every $c \in \Sigma \setminus \Sigma_1$. Clearly, we have that $[F]_\xi = F$, for any fact or context $F$, and $[\theta] \circ \xi = \theta$ for any substitution $\theta$.

Note that, by definition, $[\Delta]_{\Sigma_1 \rightarrow \Sigma}$ and $[\theta]_{\Sigma_1 \rightarrow \Sigma}$ are defined on $\Sigma_1$, while $\xi_{\Sigma_1 \rightarrow \Sigma}$ is defined on $\Sigma$.

**Lemma 10.32** Let $P$ be an LO$_q$ program, $I$ an interpretation, and $\Sigma, \Sigma_1 \in \text{Sig}_P$ two signatures, with $\Sigma_1 \subseteq \Sigma$.

1. If $I \models_{\Sigma_1} \Delta \triangleright [C] \triangleright [\theta]$ then $I \models_{\Sigma} \Delta \triangleright [C] \triangleright [\theta]$.

2. If $[I] \models_{\Sigma_1} [\Delta] \triangleright [C] \triangleright [\theta]$ then $[I] \models_{\Sigma} [\Delta]_{\Sigma_\rightarrow \Sigma_1} \triangleright [C]_{\Sigma_\rightarrow \Sigma_1}$.

**Proof**

1. By simple induction on the derivation of $I \models_{\Sigma_1} \Delta \triangleright [C] \triangleright [\theta]$.

2. By induction on the derivation of $[I] \models_{\Sigma} \Delta \triangleright [C]$.

- If $[I] \models_{\Sigma} \top, \Delta \triangleright [C]$, immediate;
- suppose $[I] \models_{\Sigma} \mathcal{A} \triangleright [C]$ and $\mathcal{A} + [C] \in [I]_{\Sigma_1}$. It follows that there exist $B \in I$, a fact $D$ and a substitution $\theta$ (defined on $\Sigma$) such that $\mathcal{A} + [C] = B\theta + D$. Note that $B$ is defined on $\Sigma_P$ by definition of (abstract) interpretation.

Now, $[\mathcal{A}] + [C] = [\mathcal{A} + [C]] = [B\theta + D] = [B\theta] + [D]$ (remember that $B$ is defined on $\Sigma_P \subseteq \Sigma_1$) $B[\theta] + [D]$. We can conclude that $[\mathcal{A}] + [C] \in [I]_{\Sigma_1}$ (note that $B \in I$ and $[\theta]$, $[D]$ are defined on $\Sigma_1$, it follows $[I] \models_{\Sigma_1} [\mathcal{A}] \triangleright [C]$;

- suppose $[I] \models_{\Sigma} \forall x.G, \Delta \triangleright [C]$ and $[I] \models_{\Sigma} G[c/x], \Delta \triangleright [C]$, with $c \notin \Sigma$. From $\Sigma_1 \subseteq \Sigma$ we get $\Sigma_1, c \subseteq \Sigma, c$, therefore we can apply the inductive hypothesis.

It follows that $[I] \models_{\Sigma_1} G[c/x], \Delta \triangleright [C]$ iff $[I] \models_{\Sigma_1} G[c/x], [\Delta] \triangleright [C]$ iff (remember that $c \notin \Sigma \setminus \Sigma_1$) because $c \notin \Sigma$ $[I] \models_{\Sigma_1} G[c/x], [\Delta] \triangleright [C]$ by definition (remember that $c \notin \Sigma$ implies $c \notin \Sigma_1$), we get $[I] \models_{\Sigma_1} \forall x.G[c/x], [\Delta] \triangleright [C]$.
iff }I\] \models_{\Sigma_i} [\forall x. G, \Delta] \triangleright [C] \) (we assume }x\) to be disjoint with the variables introduced by the [\cdot\cdot\cdot] construction;

- the remaining cases follow by a straightforward application of the inductive hypothesis.

\[\square\]

**Proposition 10.33**

*For every LOₜ program }P\) and interpretation }I\), }\([S_P(I)]\) = }T_P([I])\).

**Proof**

- }\([S_P(I)]\) ⊆ }T_P([I])\).

We prove that for every }\Sigma \in \text{Sig}_P\), }\([S_P(I)]\) \subseteq }T_P([I])_\Sigma\). Suppose }\hat{H} + C\theta \in S_P(I)\), with }H \propto G\) variant of a clause in }P\) and }I \models_{\Sigma_P} G \triangleright C \triangleright \theta\). Suppose also that }A = ((\hat{H} + C\theta + D)\theta' \in \text{Inst}_\Sigma(Up_\Sigma(S_P(I)))) = }[Sp(I)]_\Sigma\).

We have that }I \models_{\Sigma_P} G \triangleright C \triangleright \theta\) implies }I \models_{\Sigma} G \triangleright C \triangleright \theta\) by Lemma 10.32 }i\) (remember that }\Sigma_P \subseteq \Sigma\). Therefore, by Lemma 10.29 }i\), we get }\[I\] \models_{\Sigma} G\theta\theta' \triangleright C\theta\theta' \text{ for any fact } C' \supseteq C\theta\).

Taking }C' = C\theta + D\), it follows that }\[I\] \models_{\Sigma} G\theta\theta' \triangleright C\theta\theta' + D\theta'\). Therefore, by }T_P\) definition, we have }\hat{H}\theta\theta' + C\theta\theta' + D\theta' \in (T_P([I]))_\Sigma\), i.e., }A \in (T_P([I]))_\Sigma\).

- }T_P([I]) \subseteq }[S_P(I)]\).

We prove that for every }\Sigma \in \text{Sig}_P\), }T_P([I])_\Sigma \subseteq }[S_P(I)]_\Sigma\). Suppose }A \in (T_P([I]))_\Sigma\). By definition of }T_P\), there exist a variant of a clause }H \propto G\) in }P\), a fact }C\) and a substitution }\theta\) (defined over }\Sigma\) s.t. }A = }\hat{H}\theta + C\) and }\[I\] \models_{\Sigma} G\theta \triangleright C\).

By Lemma 10.32 }ii\) we have that }\[I\] \models_{\Sigma} G\theta \triangleright C\) implies }\[I\] \models_{\Sigma_P} [G\theta] \triangleright [C]\) (hereafter, we use the notation }[\cdot\cdot\cdot]\) for }\[\cdot\cdot\cdot\]_{\Sigma \ominus \Sigma_P}\). From }H \propto G\) in }P\), we know that }G\) is defined on }\Sigma_P\). It follows easily that }[G\theta] = G[\theta]\), so that }\[I\] \models_{\Sigma_P} G[\theta] \triangleright [C]\). By Lemma 10.29 }ii\), there exist a fact }C', and substitutions }\theta'\) and }\sigma\) (defined over }\Sigma_P\) s.t. }I \models_{\Sigma_P} G \triangleright C' \triangleright \theta', [\theta]|_{FV(G)} = (\theta' \circ \sigma)|_{FV(G)}, \text{ and } C'\theta'\sigma \equiv [C]\).

By }S_P\) definition, we have }\(\hat{H} + C'\theta'\) \in S_P(I)\).

Now, }A = }\hat{H}\theta + C = }\hat{H}[\theta]\xi + [C]\xi = (\text{note that by hypothesis }\theta' \circ \sigma\text{ and }[\theta]\) coincide for variables in }G, and are not defined on variables in }H\) which do not appear in }G\) because }H \propto G\) is a variant) }\hat{H}\theta'\sigma\xi + [C]\xi \supseteq }\hat{H}\theta'\sigma\xi + C'\theta'\sigma\xi = ((\hat{H} + C')\theta')\sigma\xi \in [((\hat{H} + C')\theta')]_\Sigma \subseteq }[S_P(I)]_\Sigma\).

\[\square\]

The following corollary holds.
Corollary 10.34 For every LO\textsubscript{\textgreek{a}} program P and interpretations I and J, if I \simeq J then \(S_P(I) \simeq S_P(J)\).

Proof If I \simeq J, i.e., \([I] = [J]\), we have that \(T_P([I]) = T_P([J])\). By Proposition 10.33, it follows that \([S_P(I)] = [S_P(J)]\), i.e., \(S_P(I) \simeq S_P(J)\). □

The previous Corollary allows us to safely lift \(S_P\) definition from the lattice \(\langle \mathcal{P}(HB(P)), \subseteq \rangle\) to \(\langle \mathcal{I}, \sqsubseteq \rangle\). Formally, we define the abstract fixpoint operator as follows.

Definition 10.35 (Abstract Fixpoint Operator \(S_P\)) Given an LO\textsubscript{\textgreek{a}} program P and an equivalence class \([I]_{\simeq}\) of \(\mathcal{I}\), the abstract fixpoint operator \(S_P\) is defined as follows:

\[S_P([I]_{\simeq}) = [S_P(I)]_{\simeq}\]

where \(S_P(I)\) is defined in Definition 10.31.

For the sake of simplicity, in the following we will often use I to denote its class \([I]_{\simeq}\), and we will simply use the term (abstract) interpretation to refer to an equivalence class, i.e., an element of \(\mathcal{I}\). The abstract fixpoint operator \(S_P\) satisfies the following property.

Proposition 10.36 (Monotonicity and Continuity) For every LO\textsubscript{\textgreek{a}} program P, the abstract fixpoint operator \(S_P\) is monotonic and continuous over the lattice \(\langle \mathcal{I}, \sqsubseteq \rangle\).

Proof

Monotonicity.

We prove that if \(I \sqsubseteq J\), then \(S_P(I) \sqsubseteq S_P(J)\), i.e., \([S_P(I)] \sqsubseteq [S_P(J)]\). To prove this latter condition, we will use the characterization given by Proposition 10.24. Suppose \(A = (\widehat{H} + C)\theta \in S_P(I)\), with \(H \arrows G\) variant of a clause in P and \(I \models_{\Sigma_P} G \bullet C \bullet \theta\).

By Lemma 10.30 \(i\), there exist a fact \(C'\), and substitutions \(\theta'\) and \(\sigma\) (note that they are defined over \(\Sigma_P\)) s.t. \(J \models_{\Sigma_P} G \bullet C' \bullet \theta'\), \(\theta |_{FV(G)} = (\theta' \circ \sigma) |_{FV(G)}\), \(C'\theta'\sigma \simeq C\theta\). Let \(C\theta = C'\theta'\sigma + D\), with D a fact defined over \(\Sigma_P\). By \(S_P\) definition, \(B = (\widehat{H} + C')\theta' \in S_P(J)\).

Now, \(A = (\widehat{H} + C)\theta = \widehat{H}\theta + C\theta = \widehat{H}\theta'\sigma + C'\theta'\sigma + D\) (note in fact that by hypothesis \(\theta'\sigma\) and \(\theta\) coincide for variables in \(G\), and are not defined on variables in \(H\) which do not appear in \(G\) because \(H \arrows G\) is a variant). Therefore, we have that \(A = \widehat{H}\theta'\sigma + C'\theta'\sigma + D = B\sigma + D\).

Continuity.

We show that \(S_P\) is finitary, i.e., if \(I_1 \sqsubseteq I_2 \sqsubseteq \ldots\), then \(S_P(\bigcup_{i=1}^{\infty} I_i) \sqsubseteq \bigcup_{i=1}^{\infty} S_P(I_i)\), i.e.,
\[ S_P(\bigcup_{i=1}^{\infty} I_i) \subseteq \bigcup_{i=1}^{\infty} S_P(I_i) \]. Again, we will use the characterization given by Proposition 10.24. Suppose \( A = (\mathcal{H} + C) \theta \in S_P(\bigcup_{i=1}^{\infty} I_i) \), with \( H \circ G \) variant of a clause in \( P \) and \( \bigcup_{i=1}^{\infty} I_1 \models_{\Sigma_P} G \circ C \circ \theta \).

By Lemma 10.30, there exist \( k \in \mathbb{N} \), a fact \( C' \), and substitutions \( \theta' \) and \( \sigma \) (note that they are defined over \( \Sigma_P \)) s.t. \( I_k \models_{\Sigma_P} G \circ C' \circ \theta' \circ (\theta' \circ \sigma)_{|FV(G)} \). Let \( C \theta = C' \theta' \sigma + D \), with \( D \) a fact defined over \( \Sigma_P \). By \( S_P \) definition, \( B = (\mathcal{H} + C') \theta' \in S_P(I_k) \).

Exactly as above, we prove that \( A = (\mathcal{H} + C) \theta = \mathcal{H} \theta' \sigma + C' \theta' \sigma + D = B \sigma + D \). \( \square \)

**Corollary 10.37** For every \( LO_\varphi \) program \( P \), \( [\text{lfp}(S_P)] = \text{lfp}(T_P) \).

Let \( \mathcal{F}_{\text{sym}}(P) = \text{lfp}(S_P) \), then we have the following main theorem.

**Theorem 10.38 (Soundness andCompleteness)** For every \( LO_\varphi \) program \( P \), \( O(P) = F(P) = [\mathcal{F}_{\text{sym}}(P)]_{\Sigma_P} \).

**Proof** From Theorem 10.14 and Corollary 10.37. \( \square \)

The previous results give us an algorithm to compute the operational and fixpoint semantics of a program \( P \) via the fixpoint operator \( S_P \).

**Example 10.39** Let us consider a signature with a constant symbol \( a \), a function symbol \( f \) and predicate symbols \( p, q, r, s \). Let \( V \) be a denumerable set of variables, and \( u, v, w, \ldots \in V \). Let us consider the program \( P \) of Example 3.15, which is given below.

1. \( r(w) \circ q(f(w)) \)
2. \( s(z) \circ \forall x. p(f(x)) \)
3. \( \bot \circ q(u) \& r(v) \)
4. \( p(x) \not\exists q(x) \circ \top \)

From clause 4, and using the first rule for \( \models_{\Sigma_P} \), we get that \( S_P(\emptyset) = [\{p(x), q(x)\}]_{\Sigma} \). For simplicity, we omit the class notation, and we write \( S_P(\emptyset) = \{p(x), q(x)\} \).

We can now apply the remaining clauses to the element \( I = \{p(x), q(x)\} \) (remember that \( S_P([I]) = [S_P(I)]_{\Sigma} \)). From the first clause (see Example 10.27) we have \( I \models_{\Sigma_P} q(f(w)) \circ p(x) \circ [x' \mapsto f(w')] \). It follows \( (r(w'), p(x'))[x' \mapsto f(w')] = r(w'), p(f(w')) \in S_P(\emptyset) \). As the reader can verify (see discussion in Example 10.27), clause 2 does not yield
any further element, and the same holds for clause 3, therefore we can assume (changing \( w' \) into \( y \) for convenience)

\[
S_P \|_2 = \{ \{ p(x), q(x) \}, \{ r(y), p(f(y)) \} \}.
\]

Now, we can apply clause 3 to elements in \( S_P \|_2 \). According to Example 10.27, we have that \( I \models_{\Sigma_P} q(w') \land r(v') \supset p(x''') \supset [u' \mapsto f(y'), v' \mapsto y', x''' \mapsto f(y')] \). Therefore we get that \( (p(x''))[u' \mapsto f(y'), v' \mapsto y', x''' \mapsto f(y')] = p(f(y')) \in S_P \|_3 \). Clause 2 cannot be applied yet, for the same reasons as above. Also, note that the element \( r(y), p(f(y)) \) is now subsumed by \( p(f(y')) \). Therefore we can assume

\[
S_P \|_3 = \{ \{ p(x), q(x) \}, \{ p(f(y')) \} \}.
\]

Finally, we can apply clause 2 to \( S_P \|_3 \), using the \( \forall \)-rule for the \( \models_{\Sigma_P} \) judgment. Take \( c \not\in \Sigma_P \), and consider a renaming of the last element in \( S_P \|_3 \), \( p(f(y'')) \). Consider (a renaming of) clause 2, \( s(z') \vdash \forall x.p(f(x)) \). We have that \( I \models_{\Sigma_P,e} p(f(c)) \supset \epsilon \supset \text{nil} \), with \text{nil} being the empty substitution. Therefore we get that \( I \models_{\Sigma_P} \forall x.p(f(x)) \supset \epsilon \supset \text{nil} \), from which \( s(z') \in S_P \|_4 \). The reader can verify that no further clauses can be applied and that \( S_P \|_4 \) is indeed the fixpoint of \( S_P \), therefore we have that

\[
S_P \|_4 = S_P \|_\omega = \{ \{ p(x), q(x) \}, \{ p(f(y')) \}, \{ s(z') \} \}.
\]

Note that \( F(P) \) is defined to be \([\text{lp}(S_P)]_{\Sigma_P} \), therefore it includes, e.g., the elements \( s(a) \) (see Example 3.15), \( p(f(f(y''))) \) and \( p(f(f(y''))), q(x'') \).

**10.4 Ensuring Termination**

In this section we will rephrase the results of Section 9.3, concerning termination of the bottom-up evaluation algorithm, to the case of \( \text{LO}_v \) programs. An application of these results will be presented in Section 10.5.

Accordingly, we introduce the class of programs with monadic predicate symbols.

**Definition 10.40 (The \( \mathcal{P}_1(\text{LO}_v) \) class)** The class \( \mathcal{P}_1(\text{LO}_v) \) consists of \( \text{LO}_v \) programs built over a signature \( \Sigma \) including a finite set of constant symbols, no function symbols, and a finite set of predicate symbols with arity at most one.

**Definition 10.41 (Multisets \( \mathcal{M}_1(\text{LO}_v) \))** The class \( \mathcal{M}_1(\text{LO}_v) \) consists of multisets of (non ground) atomic formulas over a signature \( \Sigma \) including a finite set of constant symbols, no function symbols, and predicate symbols with arity at most one.
Example 10.42 Let \( \Sigma \) be a signature including a constant symbols \( a \), no function symbols, and predicate symbols \( p, q \) (with arity one) and \( r \) (with arity zero). Let \( \mathcal{V} \) be a denumerable set of variables, and \( x, y, \ldots \in \mathcal{V} \). Then the clause
\[
p(x) \mathcal{R} q(x) \mathcal{R} q(y) \mathcal{R} r \mathcal{R} (p(a) \mathcal{R} p(y)) \mathcal{R} \forall z. q(z)
\]
is in the class \( \mathcal{P}_1(LO_\forall) \), and the multiset \( \{p(x), q(x), q(a), r\} \) is in the class \( \mathcal{M}_1(LO_\forall) \). \( \square \)

The following result holds.

Proposition 10.43 The class \( \mathcal{M}_1(LO_\forall) \) is closed under applications of \( \mathcal{S}_P \), i.e., if \( I \subseteq \mathcal{M}_1(LO_\forall) \) then \( \mathcal{S}_P(I) \subseteq \mathcal{M}_1(LO_\forall) \).

Proof Immediate by Definition 10.31 and Definition 10.26. \( \square \)

Termination of the bottom-up evaluation for the class \( \mathcal{P}_1(LO_\forall) \) is stated in the following proposition.

Proposition 10.44 Let \( P \) be a \( \mathcal{P}_1(LO_\forall) \) program. Then there exists \( k \in \mathbb{N} \) such that \( \mathcal{F}_\text{sym}(P) = \bigcup_{i=0}^{k} \mathcal{S}_P \uparrow_i (\emptyset) \).

Proof The proof is carried out, similarly to that of Proposition 9.57, by proving that the entailment relation between multisets in the class \( \mathcal{M}_1(LO_\forall) \) (see Proposition 10.24) is a wqo.

Entailment between multisets in \( \mathcal{M}_1(LO_\forall) \) being a wqo is a simple consequence of Proposition 9.57. Take a multiset in the class \( \mathcal{M}_1(LO_\forall) \). First of all, perform the following transformation: for every atom \( p(a) \), where \( a \) is a constant symbol in \( \Sigma \) (note that there no other ground terms other than constants in this class) introduce a new predicate symbol, with arity zero, let it be \( p_a \), and transform the original multiset by substituting \( p_a \) in place of \( p(a) \). The resulting set of predicate symbols is still finite (the set of constants of the program is finite).

Now, it is easy to see that entailment between multisets transformed in the above way is a sufficient condition for entailment of the original multisets (note that the condition is not necessary, e.g. I cannot recognize that \( p(a) \) entails \( p(x) \)). Now, entailment between transformed multisets is a wqo. In fact, note that a transformed multiset consists of atomic formulas with a predicate symbol (with arity zero or one), where arguments are (possibly duplicated) variables. Now, duplicated variables correspond to equality constraints in the class \( \mathcal{M}_1(\text{NC}) \) of Section 9.3.

Using a similar proof to that of Proposition 9.52, and from Propositions 10.43 and 6.23, we can conclude. \( \square \)
10.5 **An Example: A Distributed Test-and-Lock Protocol**

We conclude this chapter by discussing an example, which can be seen as a specification for a distributed *test-and-lock* protocol for a net with multiple resources, each one controlled by a monitor. We have verified protocol correctness using the tool described in Appendix A.

The protocol is as follows. A set of *resources*, distinguished by means of *resource identifiers*, and an arbitrary set of processes are given. Processes can non-deterministically request access to any resource. Access to a given resource must be exclusive (only one process at a time). Mutual exclusion is enforced by providing each resource with a *semaphore*.

Given a propositional symbol *init*, we can encode the initial states of the system as follows (we intentionally introduce a flaw which we will disclose later):

1. \( \text{init} \leftarrow \text{init} \land \text{think} \)
2. \( \text{init} \leftarrow \text{init} \land m(x, \text{unlocked}) \)
3. \( \text{init} \leftarrow \bot \)

The atom *think* represents a thinking (idle) process, while the first-order atom \( m(x, s) \) represents a *monitor* for the resource with identifier \( x \) and associated semaphore \( s \). The semaphore \( s \) can assume one of the two values *locked* or *unlocked*. Clause 1 and clause 2 can modify the initial state by adding, respectively, an arbitrary number of thinking processes and an arbitrary number of resources (with an initially unlocked semaphore). Finally, using clause 3 the atom *init* can be removed after the initialization phase.

The core of the protocol works as follows:

4. \( \text{think} \leftarrow \text{wait}(x) \)
5. \( \text{wait}(x) \leftarrow \text{think} \)
6. \( \text{wait}(x) \land m(x, \text{unlocked}) \leftarrow \text{use}(x) \land m(x, \text{locked}) \)
7. \( \text{use}(x) \land m(x, \text{locked}) \leftarrow \text{think} \land m(x, \text{unlocked}) \)

Using clause 4, a process can non-deterministically request access to any resource with identifier \( x \), moving to a waiting state represented by the atom *wait*(\( x \)). Clause 5 allows
Figure 10.2: A trace violating mutual exclusion for the test-and-lock protocol

\[
\begin{align*}
P \vdash_\Sigma \text{init}, \top, m(x, \text{locked}), m(x, \text{locked}) & \quad \top_r \\
&P \vdash_\Sigma \text{init}, \text{use}(x), \text{use}(x), m(x, \text{locked}), m(x, \text{locked}) \quad \text{bc}(8) \\
&P \vdash_\Sigma \text{init}, \text{wait}(x), \text{wait}(x), m(x, \text{unlocked}), m(x, \text{unlocked}) \quad \text{bc}(6^*) \\
&P \vdash_\Sigma \text{init}, \text{think}, \text{think}, m(x, \text{unlocked}), m(x, \text{unlocked}) \quad \text{bc}(4^*) \\
&P \vdash_\Sigma \text{init}, \text{think}, \text{think}, m(x, \text{unlocked}), m(x, \text{unlocked}) \quad \text{bc}(2^*) \\
&P \vdash_\Sigma \text{init} \quad \text{bc}(1^*)
\end{align*}
\]

Figure 10.3: A test-and-lock protocol: example trace
\{\textit{use}(x), \textit{use}(x)\}
\{\textit{m}(x, \textit{unlocked}), \textit{use}(x), \textit{init}\}
\{\textit{m}(x, \textit{unlocked}), \textit{use}(x), \textit{wait}(y)\}\n\{\textit{m}(x, \textit{unlocked}), \textit{use}(x), \textit{use}(y), \textit{m}(y, \textit{locked})\}\n\{\textit{m}(x, \textit{locked}), \textit{use}(x), \textit{m}(y, \textit{unlocked}), \textit{m}(y, \textit{unlocked}), \textit{think}\}\n\{\textit{m}(x, \textit{unlocked}), \textit{m}(x, \textit{unlocked}), \textit{wait}(y), \textit{think}\}\n\{\textit{m}(x, \textit{unlocked}), \textit{m}(x, \textit{unlocked}), \textit{use}(y), \textit{m}(y, \textit{locked}), \textit{use}(z), \textit{m}(z, \textit{locked})\}\n\{\textit{m}(x, \textit{locked}), \textit{m}(x, \textit{unlocked}), \textit{use}(y), \textit{m}(y, \textit{locked}), \textit{wait}(z)\}\n\{\textit{wait}(x), \textit{m}(y, \textit{unlocked}), \textit{m}(y, \textit{unlocked}), \textit{wait}(z)\}\n\{\textit{m}(x, \textit{unlocked}), \textit{m}(x, \textit{unlocked}), \textit{init}\}\n\{\textit{m}(x, \textit{unlocked}), \textit{m}(x, \textit{unlocked}), \textit{think}, \textit{think}\}\n\{\textit{use}(x), \textit{m}(x, \textit{unlocked}), \textit{think}\}\n
Figure 10.4: Fixpoint computed for the test-and-lock protocol

a process to go back to thinking from a waiting state. By clause 6, a waiting process can synchronize with the relevant monitor and is granted access provided the corresponding semaphore is unlocked. As a result, the semaphore is locked. The atom \textit{use}(x) represents a process which is currently using the resource with identifier \(x\). Clause 7 allows a process to release a resource and go back to thinking, unlocking the corresponding semaphore.

Using our verification tool (see Appendix A), we can now automatically verify the \textit{mutual exclusion} property for the above test-and-lock protocol. The specification of unsafe states is simply as follows:

\[\textit{use}(x) \not\Rightarrow \textit{use}(x) \circ \top\]

The test-and-lock specification can be seen to fall into the class \(\mathcal{P}_1(LO_{\forall})\). In fact, the binary predicate symbol \(m\) plays the same role as two unary predicate symbols \(m_{\text{locked}}\) and \(m_{\text{unlocked}}\) (note that the second argument of \(m\) in the initial specification is always instantiated). Termination of the fixpoint computation is therefore \textit{guaranteed} by Proposition 10.44. Running the verification algorithm, we actually find a security violation. The corresponding trace is shown in Figure 10.2. The problem of the above specification lies in clause 2. In fact, using an (externally quantified) variable \(x\) does not prevent the creation of multiple monitors for the same resource. This causes a violation of mutual exclusion when different processes are allowed to concurrently access a given resource by different monitors.

Luckily, we can fix the above problem in a very simple way. As we do not care about what resource identifiers actually are, we can elegantly encode them using universal quantifica-
{use(x), use(x)}
{m(x,y), m(x,z)}
{m(x, unlocked), use(x), use(y), m(y, z)}
{m(x, unlocked), use(x), wait(y)}
{m(x, unlocked), use(x), init}
{use(x), m(x, unlocked), think}

Figure 10.5: Fixpoint computed using invariant strengthening for the test-and-lock protocol

tion in the body of clause 2, as follows:

2’. \text{init} \leftarrow \text{init} \land \forall x. m(x, \text{unlocked})

Every time a resource is created, a new constant, acting as the corresponding identifier, is created as well. Note that by the operational semantics of universal quantification, different resources are assigned different identifiers. This clearly prevents the creation of multiple monitors for the same resource. An example trace for the modified specification is shown in Figure 10.3 (where \( P \) is the program consisting of clauses 1, 2’, 3 through 8 above, and we followed the usual conventions, with \( bc(i^*) \) denoting multiple applications of clause number \( i \)).

Now, running again our verification tool on the corrected specification (termination is still guaranteed by Proposition 10.44), with the same set of unsafe states, we get the fixpoint shown in Figure 10.4 (where, for readability, we re-use the same variables in different multisets). The fixpoint contains 12 elements and is reached in 7 steps. As the fixpoint does not contain \( \text{init} \), mutual exclusion is verified, for any number of processes and any number of resources.

We conclude by showing how it is possible to optimize the fixpoint computation by invariant strengthening (see Section 6.1.2.1). One possibility might be to apply the so-called counting abstraction, i.e., turn the above LO specification into a propositional program (i.e., a Petri net) by abstracting first-order atoms into propositional symbols (e.g. \( \text{wait}(x) \) into \( \text{wait} \), and so on), and compute the structural invariants of the corresponding Petri net. However, this strategy is not helpful in this case (no meaningful invariant is found). We can still try some invariants using some ingenuity. For instance, consider the following invariant:

9. \( m(x, y) \Rightarrow m(x, z) \leftarrow T \)

For what we said previously (different resources are assigned different identifiers) this invariant must hold for our specification. Running the verification tool on this extended specification we get the fixpoint in Figure 10.5, containing only 6 elements and converging in 4 steps. A further optimization could be obtained by adding the invariant \( \text{use}(x) \Rightarrow m(x, \text{unlocked}) \leftarrow T \) (intuitively, if someone is using a given resource, the corresponding
semaphore cannot be unlocked). In this case the computation converges immediately at
the first step.

10.6 Related Work

In this section we report on some related work, in particular for what concerns automated
deduction techniques for verification of parameterized systems. Alternative approaches
based, e.g., on model checking, have been discussed in Section 9.5. Clearly, there may
be overlaps between the two approaches. For instance, [FO97] uses an approach based
on regular sets (see Section 9.5) in a deductive setting. Specifically, a methodology is
proposed, which is able to infer, using a generalization method, a regular language to
represent reachable states for parameterized rings.

Many methods proposed for the verification of parameterized systems, can be roughly
classified into those using explicit induction, network invariants, which can be viewed as
implicit induction, and abstraction. Among those using explicit induction, there are, e.g.,
[EN95, GS92]. In [EN95], for instance, it is shown that some properties of rings of arbitrary
size can be verified by checking they hold for a finite number of instances of the problem.
The resulting instances can then be verified by model checking techniques.

 Implicit induction is used, e.g., in [WL89, LHR97, KM95]. A general structural induction
theorem for concurrent models of processes is discussed in [KM95]. The induction method
is based on representing a given specification as a process, and in finding a suitable invariant
process to apply induction. In [LHR97] the problem of verification for linear networks of
processes is reduced to the construction of a network invariant, which can be synthetized
using heuristics, e.g. widening techniques [CC77].

 Abstraction techniques are used, e.g., in [ID96, LS97, GZ98]. For instance, in [ID96] systems
with replicated components are verified by explicit enumeration in an abstract state space
in which states do not record the exact number of components. The notion of conservative
abstraction is discussed in [LGS+95]. Namely, the paper discusses property preserving
transformations, which ensure that a property for a given system can be safely verified on
a simpler abstraction. The formal results are based on the theory of abstract interpretation
[CC77]. An approach mixing abstraction and induction is the one given in [MQS00].

Finally, approaches which are more deductive in nature include for instance [RV95]. An
application of this system to specification and verification of security protocols is given
in [JRV00] (see Section 11.6). Traditional automated deduction approaches include for
instance using the PVS system [ORR+96], which combines (interactive) proof checking
with model checking techniques, and provides a meta-language for defining proof strategies.
Other deductive tools include, e.g., LCF [GMW79], Nuprl [CAB+86],and HOL [GM93].
**Summary of the Chapter.** In this chapter we have discussed a bottom-up semantics for a first-order formulation of LO, where universal quantification is used as a primitive for generating new values. We have shown that the usual ground semantics is not refined enough to capture the operational semantics of the universal quantifier. The semantics we have defined is the equivalent of the so-called non-ground success set semantics of Horn logic. The semantics is effective and complete with respect to the operational semantics, though in general evaluation can be non-terminating. As an example, we have verified that mutual exclusion holds for a parameterized version of a distributed test-and-lock protocol, where universal quantification is used to generate new resource identifiers.

In the next chapter we will apply the methodology and techniques discussed in this chapter to the security protocol domain. We will encode security protocols using first-order LO specifications, and we will see how interesting properties like, e.g., confidentiality, can be automatically validated using our backward reachability strategy.
Chapter 11

A Case-Study: Security Protocols

In this chapter we will illustrate and apply the methodology for system verification discussed in this thesis to the security protocol domain. Given the enormous increase in the development and applications of networked and distributed systems which has taken place in recent years, the field of security protocols seems to be a particularly interesting and relevant test bed for any verification tool, as witnessed by the plentiful and varied literature on this subject. The design and implementation of security protocols are difficult and error-prone. As a meaningful example, we mention the case of Needham-Schroeder authentication protocol [NS78], which only recently (seventeen years after its publication) has been shown to be flawed [Low95]. That’s why formal techniques to specify and analyze protocols have deserved great attention in the community.

In this chapter we will focus in particular on authentication protocols. Authentication protocols are a common means to synchronize and distribute information between agents (called principals) operating over a network. Typically, execution of a protocol run should provide the involved principals with some secret information, e.g. a cryptographic key, or knowledge about the other principals. Authentication protocols should be reliable enough to be used in a potentially compromised environment, so as to prevent a malicious intruder to cheat another agent about its own identity, or to get unauthorized information.

After presenting some background on security, in this chapter we will show how our verification tool can be used to specify, validate or detect attacks in security protocols. We will exemplify our methodology through a careful selection of different protocols presented in the literature on security. Formally, we will encode protocols in a fragment of first-order linear logic with universal quantification, corresponding to the language \( LO_\forall \) presented in Chapter 10. Our inspiration came from [CDL+99], where an equivalent logical fragment, using existential quantification, is presented. Following [CDL+99], we will use the universal quantifier to model the generation of new values (called nonces) which are commonly
used in the design of security protocols. A remarkable feature of our approach is that we can reason about *parametric* and *open* systems. As a result, we do not have to pose any artificial limitation on the number of parallel runs of a given protocol. In particular, we allow an agent to take part into different protocol sessions at the same time.

This chapter extends some preliminary results reported in [Boz01].

### 11.1 Introduction

In a distributed environment, it is necessary to provide the agents with a way to ensure each other’s identity, and to exchange confidential information in a secure manner. This is the role of authentication protocols. Typically, running an authentication protocol provides the involved principals with information about the environment and the other principals, which can subsequently be used to take decisions on how to act. Clearly, it is crucial to ensure that these information may not be tampered with by a malicious intruder. In general, it is very difficult or even impossible to prevent every form of malicious *listening* over a network. The design of authentication protocols must take into account the possibility of messages sent over a network to be intercepted, and the presence of malicious agents which can impersonate the role of an honest participant in a given protocol run. Authentication protocols should be designed in such a way to be resistant to every form of *attack*, so as not to disclose private information to unauthorized agents, and to prevent an agent from cheating other agents about its own identity. Cryptographic primitives are a common means to achieve these goals, but they are not sufficient to ensure authentication.

In our view, formal verification of protocols is a process which is made up of two distinct, though related, phases: (1) choosing a suitable formalism to represent the environment and protocol execution (typically, in the form of a specification logic), and to specify security properties; (2) enriching the logic with a (hopefully complete and automatic) tool to formally verify that a given specification is correct with respect to a given property.

Neither of the two phases should be underestimated. In particular, phase (1) is responsible for fixing a suitable *abstraction* to represent protocol execution. Typically, simplifying assumptions on messages exchanged by agents and on cryptographic primitives are used in order to abstract from implementation-dependent details. For instance, according to the so-called *Dolev-Yao model* [DY83], messages are considered as indivisible abstract values, rather than sequences of bits, and encryption is modeled in an idealized way. Therefore, phase (2) alone is not sufficient to guarantee correctness. One should never forget that correctness results strongly rely on the abstraction chosen in phase (1), for instance using the Dolev-Yao model we will not be able to prove that a protocol is resistant to implementation-dependent attacks due to the particular crypto-algorithm used. Even most importantly, correctness results rely upon protocol specification given in phase (1) being faithful (in a
sense to be clarified) with respect to the given abstraction one has in mind.

Here, we present a comprehensive logical environment for the specification and analysis of protocols. Our position with respect to phase (1) mentioned above is to use a specification logic which is as close as possible to the usual informal presentation which can be found in the literature on security protocols. In particular, following [CDL+99] we will use a multiset-rewriting-like formalism to represent a given set of principals executing protocol sessions by exchanging messages over a network, with universal quantification providing a logical and clean way to express creation of nonces. As far as phase (2) is concerned, protocol verification will be carried out using the backward search strategy based on the bottom-up semantics for linear logic described in this thesis.

In particular, our approach is well suited to verify properties which can be specified by means of minimality conditions (e.g., a given state is unsafe if there are at least two principals which have completed the execution of a protocol and a given shared secret has been unintentionally disclosed to a third malicious agent). Our verification algorithm has connections both with (symbolic) model checking and with theorem proving. Unlike traditional approaches based on model-checking, we can reason about parametric, infinite-state systems, thus we do not pose any limitation on the number of parallel runs of a given protocol. We also allow a principal to take part into different sessions at the same time.

First of all, in the next section we give a brief overview about security and in particular about authentication protocols.

### 11.2 Some Background on Authentication

In this section we briefly discuss some background on authentication protocols and we fix some terminology and notations which will be used throughout the chapter.

Authentication protocols are used to coordinate the activity of different parties (e.g. users, hosts, or processes) operating over a network. These parties are usually referred to as principals in security literature. Execution of an authentication protocol can take place for various reasons. Principals can run an authentication protocol, e.g., to get some kind of information (e.g. cryptographic keys), or to make sure that another entity is operational, or to exchange private information with other principals using a secure communication channel. In general, a successful run of an authentication protocol will affect the subsequent decisions which the involved principals may take.

An authentication protocol generally consists of a sequence of messages exchanged between two or more principals (e.g. two users and a coordinating entity acting as a server). The form and number of exchanged messages is usually fixed in advance and must conform to a specific format. In general, a given principal can take part into a given protocol run in
different ways, e.g. as the **initiator** of the protocol or the **responder** (it is usually said that a principal can have different **roles**). Often, a principal is allowed to take part into different protocol runs simultaneously, possibly with different roles.

The design of authentication protocols must take into account the possibility of messages to be intercepted, and the presence of malicious agents which can impersonate the role of an honest principal. One of the key issues in authentication is to ensure **confidentiality**, i.e., to avoid private information being disclosed to unauthorized clients. Another issue is to prevent malicious principals from cheating by impersonating other principals. In general, a principal should have enough information to ensure that every message received has been created recently (as part of the current protocol run) and by the principal who claims to have sent it. Replaying of old messages should be detected. Authentication protocols must be designed in such a way to be resistant to every possible form of **attack**. In particular, interception of messages can prevent completion of a protocol run, but should never cause a leak of information or compromise security.

### 11.2.1 Cryptographic Prerequisites

Cryptographic primitives are a fundamental, though not sufficient, ingredient of authentication protocols. A message to be transmitted over a network is usually referred to as **plaintext** or **datagram**. The task of a cryptographic algorithm is to convert the given message to a form which is unintelligible to anyone else except the intended receiver. The conversion phase is called **encryption** and usually depends on an additional parameter known as **encryption key**, whereas the encoded message is referred to as **ciphertext** or **cryptogram**. The reverse phase of decoding is called **decryption**, and usually requires possession of the corresponding **decryption key**.

In **symmetric key cryptography**, the encryption key and the decryption key can be easily obtained from each other by public techniques (usually they are identical). Security of communication requires the keys to be kept secret between the relevant principals. The exact way encryption works depends on the particular crypto-algorithm used. Most algorithms work by encrypting **blocks** of plaintext at a time. The most famous of this class is the so-called Data Encryption Standard (**DES**) [DES76]. In **public key cryptography**, no secret is shared between communicating principals. Each principal $A$ has a pair of keys, the first one being the **public** key, and the other the **private** key. The public key is made available and can be used to encrypt messages for principal $A$, whereas the private key is only known to $A$, who can use it to decrypt incoming messages. Some public key algorithm allow the private key to be used for encryption and the public key to be used for decryption. This mechanism is used to guarantee **authenticity** (rather than **confidentiality**) of messages. The most famous algorithm in this class was developed by Rivest, Shamir and Adleman, and is universally known as **RSA** [RSA78].
Authentication protocols (see [CJ97] for a survey) are usually classified depending on the cryptographic approach taken, e.g. symmetric key or public key protocols. Furthermore, a distinction is also made between protocols which use one or more trusted third parties (e.g. a central distribution key server) and protocols which do not.

11.2.2 A Classification of Attacks

Authentication protocols can be compromised by different forms of attacks. We present below a quick and rather informal classification (see also [CJ97]). Some examples of attacks of known protocols will be illustrated in Section 11.4.

**Freshness Attacks:** a freshness attack typically takes place when a principal is induced to accept, as part of the current protocol run, an old message which is currently being replayed by a malicious intruder. For instance, a principal can be led to accept an old and possibly compromised key as a new and legitimate one. Mechanisms for ensuring freshness of messages (e.g. timestamps) are often used to prevent such kind of attacks;

**Type Flaw Attacks:** a type flaw attack takes place when a principal is induced to erroneously interpret the structure of the current message. This may happen because, at the concrete level, a message is nothing but a flat sequence of bits. Mechanisms to prevent this kind of attacks usually rely on enriching messages with redundant information about their internal structure;

**Parallel Session Attacks:** a parallel session attack is usually carried out by a malicious intruder which forms messages for a given protocol run using messages coming from another legitimate session which is executed concurrently;

**Implementation Dependent Attacks:** implementation dependent attacks are very subtle and can depend on a number of ways a given protocol is implemented. A typical area in which implementation dependent attacks can arise is given by the subtleties of the particular crypto-algorithm used and its interaction with specific protocols.

Other forms of attacks include, e.g., binding attacks and encapsulation attacks. We refer to [CJ97] for a discussion and several examples of all the different forms of attacks.

11.2.3 The Dolev-Yao Intruder Model

Most formal approaches to protocol specification and analysis, including ours, are based on a set of simplifying assumptions, which is known as the Dolev-Yao intruder model. This
model has been developed on the basis of some assumptions described by Needham and Schroeder in [NS78] and by Dolev and Yao in [DY83].

According to the so-called Dolev-Yao model [DY83], messages are considered as indivisible abstract values, instead of sequences of bits. Furthermore, the details of the particular crypto-algorithm used are abstracted away, giving rise to a black-box model of encryption (perfect encryption). This set of assumptions simplifies protocol analysis, although it has the drawback of preventing the discovery of implementation dependent attacks.

There seems to be no standard presentation of the Dolev-Yao intruder model. In general, it consists of a set conservative assumptions on the potentialities of any possible attacker. Typically, this model tries to depict a worst-case scenario, in which there is an intruder who has complete control of the network, so that he/she can intercept messages, block further transmission and/or replay them at any time, possibly modifying them. The intruder works by decomposing messages (provided he/she knows the key they are encrypted with), and composing new messages with the information in his/her possession. The intruder can also use an internal memory to store information, in general he/she knows the identity and, in the case of public-key encryption, the public keys of the other principals. The intruder is supposed not to know the private keys of the other principals, unless they have been disclosed in some way (clearly, without this latter assumption any protocol is breakable in general).

It has been proved that the Dolev-Yao intruder is, so to say, the most powerful attacker [Cer01a] for the given model under consideration (e.g. perfect encryption), in the sense that it can simulate the activity of any other possible attacker. Furthermore, in [SMC00] it has been proved that it is not restrictive to consider a single Dolev-Yao intruder instead of multiple ones.

### 11.2.4 An Informal Protocol Notation

In the literature on security, protocols are usually presented by means of an informal notation. In this section we explain this notation illustrating the so-called Needham-Schroeder public-key authentication protocol [NS78] (see also Section 11.4.2).

The protocol, in the usual notation, is as follows.

1. \( A \rightarrow S : A, B \)
2. \( S \rightarrow A : \{K_b, B\}_{K_a^{-1}} \)
3. \( A \rightarrow B : \{N_a, A\}_{K_b} \)
4. \( B \rightarrow S : B, A \)
5. \( S \rightarrow B : \{K_a, A\}_{K_b^{-1}} \)
6. \( B \rightarrow A : \{N_a, N_b\}_{K_a} \)
7. \( A \rightarrow B : \{N_b\}_{K_b} \)
The protocol is run to achieve authentication between two principals $A$ and $B$. A central server $S$ is in charge of distributing the public keys of principals. Messages 3, 6, and 7 are the core of the protocol, while the purpose of messages 1, 2, 4 and 5 is to get public keys from the central authority. The notation $\{M\}_K$ indicates a message with content $M$ and encrypted with a key $K$. Also, by convention $A, B, \ldots$ indicate principal identifiers, $K_a$ and $K_b$ denote, respectively, $A$'s and $B$'s public keys, and $K_s^{-1}$ is the private key of server $S$ (encryption with the private key is used to ensure authenticity, see Section 11.2.1).

Now, the protocol has the following structure. A given principal $A$ acts as **initiator** of the protocol, and asks the central authority for $B$’s public key (message 1). The central authority sends back to $A$ the required key (message 2: $B$’s identity is included in the message to prevent attacks based on diverting key deliveries). Principal $A$ creates a **nonce** (i.e., a newly generated value), called $N_a$, and sends it to $B$ together with its own identity (message 3), encrypting the message with $B$’s public key. Upon receiving this message, principal $B$ decrypts the message, and in turn asks the central authority for the public key of $A$ (message 4). After getting the server’s reply (message 5), principal $B$ generates a new nonce $N_b$, and sends both nonces, $N_a$ and $N_b$, to $A$, encrypting the message with $A$’s key (message 6). When $A$ gets this message, a check is made that it contains the previously generated nonce $N_a$, and, if so, a new message, encrypted with $B$’s key and including the last nonce $N_b$, is sent to $B$ (message 7). The protocol is successfully completed provided $B$ gets the previously generated nonce $N_b$.

Completion of the protocol should convince $A$ about $B$’s identity (and vice versa) and also provide $A$ and $B$ with two shared values ($N_a$ and $N_b$) which they could use afterwards for authentication purposes. The use of **nonces** is ubiquitous in authentication protocols. Intuitively, a nonce should be considered as some sort of **random** and **unguessable** value. The purpose of nonces is to prevent a malicious intruder from attempting to break a given protocol by sending messages and pretending they have been generated by someone else. To exemplify, an attacker could possibly pretend to be principal $B$, intercept message 6 and replace it with a different message. However, the message created by the attacker will never be accepted as a legitimate message by $A$, unless it contains the nonce $N_a$. Unfortunately (for the attacker) nonce $N_a$ is not known except to $B$ (and $A$, of course), because only $B$ can decrypt message 3.

Intuitively, assuming $A$ and $B$ behave honestly and their private keys are not known to anyone else, and assuming nonces are not guessable, the protocol should prevent a malicious intruder to impersonate one of the two principals. However, under certain conditions, this protocol fails to achieve authentication [Low95]. We will discuss this point in Section 11.4.2.
11.3 Specifying Authentication Protocols

In Section 11.4, we will illustrate the specification and analysis of different examples of protocols. Our approach is based on the fragment of first-order linear logic and on the bottom-up evaluation algorithm explained in Chapter 10. As discussed in Part I, our specification language has a natural correspondence with multiset rewriting systems (see in particular Section 5.4), and, in fact, it has been inspired by the approach taken in [CDL+99]. In this section we introduce, rather informally, some generalities about the way we will use to specify protocols in Section 11.4.

First of all, we need a representation for the entities (e.g. principals and messages) involved. In particular, we will use a notation like

\[ pr(id, s) \]

to denote a principal with identifier \( id \) and internal state \( s \). The internal state \( s \) can store information about an ongoing execution of any given protocol (for instance, the identifier of another principal, which step of the protocol has been executed, the role of the principal, and so on). Typically, the state \( s \) will be a term like \( \text{init} \) (indicating the initial state of a principal, before protocol execution), or a term like \( \text{step}_i(data) \), where the constructor \( \text{step}_i \) denotes which is the last step executed and \( data \) represents the internal data of a given principal. In general, we allow more than one atom \( pr(id, s) \) inside a given configuration. In this way, we can model the possibility of a given principal to take part into different protocol runs, possibly with different roles.

Messages sent over a given network can in turn be represented by terms like

\[ n(\text{mess}_\text{content}) \]

where \( \text{mess}_\text{content} \) is the content of the message. Depending on the particular protocol under consideration, we can fix a specific format for messages. For instance, a message encrypted with the public key of a principal \( a \) could be represented as the term \( \text{enc}(\text{pubk}(a), \text{mess}_\text{content}) \).

Finally, we will use the Dolev-Yao intruder model (see Section 11.2.3) and the associated assumptions. In particular, as explained in Section 11.2.3, we need a way to store the intruder knowledge. We will use terms such as

\[ m(\text{inf}) \]

\[ m(\text{inf}) \]

\[ m(\text{inf}) \]

to represent the information in possession of the intruder (\( m \) stands for the internal memory of the intruder). At any given instant of time, we can think of the current state of a given system as a multiset of atoms representing principals and messages currently on the network, and the intruder knowledge.
Following [CDL+99], we represent the environment in which protocol execution takes place by means of: a protocol theory, which includes rules for every protocol role (typically, one rule for every step of the protocol), and an intruder theory, which formalizes the set of possible actions of a malicious intruder who tries to break the protocol. In addition, it is possible to have additional rules for the environment. Rules assume the general format

$$F_1 \otimes \ldots \otimes F_n \leftarrow \forall X_1 \ldots \forall X_k . (G_1 \otimes \ldots \otimes G_m)$$

where $F_i$, $G_i$ are atomic formulas (representing e.g. principals or messages) and $X_i$ are variables. As explained in Section 10, the standard semantics for the universal quantifier requires new values to be chosen before application of a rule. We use this behaviour to encode nonce generation during protocol runs. As a result, we get for free the assumption (required by the Dolev-Yao model) that nonces are not guessable. In the following we will use these notational conventions: free variables inside a rule are always implicitly universally quantified, and variables are written as upper-case identifiers.

As far the specification of the initial states is concerned, we take here a slightly different approach w.r.t. the specifications in the previous chapters. Namely, we allow a partial specification of the initial states (see Appendix A.3). This strategy is more flexible in that it may help us to find additional hypotheses under which a given attack might take place. As a general rule, the partial specification of the initial states we have chosen requires every principal to be in his/her initial state (represented by the term $\text{init}$) at the beginning of protocol execution. As usual, we express security properties by means of their negation, i.e., using formulas representing their violation. We will analyze properties which can be expressed by means of minimality conditions.

Finally, we conclude this section by collecting together some rules which are common to all the examples presented in Section 11.4. In particular, we have two rules for the environment:

$$e_1 \quad \text{pr}(Z, S) \leftarrow \text{pr}(Z, S) \otimes \forall \text{ID} . (\text{pr} \langle \text{ID}, \text{init} \rangle)$$
$$e_2 \quad \text{pr}(Z, S) \leftarrow \text{pr}(Z, S) \otimes \forall \text{pr}(Z, \text{init})$$

The first one allows creation of new principals (we use the universal quantifier to generate new identifiers for them), whereas the second rule allows creation of a new instance of a given principal (this allows a principal to start another execution of a given protocol with a new and possibly different role). Both rules can be fired at run-time, i.e., during the execution of a given protocol. We use the term $\text{init}$ to denote the initial state of any given principal. We also have the following two rules for the intruder theory:

$$t_1 \quad \text{pr}(Z, S) \leftarrow \text{pr}(Z, S) \otimes m(Z)$$
$$t_2 \quad \text{pr}(Z, S) \leftarrow \text{pr}(Z, S) \otimes \forall \text{N} . (m(\text{N}))$$

The first one allows the intruder to store the identifier of any principal, whereas the second rule formalize the capability of the intruder to generate new values (e.g. nonces).
11.4 Examples

In this section we present the specification and analysis of different authentication protocols, taken from the literature on security. Some generalities concerning the encoding we will use in the following have been discussed in Section 11.3. We summarize the experimental results for this section, obtained using the tool presented in Appendix A, in Table 11.1 (see Section 11.4.5).

11.4.1 Millen’s $ffgg$ protocol

We begin our survey from the so-called Millen’s $ffgg$ protocol [Mil99]. Although an artificial protocol, its analysis will be a good starting point to illustrate the capabilities of our verification tool. In fact, this protocol provides an example of a parallel session attack, which requires running at least two processes for the same role. It has been proved (see [Mil99]) that no serial attacks exist, i.e., the protocol is secure if processes are serialized.

The protocol is as follows.

1. $A \rightarrow B : A$
2. $B \rightarrow A : N_1, N_2$
3. $A \rightarrow B : \{N_1, N_2, S\}_{K_b} \%\{N_1, X, Y\}_{K_b}$
4. $B \rightarrow A : N_1, X, \{X, Y, N_1\}_{K_b}$

As usual, $N_1$ and $N_2$ stand for nonces, created by principal $B$ and included in message 2. The $m\%m'$ notation, introduced in [Low98], used in message 3 represents a message which has been created by the sender according to format $m$, but is interpreted as $m'$ by the receiver. In this case, the intuition is that upon receiving message 3, $B$ checks that the first component does correspond to the first of the two nonces previously created, while no check at all is performed on the second component of the message. In message 2, $S$ stands for a secret, of the same length as a nonce, which is in possession of $B$. The security property one is interested to analyze is whether the secret $S$ can be disclosed to a malicious intruder.

We have implemented the $ffgg$ protocol through the specification shown in Figure 11.1, while the intruder theory is presented in Figure 11.2. The specification consists of a set of protocol rules (rules $p_1$ through $p_4$ in Figure 11.1) and an intruder theory (rules $\iota_1$ through
\begin{align*}
p_1 & \quad p(A, init) \not\rightarrow pr(B, init) \rightarrow pr(A, step1(B)) \not\rightarrow pr(B, init) \not\rightarrow n(plain(A)) \\
p_2 & \quad pr(B, init) \not\rightarrow n(plain(A)) \rightarrow \forall N1.\forall N2.(pr(B, step2(A, N1)) \not\rightarrow n(plain(N1, N2))) \\
p_3 & \quad pr(A, step1(B)) \not\rightarrow n(plain(N1, N2)) \rightarrow \forall S.(pr(A, step3(B, S)) \not\rightarrow n(enc(pubk(B), N1, N2, S))) \\
p_4 & \quad pr(B, step2(A, N1)) \not\rightarrow n(enc(pubk(B), N1, X, Y)) \rightarrow pr(B, step4(A)) \not\rightarrow n(plain(N1, X), enc(pubk(B), X, Y, N1))
\end{align*}

Figure 11.1: Specification of the ffgg protocol

\(i_8\) in Figure 11.2). We remind the reader that the four rules \(e_1, e_2, t_1\) and \(t_2\) discussed in Section 11.3 are in addition to the present rules.

Protocol rules directly correspond to the informal description of the ffgg protocol previously presented. We have followed the conventions outlined in Section 11.3 to model the internal state of principals. In particular, we have a term \textit{init} denoting the initial state of a principal, and the constructors \textit{step1}, \textit{step2}, \textit{step3} and \textit{step4} to model the different steps of a protocol run. At every step, each principal needs to remember the identifier of the other principal he/she is executing the protocol with. In addition, at step 2 the responder stores the first nonce created (in order to be able to perform the required check, see rule \(p_4\)), and at step 3 the initiator of the protocol remembers the secret \(S\). We have modeled the secret \(S\) using the universal quantifier, as for nonces. In this way, we can get for free the requirement that the secret initially is only known to the principal who possesses it.

Finally, we have term constructors \textit{plain}(...) and \textit{enc}(...) (to be precise, we should say a family of term constructors, we find it convenient to overload the same symbol with different arities) to distinguish \textit{plain} messages from \textit{encrypted} messages.

The intruder theory is made up of rules \(i_1\) through \(i_8\) in Figure 11.2. It is an instance of the Dolev-Yao intruder theory illustrated in Section 11.2.3. Let us discuss it in more detail. Rules \(i_1\) through \(i_4\) are decomposition rules, whereas rules \(i_5\) through \(i_8\) are composition rules. We have four rules for each of the two different kinds (composition and decomposition) of messages, dealing with the different formats of messages used in the ffgg protocol. For instance, rule \(i_1\) deals with decomposition of plain messages with one component, whereas rule \(i_4\) deals with decomposition of messages with two plain components and one encrypted component, and so on. Clearly, the intruder cannot furtherly decompose encrypted components, which are stored exactly as they are, whereas plain messages are decomposed into their atomic constituents. The intruder theory we have presented is an instance of the general Dolev-Yao intruder theory, in that intruder rules have been tailored to the particular form of messages used in the specific protocol under consideration. In general, we have followed this strategy also for the other examples we will present.
\begin{align*}
i_1 & \quad n(plain(X)) \leftarrow m(plain(X)) \\
i_2 & \quad n(plain(X, Y)) \leftarrow m(plain(X)) \not\in m(plain(Y)) \\
i_3 & \quad n(enc(X, Y, Z, W)) \leftarrow m(enc(X, Y, Z, W)) \\
i_4 & \quad n(plain(X, Y), enc(U, V, W, Z)) \leftarrow m(plain(X)) \not\in m(plain(Y)) \not\in m(enc(U, V, W, Z)) \\
i_5 & \quad m(plain(X)) \leftarrow m(plain(X)) \not\in n(plain(X)) \\
i_6 & \quad m(plain(X)) \not\in m(plain(Y)) \leftarrow m(plain(X)) \not\in m(plain(Y)) \not\in n(plain(X, Y)) \\
i_7 & \quad m(enc(X, Y, Z, W)) \leftarrow m(enc(X, Y, Z, W)) \not\in n(plain(X, Y, Z, W)) \\
i_8 & \quad m(plain(Y)) \not\in m(plain(Y)) \not\in m(enc(U, V, W, Z)) \leftarrow m(plain(X)) \not\in m(plain(Y)) \not\in m(enc(U, V, W, Z)) \not\in n(plain(X, Y), enc(U, V, W, Z))
\end{align*}

Figure 11.2: Intruder theory for the $ffgg$ protocol

in the following. We refer to Section 11.5 for a discussion on how this approach could be generalized. In the case of $ffgg$ protocol, we have not given the intruder any capability to decrypt messages. This hypothesis can be relaxed (see for instance the analysis of Needham-Schroeder protocol in Section 11.4.2). The present specification is sufficient for our purposes.

Now, we can simply specify the set of unsafe states as follows:

\[ u) \quad pr(alice, step3(bob, S)) \not\in pr(bob, step4(alice)) \not\in m(plain(S)) \leftarrow \top \]

Namely, we consider a configuration unsafe if there exist two honest principals, say alice and bob, who have run the protocol to completion (i.e., they have completed, respectively, step 4 and step 3) and the secret $S$ has been disclosed to the intruder (i.e., it is eventually stored in the intruder’s internal memory).

Running our verification algorithm, we automatically find an attack, whose trace is shown in Figure 11.3. Besides the usual notational conventions, we have fixed the following shortened notations: alice has been shortened to $al$; $p^i_d$ stands for a principal $p$ after execution of step number $i$ with internal data $d$; $p^{init}$ stands for a principal in its initial state; we have omitted the $plain$ term constructor for plain messages; we have noted encrypted messages using the usual protocol notation; $M(x, y, \ldots)$ stands for the multiset $m(x), m(y), \ldots$; finally, $\Sigma_1 = \Sigma, n1, n2, \Sigma_2 = \Sigma, n1, n2, n3, n4$, and $\Sigma_3 = \Sigma, n1, n2, n3, n4, s$. The attack is exactly the parallel session one described in [Mil99]. We note that this attack is also an example of a type flaw attack, in that it relies on the secret $S$ be passed as a nonce (under the hypothesis that the lengths of the respective fields are the same).

We also remark that we have performed some further experiments, which we don’t discuss
Figure 11.3: A parallel session attack to the ffkg protocol
in detail for the sake of brevity. In particular, we wanted to ascertain the role of the two nonces \( N_1 \) and \( N_2 \) in the \textit{ffgg} protocol. According to the informal notation for the protocol introduced at the beginning of this section, principal \( B \) only checks that the first component of message \( T \) is the nonce \( N_1 \), whereas no check is performed for the second component. We have verified that imposing the check on the second component, the \textit{ffgg} protocol is safe w.r.t. the security property and the intruder theory we have presented, while removing all checks introduces \textit{serial} attacks.

We think that this example is a good illustration of the capabilities of our verification tool. In fact, using the \textit{backward} evaluation strategy championed in this thesis, we are able to automatically find a parallel session attack, \textit{without encoding any prior knowledge} about the kind of attacks to look for. In particular, according to [Mil99] the \textit{ffgg} can be generalized to protocols which only admit \textit{higher-order} parallel attacks (i.e., attacks which take place only in presence of three or more concurrent roles for the same principal). Using our tool we can automatically find such attacks, if any exists. This distinguishes our methodology from most approaches based on model-checking, which operate on a finite-state abstraction of a given protocol, and require the number of principals and the number of roles to be fixed \textit{in advance}.

### 11.4.2 The Needham-Schroeder Protocol

In this section we analyze the Needham-Schroeder public-key authentication protocol, previously discussed in Section 11.2.4. For the sake of precision, we restrict our attention to the fragment of the Needham-Schroeder protocol where the key distribution phase (i.e., the messages exchanged with the trusted server) has been omitted.

The resulting protocol, corresponding to messages 3, 6, and 7 of Section 11.2.4, is as follows:

1. \( A \rightarrow B : \{N_a, A\}_K_b \)
2. \( B \rightarrow A : \{N_a, N_b\}_K_a \)
3. \( A \rightarrow B : \{N_b\}_K_b \)

and has been implemented using the specification illustrated in Figure 11.4 and the intruder theory in Figure 11.5. Let us discuss the rules in more detail.

The protocol rules have been encoded in the same way as for the \textit{ffgg} protocol of Section 11.4.1, and directly correspond to the informal notation previously introduced. This time, all messages sent over the network are encrypted, therefore we have modeled them using atomic formulas like \( n(\text{pubk}(id), \text{mess\_content}) \), where \( id \) is a principal identifier, whereas \( \text{mess\_content} \) is a term of the form \( \text{mess}(\ldots) \). Internal states of principals have been enriched in order to express the security violations we will present in the following.

The intruder theory is shown in Figure 11.5. We consider here a more extended intruder
Figure 11.4: Specification of the Needham-Schroeder protocol

theory w.r.t. the one for the ffyg protocol in Figure 11.2. Namely, we give the intruder the capability to decrypt messages addressed to himself/herself. Given that private keys are not exchanged, the intruder will never be able to know the private keys of some other principal, therefore we still assume that the intruder is not able to decrypt messages other than the ones intended for himself/herself. Rule \(i_1\) and \(i_2\) deal with interception: the intruder can intercept any message and replay it (possibly more than once) at any given time in the future. Rule \(i_3\) states that the intruder can decompose any message addressed to himself/herself. Rules \(i_4\) and \(i_5\) are the usual rules for decomposition of messages (with one or two components). Rules \(i_6\) and \(i_7\) are the corresponding rules for composition of messages (with one or two components). Finally, using rule \(i_8\) the intruder can send a newly composed message to any principal.

Now, the specification of unsafe states is as follows:

\[
\begin{align*}
\text{u}_1 & \quad \text{pr}(\text{alice}, \text{step}3(\text{NA}, \text{NB}, \text{bob})) \not\Rightarrow \text{m}(\text{NA}) \leftarrow \top \\
\text{u}_2 & \quad \text{pr}(\text{alice}, \text{step}3(\text{NA}, \text{NB}, \text{bob})) \not\Rightarrow \text{m}(\text{NB}) \leftarrow \top \\
\text{u}_3 & \quad \text{pr}(\text{bob}, \text{step}4(\text{NA}, \text{NB}, \text{alice})) \not\Rightarrow \text{m}(\text{NA}) \leftarrow \top \\
\text{u}_4 & \quad \text{pr}(\text{bob}, \text{step}4(\text{NA}, \text{NB}, \text{alice})) \not\Rightarrow \text{m}(\text{NB}) \leftarrow \top
\end{align*}
\]

Namely, a state is unsafe if there exist two principals, say \textit{alice} and \textit{bob}, such that either \textit{alice} has run the protocol to completion with \textit{bob}, or \textit{bob} with \textit{alice}, and at least one of the two nonces has been disclosed to the intruder. We call the security property specified by the above rules strong correctness.

We can now run our verification tool on the resulting specification. As observed by Lowe [Low95], Needham-Schroeder protocol is not safe w.r.t. the above security properties. In fact, we find the attack shown in Figure 11.6, corresponding to the one presented in [Low95] (we have followed the same conventions as in Figure 11.3, with \(\Sigma_1 = \Sigma, na\) and \(\Sigma_2 = \Sigma, na, nb\)). The attack takes place because \textit{alice} decides to contact the intruder, without knowing he/she is cheating. Thus, the intruder is able to impersonate \textit{alice} and
\(i_1\) \(n(\text{pubk}(A), M) \vdash \text{intercept}(n(\text{pubk}(A), M))\)
\(i_2\) \(\text{intercept}(n(\text{pubk}(A), M)) \vdash \text{intercept}(n(\text{pubk}(A), M)) \not\subseteq n(\text{pubk}(A), M)\)
\(i_3\) \(n(\text{pubk}(\text{intruder}), M) \vdash \text{dec}(\text{M})\)
\(i_4\) \(\text{dec}(\text{mess}(M)) \vdash m(M)\)
\(i_5\) \(\text{dec}(\text{mess}(M, N)) \vdash m(M) \not\subseteq m(N)\)
\(i_6\) \(m(M) \vdash m(M) \not\subseteq \text{comp(\text{mess}(M))}\)
\(i_7\) \(m(M) \not\subseteq m(N) \vdash m(M) \not\subseteq m(N) \not\subseteq \text{comp(\text{mess}(M, N))}\)
\(i_8\) \(\text{comp}(C) \vdash n(\text{pubk}(Z), C)\)

Figure 11.5: Intruder theory for the Needham-Schroeder protocol

\[
P \vdash \Sigma_2 \Gamma_t, a^{i_3}_{na,nb,i}, i^{\text{init}}, m(\text{al}, nb) \quad \text{M} \quad b^{(i_3)}
\]

\[
P \vdash \Sigma_2 a^{i_3}_{na,nb,i}, bob^{i_4}_{na,nb,al}, i^{\text{init}}, m(\text{na}, \text{al}, nb) \quad b^{(i_4)}
\]

\[
P \vdash \Sigma_2 a^{i_3}_{na,nb,i}, bob^{i_4}_{na,nb,al}, i^{\text{init}}, m(\text{na}, \text{al}, nb), \text{comp(\text{mess}(\text{nb}))} \quad b^{(i_5)}
\]

\[
P \vdash \Sigma_2 a^{i_3}_{na,nb,i}, bob^{i_4}_{na,nb,al}, i^{\text{init}}, m(\text{na}, \text{al}, nb), \text{dec(\text{mess}(\text{nb}))} \quad b^{(i_6)}
\]

\[
P \vdash \Sigma_2 a^{i_3}_{na,nb,i}, bob^{i_4}_{na,nb,al}, i^{\text{init}}, m(\text{na}, \text{al}, nb), n(\{\text{mess(\text{nb})}\})_{\text{Kbob}} \quad b^{(i_7)}
\]

\[
P \vdash \Sigma_1 a^{i_3}_{na,i}, bob^{i_4}_{na,al}, i^{\text{init}}, m(\text{na}, \text{al}, nb), n(\{\text{mess(\text{na, nb})}\})_{\text{Kbob}} \quad b^{(i_8)}
\]

\[
P \vdash \Sigma_1 a^{i_3}_{na,i}, bob^{i_4}_{na,al}, i^{\text{init}}, m(\text{na}, \text{al}, nb), \text{comp(\text{mess}(\text{na, al}))} \quad b^{(i_9)}
\]

\[
P \vdash \Sigma_1 a^{i_3}_{na,i}, bob^{i_4}_{na,al}, i^{\text{init}}, m(\text{na}, \text{al}), \text{dec(\text{mess}(\text{na, al}))} \quad b^{(i_10)}
\]

\[
P \vdash \Sigma_1 a^{i_3}_{na,i}, bob^{i_4}_{na,al}, i^{\text{init}}, n(\{\text{mess(\text{na, al})}\})_{\text{Kbob}} \quad b^{(i_11)}
\]

\[
P \vdash \Sigma a^{i_3}_{na,i}, bob^{i_4}_{na,al}, i^{\text{init}}, m(\text{na}, \text{al}) \quad b^{(i_12)}
\]

Figure 11.6: An attack to the Needham-Schroeder protocol

204
cheat \textit{bob}. Note that as a result the protocol has been broken from the point of view of \textit{bob}. In fact, \textit{bob} thinks he has got authentication with \textit{alice} and that provided \textit{alice} is honest, the nonces have not been disclosed to anyone else (which is false), whereas from the point of view of \textit{alice}, she correctly thinks to have established authentication with the intruder (the nonces have been disclosed to \textit{bob}, but only because the intruder is cheating and \textit{alice} does not know that).

With this in mind, we can now try the following \textit{stronger} security violations (we call the corresponding security property \textit{weak correctness}).

\begin{align*}
  u'_1) & \quad pr(alice, step3(NA, NB, bob)) \not\Rightarrow pr(bob, step4(NA, NB, alice)) \not\Rightarrow m(NA) \leftarrow \top \\
  u'_2) & \quad pr(alice, step3(NA, NB, bob)) \not\Rightarrow pr(bob, step4(NA, NB, alice)) \not\Rightarrow m(NB) \leftarrow \top
\end{align*}

Namely, we try to ascertain whether it is possible that two honest principals \textit{alice} and \textit{bob} both believe to have completed the protocol with each other, and still at least one of the two nonces has been disclosed to the intruder (as the reader can easily verify, this is not the case for the trace of the previous attack). This time the verification algorithm terminates, proving that Needham-Schroeder protocol is safe with respect to this property.

We conclude this section by showing how the methodology of \textit{invariant strengthening}, discussed in Section 6.1.2.1, can improve the verification algorithm performance. Namely, we augment the set of security violations with this further rule:

\begin{align*}
  u'_3) & \quad pr(alice, step1(NA, bob)) \not\Rightarrow m(NA) \leftarrow \top
\end{align*}

Intuitively, this violation is never met. In fact, the nonce NA is created by \textit{alice} during step 1 and sent to \textit{bob}. If \textit{bob} is honest, he will never send it to the intruder. Adding this rule has the effect of accelerating convergence of the fixpoint computation (see the experimental results in Table 11.1). Note that the following invariant is instead violated (the attack follows the same scheme of the one in Figure 11.6):

\begin{align*}
  u'_4) & \quad pr(bob, step2(NA, NB, alice)) \not\Rightarrow m(NB) \leftarrow \top
\end{align*}

We remind that adding rules (including axioms) to the theory is always sound, in the sense that if no attack is found in the augmented theory, no attack can be found in the original one.

### 11.4.3 Corrected Needham-Schroeder

As observed by Lowe [Low95], the Needham-Schroeder protocol can be fixed with a small modification. The problem with the original protocol is that the second message exchanged does not contain the identity of the responder. Adding the responder’s identity to this
message prevents the intruder from replaying it, because now the initiator is expecting a message from the intruder. The corrected version of Needham-Schroeder protocol is

\begin{align*}
1. & \quad A \rightarrow B : \{N_a, A\}_{K_a} \\
2. & \quad B \rightarrow A : \{B, N_a, N_b\}_{K_a} \\
3. & \quad A \rightarrow B : \{N_b\}_{K_b}
\end{align*}

We need to make minor modifications to our previous specification. Namely, we modify rules \(p_2\) and \(p_3\) by adding the additional argument \(B\):

\begin{align*}
\varphi_2' & \quad pr(B, \text{init}) \nexists \ n(pubk(B), mess(NA, A)) \rightarrow N.A. (pr(B, \text{step2}(NA, NB, A)) \nexists \\
& \quad n(pubk(A), mess(B, NA, NB))) \\
\varphi_3' & \quad pr(A, \text{step1}(NA, B)) \nexists \ n(pubk(A), mess(B, NA, NB)) \rightarrow \\
& \quad pr(A, \text{step3}(NA, NB, B)) \nexists \ n(pubk(B), mess(NB))
\end{align*}

and we add two rules for composition and decomposition of messages with three components:

\begin{align*}
\psi_9 & \quad dec(mess(M, N, O)) \rightarrow m(M) \nexists m(N) \nexists m(O) \\
\psi_{10} & \quad m(M) \nexists m(N) \nexists m(O) \rightarrow m(M) \nexists m(N) \nexists m(O) \nexists \text{comp}(mess(M, N, O))
\end{align*}

We can now use our algorithm to automatically verify if the protocol satisfy strong correctness (axioms \(u_1\) through \(u_4\)). As in Section 11.4.2, we can accelerate convergence by means of invariant strengthening (see the experimental results in Table 11.1). We can use the two invariants \(u_3'\) and \(u_4'\) discussed in Section 11.4.2, which should now both hold. The verification algorithm terminates, as expected, proving that the modified version of the Needham-Schroeder protocol is correct w.r.t. the notion of strong correctness.

### 11.4.4 The Otway-Rees Protocol

The Otway-Rees protocol [OR87] provides a typical example of a type flaw attack. It is intended for key distribution between two principals communicating with a central server by means of shared keys (the protocol assumes symmetric key encryption, see Section 11.2.1).

The protocol is as follows (the form presented here is the one given in [BAN89]).

\begin{align*}
1. & \quad A \rightarrow B : N, A, B, \{N_a, N, A, B\}_{K_{as}} \\
2. & \quad B \rightarrow S : N, A, B, \{N_a, N, A, B\}_{K_{as}}, \{N_b, N, A, B\}_{K_{bs}} \\
3. & \quad S \rightarrow B : N, \{N_a, ab\}_{K_{as}}, \{N_b, ab\}_{K_{bs}} \\
4. & \quad B \rightarrow A : N, \{N_a, ab\}_{K_{as}}
\end{align*}

The protocol is run between two principals \(A\) and \(B\), communicating with a trusted server \(S\) by means of shared keys \(K_{as}\) and \(K_{bs}\). The purpose of the protocol is to get a new key \(K_{ab}\).
generated by the trusted server, to be used as a shared key in subsequent communications between $A$ and $B$. At the first step, principal $A$ generates a nonce $N$, to be used as a run identifier, and a nonce $N_a$, and sends to $B$ the plaintext $N$, $A$, $B$ and an encrypted message, readable only by the server $S$, of the form shown. In turn, principal $B$ generates a nonce $N_b$ and forwards $A$’s message to $S$, together with a similar encrypted component. The server checks that the $N$, $A$, $B$ components in both messages match, and, if so, generates a new key $K_{ab}$ and replies to $B$ with message 3 above, which includes a component intended for $B$ and one for $A$. The component intended for $A$ is forwarded to him/her by $B$ with message 4.

We have encoded the Otway-Rees protocol with the rules in Figure 11.7. Let us discuss them in more detail. The encoding is similar to the one used for Millen’s \texttt{ffgg} protocol (see Section 11.4.1), in particular we distinguish between the plaintext part of messages (term constructor \texttt{plain}) and the encrypted part (term constructor \texttt{enc}). Furthermore, we use a concatenation operator \texttt{cons} to glue together different components inside the plaintext or the ciphertext. In this way, we are able to capture type flaw attacks. We use an identifier $s$ for the trusted server, and the notation $sk(id, s)$ to denote the shared key between principal $id$ and $s$. Rules $p_1$ through $p_5$ are a direct translation of the protocol steps written in the usual notation. As usual, we have denoted the internal state of principals by means of term constructors like \texttt{step}. When the protocol is run to completion, the internal state of

\begin{verbatim}
p_1) \texttt{pr}(A, init) \not\rightarrow \texttt{pr}(B, init) \rightarrow \forall N.\forall N_A.
(n(\texttt{plain}(\texttt{cons}(N, \texttt{cons}(A, B)))), \texttt{enc}(sk(A, s), \texttt{cons}(N_A, \texttt{cons}(N, \texttt{cons}(A, B)))))) \not\rightarrow \texttt{pr}(A, \texttt{step}1(B, N, NA)))
p_2) n(\texttt{plain}(\texttt{cons}(N, \texttt{cons}(A, B)))), \texttt{enc}(sk(A, s), \texttt{cons}(N_A, \texttt{cons}(N, \texttt{cons}(A, B)))))) \not\rightarrow \texttt{pr}(B, \texttt{init}) \rightarrow \forall N_B.(\texttt{pr}(B, \texttt{step}2(A, N, NB)) \not\rightarrow \texttt{enc}(sk(B, s), \texttt{cons}(N_B, \texttt{cons}(N, \texttt{cons}(A, B))))))
p_3) n(\texttt{plain}(\texttt{cons}(N, \texttt{cons}(A, B)))), \texttt{enc}(sk(A, s), \texttt{cons}(N_A, \texttt{cons}(N, \texttt{cons}(A, B))))), \texttt{enc}(sk(B, s), \texttt{cons}(N_B, \texttt{cons}(N, \texttt{cons}(A, B)))))) \not\rightarrow \texttt{pr}(s, \texttt{init}) \rightarrow \forall KAB.n(\texttt{plain}(N), \texttt{enc}(sk(A, s), \texttt{cons}(N_A, KAB))), \texttt{enc}(sk(B, s), \texttt{cons}(N_B, KAB))))
p_4) n(\texttt{plain}(N), \texttt{enc}(sk(A, s), \texttt{cons}(N_A, KAB))), \texttt{enc}(sk(B, s), \texttt{cons}(N_B, KAB)))) \not\rightarrow \texttt{pr}(B, \texttt{step}2(A, N, NB)) \rightarrow \texttt{pr}(B, \texttt{step}3(A, KAB)) \not\rightarrow \texttt{enc}(sk(A, s), \texttt{cons}(N_A, KAB)))
p_5) \texttt{pr}(A, \texttt{step}1(B, N, NA)) \rightarrow \texttt{n}(\texttt{plain}(N), \texttt{enc}(sk(A, s), \texttt{cons}(N_A, KAB)))) \rightarrow \texttt{pr}(A, \texttt{step}4(B, KAB))
\end{verbatim}

Figure 11.7: Specification of the Otway-Rees protocol
i_1) n(plain(X), enc(sk(K, s), Y)) ⊑ m(plain(X)) ⊝ m(enc(sk(K, s), Y))
i_2) m(plain(X), enc(sk(K, s), Y), enc(sk(H, s), W)) ⊑ m(plain(X)) ⊝ m(enc(sk(K, s), Y)) ⊝ m(enc(sk(H, s), W))
i_3) m(plain(cons(X, Y))) ⊑ m(plain(cons(X, Y))) ⊝ m(plain(X)) ⊝ m(plain(Y))
i_4) m(plain(X)) ⊝ m(plain(Y)) ⊑ m(plain(X)) ⊝ m(plain(Y)) ⊝ m(plain(cons(X, Y)))
i_5) m(plain(X)) ⊝ m(enc(sk(K, s), Y)) ⊑ m(plain(X)) ⊝ m(enc(sk(K, s), Y)) ⊝ n(plain(X), enc(sk(K, s), Y))
i_6) m(plain(X)) ⊝ m(enc(sk(K, s), Y)) ⊝ m(enc(sk(H, s), W)) ⊑ m(plain(X)) ⊝ m(enc(sk(K, s), Y)) ⊝ m(enc(sk(H, s), W)) ⊝ m(plain(X), enc(sk(K, s), Y), enc(sk(H, s), W))

Figure 11.8: Intruder theory for the Otway-Rees protocol

each of the two involved principals contains the identity of the other principal and the new shared key obtained from the server.

The intruder theory in presented in Figure 11.8. Rules i_1 and i_2 are the usual decomposition rules for, respectively, messages with one encrypted component and with two encrypted components. Rules i_3 and i_4 allow the intruder to arbitrarily decompose and re-assemble plaintext messages, whereas encrypted messages are stored as they are. Finally, rules i_5 and i_6 allow the intruder to create new messages from stored components.

We wish to verify if the intruder can get the shared key which comes from a protocol run between two honest principals, say alice and bob. The specification of unsafe states is straightforward:

\[ u_1 \) pr(bob, step3(alice, KAB)) ⊝ m(plain(KAB)) ⊑ ⊤ \]
\[ u_2 \) pr(alice, step4(bob, KAB)) ⊝ m(plain(KAB)) ⊑ ⊤ \]

Running our verification tool, we automatically find the type flaw attack described in [CJ97]. The corresponding trace is shown in Figure 11.9 (we have followed the usual conventions, with \( Σ_i = Σ, n1, n4 \)). The attack takes place because a malicious intruder can intercept the first message, and, after stripping it of the A and B components (in the plaintext part), replay it as the last message of the protocol. The attack is successful under the hypothesis that the triple N, A, B may be erroneously accepted, by the initiator of the protocol, as the desired key. This is clearly a security flaw because the triple N, A, B is sent in clear in the first message, and therefore publicly known.

The Otway-Rees protocol provides a classical example of a type flaw attack. This kind
\[
\begin{align*}
P \vdash \Sigma_1 \top, \text{bob}^{\text{init}}, M(\{na \cdot n1 \cdot al \cdot bob\}_{K_a}, n1, al \cdot bob) & \quad \top_r \\
P \vdash \Sigma_1 a_{bob,n1,na}^{1, \text{bob}^{\text{init}}}, M(n1 \cdot al \cdot bob, \{na \cdot n1 \cdot al \cdot bob\}_{K_a}, n1, al \cdot bob) & \quad bc^{(p2)} \\
P \vdash \Sigma_1 a_{bob,n1,na}^{1, \text{bob}^{\text{init}}}, M(n1 \cdot al \cdot bob, \{na \cdot n1 \cdot al \cdot bob\}_{K_a}, n1, al \cdot bob) & \quad bc^{(i5)} \\
P \vdash \Sigma_1 a_{bob,n1,na}^{1, \text{bob}^{\text{init}}}, M(n1 \cdot al \cdot bob, \{na \cdot n1 \cdot al \cdot bob\}_{K_a}) & \quad bc^{(i3)} \\
P \vdash \Sigma_1 a_{bob,n1,na}^{1, \text{bob}^{\text{init}}}, n(n1 \cdot al \cdot bob, \{na \cdot n1 \cdot al \cdot bob\}_{K_a}) & \quad bc^{(p1)} \\
P \vdash \Sigma a^{\text{init}}, bob^{\text{init}} & 
\end{align*}
\]

Figure 11.9: An attack to the Otway-Rees protocol

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Invar</th>
<th>Steps</th>
<th>Size</th>
<th>MSize</th>
<th>Time</th>
<th>Verified</th>
</tr>
</thead>
<tbody>
<tr>
<td>Millen’s fgg</td>
<td>14</td>
<td>306</td>
<td>677</td>
<td>1335</td>
<td></td>
<td>attack</td>
</tr>
<tr>
<td>(corrected)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Needham-Schroeder</td>
<td>13</td>
<td>294</td>
<td>323</td>
<td>45</td>
<td></td>
<td>attack</td>
</tr>
<tr>
<td>(strong correctness)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Needham-Schroeder</td>
<td>13</td>
<td>304</td>
<td>755</td>
<td>516</td>
<td></td>
<td>yes</td>
</tr>
<tr>
<td>(weak correctness)</td>
<td>(\checkmark)</td>
<td>12</td>
<td>299</td>
<td>299</td>
<td>63</td>
<td>yes</td>
</tr>
<tr>
<td>Corrected Needham-Schroeder</td>
<td>14</td>
<td>1575</td>
<td>1575</td>
<td>791</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>(strong correctness)</td>
<td>(\checkmark)</td>
<td>9</td>
<td>402</td>
<td>402</td>
<td>31</td>
<td>yes</td>
</tr>
<tr>
<td>Otway-Rees</td>
<td>5</td>
<td>10339</td>
<td>10339</td>
<td>4272</td>
<td></td>
<td>attack</td>
</tr>
</tbody>
</table>

Table 11.1: Analysis of authentication protocols: experimental results

of attacks are very pervasive in authentication. Another classical example of type-flawed protocol is the Yahalom protocol [CJ97].

11.4.5 Experimental Results

In Table 11.1 we summarize the experimental results for the examples of Section 11.4. We use the following conventions: a symbol \(\checkmark\) in column 'Invar' indicates use of the invariant strengthening technique; the column ‘Steps’ denotes the number of iterations needed to reach a fixpoint (or before finding an attack); ‘Size’ is the number of multisets contained in the fixpoint (or at the time the attack was found), while ‘MSize’ is the maximum number
of multisets computed at any step; ‘Time’ is the execution time (in seconds).

11.5 Future Work

In this section we discuss some lines of research for future work. We refer to Appendix A for a discussion on the verification tool we have used to carry out the experiments discussed in this chapter, and on the future improvements we plan to implement. As an example, in order to perform protocol correctness analysis, we would like to support automatic or semi-automatic generation of invariants and pruning conditions. On the theoretical side, we have also singled out the following points.

We want to study a (possibly automatic) translation between the usual informal description of protocols and our representation. As suggested by the examples presented in this chapter, a one-to-one translation (one rule for every step) could be sufficient, provided we have a way to store the information about the internal state of principals.

Furthermore, we want to study a general formulation of the Dolev-Yao intruder theory, which could be optimized and selectively refined for analyzing the specific protocol at hand. This refinement and optimization phase can be useful in order to analyze a given protocol under different assumptions for the intruder capabilities, and it is also crucial for efficiency considerations, as exemplified by the experiments we have presented. Specifically, optimizing the rules for composition and decomposition of messages seems to be crucial to improve the verification algorithm performance. We plan to use techniques like folding/unfolding to automatize this process. A similar technique is used for instance in [Bla01].

We also consider the possibility of enriching the multiset rewriting formalism underlying our specification language with a more general term rewriting theory based on equational theories and AC unification, as done for instance in [Mea96, DMT98, JRV00].

We want to carry out further experiments on other security protocols presented in the literature. In particular, we plan to combine our verification tool with a constraint solver, along the lines discussed in Chapter 9, to analyze protocols which make use of timestamps to ensure freshness of messages exchanged between principals. Other classes of protocols which could be worth investigating include fair exchange protocols and looping protocols like Kerberos [BAN89].

Another topic we would like to investigate is typed multiset rewriting [Cer01b], which extends multiset rewriting with a typing theory based on dependent types with subtyping. Dependent types can be used to enforce dependency between an encryption key and its owner. The paper [Cer01b] also presents some extensions which increase the flexibility of multiset rewriting specifications, e.g. using memory predicates to remember information across role executions.
Finally, an open question is the problem of non-termination. In the few examples we have presented, our algorithm is always terminating, even without invariant strengthening. Although secrecy has been proved to be undecidable, even for finite-length protocols with data of bounded complexity [CDL+99], one may ask if a more restricted subclass of protocols exists, for which the verification algorithm presented here is always terminating.

### 11.6 Related Work

In this chapter we have presented security protocols as a possible application field for our methodology based on a multiset-rewriting-like specification language and on a backward search verification strategy. Our verification procedure is tailored to study security violations which can be specified by means of *minimality conditions*. While this may rule out interesting properties, e.g. questions of belief [BAN89], the proposed approach can be used to study secrecy and confidentiality properties. No artificial limit is imposed on the number of simultaneous sessions we are able to analyze.

We have performed some experiments on different authentication protocols which show that the methodology we propose can be effective either to find attacks or to validate existing protocols. We plan to overcome some current limitations of our approach, in particular we plan to refine and automatize the specification phase of protocols and of the intruder theory, as discussed in Section 11.5. We expect further results from future work on this point. Below we discuss some related work in more detail.

A wide research area in security protocol analysis is related to rewriting logic [Mes92]. For instance, we mention [Cir01], which specifies security protocols as rewriting theories which can be executed in the ELAN system [BKK+96]. A similar approach is followed in [DMT98], where the target executable language is instead Maude [CELM96]. Perhaps the more interesting work in this class is [JRV00]. This work presents an automatic compilation process from security protocol descriptions into rewrite rules. The resulting specifications are then executed using the daTac theorem prover [Vig95]. As a difference with [DMT98], which is based on matching, the execution strategy of [JRV00] relies on narrowing and AC unification. Our approach, based on multiset unification, is clearly closer to the latter approach, although currently we do not support equational theories. All of the above approaches are limited to protocol debugging, therefore they can find attacks mounted on a given protocol, but they cannot be used to analyze correctness. Also, a crucial difference is that all the above works are based on a forward breadth-first-search strategy, while effectiveness of our verification algorithm strongly relies on a backward search strategy. Another approach which shares some similarity with ours is [Del01], where a specification for security protocols based on rewriting and encoded in a subset of intuitionistic logic is presented. The author uses universal quantification to generate nonces, like us, and
embedded implication to store the knowledge of agents. This approach is still limited to
protocol debugging.

An alternative approach to verifying security protocols is based on model checking. For
instance, the FDR model checking tool was used by Lowe [Low96] to analyze the Needham-
Schroeder public-key protocol and the attack previously reported in [Low95]. Other works
which fall into this class are [MCJ97, RB99]. All these approaches have in common the
use of some kind of abstraction to transform the original problem into a finite-state model-
checking problem, which is then studied by performing a forward reachability analysis.
Using a finite-state approximation has the advantage of guaranteeing termination, however
it only allows one to analyze a fixed number of concurrent protocol runs, an approach which
is infeasible as this number increases. As a difference, we use a symbolic representation
for infinite sets of states and a backward reachability verification procedure, which avoid
putting limitations on the number of parallel sessions we are able to analyze.

Theorem proving techniques are used in [Pau98], where protocols are inductively defined
as sets of traces, and formally analyzed using the theorem prover Isabelle [Pau94]. Here,
analysis is a semi-automatic process which can take several days. The NRL protocol
analyzer [Mea96] provides a mixed approach. It is based on protocol specifications given
via Prolog rules, and enriched via a limited form of term rewriting and narrowing to manage
symbolic encryption equations. Similarly to us, verification is performed by means of a
symbolic model-checker which relies on a backward evaluation procedure which takes as
input a set of insecure states. The analyzer needs to be fed with some inductive lemmas
by the user, in the same way theorem provers need to be guided by the user during the
proof search process.

In [Bla01], the author proposes an optimized specification of security protocols based on
an “attacker view” of protocol security, specified by means of Prolog rules, as in [Mea96].
The approach is effective, and has been applied to prove correctness of a number of real
protocols. The verification algorithm performs a backward depth-first search, which seems
to be closely related to our evaluation strategy, and uses an intermediate code optimization
using a technique similar to unfolding, which we plan to study as future work. On the other
hand, we think that the multiset rewriting formalism which we use is more amenable to
an automatic translation from the usual protocol notation. As we stressed in Section 11.1,
ensuring faithfulness between the intended semantics of a protocol and its specification is
necessary to prove correctness. Also, with respect to [Bla01], we use a cleaner treatment
for nonces, and we don’t have to use approximations (which may introduce false attacks)
except for invariant strengthening, which can be controlled by the user.

Finally, we mention some works concerning the process of translation from the usual in-
formal notation for protocols, which we plan to study as part of our future work. Existing
approaches include Casper [Low98], a compiler from protocol specifications into the CSP
process algebra, oriented towards verification in FDR, and CAPSL [Mil97], a specification
language which can be compiled into an intermediate language and used to feed tools like Maude [DMT98] or the NRL analyzer [Mea96]. Finally, [JRV00] presents an automatic compilation process into rewriting rules which is able to manage infinite-state models.

Summary of the Chapter. In this chapter we have illustrated the use of our framework based on linear logic for the specification and analysis of authentication protocols. Protocols and security properties can be specified using a uniform logic corresponding to multiset rewriting enhanced with nonce management via universal quantification, and validated by means of a bottom-up evaluation algorithm which performs backward search starting from a protocol/intruder theory and a set of axioms representing security violations.

We have presented a set of examples taken from classic literature on security protocols, and we have shown that our verification methodology can be used both for finding design errors and for validating protocols with respect to a given intruder theory and a given security property. Unlike most traditional approaches based on model-checking, our verification strategy does not pose any limitation on the number of simultaneous sessions which can be analyzed, and allows any principal to take part into different protocol runs simultaneously, possibly with different roles.
Chapter 12

Conclusions

In this thesis we have presented a specification language, based on a fragment of Girard’s linear logic, which is suitable for the specification of concurrent systems, and in particular of parameterized and infinite-state systems. Furthermore, we have enriched the language with a novel computational model based on bottom-up evaluation of linear logic programs, which has strong connections with symbolic model checking procedures. We have shown different classes of parameterized and infinite-state systems which can be validated using this machinery. Summarizing, we believe that this thesis provides some original contribution at least on the following points:

- Regarding the specification language in itself, we think that a strong point in favour of our approach is the seamless integration of specialized constraint solvers into a substratum consisting of a multiset-rewriting-like logic. Further extensions like, e.g., equational rewriting theories or more complex type theories might be plugged into our system with a limited effort. Finally, universal quantification in goals provides a logical and clean way to specify name generation. We have shown the last point in the test-and-lock example of Section 10.5 and in Chapter 11, where the quantifier is used to generate nonces;

- we have defined a new bottom-up semantics for a fragment of first-order linear logic (with or without constraints). In particular, we have extended the so-called C-semantics of [FLMP93, BGLM94] to a fragment of first-order linear logic including universal quantification in goals. As a result, we have also obtained the extension of the C-semantics to Horn clauses with embedded universal quantification;

- as a result of our semantical investigation, we have obtained decidability results for some fragments of first-order linear logic (with constraints or without constraints), with applications for the verification of infinite-state systems (as in the case of the ticket protocol). We expect further results in this direction from our future work;
• we have presented a verification procedure which can be used either to detect design errors (e.g. attacks for authentication protocols) or to validate infinite-state systems. In particular, this procedure is suitable to study a wide class of safety properties. Its strength is due to the following reasons: a symbolic computation strategy based on first-order unification (in the case of the language $\text{LO}_\forall$) or on specialized constraint solvers (in the case of $\text{LO}(\mathcal{C})$); the possibility of expressing safety properties via upward-closed sets; a backward evaluation strategy (we remark that using a forward strategy would not yield the same results); a subsumption mechanism (and also static analysis techniques) to relieve the state explosion problem;

• finally, we have applied our verification procedure to verify safety properties for a wide class of parameterized and infinite-state systems ranging from mutual exclusion protocols, to broadcast and authentication protocols. We have also obtained some original results. For instance, we have verified, to our knowledge for the first time, a parameterized formulation, admitting an arbitrary number of clients and servers, of the so-called ticket protocol. It is important to note that our specification of the ticket protocol is faithful to its original formulation, in fact we model integer variables using integer variables (which can grow unboundedly). In other words, we do not abstract away their values, and we do not use reduction to a finite-state problem as in some alternative approaches.

We summarize in Table 12.1 all the experiments that we have carried out in this thesis, using the tool presented in Appendix A. The column 'Section' indicates the section in which the relevant experiment has been presented, a symbol √ in column 'Static Opt' indicates the use of static analysis techniques like invariant strengthening or pruning (the abstraction $\alpha$ of Definition 9.60 is always used in the case of the ticket algorithm); the column ‘Steps’ denotes the number of iterations needed to reach a fixpoint (or before finding a security violations); ‘Size’ is the number of multisets contained in the fixpoint (or at the time the violation was found); finally, ‘Time’ is the execution time in seconds (not applicable for the readers/writers example which has been verified by hand).

12.1 Future Work

We conclude by outlining the directions of future research we believe more promising.

We are considering enriching our specification language with a type theory, possibly based on dependent types, along the tradition of logical frameworks [Pfe01, CP02]. Dependent types provide more flexible specifications, for instance, in security protocol specifications, they can be used to enforce dependency between an encryption key and its owner [Cer01b]. More importantly, dependent types allow one to represent and manipulate computations.
<table>
<thead>
<tr>
<th>Example</th>
<th>Section</th>
<th>Static Opt</th>
<th>Steps</th>
<th>Size</th>
<th>Time</th>
<th>Verified</th>
</tr>
</thead>
<tbody>
<tr>
<td>Producer/Comsumer</td>
<td>7.3.1</td>
<td></td>
<td>13</td>
<td>16</td>
<td>56</td>
<td>yes</td>
</tr>
<tr>
<td>A Petri Net For Mutual Exclusion</td>
<td>7.3.2</td>
<td>√</td>
<td>7</td>
<td>14</td>
<td>&lt; 1</td>
<td>yes</td>
</tr>
<tr>
<td>A Petri Net For Mutual Exclusion (Dyn.Gen.)</td>
<td>7.3.2</td>
<td>√</td>
<td>7</td>
<td>20</td>
<td>&lt; 1</td>
<td>yes</td>
</tr>
<tr>
<td>Readers/Writeers</td>
<td>8.4</td>
<td>√</td>
<td>1</td>
<td>4</td>
<td>n.a.</td>
<td>yes</td>
</tr>
<tr>
<td>Ticket Protocol</td>
<td>9.4</td>
<td>√</td>
<td>17</td>
<td>201</td>
<td>126</td>
<td>yes</td>
</tr>
<tr>
<td>(Multi-client, Single-server)</td>
<td></td>
<td></td>
<td>9</td>
<td>23</td>
<td>&lt; 1</td>
<td>yes</td>
</tr>
<tr>
<td>Ticket Protocol, Dyn.Gen.</td>
<td>9.4</td>
<td>√</td>
<td>17</td>
<td>222</td>
<td>150</td>
<td>yes</td>
</tr>
<tr>
<td>(Multi-client, Single-server)</td>
<td></td>
<td></td>
<td>10</td>
<td>32</td>
<td>&lt; 1</td>
<td>yes</td>
</tr>
<tr>
<td>Ticket Protocol, Dyn.Gen.</td>
<td>9.4</td>
<td>√</td>
<td>19</td>
<td>141</td>
<td>15</td>
<td>yes</td>
</tr>
<tr>
<td>(Multi-client, Multi-server)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Test and Lock</td>
<td>10.5</td>
<td></td>
<td>7</td>
<td>12</td>
<td>&lt; 1</td>
<td>yes</td>
</tr>
<tr>
<td>Millen’s ffgg Protocol</td>
<td>11.4.1</td>
<td></td>
<td>14</td>
<td>306</td>
<td>1335</td>
<td>attack</td>
</tr>
<tr>
<td>Millen’s ffgg Protocol</td>
<td>11.4.1</td>
<td></td>
<td>14</td>
<td>27</td>
<td>2419</td>
<td>yes</td>
</tr>
<tr>
<td>(corrected)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Needham-Schroeder Protocol</td>
<td>11.4.2</td>
<td></td>
<td>13</td>
<td>294</td>
<td>45</td>
<td>attack</td>
</tr>
<tr>
<td>(strong correctness)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Needham-Schroeder Protocol</td>
<td>11.4.2</td>
<td>√</td>
<td>13</td>
<td>304</td>
<td>516</td>
<td>yes</td>
</tr>
<tr>
<td>(weak correctness)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Corrected Needham-Schroeder</td>
<td>11.4.3</td>
<td></td>
<td>14</td>
<td>1575</td>
<td>791</td>
<td>yes</td>
</tr>
<tr>
<td>(strong correctness)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Otway-Rees Protocol</td>
<td>11.4.4</td>
<td></td>
<td>5</td>
<td>10339</td>
<td>4272</td>
<td>attack</td>
</tr>
</tbody>
</table>

Table 12.1: A summary of the experiments carried out in the thesis

as objects. In the context of security protocols, this would amount to representing traces of attacks as terms.

We are currently making further experiments on other classes of protocols, including the field of security protocols. We want to carry out more experiments on different domains (other than integer numbers) and we want to study new abstraction techniques (see Section 9.3) and automatic methods to generate invariants for pruning the search space (see Section 6.1.2). Furthermore, we are investigating the issue of termination of the bottom-up evaluation algorithm for more extended linear logic fragments than the monadic fragments discussed in Section 9.3 and Section 10.4.

We also want to find out whether it is possible to study different classes of problems or different classes of properties (e.g. liveness properties) for infinite-state systems. A starting point could be [FS01], where it is shown that the construction of well-structured transition systems can be used to deal with the termination, inevitability and boundedness problems. Solution of these problems is based on tree-saturation methods, which represent computations by means of a finite tree-like structure.
We believe that improving the implementation of our verification tool (see Appendix A) could allow us to analyze more complex protocols than the ones considered so far. In particular, we need to optimize multiset unification (and the related operation of subsumption) and term representation. We are currently studying *ad-hoc* data structures for representing (constrained) multisets, when dealing with specific domains.

From a logic programming perspective, we can extend the results presented in this thesis either by considering more complex linear logic languages [And92, HM94, Mil96, DM01], or more complex observational semantics, like the S-semantics of [FLMP93, BGLM94]. We would also like to extend the decidability results presented in Section 9.3 and Section 10.4. Finally, we would also investigate the connection between our multiset rewriting logic and concurrent constraint programming languages [Sar93], where constraints are used, at the programming level, as a means to provide communication and synchronization primitives between processes running in parallel. In particular, we want to consider the *tccp* language described in [dBGM00], which deals with a timed extension of CCP languages, suitable to specify *reactive* systems.
Bibliography


227


231


Appendix A

Verification Tool

In order to run the experiments described in chapters 7 through 11, we have built a (prototypical) verification tool implementing the backward reachability algorithm for the covering problem discussed in Chapter 6. In this section we give a brief overview about the use and implementation of the tool.

A.1 General Description

The verification tool can be viewed as an implementation of the (semi)-algorithm for the covering problem presented in Figure 6.3. In particular, it supports computation of the bottom-up semantics for first-order LO programs with clauses built upon the logical connectives \( \exists, \rightarrow, \top, \) and the universal quantifier \( \forall \) in goals. The verification algorithm exploits the upward-closure property of LO axioms of the form \( H \rightarrow \top \) and is therefore tailored to solve the so-called covering problem (see Section 6.1.1). For this reason, the logical constant \( 1 \) (and consequently the multiplicative conjunction \( \otimes \)) are not supported. Furthermore, the additive conjunction \( \& \) is currently not supported (though it could be added with a limited effort). In order to run the experiments described in Chapter 9, a (core of a) specialized constraint solver has been implemented. In particular, the tool supports the DC-constraints of Chapter 9.

The algorithm evaluates the bottom-up semantics of an LO program, producing in output the backward reachability set computed at every iteration. The fixpoint is computed (modulo state-explosion) any time termination is guaranteed. In case an attack (i.e., a counterexample violating the property under investigation) is found, the corresponding execution trace can be recovered and examined.

In Section A.3 we give more details about how the algorithm works.
A.2 Implementation Notes

Following [EP91], the verification tool has been implemented in Standard ML (see [Pau96] or [Har01] for a tutorial). We chose SML because it provides high-level programming features together with a reasonable computational efficiency.

The verification tool has been built as a collection of independent modules implementing, respectively: language parsing, term management, a specialized constraint solver for a subset of linear constraints, (constrained) multisets, the bottom-up evaluation procedure, and a user frontend. A parser supporting the languages $\text{LO}_\forall$ and $\text{LO}(C)$ has been produced using ML-Lex [AMT94] and ML-Yacc [TA00], a lexical analyzer and an automatic parser generator available for the SML environment. Some parts of the code dealing with term management, e.g. unification, have been adapted from the source code for an interpreter for the language Lolli [HM94], originally written by J. Hodas.

A.3 How It Works

The verification tool takes as input a given system specification as a set of LO clauses and a set of LO axioms representing the set of final states, a specification of the set of initial states, and possibly some pruning invariants (see Section 6.1.2.2). The output consists of the sets of configurations computed at each iteration and the resulting fixpoint, if the computation is terminating. More details are given below.

**Initial States.** The tool allows one to give a partial specification of the set of the initial states. It is possible to require a certain atom (or multiset of atoms) to belong to every initial state and/or impose an atom (multiset of atoms) not to belong to any initial state. The verification algorithm signals it has possibly found an attack every time it finds a state which is compatible with the partial specification and the user must take care of checking if it is really an attack or not. Exact specification of the set of initial states is also possible. However, partial specification is more flexible because it may help one to find additional hypotheses under which a security violation might take place.

**Invariant Strengthening.** The tool supports the invariant strengthening technique (see Section 6.1.2.1, consisting in enriching the set of the states the bottom-up evaluation starts from (i.e., enriching the set of logical axioms). We have also plugged the abstraction function $\alpha$ of Section 9.3, which can be set to validate specifications written in LO enriched with a subclass of linear integer constraints.
Procedure Symbolic Fixpoint

input $P$: a set of LO clauses
Ax: a set of LO axioms
Initial: a set of initial states
Invar: pruning invariants

output $F_{sym}(P)$

begin
  $Step := 1$
  $Interp := S_{Ax}(\emptyset)$; $Last := Interp$

  repeat
    $Step := Step + 1$
    $Last := S_P(Last)$
    $Last := prune(Last, Invar)$
    $(Interp, Last) := cut_{subsumed}(Interp, Last)$
    print $Step$, $Interp$
    if $Last \cap Initial \neq \emptyset$ then print “attack found”
  until $Last = \emptyset$
  print “fixpoint”

end

Figure A.1: Pseudo-algorithm for bottom-up evaluation

Pruning. The tool supports the pruning methodology (see Section 6.1.2.2). Specifically, it is possible to specify pruning invariants in the form of upward-closed sets of states which are known to be unreachable. Upward-closed sets of states can be specified as LO axioms. The tool checks every computed configuration against the pruning invariants, and possibly cuts the search space accordingly. At the moment, pruning invariants must be provided by the user and are not tested for correctness (see also Section A.5).

Subsumption. At every iteration of the fixpoint computation, subsumption checks are performed in order to discard redundant information. Intuitively, an element (constraint in the terminology of [AJ01b]) is redundant if its denotation is contained in the denotation of some previously computed element. Subsumption checks work as follows. Every newly generated element is checked against the already computed ones. If it is subsumed, then it can be immediately discarded, otherwise it is inserted. Elements which become redundant as a result of this insertion are in turn discarded.

Evaluation Algorithm. The bottom-up evaluation algorithm, written in pseudo-code, in presented in Figure A.1. At any given iteration, the variable $Interp$ contains the current (global) interpretation computed so far, while the variable $Last$ contains the subset
of elements of \( \text{Interp} \) computed during the last step. The subroutine \( \text{prune} \) prunes a given interpretation with respect to the given set of pruning invariants. The subroutine \( \text{cut} \), \( \text{subsumed} \) takes \( \text{Interp} \) and \( \text{Last} \) as input and inserts elements of \( \text{Last} \) into \( \text{Interp} \) performing the subsumption checks described in the previous paragraph (subsumed elements are removed from \( \text{Last} \)).

### A.4 Experimental Environment

In Chapters 7 through 11 we have presented some experiments which we have carried out using our verification tool. All experiments have been executed on a Pentium III 450 Mhz, under Linux 2.2.13-0.9, and running Standard ML of New Jersey, Version 110.0.7.

### A.5 Future Work

We have singled out a number of improvements of the current prototype which we want to carry out. On the theoretical level, we plan to enrich our tool with support for (automatic or semi-automatic) generation of invariants (see Section 6.1.2.1) and cut conditions (see 6.1.2.2). To this aim, we plan to integrate a module for automatic generation of structural invariants for Petri nets [CFGR95]. We also plan to interface our tool with general constraint solvers for different domains.

From an implementation point of view, some improvements are needed in order to get a better performance. In particular, the critical operation is multiset unification (directly implemented in SML), which requires to consider (classical) unification between sub-multisets of terms of the original multisets. The most expensive phase of the verification algorithm is by far the subsumption check, which makes heavy use of multiset unification. Subsumption is indeed the basis of the verification algorithm’s power. Future work includes implementing some kind of indexing to order terms inside multisets, so to make unification faster, and optimizing the subsumption phase (e.g., using heuristics). We also need to optimize the term representation, which is currently very naive.