Verification and Validation of Logic Programs

Salvatore Ruggieri

March 1999

Addr: Corso Italia 40, 56125 Pisa, Italy
Tel: +39-050-887111 — Fax: +39-050-887226 — E-mail: ruggieri@di.unipi.it
http://www.di.unipi.it/~ruggieri
Thesis Supervisor:
Prof. Dino Pedreschi
Abstract

The original contribution of this thesis consists of the systematic development of a comprehensive theoretical framework capable of addressing several verification and validation properties on the basis of a few simple, unifying principles. Our starting point is the study of the specific problem of termination of logic programs and queries. We offer sound and complete declarative characterization of the class of programs and queries that universally terminate with respect to the leftmost selection rule (called acceptable); with respect to some selection rule (called fair-bounded); and that have finitely many refutations (called bounded). These characterizations are then systematically lifted to proof methods for weak total correctness. The proof methods are based on Hoare’s style triples \( \{ \text{Pre} \} P \{ \text{Post} \} \) which, for a logic program \( P \), specify the admissible input and expected output by means of pre- and postconditions. We thoroughly investigate the proof theory derived from the characterization of acceptable programs, which addresses properties of (weak) partial correctness, (weak) total correctness, call and success patterns characterization, modularity, correct and computed instances characterization for Prolog programs with negation and arithmetic built-in’s. We illustrate the broad applicability of the adopted verification principles on the case study of the Vanilla meta-interpreter. As a by-result, a general criterion for reasoning about meta-interpreters is introduced, and its applicability demonstrated. The characterization of terminating programs and queries represent also the starting point for the development of a formal framework for testing and debugging of logic programs, two issues in validation of programs. The framework is based on the notion of decidability of logic program semantics and observables. We present decision procedures for sub-classes of acceptable, fair-bounded and bounded programs and provide an implementation for them in the form of Prolog meta-programs. The decision procedures are then recognized to be automatic tools for testing and debugging logic programs. Finally, we extend the approach to the family of constraint logic programming languages.
in memory of my mother
# Contents

## Acknowledgments

### 1 Introduction

1.1 Motivations .................................................. 1
1.2 Thesis Contribution and Development .......................... 3
1.3 Plan of the Thesis ............................................ 6
1.4 Background and Notation ...................................... 7

## 2 Termination of Logic Programs

2.1 Universal Termination ........................................ 11
2.2 Universal Termination w.r.t. All Selection Rules ............... 13
  2.2.1 Recurrent Programs ..................................... 14
2.3 Left Termination .............................................. 16
  2.3.1 Acceptable Programs ..................................... 19
  2.3.2 Soundness ................................................ 22
  2.3.3 Completeness ............................................. 26
2.4 Ǝ-Universal Termination ...................................... 29
  2.4.1 Fair-bounded Programs .................................. 31
  2.4.2 Soundness ................................................ 36
  2.4.3 Completeness ............................................. 40
2.5 Bounded Nondeterminism and Termination ........................ 44
  2.5.1 Bounded Programs ......................................... 46
  2.5.2 Soundness ................................................ 48
  2.5.3 Completeness ............................................. 50
  2.5.4 From Bounded Nondeterminism to Universal Termination .... 52
2.6 Arithmetic Built-in's ......................................... 55
2.7 Classes of Terminating Programs ............................... 57
2.8 Inferring Termination ........................................ 59
2.9 Related Work .................................................. 64
2.10 Conclusion .................................................... 67
3 From Termination to Verification 69
  3.1 Logic Program Verification .................................. 69
     3.1.1 Which Semantics for Program Verification? .......... 69
     3.1.2 Specifications and Correctness .................. 71
     3.1.3 Proof Relations .................................. 73
  3.2 From Termination to Weak Total Correctness .......... 74
  3.3 Acceptable Programs ................................... 78
  3.4 Fair-bounded Programs ................................ 81
  3.5 Bounded Programs .................................... 86
  3.6 On Proof Methods and Proof Relations ................. 89
  3.7 Related Work ......................................... 90
  3.8 Conclusion ........................................... 95

4 Verification of Prolog programs 97
  4.1 Proof Theory .......................................... 97
     4.1.1 The Proof Method ................................ 98
     4.1.2 Modularity ..................................... 104
     4.1.3 Persistency, Call Patterns and Success Patterns .. 108
     4.1.4 Weak Partial Correctness ....................... 110
     4.1.5 Partial Correctness ................................ 111
     4.1.6 Correct and Computed Instances .................. 116
     4.1.7 Weak Total Correctness ......................... 118
     4.1.8 Total Correctness ................................ 121
     4.1.9 Weakest (Liberal) Preconditions ................. 123
  4.2 Applications ............................................ 128
     4.2.1 Arithmetic Built-in's ............................. 129
     4.2.2 Modular Verification ............................... 130
     4.2.3 The Occur-check Problem .......................... 132
     4.2.4 Meta-interpreters ................................. 134
     4.2.5 Semantics Decidability ............................ 134
  4.3 General Logic Programs .................................. 135
     4.3.1 LDNF-resolution .................................. 135
     4.3.2 LDNF-resolution ................................. 137
     4.3.3 Completeness of LDNF-resolution ................. 146
  4.4 Related Work ........................................... 148
  4.5 Conclusion ............................................. 153

5 Verification of Meta-interpreters 155
  5.1 Meta-Interpreters ....................................... 155
     5.1.1 Correctness ..................................... 156
     5.1.2 Vanilla .......................................... 157
  5.2 Vanilla .................................................. 160
     5.2.1 Specifying Pre and Post .......................... 160
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.2.2</td>
<td>Persistency, Call Patterns and Success Patterns</td>
<td>162</td>
</tr>
<tr>
<td>5.2.3</td>
<td>From (Weak) Partial Correctness to (Weak) Declarative Soundness</td>
<td>164</td>
</tr>
<tr>
<td>5.2.4</td>
<td>From Partial Correctness to Declarative Correctness</td>
<td>164</td>
</tr>
<tr>
<td>5.3</td>
<td>Proving declarative correctness</td>
<td>167</td>
</tr>
<tr>
<td>5.3.1</td>
<td>A General Method</td>
<td>167</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Amalgamation</td>
<td>169</td>
</tr>
<tr>
<td>5.3.3</td>
<td>Extensions</td>
<td>176</td>
</tr>
<tr>
<td>5.4</td>
<td>Vanilla Ctd</td>
<td>178</td>
</tr>
<tr>
<td>5.4.1</td>
<td>Correct and Computed Instances</td>
<td>178</td>
</tr>
<tr>
<td>5.4.2</td>
<td>(Weak) Total Correctness</td>
<td>180</td>
</tr>
<tr>
<td>5.5</td>
<td>Related Work</td>
<td>183</td>
</tr>
<tr>
<td>5.6</td>
<td>Conclusion</td>
<td>185</td>
</tr>
</tbody>
</table>

6 From Termination to Validation | 187 |
| 6.1     | Semantics Decidability and Testing | 188 |
| 6.1.1   | Relating Semantics Decidability to Testing | 189 |
| 6.1.2   | Acceptable, Fair-bounded and Bounded Programs | 191 |
| 6.1.3   | Decision Procedures | 198 |
| 6.1.4   | Prolog Implementation of the Decision Procedures | 201 |
| 6.1.5   | Extensions of the Approach | 209 |
| 6.1.6   | Experimental Results and a Compilation-Oriented Approach | 214 |
| 6.2     | Declarative Debugging of Missing Answers | 217 |
| 6.2.1   | Program Semantics and Missing Answers | 218 |
| 6.2.2   | Shapiro’s Debugger | 219 |
| 6.2.3   | Semantics Decidability | 221 |
| 6.2.4   | Declarative Debuggers | 223 |
| 6.2.5   | A Completeness Result | 232 |
| 6.3     | Related Work | 233 |
| 6.4     | Conclusion | 235 |

7 From LP to CLP | 237 |
| 7.1     | Constraint Logic Programs | 238 |
| 7.2     | From LP to Ideal CLP | 243 |
| 7.2.1   | Acceptable CLP programs | 243 |
| 7.2.2   | Bounded CLP programs | 247 |
| 7.3     | From Ideal CLP to CLP(\(\mathcal{R}\)) | 249 |
| 7.3.1   | The incomplete solver of CLP(\(\mathcal{R}\)) | 249 |
| 7.3.2   | Proving Termination of CLP(\(\mathcal{R}\)) programs | 251 |
| 7.4     | Related work | 257 |
| 7.5     | Conclusion | 259 |
Contents

8 Conclusions 261
  8.1 Summary ............................................. 261
  8.2 Future Research ...................................... 264

Bibliography 267

Index 278
Acknowledgments

I am very grateful to the people that helped me during all these years.

My advisor Dino Pedreschi is the person who most influenced my research style, with his friendly guidance and encouragement.

I am grateful to Franco Turini and Carlo Montangero for the stimulating review of my work in the annual advancement reports. Also, I would like to express my gratefulness to the thesis reviewers, Krzysztof Apt and Jan Maluszyński, for their pertinent comments and suggestions. I want also to thank people in my research group, and the colleagues that shared with me courses and projects. Among the others, special thanks to Simone Contiero, Giuseppe Manco, and Chiara Renso.

During these years, I enjoyed interesting discussions on verification and validation of logic programs with many researchers. In particular, I want to express my appreciation to Pierre Deransart, Maurizio Gabbrielli, Marianne Kalsbeek, Oliver Jack, Fred Mesnard, and Alberto Pettorossi. Also, thanks to Danny De Schreye, Fred, Maurizio and Pierre for their helpful comments on a draft of the thesis.

A lot of friends supported me during the years spent in Pisa. Sincere thanks to all of them, and particularly to Gianluca and Luciano for their encouragement.

I am conscious that I would have had no opportunity without the sacrifices made by my parents Antonio and Teresa, and by my family. Last but not least, thanks to Tiziana for her support and patience, and especially for her love.
Chapter 1

Introduction

Normally, there is some distinguished interpretation, called the intended interpretation which gives the principal meaning of the symbols. Naturally, the intended interpretation of a formula should be a model of the formula.

J. W. Lloyd [106]

1.1 Motivations

Logic programming is a simple, powerful and versatile formalism which originated from the discovery that a subset of first order logic can be given an elegant computational interpretation. On the one hand, the family of logic programming languages bases its theoretical foundations on the rigorous mathematical framework of formal logic. On the other hand, research in design and implementation of logic programming systems has led to expressive, practical, and efficient programming languages and systems, which have deserved the study of appropriate programming methodologies and techniques.

Due to its mathematical roots, the family of logic programming languages is then advocated as an ideal support to declarative programming. The milestone work of Kowalski [101] foresaw a separation of concerns between the logic and the control components of programs. Logic is demanded to programmers, who write (first order logic) specifications that can be directly used as logic programs. The generation of a suitable control is demanded to the underlying logic programming system.

This ideal situation, however, is usually contradicted by practical experience. On
the one hand, direct execution of specifications may be hopelessly inefficient, and, on
the other hand, logic programming systems often exhibit slightly different semantics,
and, in many cases, offer non-declarative extensions of the pure logic programming
paradigm. For these reasons, declarative programs (just as imperative programs)
may fail to terminate, may end in run-time errors, may deliver unexpected output,
may behave differently in distinct logic programming systems.

It is therefore important to assess the correctness of a logic program with respect to its
specification, or intended interpretation. The problem of logic program verification
has received particular attention in the recent years, as witnessed by the body of
research cited in the Related Work sections. Many proof methods and techniques
have been put forward to address the various verification issues, including:

(i) termination,
(ii) partial correctness,
(iii) characterization of call and success patterns,
(iv) absence of type and run-time errors,
(v) characterization of correct and computed instances,
(vi) safe omission of the occur-check,
(vii) modular program development,
(viii) correctness of meta-programs.

However, no comprehensive framework has been proposed, capable of addressing
the various verification issues within a single proof theory on the basis of a few
simple, unifying principles. A striking comparison naturally arises with imperative
programming, where Hoare's logic thoroughly encompasses verification of sequential
programs, and provides the basis for verifying concurrent and distributed programs
(see e.g., Apt and Olderog [14]).

Also, it is important to assess the correctness of a logic program with respect to the
user requirements. In fact, even in the case that a formal proof of correctness is
provided, a program must be validated against the informal intended meaning of the
final user, which may differ from the programmer specification. Among the various
validation issues, we concentrate on formal approaches to:

(ix) testing,
(x) and debugging.

While declarative debugging of logic programs is an active research area, the problem
of providing a formal framework for testing logic programs remains substantially
unexplored.
1.2 Thesis Contribution and Development

On the basis of the above discussion, the original contribution of this thesis consists of the systematic development of a comprehensive theoretical framework capable of addressing properties (\(\bar{\alpha}\)x) on the basis of a few simple, unifying principles.

Our starting point is the study of a specific, yet crucial, problem in program analysis, namely termination of logic programs and queries.

We offer a sound and complete declarative characterization of the class of programs and queries - called acceptable - that universally terminate with respect to the leftmost selection rule, by revisiting the approach of Apt and Pedreschi [16].

While most of the research on termination has been directed towards the study of (Prolog's) leftmost selection rule, we introduce the notion of \(\exists\)-universal termination, which is closely related to the existence of a terminating control for programs and queries. A logic program \(P\) and a query \(Q\) \(\exists\)-universally terminate iff there exists a selection rule \(s\) such that every SLD-derivation of \(P\) and \(Q\) via \(s\) is finite. We design a sound and complete declarative characterization of programs and queries - called fair-bounded - that \(\exists\)-universally terminate.

Finally, we introduce the notion of bounded nondeterminism of programs and queries, namely the property that they have finitely many refutations via any selection rule, and provide a declarative characterization of this class of programs and queries - which we call bounded. \(\exists\)-universal termination implies bounded nondeterminism. On the other hand, bounded programs and queries can be transformed into terminating programs and queries by pruning SLD-trees at an appropriate level.

The declarative characterizations found provide us with practical proof methods for termination analysis, easy to apply in paper & pencil proofs, having modular proof obligations, and being suitable for automation.

We proceed by introducing some properties that are of interest in program verification, such as (weak) partial correctness, (weak) total correctness, persistence of the proof relation and call patterns characterization. Then, we combine the termination proof methods above with a well-known correctness proof method, called the inductive method, in order to obtain a unifying framework able to reason on those properties. The interesting outcome of this approach is that the characterizations of terminating programs are systematically lifted to proof methods for weak total correctness. The resulting proof methods are based on Hoare's style triples \(\{Pre\} \ P \{Post\}\) which, for a logic program \(P\), specify the admissible input and expected output by means of preconditions \(Pre\) and postconditions \(Post\).

Although the logic programming version of a triple is defined on purely logical terms and the notions of correctness refer to the least Herbrand model semantics, the proof theories can be readily applied to reason about operational and run-time properties, thus abstracting away from the subtleties of the procedural interpretation of logic programming—unification and the logical variable, the search strategy, to mention a few. In this sense, the resulting verification methods are carefully designed
as compromises between generality and expressiveness from the one side, and ease of use from the other side.

Next, we investigate in deep the proof theory derived from acceptable programs and queries. The resulting proof relation $\vdash_i \{Pre\} P \{Post\}$ addresses properties (i-vii) for Prolog programs with negation and arithmetic built-in's.

In addition, we introduce the notions of weakest (liberal) preconditions and strongest postconditions. The strongest postcondition of a program $P$ and a precondition $Pre$ is shown to coincide with the intersection of the least Herbrand model of $P$ and the precondition $Pre$, namely with the relevant subset of the least Herbrand model. We provide a characterization of the weakest (liberal) preconditions of a program $P$ and a postcondition $Post$ as ordinal closures of a function $\psi_{P,Post}$ defined over the lattice of Herbrand interpretations.

Moreover, as a by-result of the properties of general programs which are in the proof relation, we state a rather general form of completeness of SLDNF-resolution.

For obvious reasons of presentation, the $\vdash_i$ proof method is introduced in an incremental way, by a stepwise definition of increasingly higher levels of verification, from weak partial correctness up to full-armed total correctness. This is a standard presentation style adopted in many textbooks on Hoare's logic for imperative programming.

We illustrate the broad applicability of the adopted verification principles on the case study of the Vanilla meta-interpreter. In the computational logic paradigm, meta-programming is a natural and powerful tool, and a number of meta-interpreters have been introduced and proved correct with respect to their intended behavior.

However, the task of proving correctness has been largely performed using ad-hoc techniques, depending case by case on the semantics, the particular meta-program and the range of properties one was interested in verifying. No uniform and general method has been proposed for tackling the problem by simple and powerful tools. In addition, the proofs of correctness are typically based on operational reasonings, without any chance to exploit the declarative reading of programs.

We find that, under certain natural assumptions, all interesting verification properties lift up from the object program to the Vanilla, including:

- partial correctness,
- termination,
- absence of run-time errors,
- call and success patterns characterization,
- correct and computed instances characterization.

Interestingly, it is possible to establish these results on the basis of purely declarative reasoning, using the proof method based on the $\vdash_i$ relation.
As a by-result of the case study of Vanilla, we introduce a general criterion for reasoning about generic meta-interpreters, and demonstrate its applicability. New results are obtained, including results related to amalgamation, reflection down/up, parameterization and extensions, termination, and absence of errors.

The characterizations of terminating programs and queries represent also the starting point for the development of a formal framework for testing and debugging logic programs, two issues in validation of programs. The formal framework is based on the notion of decidability of logic program semantics and observables – focusing in particular on the C-semantics and the S-semantics. While semantics decidability may seem a pure theoretical notion, we recognize the existence of a tight relation between semantics decidability and testing of logic programs, by showing that the two problems are equivalent.

We present decision procedures for sub-classes of acceptable, fair-bounded and bounded programs and provide an implementation for them in the form of Prolog meta-programs. The meta-programming approach also reveals to be successful in modeling extensions to programs with arithmetic, meta-programs, general programs and other declarative semantics, such as the finite failure set, the closed word assumption set, and the computed answers with depth.

The decision procedures are then recognized to be automatic tools for testing logic programs. Software testing is an important stage in program development. It covers more than one third of the development time, and requires a high degree of specialization of the developers. In our terminology the testing problem consists of checking whether or not the formal semantics of a program includes a given finite set of atoms. This set represents a collection of test cases provided by the requirement documents (validation testing), or the formal specification (verification testing), or a previous version of the program (regression testing).

Also, we discuss some preliminary experimentations on efficiency of the meta-programming implementation, and outline an efficient compilation-oriented approach that overcome the overhead due to meta-programming.

Some specializations of the decision procedures are employed as the basic components for a declarative debugging approach of missing answers, namely of those queries which are valid in the intended meaning of a program but that are not in its actual semantics. A missing answer originates from a “failure” in the construction of a proof tree for a valid query and is usually detected during testing. The reason of such a failure is the presence of uncovered atoms, i.e. of atoms \( A \) in the intended interpretation of the program, for which there is no immediate justification in the program in order to deduce \( A \).

Many debuggers in the literature find uncovered atoms starting from missing answers that have a \emph{finitely failed SLD-tree}. As we will point out, this assumption is restrictive in some cases, and it is due to a well-known limitation of the \emph{negation as failure} rule. We propose two declarative debuggers of missing answers for C- and S-semantics that are correct for any program, and complete and terminating for the
class of acceptable logic programs.

Summarizing, the approach followed in this thesis for the analysis of logic programs starts with the characterization of terminating programs and queries. From these characterizations, verification and validation proof theories are developed and investigated in a systematic way. Therefore, when trying to apply the same approach to one of the many extensions of pure logic programming, the basic step is to find out declarative characterizations of terminating programs and queries. At least in principle, verification and validation proof theories should systematically be derived from those characterizations by generalizing the approach followed in this thesis.

We investigate how to lift the classes of terminating programs and queries to one of the most successful extension of logic programming, namely to the family of constraint logic programming (CLP) languages, which merge logic programming with constraint solving. We show that for a large class of languages, called \textit{ideal}, acceptable and bounded programs and queries can be generalized in a natural and intuitive way.

Unfortunately implementations of CLP languages are not complete, in general, in the sense that the constraint solver is not complete with respect to the declarative semantics of the constraint domain. Since the characterizations of terminating programs and queries we have introduced are tightly linked to the declarative semantics, the natural extensions of those characterizations to CLP languages may not be sound termination proof methods, in general. We study a specific CLP system whose constraint solver is not complete, namely the CLP(\(R\)) system, whose domain is the set of real numbers. We propose some approaches to prove termination of CLP(\(R\)) programs. On the one side, acceptability is coupled with an extension to CLP(\(R\)) of the well-known notion of modes, thus producing a sound proof method. On the other side, a transformational approach is proposed, which leads to a sound and complete proof method.

\section{Plan of the Thesis}

In Chapter 2, the problem of termination is investigated, by introducing some declarative characterizations of terminating programs (including also arithmetic built-in's) and queries.

In Chapter 3, several notions of program correctness are introduced, and the desirable properties of proof relations are recalled. Then the characterizations of Chapter 2 are systematically lifted to verification proof methods.

One of those methods, the one deriving from programs terminating w.r.t. the (Prolog's) leftmost selection rule, is deeply studied in Chapter 4 with reference to Prolog programs, possibly including arithmetic built-in's and negative literals.

Chapter 5 illustrate the broad applicability of the method of Chapter 4 by means of
the case study of the vanilla meta-interpreter. As a by-result of the case study, a general criterion for reasoning about generic meta-interpreters is introduced.

In Chapter 6, semantics and observable decidability is shown for sub-classes of the terminating programs characterized in Chapter 2. Based on semantics decidability, a framework for testing and debugging is developed.

In Chapter 7, we extend the approach followed in the thesis to the family of constraint logic programming languages. In particular, we investigate how the characterizations of terminating programs and queries extend to the CLP Scheme.

Finally, in Chapter 8 we summarize the conclusions of our research, and discuss some lines for future research in verification and validation of (extensions of) logic programs.

1.4 Background and Notation

We briefly introduce standard syntax and terminology of logic programming as well as some basic results. Unless otherwise specified, we substantially adhere to Apt’s book [10] notation. Others self-contained introductions to logic programming can be found in Lloyd’s book [106], and in Apt’s contribution to the Handbook of Theoretical Computer Science [9].

Languages

A signature is a pair $\langle \Sigma_L, \Pi_L \rangle$ of sets of function and predicate symbols. Function symbols are denoted by $f, g, h, \ldots$. Predicate symbols are denoted by $p, q, \ldots$. To each function symbol a non-negative arity is assigned. In particular, 0-ary function symbols are called constants and denoted by $a, b, c, \ldots$.

A (first order) alphabet consists of a signature, of an infinite and fixed set of variables, the connectives $\neg$, $\land$, $\lor$, $\rightarrow$, $\leftarrow$, the quantifiers $\forall$ and $\exists$, and the punctuation symbols “(“ “)”, “,” “.”. Variables are denoted by $x, y, z, \ldots$.

The sets of terms, atoms, literals, clauses and well-formed formulas on an alphabet are defined in the standard way. Terms are denoted by $T, S, \ldots$. Atoms are denoted by $A, B, \ldots$. Terms, atoms etc. with no variables are called ground. Ground terms are denoted by lower case letters $t, s, u, \ldots$. An atom is called pure if it is of the form $p(x_1, \ldots, x_n)$ where $x_1, \ldots, x_n$ are distinct variables.

The (first order) language given by an alphabet consists of the set of all well-formed formulas on the alphabet. Therefore, once fixed the set of variables, first order languages are determined by their signature. In this sense, we confuse language and signature.

Let $L$ be a language. We denote by $U_L$ the set of ground terms on $L$, and call it the Herbrand Universe of $L$; by $B_L$ the set of ground atoms on $L$, and call it the Herbrand Base of $L$; and by $Atom_L$ the set of atoms on $L$. Given two languages $L$
with signature \((\Sigma_L, \Pi_L)\) and \(M\) with signature \((\Sigma_M, \Pi_M)\), we say that \(M\) extends \(L\) iff \(\Sigma_M \supseteq \Sigma_L\) and \(\Pi_M \supseteq \Pi_L\).

We refer the reader to the cited textbooks for an introduction to substitutions, instances, unifiers and most general unifiers (in short, mgu’s), and the presentation of unification algorithms. Substitutions are denoted by \(\theta, \gamma, \ldots\). For an atom \(A\), we denote by \([A]_L\) the set of ground atoms of \(L\) which are instances of \(A\).

Queries and Programs

A query is a finite sequence \(A_1, \ldots, A_n\) \((n \geq 0)\) of atoms. \(Query_L\) denotes the set of queries on \(L\). Queries are denoted by \(Q, R, \ldots\).

A logic program (or, simply, a program) is a finite set of definite Horn clauses. In logic programming notation, clauses are written in the form

\[
A \leftarrow B_1, \ldots, B_n.
\]

where \(A\) is an atom and \(B_1, \ldots, B_n\) is a query. \(A\) is called the clause head, while \(B_1, \ldots, B_n\) the clause body. If \(n = 0\) the clauses is called unit clause and is also written in the form \(A\). The symbols “,” and “\(\leftarrow\)” are used to denote conjunction and reverse implication, respectively. We denote programs by \(P, V, \ldots\).

The language generated by a program \(P\) is \(L_P = (\Sigma_P, \Pi_P)\), where \(\Sigma_P\) is the set of function symbols appearing in \(P\), and \(\Pi_P\) is the set of predicate symbols appearing in \(P\). \(B_P\) is an abbreviation for \(B_{\Sigma_P}\).

We denote by \(n_P\) the maximum number of atoms occurring in the body of a clause of program \(P\). We write \(A \leftarrow B_1, \ldots, B_n \in ground_L(P)\) iff \(A \leftarrow B_1, \ldots, B_n\) is a ground instance of a clause from \(P\).

Throughout this thesis, we consider a fixed language \(L\) in which programs and queries are written. All the results are parametric with respect to \(L\), provided that \(L\) is rich enough to contain the symbols of the programs and queries under consideration.

SLD-resolution

We refer to Apt’s book [10] for the definition of SLD-derivations, computed answer substitutions, SLD-refutations, SLD-trees, \ldots. Given a computed answer substitution \(\theta\) for a program \(P\) and a query \(Q\), we say that \(Q\theta\) is a computed instance of \(P\) and \(Q\).

Here, we represent a SLD-derivation for a program \(P\) and a query \(Q_0\) as a (possibly infinite) sequence \(Q_0, Q_1, \ldots, Q_n\) of queries, abstracting away – when not necessary – from others technical details such as the mgu’s and the input clauses employed. If the SLD-derivation is finite, we say that its length is \(n\). A SLD-derivation is ground if every query in it is ground. SLD-derivations are denoted by \(\xi, \mu, \ldots\).

\(FF_P^L\) is the finite failure set of program \(P\), i.e. the set of ground atoms that have a finitely failed SLD-tree.
HIS stands for the set of initial fragments of SLD-derivations in which the last query is non-empty. Elements in HIS are denoted by $\xi^c$, $\mu^c$, $\ldots$. A selection rule is a function which, when applied to an element in HIS yields an occurrence of an atom in its last query. Selection rules are denoted by $s$, $f$, $g$, $\ldots$.

LD-resolution is SLD-resolution with the leftmost selection rule, namely the rule that always selects the leftmost atom in the last query of an element in HIS. A selection rule $f$ is fair if for every SLD-derivation $\xi$ via $f$ either $\xi$ is finite or for every atom $A$ in $\xi$, (some further instantiated version of) $A$ is eventually selected. The round-robin selection rule $rr$ selects atoms in the last query of $\xi^c$ as follows. If $\xi^c$ consists only of the initial query $\xi$, then the leftmost atom is selected. Otherwise, if at the previous step in $\xi^c$, the atom $A$ in the query $Q$ was selected, and $A$ was not the rightmost atom in $Q$, then the (instantiated version of) the atom following $A$ in $Q$ is selected. Finally, if at the previous step of $\xi^c$, the rightmost atom $A$ was selected, then the leftmost atom of the last query in $\xi^c$ is selected. Obviously $rr$ is a fair selection rule.

Interpretations and Fixpoints

Herbrand interpretations, Herbrand models, operators on complete partial orders and their fixpoints are defined in the usual way [10]. We identify Herbrand interpretations with subsets of $B_L$. For two Herbrand interpretations $I$, $J$, we write $I \rightarrow J$ to denote the set $(B_L \setminus I) \cup J$.

Given a Herbrand interpretation $I$ and a query $Q$ we write $I \models Q$ if $I$ is a model of $Q$. In particular, if $A$ is a ground atom then $I \models A$ iff $A \in I$. $\text{ground}_L(Q)$ denotes the set of ground instances of a query $Q$. We denote by $I^c$ the complement of $I$, i.e. $B_L \setminus I$.

For an operator $f$ and an ordinal $\alpha$, $f \uparrow \alpha$ and $f \downarrow \alpha$ denote the upward and downward ordinal powers of $f$, respectively. $\omega$ denotes the smallest limit ordinal, apart from 0. $gfp(f)$ and $lfp(f)$ denote the greatest fixpoint and the least fixpoint of $f$.

Logical consequence is denoted by $\models$. A correct instance of a program $P$ and a query is any instance $Q$ of the query such that $P \models Q$.

By $M_L^P$ we denote the least Herbrand model of the program $P$ with $L$ as the underlying language. We recall that $M_L^P$ coincides with:

- the set of ground atoms that are logic consequences of $P$,
- the set of ground atoms that have a SLD-refutation,
- the least fixpoint of the immediate consequence operator which maps Herbrand interpretations into Herbrand interpretations as follows:

$$T_P(I) = \{ A \in B_L \mid \exists B_1, \ldots, B_n \in \text{ground}_L(P) \}
I \models B_1, \ldots, B_n \}.$$
• the upward ordinal closure of $T_P$, namely $T_P \uparrow \omega$.

Also, we recall that a Herbrand interpretation $I$ is a model of $P$ iff $T_P(I) \subseteq I$.

**Proof Trees**

A *proof tree* (introduced by Clark [42]) for a program $P$ and an atom $A$ is a labeled tree $T$ such that: the root of $T$ is $A$; if $B$ is a labeled node and $B_1, \ldots, B_n$ are its children nodes then $B \leftarrow B_1, \ldots, B_n$ is an instance of a clause from $P$; if $B$ is a labeled leaf then $B \leftarrow$ is an instance of a clause from $P$. A *ground proof tree* is a proof tree where all nodes are ground atoms.

Clark [42] showed that $M_P^c$ coincides with the set of ground atoms $A$ for which there exists a ground proof tree for $P$ and $A$.

**Multisets**

A *multiset*, sometimes called *bag*, is an unordered sequence. We denote a multiset consisting of elements $a_1, \ldots, a_n$ by $\text{bag}(a_1, \ldots, a_n)$. The set of multisets of elements belonging to $W$ is denoted by $\text{bag}(W)$. Let $>$ be a (non-reflexive) ordering over the set $W$. The *multiset ordering over* $(W, >)$ is an ordering on finite multisets of $W$, and is denoted by $\succ_m$. It is defined as the transitive closure of the relation $\succ$ in which $x \succ y$ if $y$ can be obtained from $x$ by replacing an element $a$ of $x$ by a finite (possibly zero) number of elements $b \in W$ such that $a > b$. We write $x \succeq_m y$ iff $x \succ_m y$ or $x = y$.

It is well-known (see e.g. Dershowitz [68]) that the multiset ordering over a well-founded partial ordering is again well-founded. In particular, the multiset ordering over the set of natural numbers with their usual ordering is well-founded.

**General Programs and Miscellaneous**

A *general program* (resp., query) is obtained by allowing negative literals in clause bodies (resp., in queries). A negative literal is denoted by $\neg A$, where $A$ is an atom. We refer the reader to Lloyd's book for notation, definitions and results on general programs and SLDNF-resolution. Here, we only recall that $\text{comp}(P)$ denotes Clark's completion of the general program $P$.

We use Prolog's representation of lists. In particular, for $n \geq 1$, $[t_1, \ldots, t_n | t]$ is an abbreviation for $[t_0 | [t_1, \ldots, t_n | t]]$; and $[t_1, \ldots, t_n]$ is an abbreviation for $[t_0 | [t_1, \ldots, t_n | []]]$. The set of ground lists is denoted by $\text{GList}$. The list concatenation operator $* : \text{GList} \times \text{GList} \rightarrow \text{GList}$ is defined as follows: $[t_1, \ldots, t_n] * [s_1, \ldots, s_m] = [t_1, \ldots, t_n, s_1, \ldots, s_m]$. In general, the set of lists whose elements belong to $\alpha$, is denoted by $\text{List}(\alpha)$.

$N$ is the set of natural numbers. For a set $S$ and a natural $n$, $S^n$ denotes the set of $n$-tuples of elements from $S$. For a finite and non-empty set $S$ of numbers, $\max S$ is the maximum number in $I$. When $S$ is empty, $\max S = 0$. 
Chapter 2
Termination of Logic Programs

The starting point of the thesis work is the study of a specific, albeit crucial, problem in logic program verification, namely the declarative characterization of universal termination of programs and queries. The intended objective is to find out characterizations that provide us with practical proof methods for termination analysis. In this Chapter, we recall the notions of universal termination and left termination, while introducing ∃-universal termination and bounded nondeterminism. We offer sound and complete declarative characterizations of programs and queries that are terminating in the sense of the above notions. The characterizations turn out to be termination proof methods that are useful in practice, that are easy to apply in paper & pencil proofs, that have proof obligations allowing for modular proofs, and that are suitable for automation.

2.1 Universal Termination

The milestone work of Kowalski [101] foresaw a separation of concerns between the logic and the control components of programs. Logic is demanded to programmers, who write specifications that can be directly used as programs. The generation of a complete control is demanded to the underlying logic programming system. By a complete control, it is usually meant a selection rule s such that every logical consequence of a program and a query has a refutation via s. By Strong Completeness of SLD-resolution (see Apt [10]), any selection rule is complete in this sense. However, a stronger form of completeness is usually intended, which takes into account termination as well.

Definition 2.1.1 By a complete control for a program P and a query Q, we mean any selection rule s such that every SLD-derivation of P and Q via s is finite. □
The ideal situation where logic and control are kept separated is usually contradicted by practical experience. For efficiency reasons, early systems such as Prolog, adopted a fixed control, namely a left-to-right selection rule and a depth-first search strategy. Unfortunately, Prolog’s control is not complete in the sense of Definition 2.1.1. This fact prevents writing programs declaratively, and, in practice, a left-to-right style is customary in any Prolog program.

Second generation logic languages adopt more flexible control primitives, which allow for addressing logic and control separately. Program clauses are intended to model the logic of programs, as usual. In addition, programs are augmented by declarations or annotations that (implicitly or explicitly) specify restrictions on the admissible selection rules. In this class of languages, we include NU-Prolog [153], Gödel [88] and the Mercury [147] systems, among the others.

In this context, the problem of characterizing classes of terminating programs and queries is essential

(i) to gain a precise understanding of the class of programs and queries that have a complete control, in the sense of Definition 2.1.1, or such that a given selection rule is for them a complete control;

(ii) to provide support for paper & pencil verification of termination properties;

(iii) to serve as theoretical frameworks on which the design of automatic tools for termination analysis, compiler optimizations and program transformations can be based.

In general terms, the problem of universal termination of a program \( P \) and a query \( Q \) w.r.t. a set of admissible selection rules consists of showing that every rule in the set is a complete control for \( P \) and \( Q \), in the sense of Definition 2.1.1.

**Definition 2.1.2** A logic program \( P \) and a query \( Q \) universally terminate w.r.t. a set of selection rules \( S \) if every SLD-derivation of \( P \) and \( Q \) via any selection rule from \( S \) is finite. □

It is worth noting that, since SLD-trees are finitely branching, by König’s Lemma, “every SLD-derivation for \( P \) and \( Q \) via a selection rule \( s \) is finite” is equivalent to state that the SLD-tree of \( P \) and \( Q \) via \( s \) is finite.

In addition to universal termination, the notion of existential termination is considered in the literature. \( P \) and \( Q \) existentially terminate w.r.t. a given selection rule and a search strategy if the visit of the SLD-tree of \( P \) and \( Q \) via the selection rule and according to the search strategy either finds out a refutation or a finitely failed SLD-tree. Unfortunately, this definition takes into account the search strategy, thus departing from the intended potential of logic programming as a declarative paradigm. In this thesis, we substantially concentrate on universal termination.
2.2 Universal Termination w.r.t. All Selection Rules

The early approaches to the characterization of terminating programs focused on universal termination w.r.t. all selection rules. Indeed, this is a strong property, holding only for basic programs and queries. In the following, we briefly recall the approach of Bezem [22, 23], which defined the class of recurrent programs and queries. First, we introduce the notion of level mapping.

**Definition 2.2.1** A *level mapping* is a function $\ | : B_L \rightarrow N$ of ground atoms to natural numbers.

For a ground atom $A$, $|A|$ is called the level of $A$. \hfill $\Box$

Level mappings play the role of termination functions. They have been investigated since the early studies on termination of logic programs, especially in the context of automatic inference. The following example introduces some well-known classes of level mappings.

**Example 2.2.2 (Norms and Level Mappings)** A *norm* $\ | : U_L \rightarrow N$ is a function from ground terms into natural numbers. $\ |$ is called linear if for every ground term $f(t_1, \ldots, t_n)$:

$$\ | f(t_1, \ldots, t_n) | = a_0 + a_1 |t_1| + \ldots + a_n |t_n|,$$

i.e. the level of $f(t_1, \ldots, t_n)$ is a linear combination of the levels of arguments $t_1, \ldots, t_n$, where for $i \in [1, n]$ $a_i$ is a constant natural number depending on $f$. In the case $a_1 = \ldots = a_n = 1$, we say that the norm is *semi-linear*.

As an example, the *size* norm is semi-linear:

$$\text{size}(f(t_1, \ldots, t_n)) = 1 + \text{size}(t_1) + \ldots + \text{size}(t_n)$$

if $n > 0$

$$\text{size}(a) = 0$$

if $a$ is a constant.

Intuitively, the size $\text{size}(t)$ of a ground term $t$ is the number of function symbols occurring in it, excluding constants. Another widely used semi-linear norm is the list-length function, defined as follows:

$$|f(\ldots)| = 0 \quad \text{if} \quad f \neq [\ldots.]$$

$$|[x; t]| = 1 + |t| \quad \text{otherwise.}$$

In particular, for a ground list $[t_1, \ldots, t_n]$ the list-length is $n$. An example of non-linear norm is the *term depth* norm:

$$\text{depth}(f(t_1, \ldots, t_n)) = 1 + \max \{\text{depth}(t_i) \mid i \in [1, n]\}$$

if $n > 0$

$$\text{depth}(a) = 0$$

if $a$ is a constant.
A common way of defining level mappings is to express the level of a ground atom as a (possibly linear) function of the level of its arguments calculated w.r.t. some given norms. For instance, one could define a linear level mapping such that:

\[ |\text{append}(xs, ys, zs)| = |xs| + |zs|, \]

or a nonlinear one such that \( |\text{append}(xs, ys, zs)| = \min\{ |xs|, |zs| \} \).

\[ \square \]

### 2.2.1 Recurrent Programs

We are now ready to introduce the class of recurrent programs. Intuitively, a program is recurrent if the level of the body-atoms of a ground instance of a clause is lower than the level of the head of the ground instance.

**Definition 2.2.3** Let \( \| \) be a level mapping. A logic program \( P \) is **recurrent by \( \| \)** iff for every \( A \leftarrow B_1, \ldots, B_n \) in \( \text{ground}_x(P) \):

\[ \text{for } i \in [1, n] \quad |A| > |B_i|. \]

\[ \square \]

**Example 2.2.4 (Satisfiability)** The following program \( \text{SAT} \) for propositional satisfiability:

\[
\begin{align*}
\text{satisfiable(Formula)} & \leftarrow \text{there is a true instance of Formula} \\
\text{satisfiable(true)} & \\
\text{satisfiable}(X \land Y) & \leftarrow \\
& \quad \text{satisfiable}(X), \text{satisfiable}(Y) \\
\text{satisfiable(not X)} & \leftarrow \text{invalid}(X) \\
\text{invalid(false)} & \\
\text{invalid(X \land Y)} & \leftarrow \text{invalid}(X) \\
\text{invalid(X \land Y)} & \leftarrow \text{invalid}(Y) \\
\text{invalid(not X)} & \leftarrow \text{satisfiable}(X)
\end{align*}
\]

is readily checked to be recurrent by \( \| \), where

\[ |\text{satisfiable}(t)| = |\text{invalid}(t)| = \text{size}(t). \]

As an example, consider a ground instance of the second clause:

\[
\text{satisfiable}(x \land y) \leftarrow \\
\text{satisfiable}(x), \text{satisfiable}(y)
\]
We calculate:

\[
|\text{satisfiable}(x \land y)| = \text{size}(x \land y) \\
= 1 + \text{size}(x) + \text{size}(y) \\
> \text{size}(x) \\
= |\text{satisfiable}(x)|,
\]

and similarly for the second body atom. Finally, notice that Definition 2.2.3 imposes no proof obligations for unit clauses. \(\square\)

The definition extends to queries in a systematic way. In fact, the proof obligations for a query \(Q\) are derived from those for the program \(\{p \leftarrow Q\}\), where \(p\) is a fresh predicate symbol. Intuitively, this operation is justified by the fact that the termination behavior of the query \(Q\) and a program \(P\) is the same as for the query \(p\) and the program \(P \cup \{p \leftarrow Q\}\).

**Definition 2.2.5** Let \(\triangledown\) be a level mapping. A query \(Q\) is recurrent by \(\triangledown\) iff there exists \(k \in \mathbb{N}\) such that for every \(A_1, \ldots, A_n \in \text{ground}_k(Q)\):

\[
\text{for } i \in [1,n] \quad k > |A_i|.
\]

Intuitively, the natural \(k\) in the definition above plays the role of the level of the atom \(p\).

**Example 2.2.6 (Satisfiability Ctd)** Consider again Example 2.2.4. The query \(\text{satisfiable}(X)\) is recurrent iff for every instance \(x\) of \(X\), we have that \(\text{size}(x)\) is bounded by a fixed natural \(k\). Obviously, this is possible iff \(X\) is a ground term. For instance, the query \(\text{satisfiable}(\text{not}(\text{true}) \land \text{false})\) is recurrent, whilst the query \(\text{satisfiable}(\text{false} \land X)\) is not. \(\square\)

It is worth noting that a program and a query are recurrent depending on the chosen level mapping. In this sense, level mappings specify the directionality of predicate arguments in recurrent atomic queries, namely which arguments must be (somewhat) necessarily specified.

**Example 2.2.7 (Append)** Consider the program \textsc{append}:

\[
\text{append}([\ ], Ys, Ys) \leftarrow \\
\text{Zs is the result of concatenating the lists Xs and Ys.}
\]

\[
\text{append}([X | Xs], Ys, [X | Zs]) \leftarrow \text{append}(Xs, Ys, Zs).
\]
It is easy to check that \textsc{append} is recurrent by the level mapping $|\text{append}(xs, ys, zs)| = |xs|$ and also by $|\text{append}(xs, ys, zs)| = |zs|$. In the two cases, we get different classes of queries which are recurrent by the same level mapping. For instance, $\text{append}([X, a, Y], [a], Z)$ is recurrent by the former level mapping, whilst it is not so by the latter. In contrast, $\text{append}(Z, [a], [X, a, Y])$ is recurrent by the latter, whilst it is not by the former. The level mapping:

$$|\text{append}(xs, ys, zs)| = \min(|xs|, |zs|)$$

combines the advantages of both of them. \textsc{append} is easily seen to be recurrent by it. Moreover if $X$ is a list or $Z$ is a list, $\text{append}(X, Y, Z)$ is recurrent by the level mapping above.

The termination properties of recurrent programs are summarized in the following Theorem.

**Theorem 2.2.8 (Bezem [22])** Let $P$ be a program and $Q$ a query.

If $P$ and $Q$ are both recurrent by a level mapping $\mid \mid$ then they universally terminate w.r.t. all selection rules.

Conversely, if $P$ and $Q$ universally terminate w.r.t. all selection rules, and $P$ and every ground query universally terminate w.r.t. all selection rules, then $P$ and $Q$ are both recurrent by a level mapping $\mid \mid$. \hfill $\square$

Notice that completeness is not stated in full general terms, i.e. recurcency is not a complete proof method for universal termination w.r.t. all selection rules. Informally speaking, incompleteness is due to the use of level mappings, which are functions that must specify a value for every ground atom. As a consequence, if $P$ is recurrent then every ground query is recurrent as well, hence terminating. We provide a general completeness result in the next section for a class of programs containing recurrent programs.

### 2.3 Left Termination

The stage following the study of universal termination w.r.t. all selection rules was the investigation of termination w.r.t. specific selection rules, and in particular w.r.t. Prolog’s leftmost selection rule. Recurrent programs and queries are too restrictive to deal with Prolog programs, as a larger class of programs and queries is terminating when considering a specific selection rule.

**Example 2.3.1 (Even - Apt and Pedreschi [7])** Examine the following program \textsc{even}:

\begin{verbatim}
EVEN/:
\end{verbatim}
### 2.3. Left Termination

\[ \text{even}(X) \leftarrow \]
\[ X \text{ is an even natural number.} \]
\[ \text{even}(s(s(X))) \leftarrow \text{even}(X). \]
\[ \text{even}(0). \]
\[ \text{lte}(X, Y) \leftarrow \]
\[ X, Y \text{ are natural numbers s.t. } X \text{ is smaller or equal than } Y. \]
\[ \text{lte}(s(X), s(Y)) \leftarrow \text{lte}(X, Y). \]
\[ \text{lte}(0, Y). \]

\textit{EVEN} is recurrent by defining:

\[
\begin{align*}
| \text{even}(x) | &= \text{size}(x) \\
| \text{lte}(x, y) | &= \min\{\text{size}(x), \text{size}(y)\}.
\end{align*}
\]

Now consider the query:

\[ Q = \text{lte}(X, s^{100}(0)), \text{even}(X) \]

which is supposed to compute the even numbers not exceeding 100. One can show that all I.D.-derivations of \textit{EVEN} and \( Q \) are finite, whereas there exists an infinite SLD-derivation when the rightmost selection rule is used. As a consequence of Theorem 2.2.8 the query \( Q \) is not recurrent, although it can be evaluated by a finite Prolog computation.

This example is a contrived instance of the \textit{generate-and-test} programming technique. This technique involves two procedures, one which generates the set of candidates, and another which tests whether these candidates are solutions to the problem. Actually, most Prolog programs that are implementations of the "generate-and-test" technique are not recurrent, as they heavily depend on the left-to-right order of evaluation, like the above query. In conclusion, this example shows that, even for recurrent programs, the class of recurrent queries is not sufficiently large to capture termination via a specific selection rule.

Left termination is an abbreviation for universal termination w.r.t. the singleton consisting of the leftmost selection rule.

**Definition 2.3.2** A logic program \( P \) and query \( Q \) universally left terminate if they universally terminate w.r.t. the set consisting of only the leftmost selection rule. \( \Box \)

**Example 2.3.3** (\textsc{Naive Reverse}) The following \textsc{Naive Reverse} program is often used as a benchmark for Prolog applications:
reverse(Xs, Ys) ← Ys is a reverse of the list Xs.

reverse([X | Xs], Ys) ←
    reverse(Xs, Zs),
    append(Zs, [X], Ys).
reverse([], []).

It is easy to check that the ground query reverse(xs, ys), for a list xs with at least two elements and an arbitrary list ys has an infinite SLD-derivation, obtained by using the selection rule which selects the leftmost atom at the first two steps, and the second leftmost atom afterwards. By Theorem 2.2.8 NAIVE REVERSE is not recurrent by any level mapping. However, all LD-derivations for the query above are finite, i.e. NAIVE REVERSE and the query left terminate.

Left-termination was addressed by Apt and Pedreschi in [15], which introduced the class of acceptable logic programs. Further studies [7, 16] showed that this class is large enough to include most of common Prolog programs. However, their characterization encountered a completeness problem similar to the one highlighted for Theorem 2.2.8.

Example 2.3.4 (Transp) An example of interesting programs which terminate on a strict subset of B_L only, is the following program TRANS:

trans(x, y, e) ← x ~_e y for a DAG e

trans(X, Y, E) ←
    member([X, Y], E).

trans(X, Y, E) ←
    member([X, Z], E),
    trans(Z, Y, E).

member(X, [X|Xs]).
member(X, [Y|Xs]) ←
    member(X, Xs).

In the intended meaning of the program, trans(x, y, e) succeeds iff x ~_e y, i.e. if [x, y] is in the transitive closure of a direct acyclic graph (DAG) e, which is represented as a list of pairs of nodes. It is readily checked that if e is not a DAG, i.e. it contains a cycle, then infinite derivations may occur. However, in the intended use of the program, this case is not a programming error, since the informal specification explicitly states that e is a DAG.

Notice that this is precisely the same reasoning followed in imperative programming. A Pascal procedure for the visit of a DAG is intended to terminate for the
specified data structure, while nothing is said about its behavior for unintended inputs.

In the approach of Apt and Pedreschi, \textsc{TRANS} cannot be reasoned about, since the same incompleteness problem of recurrent programs holds, namely they characterize a class of programs that (left) terminate for every ground query.

The cause of the restricted form of completeness of Theorem 2.2.8 lies in the use of level mappings, which must specify a natural number for every ground atom — hence termination is forced for every ground query. A more subtle consequence of using level mappings is that one must specify values also for uninteresting atoms, such as \texttt{trans}(x, y, e) when e is not a DAG.

Our solution is to consider \textit{extended level mappings} instead of level mappings.

\textbf{Definition 2.3.5} An extended level mapping is a function \(||: B_L \rightarrow N^\infty|\) of ground atoms to \(N^\infty\), where \(N^\infty = N \cup \{\infty\}\). For a ground atom \(A\), \(|A|\) is called the level of \(A\).

In contrast to the more standard Definition 2.2.1 of level mapping, we included \(\infty\) in the codomain of extended level mappings. The rationale is to use \(\infty\) as a means to model non-termination and uninteresting instances of program clauses. First, we extend the \(>\) order on naturals to a relation \(\triangleright\) on \(N^\infty\).

\textbf{Definition 2.3.6} We define the relation \(n \triangleright m\) for \(n, m \in N^\infty\) as follows:

\[
    n \triangleright m \quad \text{iff} \quad n = \infty \text{ or } n > m.
\]

We write \(n \trianglerighteq m\) iff \(n \triangleright m\) or \(n = m\).

Therefore, \(\infty \triangleright m\) for every \(m \in N^\infty\). With this additional notation we are now ready to introduce (a revised definition of) acceptable programs and queries.

\subsection{Acceptable Programs}

A program \(P\) is acceptable if for every ground instance of a clause from \(P\) the level of the head is greater than the level of each atom in the body such that the body atoms to its left are true in a model of the program.

\textbf{Definition 2.3.7} Let \(||\) be an extended level mapping, and \(I\) a Herbrand interpretation (i.e., a subset of \(B_L\)). A logic program \(P\) is acceptable by \(||\) and \(I\) iff \(I\) is a model of \(P\), and for every \(A \leftarrow B_1, \ldots, B_n\) in \(\text{ground}_L(P)\):

\[
    \text{for } i \in [1, n] \quad I \models B_1, \ldots, B_{i-1} \quad \text{implies} \quad |A| \triangleright |B_i|.
\]
An immediate consequence of this definition is that every program recurrent by \( \succ \) is acceptable by \( \succ \) and any model \( I \).

**Example 2.3.8 (Naive Reverse Ctd)** Consider the **Naive Reverse** program. An intuitive level mapping is the following:

\[
\begin{align*}
| reverse(xs, ys) | &= | xs | + 1 \\
| append(xs, ys, zs) | &= | xs |.
\end{align*}
\]

As observed in Example 2.3.3, the program is not recurrent by any level mapping. The problem with recurrency is in proving for every ground instance of the first clause:

\[
\text{reverse}([x \mid xs], ys) \leftarrow \\
\text{reverse}(xs, zs), \\
\text{append}(zs, [x], ys).
\]

that the level of the head is greater than the level of the second body atom. For the level mapping above, we should show:

\[
| reverse([x \mid xs], ys) | = | xs | + 2 > | zs | = | append(zs, [x], ys) |
\]

for every \( xs, zs \). This is obviously not true. When committing to a specific selection rule, the decreasing requirement can be weakened. In the case of acceptability, the decreasing is required only when the body atom \( reverse(xs, zs) \) is in some specified model of **Naive Reverse**. In practice, the model provides us with further information for establishing the decreasing. We define:

\[
I = \{ reverse(xs, ys) \mid | xs | = | ys | \} \cup \\
\{ append(xs, ys, zs) \mid | zs | = | xs | + | ys | \}.
\]

It is immediate to observe that \( I \) is a model of **Naive Reverse**. As an example, consider the ground clause above. If its body is true in \( I \) then \( | ys | = | zs | + | [x] | = | zs | + 1 \) and \( | xs | = | zs | \). This implies \( | ys | = | xs | + 1 \), hence the head of the ground clause is true in \( I \).

The acceptability proof obligations for ground instances of the first clause are then:

\[
\begin{align*}
(i) & \quad | reverse([x \mid xs], ys) | \triangleright reverse(xs, zs) \\
(ii) & \quad I \models reverse(xs, zs) \Rightarrow \\
& \quad | reverse([x \mid xs], ys) | \triangleright | append(zs, [x], ys) |.
\end{align*}
\]

For the given \( \succ \) and \( I \), we calculate:

\[
\begin{align*}
(i) & \quad | xs | + 2 \triangleright | xs | + 1 \\
(ii) & \quad | xs | = | zs | \Rightarrow | xs | + 2 \triangleright | zs |,
\end{align*}
\]

which are of immediate verification. \( \square \)
The definition of acceptable queries is systematically derived from that of acceptable programs, in the same way as we did for recurrent programs and queries (see Section 2.2.1).

**Definition 2.3.9** Let \(||\) be an extended level mapping, and \(I\) a Herbrand interpretation. A query \(Q\) is acceptable by \(||\) and \(I\) iff there exists \(k \in \mathbb{N}\) such that for every \(A_1, \ldots, A_n \in \text{ground}_I(Q)\):

\[
\text{for } i \in [1, n] \quad I \models A_1, \ldots, A_{i-1} \quad \implies \quad k \triangleright |A_i|.
\]

\(\square\)

**Example 2.3.10** (Even Ctd) Consider again the program \(\text{EVEN}\) and the query:

\[
\text{lte}(X, s^{100}(0)), \text{even}(X).
\]

We observed that the program is recurrent, while the query was not. On the contrary, \(\text{EVEN}\) and the query above are acceptable by the extended level mapping of Example 2.3.1 and the model:

\[
I = \{ \text{lte}(x, y) \mid \text{size}(x) \leq \text{size}(y) \} \cup [\text{even}(X)]_\mu.
\]

Consider a ground instance of the query:

\[
\text{lte}(x, s^{100}(0)), \text{even}(x).
\]

We claim that by fixing \(k = 101\), the proof obligations of Definition 2.3.9 are satisfied. In fact:

\[
k \triangleright 100 = |\text{lte}(x, s^{100}(0))|,
\]

and, if \(\text{size}(x) \leq \text{size}(s^{100}(0)) = 100\)

\[
k \triangleright 100 \triangleright \text{size}(x) = |\text{even}(x)|.
\]

\(\square\)

The introduction of \(\infty\) in the codomain of extended level mappings allows us to exclude uninteresting atoms from the termination analysis. The need for reasoning on a subset of queries is motivated by the fact that logic programs are untyped, and then queries may have atoms that in the intended interpretation of the programmer are ill-typed. Another reason for introducing \(\infty\) is the fact that a program may terminate for a strict subset of ground atoms only.

**Example 2.3.11** (Transp Ctd) Consider the \(\text{TRANSP}\) program again. We have pointed out that in the intended use of the program, \(e\) is supposed to be a DAG. Let us define:

\[
I = [\text{trans}(X, Y, E)]_\mu \cup \{ \text{member}(x, e) \mid x \text{ is in the list } e \}
\]

\[
|\text{trans}(x, y, e)| = \begin{cases} 
|e| + 1 + \text{Card}\{v \mid x \sim_x e, v\} & \text{if } e \text{ is a DAG} \\
\infty & \text{otherwise}
\end{cases}
\]

\[
|\text{member}(x, e)| = |e|.
\]
where \( \text{Card} \) is the set cardinality operator. It is not difficult to check that \( \text{TRANS} \) is acceptable by \( \mid \mid \) and \( I \). In particular, consider a ground instance of the second clause:

\[
\text{trans}(x, y, e) \leftarrow \\
\text{member}([x, z], e), \\
\text{trans}(z, y, e).
\]

It is immediate to see that \( I \) is a model of it. In addition, we have to show the proof obligations:

\[
(i) \quad |\text{trans}(x, y, e)| \triangleright |\text{member}([x, z], e)| \\
(ii) \quad [x, z] \text{ is in } e \Rightarrow |\text{trans}(x, y, e)| \triangleright |\text{trans}(z, y, e)|.
\]

The first one is easy to show since, \( |\text{trans}(x, y, e)| \triangleright |e| \). Focusing on the second one, we distinguish two cases. If \( e \) is not a DAG, the conclusion is immediate. Otherwise, \([x, z]\) in \( e \) implies that:

\[
\text{Card}\{ v \mid x \rightarrow_e v \} > \text{Card}\{ v \mid z \rightarrow_e v \},
\]

and then:

\[
|\text{trans}(x, y, e)| \\
= |e| + 1 + \text{Card}\{ v \mid x \rightarrow_e v \} \\
\triangleright |e| + 1 + \text{Card}\{ v \mid z \rightarrow_e v \} \\
|\text{trans}(z, y, e)|.
\]

We conclude by observing that for a DAG \( e \), the queries \( \text{trans}(x, y, e) \) and \( \text{trans}(x, y, e) \) are acceptable by \( \mid \mid \) and \( I \). The first one is intended to compute all nodes \( y \) such that \( x \rightarrow_e y \), while the second one computes the binary relation \( \rightarrow_e \).

\[
\square
\]

### 2.3.2 Soundness

In this section, we show that if a program and a query are acceptable then they left terminate. Therefore, acceptability provides us with a sound method for proving left termination. We start by showing that the notion of acceptability is persistent along SLD-derivations.

**Lemma 2.3.12 (Persistency)** Let \( P \) be a program and \( Q \) a query both acceptable by \( \mid \mid \) and \( I \). Every SLD-resolution \( Q' \) of \( P \) and \( Q \) is acceptable by \( \mid \mid \) and \( I \).

**Proof.** First of all, we observe that for any substitution \( \theta \), directly from Definition 2.3.1, \( Q\theta \) is acceptable by \( \mid \mid \) and \( I \) by using some fixed bound \( \leq N \). Let \( \theta \) be
now the mgu of the selected atom in $Q$ and the input clause head. Assume that $Q\theta = A_1, \ldots, A_n$, and that $c: A_k \leftarrow B_1, \ldots, B_m$ is the instantiation by $\theta$ of the input clause. Then $Q'$ is

$$A_1, \ldots, A_{k-1}, B_1, \ldots, B_m, A_{k+1}, \ldots, A_n.$$  

Let now $Q_{\text{m}} = A_1', \ldots, A_{k-1}', B_1', \ldots, B_m', A_{k+1}', \ldots, A_n'$ be a ground instance of $Q'$. Then there exists

$$A_1', \ldots, A_n'$$  

ground instance of $Q\theta$, with $A_k' \leftarrow B_1', \ldots, B_m'$ ground instance of $c$.

Let us show the proof obligations of Definition 2.3.9 for $Q_{\text{m}}$.

Let $i \in [1, k-1]$. If $I \models A_1', \ldots, A_{k-1}'$ then, since $Q\theta$ is acceptable, $\text{bound} \triangleright |A_i'|$.

Let $i \in [1, m]$. If $I \models A_1', \ldots, A_{k-1}', B_1', \ldots, B_m'$ then, since $P$ and $Q\theta$ are acceptable, $\text{bound} \triangleright |A_i'| \triangleright |B_i'|$.

Let $i \in [k+1, n]$. If $I \models A_1', \ldots, A_{k-1}', B_1', \ldots, B_m', A_{k+1}', \ldots, A_n'$ then, since $I$ is a model of $P$, $I \models A_k'$. Therefore,

$$I \models A_1', \ldots, A_{k-1}', A_k', A_{k+1}', \ldots, A_n'.$$

Since $Q\theta$ is acceptable, we conclude $\text{bound} \triangleright |A_i'|$.

Next we associate a finite multiset over $N$ to acceptable queries.

**Definition 2.3.13** Let $Q = A_1, \ldots, A_n$ be a query acceptable by $\ll | \rr$ and $I$. We define the sets $^\circ |Q|_i^j$ for $i \in [1, n]$ as follows:

$$^\circ |Q|_i^j = \{ |A_i'| \mid A_1', \ldots, A_n' \in \text{ground}_c(Q) \land I \models A_1', \ldots, A_{i-1}' \}.$$  

We define $^\circ |Q|^j$ as the finite multiset

$$^\circ |Q|^j = \text{bag}(\text{max}^x |Q|_1^j, \ldots, \text{max}^x |Q|_n^j).$$  

where $k$ is the maximum in $[1, n]$ such that $I \models \exists (A_1, \ldots, A_{k-1})$.  

We observe that the definition is well-formed. By Definition 2.3.9, the sets $^\circ |Q|_i^j$ for $i \in [1, n]$ are finite, and then there exists the maximum (which is 0 in case of empty sets). The following lemma shows a crucial relation between a query and its SLD-resolvents.

**Lemma 2.3.14** Let $P$ be a program and $Q$ a query both acceptable by $\ll | \rr$ and $I$. Consider any SLD-resolvent $Q'$ of $P$ and $Q$. Then

(i) $^\circ |Q'| \geq_{m}^\circ |Q|^j$,

(ii) if $Q'$ is a LD-resolvent, then $^\circ |Q'| \succ_{m}^\circ |Q|^j$.  

Proof. First of all, we observe that for every substitution \( \theta \),

\[
^a|Q|^i \succeq_m ^a|Q\theta|^i.
\]  

(2.1)

In fact, by Definition 2.3.13 \(^a|Q|^i \supseteq ^a|Q\theta|^i\) holds for \( i \in [1, k] \), where \( k \) is the maximum number such that the query obtained by considering the \( k \) leftmost atoms in \( Q\theta \) is satisfiable in \( I \).

Let \( \theta \) be now the mgu of the selected atom in \( Q \) and the input clause head. Assume that \( Q\theta = A_1, \ldots, A_n \), and that \( c : A_k \leftarrow B_1, \ldots, B_m \) is the instantiation by \( \theta \) of the input clause. Then \( Q' \) is

\[
A_1, \ldots, A_{k-1}, B_1, \ldots, B_m, A_{k+1}, \ldots, A_n.
\]

Let now \( Q_{in} = A'_1, \ldots, A'_{k-1}, B'_1, \ldots, B'_m, A'_{k+1}, \ldots, A'_n \) be a ground instance of \( Q' \). Then there exists

\[
A'_1, \ldots, A'_n
\]

ground instance of \( Q\theta \), with \( A'_k \leftarrow B'_1, \ldots, B'_m \) ground instance of \( c \).

Consider now any \( x \in \,^a|Q'|^i \).

Let \( i \in [1, k-1] \). Then \( I \models A'_1, \ldots, A'_{i-1} \) and \( x = |A'_i| \) for some \( Q_{in} \). This implies that \( x \) is in \( ^a|Q\theta|^i \).

Let \( i \in [1, m] \). Then \( I \models A'_1, \ldots, A'_{k-1}, B'_1, \ldots, B'_{i-1} \) and \( x = |B'_i| \) for some \( Q_{in} \). Since \( P \) is acceptable, there exist \( y = |A'_k| \) \( \in \,^a|Q\theta|^i \) such that \( y > x \).

Let \( i \in [k+1, n] \). Then

\[
|A'_1, \ldots, A'_{k-1}, B'_1, \ldots, B'_m, A'_{k+1}, \ldots, A'_{i-1}|
\]

and \( x = |A'_i| \) for some \( Q_{in} \). Since \( I \) is a model of \( P \), \( I \models A'_k \). Therefore, \( I \models A'_1, \ldots, A'_{k-1}, A'_k, A'_{k+1}, \ldots, A'_{i-1} \), which implies \( x \in \,^a|Q\theta|^i \).

Summarizing:

\[
\begin{align*}
&\text{for } i \in [1, k-1] & \max_{A'_i}^a|Q'|^i \leq \max_{A'_i}^a|Q\theta|^i, \\
&\text{for } i \in [1, m] & \forall \ x \in \,^a|Q'|^i \exists \ y \in \,^a|Q\theta|^i \ y > x, \\
&\text{for } i \in [k+1, n] & \max_{A'_i}^a|Q'|^i \leq \max_{A'_i}^a|Q\theta|^i.
\end{align*}
\]  

(2.2) (2.3) (2.4)

Let \( h \) be the maximum in \([1, n]\) such that \( I \models (A_1, \ldots, A_{h-1}) \), and \( h' \) be the maximum in \([1, n-1 + m]\) such that the query obtained by considering the \( h' \) leftmost atoms in \( Q' \) is satisfiable in \( I \). We now distinguish the two conclusions of the Theorem.

(i). We calculate:

\[
^a|Q|^i \succeq_m \{ (2.1) \} ^a|Q\theta|^i
\]
We are in the position to state soundness of the proposed proof method.

**Theorem 2.3.15 (Termination Soundness)** Let $P$ be a program and $Q$ a query both acceptable by $\| \|$ and $I$. Then $P$ and $Q$ left terminate.

**Proof.** Suppose that there exists an infinite LD-derivation $Q, Q_1, \ldots, Q_i, \ldots$ of $P$ and $Q$. By Lemma 2.3.12, every $Q_i$ is acceptable by $\| \|$ and $I$. This implies, by Lemma 2.3.14 (ii), that:

$$\stackrel{\sigma}{\triangleright}_m Q_1 \triangleright_m \ldots \triangleright_m \stackrel{\sigma}{\triangleright}_m Q_i \triangleright_m \ldots$$

is an infinite descending chain of bags over naturals. This is impossible since the finite multiset ordering over naturals is well-founded. \qed

**Example 2.3.16 (Naive Reverse Ctd)** `NAIVE REVERSE` has been shown to be acceptable by $\| \|$ and $I$, where:

- $\text{reverse}(xs, ys) = |xs| + 1$
- $\text{append}(xs, ys, zs) = |xs|$

$$I = \{ \text{reverse}(xs, ys) \mid |xs| = |ys| \} \cup \{ \text{append}(xs, ys, zs) \mid |zs| = |xs| + |ys| \}.$$
Given a (not necessarily ground) list $Xs$, the query $\text{reverse}(Xs, Ys)$ is acceptable by $[]$ and $I$. Consider, in fact, any ground instance $\text{reverse}(xs, ys)$ of its. We have that $|\text{reverse}(xs, ys)|$ is equal to $|xs|$, i.e., the list length of $Xs$. Therefore, by fixing $k$ to the list-length of $Xs$ plus one, the proof obligations of Definition 2.3.9 are satisfied. By Theorem 2.3.15, NAIVE REVERSE and the query above left terminate.

2.3.3 Completeness

In this section, we show that if $P$ and $Q$ left terminate then they are acceptable by some $[]$ and $I$. Therefore, acceptability is a sound and complete proof method for left termination. First, we introduce a further definition.

**Definition 2.3.17** Let $P$ be a program, $Q$ a query, and $s$ a selection rule.

We define $\text{length}^P_s(Q)$ as $\infty$ if there exists an infinite SLD-derivation of $P$ and $Q$ via $s$, and as the maximum length of such a SLD-derivation otherwise.

We observe that the definition of $\text{length}^P_s$ is well-formed. In fact, since SLD-trees are finitely branching, by König’s Lemma if there is no infinite SLD-derivation of $P$ and $Q$ via $s$ then there are finitely many SLD-derivations of $P$ and $Q$, hence the maximum length exists. Another useful property of $\text{length}^P_s$, when $s$ is the leftmost selection rule or the round-robin selection rule, is the following.

**Lemma 2.3.18** Let $P$ be a program, $Q$ a query, $A$ an atom and $s$ the leftmost or the round-robin selection rule. Then:

(i) for every $Q'$ SLD-resolvent of $P$ and $A$ via $s$

$$\text{length}^P_s(A) \triangleright \text{length}^P_s(Q'),$$

(ii) for every $Q'$ instance of $Q$, $\text{length}^P_s(Q) \triangleright \text{length}^P_s(Q').$

**Proof.** (i) We observe that, in the case of the leftmost and round-robin selection rules, the atoms selected in a derivation starting with $Q'$ are the same selected after the resolution of $A$. The conclusion follows then by definition of $\text{length}^P_s$. Let us show (ii). Consider a SLD-resolvent $Q'_1$ of $P$ and $Q'$ via $s$. Since $Q'$ is an instance of $Q$, the atoms selected by $s$ in $Q'$ and in $Q$ occur in the same position. Therefore, there exists a SLD-resolvent $Q_1$ of $P$ and $Q$ via $s$ such that $Q'_1$ is an instance of $Q_1$. In general, for every SLD-derivation $Q'_1, Q'_2, \ldots, Q'_n, \ldots$ via $s$ there exists:

$$Q, Q_1, \ldots, Q_n, \ldots$$

(prefix of a) SLD-derivation via $s$ such that $Q'_i$ is an instance of $Q_i$ for $i \geq 1$. Therefore, $\text{length}^P_s(Q) \triangleright \text{length}^P_s(Q').$

The following lemma is crucial in showing that acceptability is a complete termination proof method for left termination.
2.3. Left Termination

Lemma 2.3.19 Let $P$ be a program. Then there exist an extended level mapping $\parallel$ and a Herbrand interpretation $I$ such that:

(i) $P$ is acceptable by $\parallel$ and $I$, and

(ii) for every $A \in B_L$, $|A| \in N$ iff every LD-derivation of $P$ and $A$ is finite.

Proof. We define $I = M_P^I$ and $|A| = \text{length}_{LD}^P(A)$.

First, we observe that (ii) is immediate by definition of $|\parallel|$. Let us now consider (i). We show the proof obligations of Definition 2.3.7. Obviously, $M_P^I$ is a model of $P$. Consider now $A \leftarrow B_1, \ldots, B_n$ in $\text{ground}_L(P)$, and $i \in [1, n]$. Suppose that $I \models B_1, \ldots, B_{i-1}$. Since $I = M_P^I$, by Strong Completeness of SLD-resolution, there exists a LD-refutation $\xi$ of $P$ and the query $B_1, \ldots, B_{i-1}$. We claim that:

\[ \text{length}_{LD}^P(B_1, \ldots, B_n) \geq \text{length}_{LD}^P(B_i). \] (2.5)

In fact, consider a (possibly infinite) LD-derivation $\xi'$ of $P$ and $B_i$. Let $\xi'$ be the prefix of the LD-derivation of $P$ and $B_1, \ldots, B_n$ obtained by selecting clauses first according to $\xi$ and then according to $\xi'$. We have that the length of $\xi'$ is greater or equal than the length of $\xi$, and then (2.5) holds. Observing that $B_1, \ldots, B_n$ is an instance of a LD-resolvent $Q$ of $P$ and $A$, we now calculate:

\[ |A| = \text{length}_{LD}^P(A) \]
\[ \triangleright \{ \text{Lemma 2.3.18 (i)} \} \]
\[ \text{length}_{LD}^P(Q) \]
\[ \triangleright \{ \text{Lemma 2.3.18 (ii)} \} \]
\[ \text{length}_{LD}^P(B_1, \ldots, B_n) \]
\[ \triangleright \{ (2.5) \} \]
\[ \text{length}_{LD}^P(B_i) = |B_i|. \]

Finally, we state completeness of the proof method.

Theorem 2.3.20 (Termination Completeness) Assume that a program $P$ and a query $Q$ left terminate. Then there exist an extended level mapping $\parallel$ and a Herbrand interpretation $I$ such that $P$ and $Q$ are both acceptable by $\parallel$ and $I$. \(\Box\)

Proof. Consider the program $P' = P \cup \{ \text{new} \leftarrow Q \}$, defined on a language obtained by adding to $L$ the fresh predicate symbol new. By Lemma 2.3.19 (i), there exist $\parallel$ and $I$ such that $P'$ is acceptable by $\parallel$ and $I$. Since new is fresh, left termination of $P$ and $Q$ implies that every LD-derivation of $P'$ and new is finite. By Lemma 2.3.19 (ii), we conclude that $|\text{new}| \in N$. Since the definition of acceptability is modular, $P$ is acceptable by the restrictions of $\parallel$ and $I$ on $L$, i.e. not including new.
Turning the attention on $Q$, since $\mathbf{new} \leftarrow Q$ is acceptable by $\mathbf{||}$ and $I$, we have that for every ground instance $A_1, \ldots, A_n$ of $Q$, for $i \in [1, n]$

$$I \models A_1, \ldots, A_{i-1} \implies |\mathbf{new}| \triangleright |A_i|.$$ 

In conclusion $Q$ is acceptable by the restrictions of $\mathbf{||}$ and $I$ on $L$, by fixing $k = |\mathbf{new}|$ in Definition 2.3.9.

Summarizing, in the last two sections we showed that the class of acceptable logic programs and queries precisely characterizes the notion of left termination.

Example 2.3.21 (Permutation) The well-known program PERMUTATION checks whether two lists are permutations of each other.

\begin{verbatim}
perm(Xs, Ys) ← Ys is a permutation of the list Xs.
perm([], []). 
perm([X|Xs], Ys) ←
    delete(X, Ys, Zs),
    perm(Xs, Zs).
delete(X, [X|Y], Y). 
delete(X, [H|Y], [H|Z]) ←
    delete(X, Y, Z).
\end{verbatim}

Consider now the query $\text{perm}([a, b], Ys)$, whose intended meaning is to find out all permutations of the list $[a, b]$. We claim that the program and the query cannot be acceptable by the same extended level mapping $\mathbf{||}$ and Herbrand interpretation $I$. On the contrary, assume they are both acceptable by $\mathbf{||}$ and $I$. By Definition 2.3.9 there would exist $k \in \mathbb{N}$ such that $k \triangleright |\text{perm}([a, b], Ys)|$ for every ground term $Ys$. By Definition 2.3.7, for every ground instance

$$\text{perm}([a,b], Ys) \leftarrow \text{delete}(a, Ys, zs), \text{perm}([b], zs).$$

we have:

$$k \triangleright |\text{perm}([a,b], Ys)| \triangleright |\text{delete}(a, Ys, zs)|.$$ 

(2.6)

On the other hand, from the proof obligations of Definition 2.3.7 applied to the last clause of PERMUTATION, we have that for every $n \geq 1$:

$$|\text{delete}(a, [b_1, \ldots, b_n], [b_1, \ldots, b_n])|$$

\[ \triangleright |\text{delete}(a, [b_2, \ldots, b_n], [b_2, \ldots, b_n])|. $$

This implies that $|\text{delete}(a, [b_1, \ldots, b_n], [b_1, \ldots, b_n])| \triangleright n$, and for $n = k$ we obtain a contradiction of (2.6).

In conclusion, since the program and the query are not acceptable by the same $\mathbf{||}$ and $I$, by Theorem 2.3.20 we conclude that they do not left terminate. \hfill \square
2.4 Θ-Universal Termination

Example 2.3.21 shows that the leftmost selection rule, i.e., Prolog's control, is not complete in the sense of Definition 2.1.1. In general, a left-to-right style is customary in any Prolog program, thus losing in declarativeness. This and other problems with Prolog semantics motivated the design of more high level, expressive, efficient and practical logic programming systems, which adopt more flexible selection rules. In the next definition, we introduce Θ-universal termination of logic programs, claiming that it is an essential concept for the concerns of separating the development of the logic part of programs from the problem of associating a complete control to them.

Definition 2.4.1 A logic program $P$ and a query $Q$ Θ-universally terminate iff there exists a selection rule $s$ such that every SLD-derivation of $P$ and $Q$ via $s$ is finite.

If $P$ and $Q$ Θ-universally terminate then it is possible, at least in principle, to associate to them a complete control, in the sense of Definition 2.1.1. On the other hand, if $P$ and $Q$ does not Θ-universally terminate, then no logic programming system can be complete in the sense of Definition 2.1.1. Nevertheless, completeness can still be achieved by adopting transformational techniques, or pruning mechanisms, or, more in general, by modifying the operational interpretation of $P$ and $Q$ provided by SLD-resolution.

Example 2.4.2 (Permutation Ctd) Consider again the PERMUTATION program of Example 2.3.21 and the query perm([a, b], Ys). We have seen that they do not left terminate. However, we observe that they universally terminate w.r.t. the rightmost selection rule, hence they Θ-universally terminate.

As a general rule, termination via the rightmost selection rule of a program $P$ and a query $Q$ (or, more in general, via any fixed selection rule) can be proved by showing acceptability of $P'$ and $Q'$ obtained by reversing (or reordering according to the fixed rule) body atoms in $P$ and atoms in $Q$.

At first sight, the definition of Θ-universal termination looks rather different from Definition 2.1.2, the definition of universal termination. On the contrary, we show that Θ-universal termination coincide with universal termination with respect to the set of fair selection rules. Therefore, any fair selection rule is a complete control for any logic program and query for which a complete control exists, in the sense of Definition 2.1.1.

Theorem 2.4.3 A logic program $P$ and a query $Q$ Θ-universally terminate iff they universally terminate w.r.t. the set of fair selection rules.
Proof. The if part is immediate. Conversely, suppose that there exists an infinite SLD-derivation \( \xi \) via a fair selection rule \( f \). Let \( s \) be any other selection rule. Since \( f \) is fair, by reasoning as in the Switching Lemma (see \([10, \text{Lemma 3.32}]\)), we can switch the order of selection of atoms in \( \xi \) accordingly to \( s \), thus obtaining an infinite SLD-derivation of \( P \) and \( Q \) via \( s \). \( \square \)

Example 2.4.4 (ProdCons) The following program \textsc{ProdCons} abstracts a (concurrent) system composed of a producer and a consumer.

\[
\begin{align*}
(s) \quad \text{system}(N) & \leftarrow \\
& \quad \text{prod}(Bs), \text{cons}(Bs, N). \\
(p1) \quad \text{prod}([s(0) \mid Bs]) & \leftarrow \\
& \quad \text{prod}(Bs). \\
(p2) \quad \text{prod}([s(s(0)) \mid Bs]) & \leftarrow \\
& \quad \text{prod}(Bs). \\
& \quad \text{prod}([]). \\
(c) \quad \text{cons}([D \mid Bs], s(N)) & \leftarrow \\
& \quad \text{cons}(Bs, N), \text{wait}(D). \\
& \quad \text{cons}([], 0). \\
& \quad \text{wait}(0). \\
(w) \quad \text{wait}(s(D)) & \leftarrow \\
& \quad \text{wait}(D).
\end{align*}
\]

For notational convenience, we identify the term \( s^n(0) \) with the natural number \( n \). Intuitively, \text{prod} is the producer of a non-deterministic sequence of 1's and 2's, and \text{cons} the consumer of the sequence. The shared variable \( Bs \) in clause \((s)\) plays the role of an unbounded buffer. Moreover, since it is realistic to assume that consumption depends on \( n \), we model consumption by \text{wait}. The overall system is started by the query \text{system}(n), and stops when \text{cons} has consumed \( n \in N \) messages.

Notice that \textsc{ProdCons} and a query \text{system}(n) have infinite SLD-derivations via both the leftmost and the rightmost selection rules. Actually, viewed as a concurrent system, the program needs the assumption of fairness in order to terminate.

Logic programs, in fact, have a natural interpretation in terms of nondeterministic concurrent systems, where atoms model processes, shared variables model multi-party communication channels, clauses model process activations, and queries model dynamic networks of parallel processes. We refer the reader to Shapiro \([145]\) for a collection of papers on the concurrent interpretation of logic programs, and to Tick \([154]\) for a recent survey. \( \square \)

We propose a characterization of \( \exists \)-universal termination which is indeed a characterization of universal termination with respect to fair selection rules. Soundness and completeness of the characterization are proved by exploiting well-known properties
of fair selection rules. In particular, the following relation between finitely failed SLD-derivations and downward ordinals of the immediate consequence operator will be useful.

**Lemma 2.4.5** Let $P$ be a program, and $Q$ a query. Then $T_P \downarrow i \not\models \exists Q$ for some $i \in N$ iff every SLD-derivation of $P$ and $Q$ via any fair selection rule is failed.

**Proof.** Consider the program $P' = P \cup \{ p \leftarrow Q \}$ where $p$ is a fresh predicate symbol. By definition of $T_P$, $T_P \downarrow i \not\models \exists Q$ for some $i \geq 0$ iff $p \not\in T_P \downarrow i + 1$ for some $i \geq 0$ iff $p \not\in T_P \downarrow \omega$. We recall (see [9, Theorem 5.6]) that $p \not\in T_P \downarrow \omega$ iff every SLD-derivation of $P'$ and $p$ via any fair selection rule is failed. Since $p$ is a fresh symbol, we conclude that $T_P \downarrow i \not\models \exists Q$ for some $i \geq 0$ iff every SLD-derivation of $P$ and $Q$ via any fair selection rule is failed. □

### 2.4.1 Fair-bounded Programs

We offer a characterization of $\exists$-universal termination by means of the notions of fair-bounded programs and queries, that provide us with a sound and complete method of proving $\exists$-universal termination. The definition of fair-boundedness is purely declarative, in the sense that neither any procedural notion is needed in order to prove a program fair-bounded nor the definition reflects some fixed ordering of the atoms.

**Definition 2.4.6** Let $\| \|$ be an extended level mapping, and $I$ a Herbrand interpretation. A logic program $P$ is fair-bounded by $\| \|$ and $I$ iff $I$ is a model of $P$ such that for every $A \leftarrow B_1, \ldots, B_n$ in $\text{ground}_I(P)$:

\[ (a) \ I \models B_1, \ldots, B_n \text{ implies for every } i \in [1,n] \quad |A| \triangleright |B_i|, \text{ and} \]

\[ (b) \ I \not\models B_1, \ldots, B_n \text{ implies there exists } i \in [1,n] \quad I \not\models B_i \land |A| \triangleright |B_i|. \quad \square \]

We observe that some valuable properties follow directly from Definition 2.4.6. First, the proof obligations are modular, in the sense that program clauses are taken into consideration separately, one at a time. Second, the hypothesis of conditions $(a)$ and $(b)$ are mutually exclusive.

The next definition extends fair-boundedness to queries. The definition is derived from the definition of fair-bounded programs with the same methodology followed in the case of recurrency and acceptability.

**Definition 2.4.7** Let $\| \|$ be an extended level mapping, and $I$ a Herbrand interpretation. A query $Q$ is fair-bounded by $\| \|$ and $I$ iff there exists $k \in N$ such that for every $A_1, \ldots, A_n \in \text{ground}_I(Q)$:

\[ (a) \ I \models A_1, \ldots, A_n \text{ implies for every } i \in [1,n] \quad k \triangleright |A_i|, \text{ and} \]

\[ (b) \ I \not\models A_1, \ldots, A_n \text{ implies there exists } i \in [1,n] \quad I \not\models A_i \land k \triangleright |A_i|. \quad \square \]
(b) $I \not \models A_1, \ldots, A_n$ implies there exists $i \in [1, n] \quad I \not \models A_i \land k \uparrow \{A_i\}$. □

**Example 2.4.8 (ProdCons Ctd)** The **prodcons** program is fair-bounded. First, we introduce the list-max norm:

\[
\text{lmax}(f(x_1, \ldots, x_n)) = 0 \quad \text{if } f \neq \text{[]} \text{.}
\]

\[
\text{lmax}([x|x]) = \max \{\text{lmax}(xs), \text{size}(x)\} \quad \text{otherwise}.
\]

Note that for a ground list $xs$, $\text{lmax}(xs)$ equals the maximum size of an element in $xs$. Then we define:

\[
I = \quad [\text{system}(N)]_L
\]

\[
\cup \{ \text{prod}(bs) \mid bs \text{ list of 1's and 2's } \}
\]

\[
\cup \{ \text{cons}(bs, n) \mid |bs| = \text{size}(n) \}
\]

\[
\cup [\text{wait}(X)]_L
\]

\[
|\text{system}(n)| = \text{size}(n) + 3
\]

\[
|\text{prod}(bs)| = |bs|
\]

\[
|\text{cons}(bs, n)| = \begin{cases} 
\text{size}(n) + \text{lmax}(bs) & \text{if } \text{cons}(bs, n) \in I \\
\text{size}(n) & \text{if } \text{cons}(bs, n) \notin I
\end{cases}
\]

\[
|\text{wait}(t)| = \text{size}(t).
\]

Let us show the proof obligations of Definition 2.4.6. Those for unit clauses are trivial. Consider then the recursive clauses $(s), (p1), (p2), (c)$, and $(w)$.

(w) $I$ is obviously a model of $(w)$. In addition, $|\text{wait}(s(d))| = \text{size}(d) + 1 \uparrow \text{size}(d) = |\text{wait}(d)|$. This implies $(a, b)$.

(c) Consider a ground instance

\[
\text{cons}([d \mid bs], s(n)) \leftarrow \text{cons}(bs, n), \text{wait}(d).
\]

of $(c)$. If $I \models \text{cons}(bs, n), \text{wait}(d)$ then $|bs| = \text{size}(n)$, and then:

\[
|[d \mid bs]| = |bs| + 1 = \text{size}(n) + 1 = \text{size}(s(n)),
\]

i.e. $I \models \text{cons}([d \mid bs], s(n))$. Therefore, $I$ is a model of $(c)$. Let us show proof obligations $(a, b)$ of Definition 2.4.6.

(a) Suppose that $I \models \text{cons}(bs, n), \text{wait}(d)$. We have showed that $I \models \text{cons}([d \mid bs], s(n))$. We calculate:

\[
|\text{cons}([d \mid bs], s(n))|
\]

\[
= \text{size}(n) + 1 + \max \{\text{lmax}(bs), \text{size}(d)\}
\]

\[
\uparrow \text{size}(n) + \text{lmax}(bs)
\]

\[
= |\text{cons}(bs, n)|
\]
and

\[
|\text{cons}([d \mid bs], s(n))| \\
\geq \text{size}(n) + 1 + \max\{lmax(bs), \text{size}(d)\} \\
\geq \text{size}(d) \\
= |\text{wait}(d)|.
\]

These two inequalities show that (a) holds.

(b) If \( I \not\models \text{cons}(bs, n) \), \( \text{wait}(d) \) then necessarily \( I \not\models \text{cons}(bs, n) \). This and

\[
|\text{cons}([d \mid bs], s(n))| \\
\geq \text{size}(n) + 1 \\
\geq \text{size}(n) \\
= \{ I \not\models \text{cons}(bs, n) \} \\
|\text{cons}(bs, n)|
\]

show (b).

(p1) \( I \) is obviously a model of (p1). Moreover:

\[
|\text{prod}([s(0) \mid bs])| = |bs| + 1 \geq |bs| = |\text{prod}(bs)|
\]

implies (a) and (b).

(p2) This case is analogous to the previous one.

(s) Consider a ground instance

\[
\text{system}(n) \leftarrow \text{prod}(bs), \text{cons}(bs, n).
\]

of (s). Obviously \( I \) is a model of (s). Let us show (a,b).

(a) Suppose that \( I \models \text{prod}(bs), \text{cons}(bs, n) \). Then \( bs \) is a list of 1’s and 2’s. This implies \( lmax(bs) \leq 2 \). Moreover, we have that \( |bs| = \text{size}(n) \).

We calculate:

\[
|\text{system}(n)| = |s(n)| + 3 \\
\geq \{ |bs| = \text{size}(n) \} \\
|bs| \\
= |\text{prod}(bs)|
\]

and

\[
|\text{system}(n)| = \text{size}(n) + 3 \\
\geq \{ lmax(bs) \leq 2 \} \\
\text{size}(n) + lmax(bs) \\
= |\text{cons}(bs, n)|.
\]

These two inequalities show (a).
(b) Suppose that $I \not\models \text{prod}(bs)$, $\text{cons}(bs, n)$. We distinguish two cases. If $I \not\models \text{cons}(bs, n)$ then:

$$|\text{system}(n)| = \text{size}(n) + 3 \quad \Rightarrow \quad \text{size}(n) = |\text{cons}(bs, n)|.$$ 

If $I \models \text{cons}(bs, n)$ and $I \not\models \text{prod}(bs)$ then:

$$|\text{system}(n)| = \text{size}(n) + 3$$

$$\Rightarrow \quad \{ I \models \text{cons}(bs, n) \implies |bs| = \text{size}(n) \}$$

$$|bs| = |\text{prod}(bs)|.$$ 

We conclude this example by noting that for every $n \in N$ the query $\text{system}(n)$ is fair-bounded by $| |$ and $I$.

Let us discuss more in detail the meaning of proof obligations (a) and (b) in Definition 2.4.6. Consider a ground instance $A \leftarrow B_1, \ldots, B_n$ of a clause.

If the body $B_1, \ldots, B_n$ is true in the model $I$, then there might exist a SLD-refutation for it. (a) is then intended to bound the length of the refutation.

If the body is not true in the model $I$, then it cannot have a refutation. In this case, termination actually means that there is an atom in the body that has a finitely failed SLD-tree. (b) is then intended to bound the depth of the finitely failed SLD-tree.

These intuitions clarify why in Example 2.4.8 the level mapping for the $\text{cons}$ atoms distinguishes two cases. When $\text{cons}(bs, n)$ is in $I$, we bound the length of a possible SLD-refutation, while when it is not in $I$ we bound the depth of a finitely failed SLD-tree. However, since the proof method is purely declarative, in practice we do not reason about operational notions such as refutations and SLD-trees, but on models and (extended) level mappings.

**Example 2.4.9 (Permutation Ctd)** Consider again the **PERMUTATION** program of Example 2.3.21 and the query $\text{perm}([a, b], ys)$. Let us show they are fair-bounded by $| |$ and $I$, where:

$$|\text{perm}(xs, ys)| = |xs|$$

$$|\text{delete}(x, xs, ys)| = |ys|.$$

$$I = \{ \text{perm}(xs, ys) \mid |xs| = |ys| \}$$

$$\{ \text{delete}(x, xs, ys) \mid |xs| = |ys| + 1 \}.$$ 

It is readily checked that the query is fair-bounded by $| |$ and $I$, and that $I$ is a model of **PERMUTATION**. Also, the only non-trivial proof obligations are those regarding the second clause. Let
\[ \text{perm}([x|xs], ys) \leftarrow \text{delete}(x, ys, zs), \text{perm}(xs, zs). \]

be a ground instance of that clause. If the body is true in \( I \), then \( |xs| = |zs| \). This implies:

\[
|\text{perm}([x|xs], ys)| = |xs| + 1 \\
\triangleright |zs| + 1 \\
|\text{delete}(x, ys, zs)|,
\]

\[
|\text{perm}([x|xs], ys)| = |xs| + 1 \\
\triangleright |zs| \\
|\text{perm}(xs, zs)|.
\]

This shows proof obligation \((a)\). Consider now \((b)\), and suppose that the body is not true in \( I \). We distinguish two cases. On the one hand, if \( I \not\models \text{perm}(xs, zs) \), then

\[
|\text{perm}([x|xs], ys)| = |xs| + 1 \\
\triangleright |zs| \\
|\text{perm}(xs, zs)|
\]

implies proof obligation \((b)\). On the other hand, if \( I \not\models \text{delete}(x, ys, zs) \) and \( I \models \text{perm}(xs, zs) \), then \( |xs| = |zs| \) implies

\[
|\text{perm}([x|xs], ys)| = |xs| + 1 \\
= |zs| + 1 \\
\triangleright |zs| \\
|\text{delete}(x, ys, zs)|,
\]

which concludes the proof. \( \square \)

Summarizing, we claim that proving that a program is fair-bounded is simple and practical in paper & pencil proofs, as recurrency and acceptability are. In fact, proof obligations restrict to consider ground instances of clauses and queries. Substitutions and non-ground terms have not to be taken into account, since the method automatically lifts up to non-ground queries \( Q \) by considering every ground instance of \( Q \). In contrast, the model and the extended level mapping have to be chosen more carefully than in the case of acceptability, due to more binding proof obligations. Intuitively, the model \( I \) is a description of some property of the declarative interpretation of the program, namely the least Herbrand model. However, as a consequence of the intended meaning of proof obligation \((b)\) in Definition 2.4.6, the complement of \( I \) is
necessarily included in the finite failure set of the program. This intuition will be
formally stated later in Lemma 2.4.11.

As in the case of acceptable programs, the inclusion of \( \infty \) in the codomain of extended
level mapping allows for excluding \textit{unintended atoms} and \textit{non-terminating atoms}
from the termination analysis. In fact, if \( |A| = \infty \) then \( (a, b) \) in Definition 2.4.6 are
trivially satisfied.

### 2.4.2 Soundness

In this section, we show that if a program and a query are fair-bounded then they \( \exists \)-
universally terminate. Therefore, fair-boundedness provides us with a sound method
for proving \( \exists \)-universal termination. First of all, we show that the notion of fair-
boundedness is persistent along SLD-derivations.

**Lemma 2.4.10 (Persistency)** Let \( P \) be a program and \( Q \) a query both fair-bounded by \( | | \) and \( I \). Every SLD-resolvent \( Q' \) of \( P \) and \( Q \) is fair-bounded by \( | | \) and \( I \).

**Proof.** First of all, we observe that for any substitution \( \theta \), directly from
Definition 2.4.7, \( Q\theta \) is fair-bounded by \( | | \) and \( I \) by using some fixed \( bound \in N \). Let \( \theta \) be
now the mgu of the selected atom in \( Q \) and the input clause head. Assume that
\( Q\theta = A_1, \ldots, A_n \), and that \( c : A_k \leftarrow B_1, \ldots, B_m \) is the instantiation by \( \theta \) of the
input clause. Then \( Q' \) is

\[ A_1, \ldots, A_{k-1}, B_1, \ldots, B_m, A_{k+1}, \ldots, A_n. \]

Let now \( Q_{in} = A'_1, \ldots, A'_{k-1}, B'_1, \ldots, B'_m, A'_{k+1}, \ldots, A'_n \) be a ground instance of \( Q' \).
Then there exists

\[ A'_1, \ldots, A'_n \]

ground instance of \( Q\theta \), with \( A'_k \leftarrow B'_1, \ldots, B'_m \) ground instance of \( c \).

Let us show the proof obligations of Definition 2.4.7.

(a) Suppose that \( I \models Q_{in} \). We distinguish two cases.

\( (a1) \) If \( I \models A'_1, \ldots, A'_n \), then \( bound \triangleright A'_k \) by Definition 2.4.7 (a). Moreover,
\( I \not\models B'_1, \ldots, B'_m \). By Definition 2.4.6 (b) there exists \( i \in [1, m] \) such that \( A'_i \triangleright B'_i \)
and \( I \not\models B'_i \). We conclude that \( I \not\models B'_i \) and \( bound \triangleright A'_k \triangleright B'_i \].

(b) Suppose that \( I \not\models Q_{in} \). We distinguish two cases.

\( (b1) \) If \( I \models A'_1, \ldots, A'_n \), then \( bound \triangleright A'_k \) by Definition 2.4.7 (a). Moreover,
\( I \not\models B'_1, \ldots, B'_m \). By Definition 2.4.6 (b) there exists \( i \in [1, m] \) such that \( A'_i \triangleright B'_i \)
and \( I \not\models B'_i \). We conclude that \( I \not\models B'_i \) and \( bound \triangleright A'_k \triangleright B'_i \] .

(b2) If \( I \not\models A'_1, \ldots, A'_n \), then by Definition 2.4.7 (b), there exists \( i \in [1, n] \) such that
\( I \not\models A'_i \) and \( bound \triangleright A'_k \). We distinguish two cases.

If \( i \neq k \) then we have the conclusion, since \( A'_i \) is in \( Q_{in} \).

Suppose, on the contrary, that \( I \not\models A'_k \land bound \triangleright A'_k \). Since \( I \) is a model of \( P \),
\( I \not\models A'_k \) implies \( I \not\models B'_1, \ldots, B'_m \). By Definition 2.4.6 (b), there exists \( i \in [1, m] \)
such that $|A'_i| \triangleright |B'_i|$ and $I \not\models B'_i$. We conclude that $I \not\models B'_i$ and $bound \triangleright |A'_i| \triangleright |B'_i|$. □

The following property of fair-bounded programs can be intuitively read as follows. The ground atoms in the complement of the Herbrand interpretation $I$ used to prove $P$ fair-bounded either have level equal to $\infty$ or belong to the finite failure set of $P$.

**Lemma 2.4.11** Let $P$ be a program fair-bounded by $\models$ and $I$. Then for every ground atom $A \notin I$, if $|A| \neq \infty$ then $A \notin T_P \downarrow |A| + 1$.

**Proof.** Suppose that $A \notin I$ and $|A| \neq \infty$. The proof proceeds by induction on $|A|$.

1. $|A| = 0$ We claim that no $A \leftarrow B_1, \ldots, B_n$ is in $\text{ground}_k(P)$. Otherwise, since $I$ is a model of $P$ and $A \notin I$ then $I \not\models B_1, \ldots, B_n$. By Definition 2.4.6 (b) there exists $i \in [1, n]$ such that $0 \triangleright |B_i|$, which is impossible. By definition of $T_P$, we conclude that $A \notin T_P \downarrow 1$.

2. $|A| > 0$ We distinguish two cases.

Suppose that there is no $A \leftarrow B_1, \ldots, B_n$ in $\text{ground}_k(P)$. By definition of $T_P$, we have that $A \notin T_P \downarrow 1$. By monotonicity of $T_P$, we conclude $A \notin T_P \downarrow |A| + 1$.

Suppose, on the contrary, that there exists $A \leftarrow B_1, \ldots, B_n$ in $\text{ground}_k(P)$. Since $I$ is a model of $P$ and $A \notin I$ then $I \not\models B_1, \ldots, B_n$. By Definition 2.4.6 (b) there exists $i \in [1, n]$ such that $I \not\models B_i$ and $|A| \triangleright |B_i|$. By induction hypothesis, we have that $B_i \notin T_P \downarrow |B_i| + 1$. By monotonicity of $T_P$, we observe that $B_i \notin T_P \downarrow |A|$. By definition of $T_P$, we conclude that $A \notin T_P \downarrow |A| + 1$. □

To show termination soundness, we follow the same approach as for acceptability. First, we associate a finite multiset over $N$ to fair-bounded queries.

**Definition 2.4.12** Let $Q = A_1, \ldots, A_n$ be a query fair-bounded by $\models$ and $I$. We define the sets $\mathbb{b}|Q|_i^j$ for $i \in [1, n]$ as follows:

$$\mathbb{b}|Q|_i^j = \{ |A'_i| \mid A'_1, \ldots, A'_n \in \text{ground}_k(Q) \land I \models A'_1, \ldots, A'_n \}.$$ 

We define $\mathbb{b}|Q|^j$ as the finite multiset

$$\mathbb{b}|Q|^j = \text{bag}(\max |Q|_1^j, \ldots, \max |Q|_n^j),$$

if $I \models \exists (A_1, \ldots, A_n)$, and $\mathbb{b}|Q|^j = \text{bag}()$ if $I \not\models \exists (A_1, \ldots, A_n)$. □

We observe that the definition is well-formed. By Definition 2.4.7, the sets $\mathbb{b}|Q|_j^i$ for $i \in [1, n]$ are finite, and then there exists the maximum (which is 0 in case of empty sets). The following lemma shows a crucial relation between a query and its SLD-resolvents.
Lemma 2.4.13 Let $P$ be a program and $Q$ a query both fair-bounded by $|I|$ and $I$. For every SLD-resolvent $Q'$ of $P$ and $Q$, we have that:

(i) $^b|Q'|^b \succeq _m^b|Q|^b$, and

(ii) if $I \models \exists Q'$ then $^b|Q'|^b \succ _m^b|Q|^b$.

Proof. First of all, we observe that for every substitution $\theta$,

$$^b|Q|^b \succeq _m^b|Q\theta|^b. \tag{2.7}$$

In fact, by Definition 2.4.12 $^b|Q|^b \succeq _m^b|Q\theta|^b$ holds for $i \in [1,n]$, where $n$ is the number of atoms in $Q$.

Let $\theta$ be now the mgu of the selected atom in $Q$ and the input clause head. Assume that $Q\theta = A_1, \ldots, A_n$, and that $c : A_k \leftarrow B_1, \ldots, B_m$ is the instantiation by $\theta$ of the input clause. Then $Q'$ is

$$A_1, \ldots, A_{k-1}, B_1, \ldots, B_m, A_k, A_{k+1}, \ldots, A_n.$$

First, suppose that $I \not\models \exists Q'$. Then we have to show only (i), which is immediate by observing that $^b|Q|^b \succeq _m^b|Q\theta|^b$. On the other hand, assume now that $I \models \exists Q'$. Then for every $i \in [1,n+m-1]$:

$$^b|Q|^b \not= \emptyset. \tag{2.8}$$

Let now $Q_{in} = A'_1, \ldots, A'_{k-1}, B'_1, \ldots, B'_m, A'_{k+1}, \ldots, A'_n$ be a ground instance of $Q'$ such that $I \models Q_{in}$. Then there exists

$$A'_1, \ldots, A'_n$$

ground instance of $Q\theta$, with $A'_k \leftarrow B'_1, \ldots, B'_m$ ground instance of $c$. As a consequence for every $i \in [1,n], i \not= k$, we have $|A'_i| \in _m^b|Q\theta|^b$. This implies:

for $i \in [1,k-1]$ \quad $\max^b|Q'|^b_i \leq \max^b|Q\theta|^b_i$, \quad \tag{2.9}

for $i \in [k+1,n]$ \quad $\max^b|Q'|^b_{i+m-1} \leq \max^b|Q\theta|^b_i$. \quad \tag{2.10}

Moreover, since $I \models B'_1, \ldots, B'_m$, by Definition 2.4.6 (a), we have that $|A'_i| \triangleright |B'_i|$ for every $i \in [1,m]$. By the assumption that $Q$ is fair-bounded, we have that $|A'_i| \in N$ and then $|A'_i| > |B'_i|$ for every $i \in [1,m]$. Summarizing:

for $i \in [1,m]$ \quad $\forall x \in _m^b|Q'|^b_{i+k-1} \exists y \in _m^b|Q\theta|^b_i \, y > x. \tag{2.11}$

In conclusion, we calculate:

$$^b|Q|^b \succeq _m \{ (2.7) \}$$
which implies (i-ii).

The next theorem shows that no SLD-derivation of fair-bounded programs and queries is infinite via a fair selection rule.

**Theorem 2.4.14** Let $P$ be a program and $Q$ a query both fair-bounded by $\| \|$ and $I$. Then every SLD-derivation of $P$ and $Q$ via a fair selection rule is finite.

**Proof.** Suppose that there exists an infinite SLD-derivation $Q_i, \ldots, Q_n$ of $P$ and $Q$ via a fair selection rule. By Lemma 2.4.10, every $Q_i$ is fair-bounded by $\| \|$ and $I$. We distinguish two cases depending whether or not for every $i \geq 1$, $I \models \exists Q_i$.

Suppose that $I \nvDash \exists Q_i$ for some $i \geq 1$. By Definition 2.4.7 (b), there exists $k \in N$ such that for every $A_1, \ldots, A_n$ ground instance of $Q_i$, there exists $j \in [1, n]$ such that $I \nvDash A_j$ and $k > |A_j|$. By Lemma 2.4.11, $A_j \not\in T_P \downarrow |A_j| + 1$. By monotonicity of $T_P$, $A_j \not\in T_P \downarrow k$. Summarizing, $T_P \downarrow k \nvDash \exists Q_i$. By Lemma 2.4.5, every fair SLD-derivation of $P$ and $Q_i$ is failed, hence finite. This contradicts the assumption that there exists an infinite fair SLD-derivation.

Suppose now that for every $i \geq 1$, $I \models \exists Q_i$.

By Lemma 2.4.13 (ii), $|Q_i| \succeq_m \ldots \succeq_m |Q_i| \succeq_m \ldots$ is an infinite descending chain of bags over naturals. This is impossible since the finite multiset ordering over naturals is well-founded. \qed

We are in the position to state soundness of the proposed proof method.

**Theorem 2.4.15 (Termination Soundness)** Let $P$ be a program and $Q$ a query both fair-bounded by $\| \|$ and $I$. Then $P$ and $Q$ $\exists$-universally terminate.

**Proof.** By Theorem 2.4.14, every SLD-derivation of $P$ and $Q$ via a fair selection rule is finite. By Theorem 2.4.3, we conclude that $P$ and $Q$ $\exists$-universally terminate. \qed

**Example 2.4.16** Let us consider Example 2.4.8. We showed that PRODCONS and the query system$(n)$ are both fair-bounded by $\| \|$ and $I$, where $n \in N$. By Theorem 2.4.14, we conclude that every fair SLD-derivation of PRODCONS and system$(n)$ is finite.

Analogously, we have that the PERMUTATION program and the query perm([a,b], Ys) of Example 2.4.9 $\exists$-universally terminate. \qed
We conclude this section with a corollary showing that the upward and downward ordinal closures of fair-bounded programs coincide on the set \( \{ A \mid |A| \neq \infty \} \).

**Corollary 2.4.17** Let \( P \) be a program fair-bounded by \( \| \| \) and \( I \). For every \( A \in B_L \) such that \( |A| \neq \infty \), we have that

\[
A \in T_P \downarrow \omega \quad \text{iff} \quad A \in T_P \uparrow \omega.
\]

**Proof.** Since \( |A| \in N \), we have that the query \( A \) is fair-bounded. By Theorem 2.4.14, every SLD-derivation of \( P \) and \( A \) via a fair selection rule \( s \) is finite. Then either there exists a SLD-refutation or every SLD-derivation via \( s \) is failed. We recall that there exists a refutation of \( P \) and \( A \) iff \( A \in T_P \downarrow \omega \), and that every SLD-derivation via \( s \) is failed iff \( A \not\in T_P \downarrow \omega \) (see Lemma 2.4.5). This implies that \( A \in T_P \downarrow \omega \) iff \( A \in T_P \uparrow \omega \). \( \Box \)

### 2.4.3 Completeness

In this section, we show that if \( P \) and \( Q \) \( \exists \)-universally terminate then they are fair-bounded by some \( \| \| \) and \( I \). Therefore, fair-boundedness is a sound and complete proof method for \( \exists \)-universal termination. We start with a simple observation.

**Lemma 2.4.18** For every program \( P \), \( T_P \downarrow \omega \) is a model of \( P \).

**Proof.** Since \( T_P \downarrow \omega = \cap_{i \in N} T_P \downarrow i \), by monotonicity of \( T_P \), \( T_P(T_P \downarrow \omega) \subseteq T_P \downarrow i + 1 \), for every \( i \in N \). Since \( T_P \downarrow 0 = B_L \), this implies \( T_P(T_P \downarrow \omega) \subseteq \cap_{i \in N} T_P \downarrow i + 1 = T_P \downarrow \omega \), i.e. \( T_P \downarrow \omega \) is a model of \( P \). \( \Box \)

We recall from Definition 2.3.17 that \( \text{length}_P^Q(A) \) is the maximum length of a SLD-derivation of \( P \) and \( Q \) via the selection rule \( s \).

The following result show that fair-boundedness is a complete termination proof method with respect to fair selection rules.

**Lemma 2.4.19** Let \( P \) be a program and \( f \) a fair selection rule. Then there exist an extended level mapping \( \| \| \) and a Herbrand interpretation \( I \) such that:

(1) \( P \) is fair-bounded by \( \| \| \) and \( I \), and

(2) for every \( A \in B_L \), \( |A| \in N \) iff every SLD-derivation of \( P \) and \( A \) via \( f \) is finite.

**Proof.** We define \( I = T_P \downarrow \omega \) and \( |A| = \text{length}_P^P(A) \), where \( rr \) is the round-robin selection rule, which is fair.

First, consider (ii). As a consequence of Theorem 2.4.3, every SLD-derivation of \( P \) and \( A \) via \( f \) is finite iff every SLD-derivation of \( P \) and \( A \) via \( rr \) is finite. By definition of \( \text{length}_P^P(A) \), we have that \( |A| \in N \) iff every SLD-derivation of \( P \) and \( A \) via \( rr \) is finite, hence the conclusion (ii).
2.4. $\exists$-Universal Termination

Let us now consider $(i)$. We show the proof obligations of Definition 2.4.6. By Lemma 2.4.18, $I$ is a model of $P$. Consider now $A \leftarrow B_1, \ldots, B_n$ in $\text{ground}_I(P)$.

(a) Suppose that $I \models B_1, \ldots, B_n$. By definition of $I$, this implies that for every $k \geq 0, T_P \downarrow k \models B_1, \ldots, B_n$. By Lemma 2.4.5, we conclude that there exists at least one non-failed SLD-derivation $\xi$ of $P$ and $B_1, \ldots, B_n$ via $rr$. We claim that for $i \in [1, n]$:

$$\text{length}_P^P(B_1, \ldots, B_n) \geq \text{length}_P^P(B_i).$$

(2.12)

In fact, consider a SLD-derivation $\xi'$ of $P$ and $B_i$ via $rr$. Since $rr$ is fair and $\xi$ is non-failed, there exists a SLD-derivation of $P$ and $B_1, \ldots, B_n$ where all the atoms in $\xi'$ are eventually selected, and the other selections are made accordingly to $\xi$. Thus, we obtain a SLD-derivation of $P$ and $B_1, \ldots, B_n$ via $rr$ whose length is greater or equal than the length of $\xi'$. In conclusion, (2.12) holds. Observing that $B_1, \ldots, B_n$ is an instance of a SLD-resolvent $Q$ of $P$ and $A$ via $rr$, we calculate for $i \in [1, n]$:

$$|A| = \text{length}_P^P(A)$$

$$\triangleright \{ \text{Lemma 2.3.18 (i) } \}$$

$$\text{length}_P^P(Q)$$

$$\triangleright \{ \text{Lemma 2.3.18 (ii) } \}$$

$$\text{length}_P^P(B_1, \ldots, B_n)$$

$$\triangleright \{ (2.12) \}$$

$$\text{length}_P^P(B_i) = |B_i|.$$

(b) Suppose now that $I \nvdash B_1, \ldots, B_n$. By definition of $I$, we have that for some $i \in [1, n]$ there exists $k_i \geq 0$ such that $T_P \downarrow k_i \nvdash B_i$.

Consider now $B_h$ such that $T_P \downarrow k_h \nvdash B_h$ and $\text{length}_P^P(B_h)$ is minimum.

By Lemma 2.4.5, we have that every SLD-derivation of $P$ and $B_h$ via $rr$ is failed.

By definition of $h$ and Lemma 2.4.5, we have that for $i \neq h$ there exists a SLD-derivation $\xi_i$ of $P$ and $B_i$ via $rr$ such that $\xi_i$ is either successful or has length greater or equal than $\text{length}_P^P(B_h)$. We claim that:

$$\text{length}_P^P(B_1, \ldots, B_n) \geq \text{length}_P^P(B_h).$$

(2.13)

In fact, let $\xi$ be a SLD-derivation of $P$ and $B_h$ with length $\text{length}_P^P(B_h)$. Consider now a SLD-derivation $\xi'$ of $P$ and $B_1, \ldots, B_n$ via $rr$ where the atoms are selected accordingly to $\xi$ and to $\xi_i$ for $i \neq h$. We observe that the length of $\xi'$ is at least $\text{length}_P^P(B_h)$, since $\xi_i$ for $i \neq h$ is either successful or longer than $\xi$. In conclusion, (2.13) holds. Observing that $B_1, \ldots, B_n$ is an instance of a SLD-resolvent $Q$ of $P$ and $A$, we calculate:

$$|A| = \text{length}_P^P(A)$$
Lemma 2.3.18 (i) \( length_{re}^P(Q) \)

Lemma 2.3.18 (ii) \( length_{re}^P(B_1, \ldots, B_n) \)

Theorem 2.4.20 (Termination Completeness) Let \( P \) be a program and \( Q \) a query that \( \exists \)-universally terminate. Then there exist \( || \) and \( I \) such that \( P \) and \( Q \) are both fair-bounded by \( || \) and \( I \).

Proof. By Theorem 2.4.3, every SLD-derivation of \( P \) and \( Q \) via any fair selection rule \( f \) is finite. Consider the program \( P' = P \cup \{ \text{new} \leftarrow Q \} \), where \text{new} \ is a fresh predicate symbol. By Lemma 2.4.19 (i), \( P' \) is fair-bounded by some \( || \) and \( I \). Since \text{new} \ is a fresh symbol, the assumption of the Theorem implies that every SLD-derivation of \( P' \) and \text{new} \ via \( f \) is finite. By Lemma 2.4.19 (ii), we conclude that \( |\text{new}| \in N \). Consider now the restrictions of \( || \) and \( I \) on \( L \), i.e. not including \text{new}. Since the definition of fair-boundedness is modular, \( P \) is fair-bounded by the restrictions of \( || \) and \( I \).

Turning the attention on \( Q \), since \text{new} \leftarrow Q \ is fair-bounded by \( || \) and \( I \), we have that for every ground instance \( A_1, \ldots, A_n \) of \( Q \):

(a) if \( I \models A_1, \ldots, A_n \) then for \( i \in [1, n] \), \( |\text{new}| \models |A_i| \), and

(b) if \( I \not\models A_1, \ldots, A_n \) then there exists \( i \in [1, n] \) such that \( I \not\models A_i \) and \( |\text{new}| \models |A_i| \).

In conclusion \( Q \) is fair-bounded by the restrictions of \( || \) and \( I \), by fixing \( k = |\text{new}| \) in Definition 2.4.7.

Summarizing, in the last two sections we showed that the class of fair-bounded logic programs and queries precisely characterizes the notion of \( \exists \)-universal termination, i.e. the class of logic programs and queries for which a complete control exists in the sense of Definition 2.1.1.

Example 2.4.21 (OddEven) The \texttt{ODDEVEN} program:

\begin{verbatim}
even(X) ←
X is an even natural number.
\end{verbatim}

\begin{verbatim}
even(s(X)) ← odd(X).
\end{verbatim}
even(0).

odd(X) ← X is an odd natural number.

odd(s(X)) ← even(X).

defines the even and odd predicates, with the usual intuitive meaning. The query even(X), odd(X) is intended to check whether the program defines a number that is both even and odd. We show that ODDEVEN and the query above are not fair-bounded by any || and I. By The Termination Completeness Theorem 2.4.20, we conclude that they do not 3-universally terminate.

Suppose that ODDEVEN is fair-bounded by || and I. By Definition 2.4.6 (a,b), for \( i \geq 0 \):

\[
|\text{odd}(s^{i+1}(0))| \triangleright |\text{even}(s^i(0))| \\
|\text{even}(s^{i+1}(0))| \triangleright |\text{odd}(s^i(0))|.
\]

which imply, for \( i \geq 0 \):

\[
|\text{even}(s^i(0))| \triangleright i \\
|\text{odd}(s^i(0))| \triangleright i.
\]  

(2.14)

Suppose now that the query above is fair-bounded by the same || and I, and let \( k \) be any natural number satisfying Definition 2.4.7. Consider the ground instance even(s^{k+1}(0)), odd(s^{k+1}(0)) of the query above. We distinguish two cases.

If the ground instance is true in I, then by Definition 2.4.7 (a) and (2.14) we should have:

\[
k \triangleright |\text{even}(s^{k+1}(0))| \triangleright k,
\]

which is absurd.

If the ground instance is not true in I, then again by Definition 2.4.7 (b) and (2.14) either:

\[
k \triangleright |\text{even}(s^{k+1}(0))| \triangleright k,
\]
or

\[
k \triangleright |\text{odd}(s^{k+1}(0))| \triangleright k,
\]

which are both impossible. \( \square \)

We conclude by observing that in the case of fair selection rules, locality of termination is lost, in the following sense. While for left termination, if a program and a query \( Q \) left terminate then there exists an atom (the leftmost one) in \( Q \) such that the program and the atom left terminate, this property does not hold for universal termination w.r.t. fair selection rules.

**Example 2.4.22** Consider the simple program:
q(a) ← q(a).
p(b) ← p(b)

We have that the queries \(q(X)\) and \(p(X)\) have only infinite SLD-derivations. On the contrary, their composition, namely the query \(q(X), p(X)\), has only finite SLD-derivation via any fair selection rule. Let us see how to reason on those queries in the framework of fair-boundedness. We define:

\[
I = \emptyset, \quad |q(a)| = |p(b)| = \infty, \quad |p(a)| = |q(b)| = 0.
\]

The program and the query \(q(X), p(X)\) are fair-bounded by \(I\) and \(|\cdot|\). In fact, there are two ground instances of the query, both of which are not true in \(I\). Therefore we have to show proof obligation \((b)\) of Definition 2.4.7. For the ground instance \(q(a), p(a)\), we observe that \(I \not\models p(a)\) and \(1 \triangleright |p(a)|\). For the ground instance \(q(b), p(b)\), \(I \not\models q(b)\) and \(1 \triangleright |q(b)|\) hold. We conclude \(\exists\)-universal termination of the query \(q(X), p(X)\).

On the contrary, the program and the query \(q(X)\) are not fair-bounded by the same \(I\) and \(|\cdot|\). In fact, by Definition 2.4.6 \((a, b), |q(a)| \triangleright |q(a)|\) must hold. Thus \(|q(a)| = \infty\). This implies, by Definition 2.4.7 \((a, b), q(a)\) cannot be fair-bounded by the same \(|\cdot|\) and \(I\). The same reasoning applies to \(p(X)\). Since fair-boundedness is a complete characterization of \(\exists\)-universal termination, the program and the query \(q(X)\) does not \(\exists\)-universally terminate.

\[\square\]

## 2.5 Bounded Nondeterminism and Termination

In the previous section, we have seen that a complete control for \(P\) and \(Q\) exists iff \(P\) and \(Q\) \(\exists\)-universally terminate. In general, however, a complete control in the sense of Definition 2.1.1 may not exist.

**Example 2.5.1 (OddEven Ctd)** Reconsider the \texttt{ODDEVEN} program and the query \(\texttt{even}(X), \texttt{odd}(X)\) of Example 2.4.21.

Even tough the program is recurrent, we have shown that \texttt{ODDEVEN} and the query above do not \(\exists\)-universally terminate. Therefore, there exists no complete control for them, in the sense of Definition 2.1.1.

Notice, however, that they have no SLD-refutation.

\[\square\]

In addition, very few systems adopt fair selection rule, due to implementation reasons.

**Example 2.5.2 (Permutation)** Reconsider the program \texttt{PERMUTATION} and the query \(\texttt{perm}(\llbracket a, b \rrbracket, Ys)\) of Examples 2.3.21 and 2.4.9.
We have shown that they do not left terminate. Although they \exists-universally terminate, it may be the case that the underlying system does not support fair selection rules. Thus, we are still left with the termination problem.

Notice, however, that the program and the query above have finitely many SLD-refutations via any selection rule. Thus, we could find out all of them by cutting SLD-trees at an appropriate level.

In the following, we introduce the notion of bounded nondeterminism of logic programs and queries, which is adapted from a similar notion in the context of imperative (parallel) programming (see e.g., [14]). We claim that it is strongly related to the notion of universal termination.

Nondeterminism arises in SLD-resolution in two ways. First, the selection of the atom to be reduced is arbitrary. By Strong Completeness of SLD-resolution (see [9]), however, the order of reductions is immaterial. Second, the choice of the input clause is made nondeterministically from a number of alternatives. In the literature, don't know and don't care nondeterminism are usually distinguished. In the case of don't care nondeterminism, only one choice is pursued at each resolution step: any choice will lead to a solution, and it does not matter which particular solution is found. For don't know nondeterminism the choice matters, but the correct one is not known at the time the choice is made, thus all alternatives must be considered.

The following definition introduces the concept of bounded nondeterminism.

**Definition 2.5.3** Let $P$ be a program and $Q$ a query. We say that $P$ and $Q$ have bounded nondeterminism iff for every selection rule $s$ there are finitely many SLD-refutations of $P$ and $Q$ via $s$.

The relation between this definition and the notion of universal termination is tight. In fact, if $P$ and $Q$ $\exists$-universally terminate, then $P$ and $Q$ have bounded nondeterminism. Conversely, if $P$ and $Q$ have bounded nondeterminism then there exists an upper bound to the length of the SLD-refutations of $P$ and $Q$. If the upper bound is known, then we can syntactically transform $P$ and $Q$ into an equivalent program and query that universally terminate w.r.t. all selection rules, i.e. any selection rule will be a complete control for them.

**Example 2.5.4** The \texttt{ODDEVEN} program and the query \texttt{even(X), odd(X)} of Example 2.4.21 have bounded nondeterminism, since they have no SLD-refutation, while they do not $\exists$-universally terminate.

However, we observe that the boundary between $\exists$-universal termination and bounded nondeterminism is not so marked. In fact, the simple program:

\[
p \leftarrow p.
\]

$P$
and the query \( p \) have not bounded nondeterminism. Therefore, they do not \( \exists \)-universally terminate. On the contrary, the program:

\[
\begin{align*}
p & \leftarrow p, q, \quad p
\end{align*}
\]

and the query \( p \) have bounded nondeterminism and, moreover, \( \exists \)-universally terminate. Finally, the program

\[
\begin{align*}
p & \leftarrow p, q, \quad p, q \\
q & \leftarrow q
\end{align*}
\]

and the query \( p \) do not \( \exists \)-universally terminate, while they have bounded nondeterminism.

In the following, we offer a declarative characterization of programs and queries that have bounded nondeterminism, by introducing the class of bounded programs and queries. A direct application to termination of the proposed theoretical framework is a source-to-source transformation for bounded programs and queries that yields programs and queries that universally terminate via any selection rule, while retaining the set of refutations.

### 2.5.1 Bounded Programs

In this section we introduce a declarative characterization of the class of programs and queries that have bounded nondeterminism.

**Definition 2.5.5** Let \( \mid \mid \) be an extended level mapping, and \( I \) a Herbrand interpretation. A logic program \( P \) is bounded by \( \mid \mid \) and \( I \) iff \( I \) is a model of \( P \) such that for every \( A \leftarrow B_1, \ldots, B_n \) in \( \text{ground}_L(P) \):

\[
I \models B_1, \ldots, B_n \quad \text{implies} \quad \text{for } i \in [1, n] \quad |A| \succ |B_i|.
\]

It is straightforward to note that the definition of bounded programs is a simplification of Definition 2.4.6 of fair-bounded programs, where proof obligation (b) is discarded. Intuitively, the definition of boundedness only requires the decreasing of the extended level mapping when the body atoms are true in some model of the program, i.e. they might have a refutation.

Also, notice that proof obligations are modular, in the sense that program clauses are taken into consideration separately, and the notion of boundedness is purely declarative, in the sense that neither any procedural notion is needed in order to prove a program bounded nor the definition reflects some fixed ordering of the atoms. The next definition extends boundedness to queries.
2.5. Bounded Nondeterminism and Termination

**Definition 2.5.6** Let $||$ be an extended level mapping, and $I$ a Herbrand interpretation. A query $Q$ is **bounded by $||$ and $I$** iff there exists $k \in \mathbb{N}$ such that for every $A_1, \ldots, A_n \in \text{ground}_L(Q)$:

$$ I \models A_1, \ldots, A_n \quad \text{implies} \quad \forall i \in [1, n] \quad k \geq |A_i|. $$

\[\square\]

**Example 2.5.7** Reconsider the program:

```
p ← p, q.
p.
q ← q.
```

and the query $p$. Let $h, k \in \mathbb{N}$. We define:

$$ I = \{ p \} \quad |p| = h \quad |q| = k. $$

Obviously, $I$ is a model of the program. We observe that for unit clauses there is not proof obligation. Consider, instead, the first and the third clause. Since their bodies are not true in $I$, the proof obligations of Definition 2.5.5 are trivially satisfied. In conclusion, the program is bounded by $||$ and $I$. Also, the query $p$ is bounded by the same level mapping and Herbrand interpretation.

Then we have no restriction on $h$ and $k$. Intuitively, they are upper bounds for the depth of proof trees for $p$ and $q$ respectively. Therefore, the best choices for $h, k$ are $h = k = 0$.

\[\square\]

**Example 2.5.8 (OddEven Ctd)** Consider again the \texttt{ODDEVEN} program. It is readily checked that it is bounded by defining:

$$ |\text{even}(x)| = |\text{odd}(x)| = \text{size}(x), $$

$$ I = \{ \text{even}(s^{2i}(0)), \text{odd}(s^{2i+1}(0)) \mid i \geq 0 \}. $$

The query \texttt{even}(X), \texttt{odd}(X) is bounded by $||$ and $I$. In fact, since no instance of its is true in $I$, Definition 2.5.6 imposes no requirement.

\[\square\]

**Example 2.5.9 (Permutation)** Consider the program \texttt{PERMUTATION} and the query \texttt{perm([a, b], Ys)} of Examples 2.3.21 and 2.4.9.

Since we showed that they are fair-bounded by the level mapping $||$ and the Herbrand interpretation $I$ of Example 2.4.9 at page 34, they are also bounded by the same $||$ and $I$.

In particular, $k = |[a, b]| + 1 = 3$ satisfies the proof obligations of Definition 2.5.6.

\[\square\]
2.5.2 Soundness

In this section, we show that if a program and a query are bounded then they have bounded nondeterminism. First, we show that the notion of boundedness is persistent along SLD-derivations.

**Lemma 2.5.10 (Persistency)** Let \( P \) be a program and \( Q \) a query both bounded by \( \| \) and \( I \). Every SLD-resolvent \( Q' \) of \( P \) and \( Q \) is bounded by \( \| \) and \( I \).

**Proof.** A simplification of the proof of Lemma 2.4.10. \( \square \)

To show that bounded programs and queries have bounded nondeterminism, we follow an approach which is similar to the approaches used for proving termination soundness in the case of acceptability and fair-boundedness. In particular, we observe that Lemma 2.4.13 extends to bounded programs.

**Lemma 2.5.11** Let \( P \) be a program and \( Q \) a query both bounded by \( \| \) and \( I \). For every SLD-resolvent \( Q' \) of \( P \) and \( Q \), we have that:

\( (i) \quad \| Q \| \geq m \| Q' \|, \) and

\( (ii) \quad \text{if } I \models \exists Q' \text{ then } \| Q \| \succ_m \| Q' \|. \)

**Proof.** A simplification of the proof of Lemma 2.4.13. \( \square \)

Next, we introduce a subrelation of \( \succ_m \), which is parametric in a natural number \( v \) that denotes the maximum number of elements that may replace an element in the \( \succ \) ordering.

**Definition 2.5.12** Let \( v \) be a natural number, and \( \text{bag}(N) \) the set of multisets on \( N \). The relation \( \succ^v_m \) is defined as the transitive closure of the relation \( \succ^v \) in which \( x \succ^v y \) if \( x \) can be obtained from \( y \) by replacing an element \( a \) of \( y \) by at most \( v \) (possibly zero) elements \( b \in N \) such that \( a > b \).

Intuitively, next Lemma shows that if proof trees of an atom \( A \) have a maximum depth \( k \), then the length of refutations of \( A \) is bounded by \( f(k) \) for a given and computable function \( f \).

**Lemma 2.5.13** Let \( P \) be a program and \( \text{bag}(N) \) the set of multisets on \( N \).

There exists a computable function \( f : N \rightarrow N \) such that for every \( k \in N \), and every sequence \( b_0, b_1, \ldots, b_n \) of bags such that

\[ \text{bag}(k) \succ^v_m b_0 \succ^v_m \ldots \succ^v_m b_n \]

it results \( f(k) > n \).
Proof. We map the elements in the bags of the chain $\text{bag}(k) \triangleright_m b_0 \triangleright_m \ldots \triangleright_m b_n$ in a maximal tree constructed as follows.

The root of the tree is $k$.

Consider a node $h$. If $h$ is replaced by (at most $v$) $a_1, \ldots, a_m$ elements lower than $h$ then there are $m$ children labeled with $a_1, \ldots, a_m$.

We observe that such a tree is $v$-branching and has at most $k + 1$ levels. It turns out that a $v$-branching tree with at most $k + 1$ levels has at most $f(k)$ nodes, where:

$$f(k) = \begin{cases} \frac{v^{k+1} - 1}{v - 1} & \text{if } v > 1 \\ k + 1 & \text{if } v = 1 \\ 1 & \text{if } v = 0. \end{cases}$$

Notice now that every element in the bags $\text{bag}(k), b_0, b_1, \ldots, b_n$ appears in the tree as defined above. Since the number of elements in $\text{bag}(k), b_0, b_1, \ldots, b_n$ is at least $n + 1$ (since every bag - apart $b_n$ - must have at least one element), we have that $n$ is bounded by the maximum number of elements in a tree of the form above. Therefore, the function $f$ satisfies the conclusion of the Lemma.

We are now in the position to show that bounded programs and queries have bounded nondeterminism.

**Theorem 2.5.14 (Soundness)** Let $P$ be a program and $Q$ a query both bounded by $\|\|$ and $I$. Then $P$ and $Q$ have bounded nondeterminism.

Proof. We show that, under the hypothesis of the theorem, the length of a SLD-refutation of $P$ and $Q$ via any selection rule $s$ is bounded by some $b \in N$. By König's Lemma and the fact that SLD-trees are finitely branching, we conclude that there are finitely many SLD-refutations of $P$ and $Q$ via $s$, i.e. $P$ and $Q$ have bounded nondeterminism.

By Definition 2.5.6 there exists $k \in N$ such that for every $B_1, \ldots, B_n$ ground instance of $Q$:

$$I \models B_1, \ldots, B_n \implies \text{for } i \in [1, n] \quad k \triangleright |B_i|.$$  

Called $v = \max\{n_P, n\}$, this implies:

$$\text{bag}(k) \triangleright_m ^v b |Q|^\prime. \quad (2.15)$$

Consider now a SLD-refutation $Q, Q_1, \ldots, Q_j$ of $P$ and $Q$ via a selection rule $s$. By Soundness of SLD-resolution, for every $i \in [1, j]$, $I \models \exists Q_i$. Moreover, by Lemma 2.5.10 for every $i \in [1, j]$, $Q_i$ is bounded by $\|\|$ and $I$. By Lemma 2.5.11 (ii) and (2.15), $\text{bag}(k) \triangleright_m ^v b |Q|^\prime \triangleright_m ^v b |Q_1|^\prime \triangleright_m ^v \ldots \triangleright_m ^v b |Q_j|^\prime$ is a descending chain of bags over naturals. Moreover, since SLD-resolution replaces an atom by at most $n_P$ atoms, the stronger statement

$$\text{bag}(k) \triangleright_m ^v b |Q|^\prime \triangleright_m ^v b |Q_1|^\prime \triangleright_m ^v \ldots \triangleright_m ^v b |Q_j|^\prime$$
holds. By Lemma 2.5.13, we get \( f(k) > j \), i.e. the length of the refutation is bounded by \( f(k) \). Since SLD-trees are finitely branching, by König's Lemma there are finitely many SLD-refutations for \( P \) and \( Q \).

**Example 2.5.15** Consider the program:

\[
p \leftarrow p, q.
p
\]

and the query \( p \) of Example 2.5.7. We have shown that they are both bounded by the same \( \| \| \) and \( I \). By Theorem 2.5.14, we conclude that they have bounded nondeterminism.

The same conclusion hold for the \textsc{oddeven} program and the query \textsc{even}(X), \textsc{odd}(X), and for \textsc{permutation} and the query \textsc{perm}([a, b], Ys).

**2.5.3 Completeness**

In this section, we show that boundedness is a complete characterization of bounded nondeterminism. First, we introduce a variant of the \( \text{length}^P_s \) function, targeted to measure refutation length rather than derivation length.

**Definition 2.5.16** For a program \( P \) and a query \( Q \), we define \( \text{rlength}^P_s(Q) \) as \( \infty \) if there exist infinitely many SLD-refutations of \( P \) and \( Q \) via the selection rule \( s \), and as the maximum length of a SLD-refutation of \( P \) and \( Q \) via \( s \) otherwise.

The analogous of Lemma 2.3.18 holds for \( \text{rlength}^P_s \).

**Lemma 2.5.17** Let \( P \) be a program, \( Q \) a query and \( s \) a selection rule. Then:

(i) for every \( Q' \) SLD-resolvent of \( P \) and \( Q \) via \( s \)

\[
\text{rlength}^P_s(Q) \models \text{rlength}^P_s(Q'),
\]

(ii) for every \( Q' \) instance of \( Q \), \( \text{rlength}^P_s(Q) \models \text{rlength}^P_s(Q') \).

**Proof.** (i) Let \( \xi' \) be a SLD-refutation of \( P \) and \( Q' \) via \( s \) whose length is \( l \). Since \( Q' \) is a SLD-resolvent of \( P \) and \( Q \), there exists a SLD-refutation \( \xi \) of \( P \) and \( Q \) that first selects the atom selected in \( Q \) by \( s \), then selects according to the selections of \( \xi' \).

By the Independence Lemma [10, Theorem 3.3], there exists a SLD-refutation of \( P \) and \( Q \) via \( s \) whose length is equal to the length of \( \xi \), namely to \( l + 1 \).

Consider now two cases.

Suppose that there exist infinitely many SLD-refutations of \( P \) and \( Q' \) via \( s \). Since SLD-trees are finitely branching, by König's Lemma lengths are unbounded, and then by reasoning as above we can find infinitely many SLD-refutations of \( P \) and \( Q \) via \( s \). Summarizing, (i) holds.
Suppose that there exist finitely many SLD-refutations of $P$ and $Q'$ via $s$. By reasoning as above in the case that $\xi'$ is the longest SLD-refutation, we get that there exists a SLD-refutations of $P$ and $Q$ via $s$ longer than $\xi'$, which implies (i).

(ii). Consider a SLD-refutation $\xi'$ of $P$ and $Q'$ via $s$. By the Lifting Theorem [10, Theorem 3.22] there exists an SLD-refutation $\xi$ of $P$ and $Q$ which is of the same length of $\xi$. By the Independence Lemma [10, Theorem 3.33] there exists a SLD-refutation $\mu$ of $P$ and $Q$ via $s$ using the same clauses of $\xi'$. Therefore, $rlength^P_s(Q) \geq rlength^P_s(Q')$.

The next lemma states that every program is bounded by an extended level mapping defined in terms of the length of SLD-refutations.

**Lemma 2.5.18** Let $P$ be a program and $s$ a selection rule. Then there exist an extended level mapping $\lVert \rVert$ and a Herbrand interpretation $I$ such that:

(i) $P$ is bounded by $\lVert \rVert$ and $I$, and

(ii) for every $A \in B_L$, $|A| \in N$ iff there are finitely many SLD-refutations of $P$ and $A$ via $s$.

**Proof.** We define $I = M^P_P = T_P \uparrow \omega$ and $|A| = rlength^P_s(A)$.

First, we observe that (ii) is immediate by definition of $\lVert \rVert$. Let us now consider (i). We show the proof obligations of Definition 2.5.5. Obviously, $I$ is a model of $P$. Consider now $A \leftarrow B_1, \ldots, B_n$ in $ground_L(P)$. Suppose that $I \models B_1, \ldots, B_n$. Since $I = M^P_P$, by Strong Completeness of SLD-resolution there exists a SLD-refutation $\xi$ of $P$ and $B_1, \ldots, B_n$ via $s$. We claim that for $i \in [1, n]$:

$$rlength^P_s(B_1, \ldots, B_n) \geq rlength^P_s(B_i). \tag{2.16}$$

In fact, consider a SLD-refutation $\xi'$ of $P$ and $B_i$ via $s$. Since $\xi'$ is successful, there exists a SLD-refutation of $P$ and $B_1, \ldots, B_n$ via $s$ where all the atoms in $\xi'$ are eventually selected, and the other selections are made accordingly to $\xi$. Thus, we obtain a SLD-refutation of $P$ and $B_1, \ldots, B_n$ via $s$ whose length is greater or equal than the length of $\xi'$, i.e. (2.16) holds. Observing that $B_1, \ldots, B_n$ is an instance of a SLD-resolvent $Q$ of $P$ and $A$, we calculate for $i \in [1, n]$:

$$|A| = rlength^P_s(A) \uparrow \{ \text{Lemma 2.5.17 (i)} \} \uparrow |A| = rlength^P_s(Q) \uparrow \{ \text{Lemma 2.5.17 (ii)} \} \uparrow rlength^P_s(B_1, \ldots, B_n) \uparrow \{ \text{2.16} \} \uparrow rlength^P_s(B_i) = |B_i|.$$
Chapter 2. Termination of Logic Programs

Let us show the completeness result.

**Theorem 2.5.19 (Completeness)** Let $P$ be a program and $Q$ a query that have bounded nondeterminism. Then there exist $||$ and $I$ such that $P$ and $Q$ are both bounded by $||$ and $I$.

**Proof.** Let $s$ be any selection rule.

Consider the program $P' = P \cup \{ \text{new} \leftarrow Q \}$, where $\text{new}$ is a fresh predicate symbol. By Lemma 2.5.18 (i), $P'$ is bounded by some $||$ and $I$. Moreover, the assumption of the Theorem implies that there are finitely many SLD-refutations of $P$ and $\text{new}$ via $s$. By Lemma 2.5.18 (ii), $[\text{new}] \in N$. Consider now the restrictions of $||$ and $I$ to $B_L$, i.e. not including $\text{new}$. Since proof obligations of Definition 2.5.5 are modular, it is readily checked that $P$ is bounded by the restrictions of $||$ and $I$. Turning the attention on $Q$, since $\text{new} \leftarrow Q$ is bounded by $||$ and $I$, we have that for every ground instance $A_1, \ldots, A_n$ of $Q$, if $I \models A_1, \ldots, A_n$ then for $i \in [1, n]$, $[\text{new}] \triangleright [A_i]$. In conclusion $Q$ is bounded by the restrictions of $||$ and $I$ on $B_L$, by fixing $k = [\text{new}]$ in Definition 2.5.6. □

### 2.5.4 From Bounded Nondeterminism to Universal Termination

So far, we have developed a theoretical framework for the analysis of bounded nondeterminism of logic programs. We have proposed a declarative characterization of the class of programs and queries that have bounded nondeterminism in terms of bounded programs and queries. As already mentioned, bounded nondeterminism is strongly related to universal termination. In fact, if $P$ and $Q$ $\exists$-universally terminate, then $P$ and $Q$ have bounded nondeterminism, and then they are bounded. Conversely, if $P$ and $Q$ have bounded nondeterminism, i.e. they are bounded, then there exists an upper bound to the length of SLD-refutations of $P$ and $Q$.

By looking into the proof of the Soundness Theorem 2.5.14, we observe that the natural number $k$ of Definition 2.5.6 of bounded queries provides us with an upper bound for the elements that appear in the multisets associated with queries along a SLD-refutation of $P$ and $Q$. Thus, we next propose a syntactic transformational approach that prunes SLD-derivations in such a way that the multisets associated to queries contain only elements that are bounded by a given natural $k$. This will provide us with a terminating control procedure for bounded programs and queries. For notational convenience, we denote by $T$ a sequence $T_1, \ldots, T_n$ of terms, with $n \geq 0$. Also, $s^k(0)$ is a shorthand for $s( \ldots s(0) \ldots )$, where $s$ is repeated $k$ times.

**Definition 2.5.20** Let $P$ be a program and $Q$ a query both bounded by $||$ and $I$, and let $k \in N$. 


We define $\text{Ter}(P)$ as the program such that:

- for every clause in $P$
  
  \[
  p_0(T_0) \leftarrow p_1(T_1), \ldots, p_n(T_n).
  \]
  with $n > 0$, the clause
  
  \[
  p_0(T_0, s(D)) \leftarrow p_1(T_1, D), \ldots, p_n(T_n, D).
  \]
  is in $\text{Ter}(P)$, where $D$ is a fresh variable,

- and, for every clause in $P$
  
  \[
  p_0(T_0).
  \]
  the clause
  
  \[
  p_0(T_0, D).
  \]
  is in $\text{Ter}(P)$, where $D$ is a fresh variable.

Finally, if $Q$ is $p_1(T_1), \ldots, p_n(T_n)$ then we define $\text{Ter}(Q, k)$ as the query:

\[
q_0(T_0, s^k(0)), \ldots, q_n(T_n, s^k(0)). \]

This definition serves to illustrate the following result, relating bounded nondeterminism and universal termination.

**Theorem 2.5.21** Let $P$ be a program and $Q$ a query both bounded by $\| \|$ and $I$, and let $k$ be a given natural number satisfying Definition 2.5.6.

Then, for every $n \in N$, $\text{Ter}(P)$ and $\text{Ter}(Q, n)$ universally terminate w.r.t. all selection rules.

Moreover, there is a bijection between SLD-refutations of $P$ and $Q$ via a selection rule $s$ and SLD-refutations of $\text{Ter}(P)$ and $\text{Ter}(Q, k-1)$ via $s$.

**Proof.** It is readily checked that $\text{Ter}(P)$ and $\text{Ter}(Q, n)$ are recurrent by a level mapping $| |$ such that $\|p(t, m)\| = \text{size}(m)$. Therefore, universal termination follows from Theorem 2.2.8.

Consider the second part of the Theorem.

First, assume that $s$ is the leftmost selection rule. We associate to a LDB-refutation $\xi = Q, Q_1, \ldots, Q_j$ for $P$ and $Q$, a LDB-derivation $\xi'$ for $\text{Ter}(P)$ and $\text{Ter}(Q, k-1)$ selecting at each step the input clause in $\text{Ter}(P)$ corresponding (accordingly to the transformation $\text{Ter}$) to the input clause in $P$ selected at the same step in $\xi$. Assume by absurd that $\xi'$ is failed, i.e. the leftmost atom $p(T, t)$ is selected in some $Q'$ and it does not unify with any clause in $\text{Ter}(P)$.

If $t$ is of the form $s(t')$, then we conclude that $p(T)$ does not unify with any clause of $P$, which is impossible since $\xi$ is a refutation.
Assume now that \( t = 0 \). We associate to every query in \( \xi' \) the bag \( \text{bag}(a_1, \ldots, a_h) \) where \( a_1, \ldots, a_h \) are the naturals occurring in the rightmost arguments of atoms in the query. Called \( Q'' \) the corresponding query in \( \xi \), it is readily checked that, for \( 1|Q''| = \text{bag}(b_1, \ldots, b_h) \), it results \( a_i \geq b_i \) for \( i \in [1, h] \). Therefore, \( 1|Q''| \) is necessarily of the form \( \text{bag}(0, b_2, \ldots, b_h) \). Since \( s \) is the leftmost selection rule, this implies that \( p(T) \) unifies with a unit clause of \( P \). This is absurd, otherwise \( p(T, t) \) would unify with the corresponding clause of \( \text{Ter}(P) \).

Thus \( \xi' \) is a LD-refutation. Finally, we observe that the mapping \( \psi \) associating \( \xi' \) to \( \xi \) is a bijection (modulo renaming apart).

Consider now the case that \( s \) is arbitrary. By Strong Completeness of SLD-resolution there is a bijection \( \phi \) mapping SLD-refutations via \( s \) to LD-derivations. Therefore, the conclusion of the Theorem follows by considering the bijection \( \phi \circ \psi \circ \phi^{-1} \). □

The intuitive reading of this result is that the transformed program and query maintain the same success semantics of the original program and query. It is worth noting that no assumption is made on the selection rule \( s \), i.e. any selection rule is a complete control for the transformed program and query.

**Example 2.5.22 (Permutation)** Reconsider the program \textsc{permutation} and the query \texttt{perm([a, b], Ys)} of Examples 2.3.21 and 2.5.9. The transformed program \textsc{Ter(permutation)} is:

\[
\begin{align*}
\text{perm}([1], [], D). \\
\text{perm}([X|Xs], Ys, s(D)) & \leftarrow \\
& \quad \text{delete}(X, Ys, Zs, D), \\
& \quad \text{perm}(Xs, Zs, D). \\
\text{delete}(X, [X|Y], Y, D). \\
\text{delete}(X, [H|Y], [H|Z], s(D)) & \leftarrow \\
& \quad \text{delete}(X, Y, Z, D).
\end{align*}
\]

and the transformed query for \( k = 3 \) is \texttt{perm([a,b], Ys, s^3(0))}. By Theorem 2.5.21, the transformed program and query running on any logic programming system provide us with a terminating control for the original program and query, modulo the extra argument added to each predicate. □

The transformations \textsc{Ter}(P) and \textsc{Ter}(Q, k) are of pure theoretical interest. The practical side of the approach can be outlined as follows. First an extended level mapping and a model are inferred such that \( P \) and \( Q \) are bounded, and a natural \( k \) is computed satisfying Definition 2.5.6. The automatic inference of level mappings and Herbrand models will be discussed in Section 2.8. Then the logic programming system (usually a compiler) associates to every atom in a derivation a counter (that plays the role of the parameter introduced by the transformation) and add to the pure SLD-resolution mechanism some (very simple) checks that restrict the search space to atoms with associated non-negative counters.
Specifically for Prolog systems, the transformation can be rephrased by using arithmetic built-in’s.

**Example 2.5.23 (Permutation Ctd)** Consider the PERMUTATION program. The following transformed program:

```prolog
perm([], [], D).
perm([X|Xs], Ys, D) ←
    D > 0,
    D1 is D - 1,
    delete(X, Ys, Zs, D1),
    perm(Xs, Zs, D1).
delete(X, [X|Y], Y, D).
delete(X, [X|Y], [Y|Z], D) ←
    D > 0,
    D1 is D - 1,
    delete(X, Y, Z, D1).
```

and query `perm([a, b], Ys, 2)` make use of arithmetic built-in’s to prune derivations. □

## 2.6 Arithmetic Built-in’s

A program with arithmetic is a logic program in which the predicates:

\[<, =<, =:, =/=, is, >=, >\]

can appear only in clause bodies. These predicates are defined for particular terms, called ground arithmetic expressions (in short, gae’s). Formally, the set `Gae` of gae’s is obtained by removing from the Herbrand universe on the signature

\[\Sigma_{Ar} = \{0, 1, -1, 2, -2, \ldots, \times^2, \times^2, /, \mod_2\}\]

all terms containing a division by zero. We denote by `Gae` the set of numerals \(\{0, 1, -1, 2, -2 \ldots\}\). For a gae \(n\), `value(n)` is the integer denoted by \(n\).

According to [150], we extend LD-resolution assuming that a program with arithmetic in which `>` appears, implicitly contains the set of unit clauses:

\[P_{>} = \{ n > m. \mid n, m \in Gae \land value(n) > value(m) \}\]

(and analogously for `<`, `=<`, `=:\`, `=/\`, `>=`). If `is` appears in the program, we consider the set:

\[P_{is} = \{ value(m) is m. \mid m \in Gae \}\].
Consider now a LD-derivation for a program with arithmetic and a query such that the atom \( n > m \) is selected. If \( n, m \) are gae's then, according to the implicit clauses, the LD-derivation fails if \( \text{value}(n) \) is lower or equal than \( \text{value}(m) \). If the value of \( n \) is greater than that of \( m \), the resolvent is the rest of the goal.

We stipulate that a LD-derivation for a program with arithmetic and a query ends in an error if an atom \( n > m \) is selected and \( n, m \) are not gae's. This is the procedural semantics of \( > \) in Prolog. A similar operational semantics is given for \( <, =<, =:, =/=, =/> \), whereas for \( \text{is} \) only the second argument is required to be a gae.

**Example 2.6.1 (Part)** Consider the program \textsc{Part}:

\[
\begin{align*}
\text{part}(X, [Y|Xs], [Y|Ls], Bs) & \leftarrow X > Y, \text{part}(X, Xs, Ls, Bs). \\
\text{part}(X, [Y|Xs], Ls, [Y|Bs]) & \leftarrow X =< Y, \text{part}(X, Xs, Ls, Bs). \\
\text{part}(X, [], [], []) & \leftarrow \\
\end{align*}
\]

for partitioning a list of gae's. If \( x \) is a gae and \( xs \) is a list of gae's, then the query \text{part}(x, xs, Ls, Bs) is intended to compute in Ls (resp., Bs) those elements in \( xs \) that are lower (resp., greater or equal) than \( x \). We recall that \textsc{Part} implicitly contains the clauses in \( P> \) and \( P=< \).

Unfortunately, as discussed by Apt [10], it is not possible to reason in a declarative way on run-time arithmetic errors within the logic programming theory. In particular, the Lifting Lemma and the SLD-resolution Completeness Theorem do not hold for programs with arithmetic. Restricted form of those results can be provided under the assumption that no SLD-derivation for \( P \) and \( Q \) via a selection rule \( s \) ends in an error. We will reason about this problem in Section 4.2.1.

Here, we are concerned with the problem of extending the notions of recurrency, acceptability, fair-boundedness, and boundedness to programs with arithmetic. Consider, as an example, the \textsc{Part} program.

**Example 2.6.2 (Part Ctd)** As a logic program, \textsc{Part} is recurrent by defining:

\[
\begin{align*}
|x > y| &= |x =< y| = 0 \\
|\text{part}(x, x, Ls, Bs)| &= |xs|.
\end{align*}
\]

We observe that there is no proof obligation for unit clauses (including the implicit ones). Let us consider now a ground instance of the first clause (for the second clause we reason symmetrically):

\[
\begin{align*}
\text{part}(x, [y | y], [y | l], bs) & \leftarrow x > y, \text{part}(x, y, l, bs). \\
\end{align*}
\]
We have that
\[ |\text{part}(x, [y \mid ys], [y \mid ls], bs)| = |ys| + 1 \]
\[ \triangleright 0 = |x \triangleright y| \]
and
\[ |\text{part}(x, [y \mid ys], [y \mid ls], bs)| = |ys| + 1 \]
\[ \triangleright |ys| \]
\[ = |\text{part}(x, ys, ls, bs)|. \]
These relations show that \textsc{part} is recurrent by \(||\). In addition, the query \text{part}(X, Ys, Ls, Bs) is recurrent by the same \(||\), provided that \(Ys\) is a list.

As highlighted by this example, Definition 2.2.3 of recurrency imposes no proof obligations for unit clauses. Similarly acceptability, fair-boundedness and boundedness, require only that the Herbrand interpretation used is a model of the unit clauses, and in particular that it is a model of the clauses \(P_{op}\) for each arithmetic predicate \(op\) appearing in \(P\). Also, we observe that in the proofs of The Soundness Theorems 2.2.8, 2.3.15, 2.4.15 and 2.5.19 we have not assumed programs to be finite sets of clauses. In addition, the property that SLD-trees are finitely branching, used in the proof of Theorem 2.5.19, continues to hold for programs with arithmetic. Therefore, all the Termination Soundness results extend to programs with arithmetic.

**Theorem 2.6.3** Let \(P\) be a program with arithmetic, and \(Q\) a query both recurrent (resp., acceptable, fair-bounded, bounded) by \(||\) (and \(I\)). Then \(P\) and \(Q\) universally terminate w.r.t. all selection rules (resp., universally left terminate, \(\exists\)-universally terminate, have bounded non-determinism), with SLD-derivations possibly ending in errors.

**Example 2.6.4** (\textsc{part Ctd}) By Theorem 2.6.3, every SLD-derivation of \textsc{part} and the query \text{part}(X, Ys, Ls, Bs) is finite, when \(Ys\) is a list. However, when \(Ys\) is not a list of gae’s or \(X\) is not a gae, SLD-derivations may end in errors.

### 2.7 Classes of Terminating Programs

We have introduced several classes of programs and queries, which provide declarative characterizations of operational notions such as universal termination and bounded nondeterminism. Looking at those classes from an operational point of view, we recognize that recurrency of a program and a query implies acceptability of them, which in turn implies fair-boundedness, which in turn implies boundedness. Below we relate these classes each other from a declarative point of view. In addition to the implications above, the results of this section are useful in practice, since they allow to reuse or to simplify termination proofs.
**Theorem 2.7.1** Let $P$ be a program and $Q$ a query, $||$ an extended level mapping and $I$ a Herbrand model of $P$. Each of the following statements implies the statements below it:

(i) $P$ and $Q$ are recurrent by $||$.

(ii) $P$ and $Q$ are acceptable by $||$ and $I$.

(iii) $P$ and $Q$ are fair-bounded by $||$ and $I$.

(iv) $P$ and $Q$ are bounded by $||$ and $I$.

**Proof.** We restrict to consider programs. The reasoning is analogous for queries.

(i $\rightarrow$ ii). Immediate from Definitions 2.2.3, and 2.2.5.

(ii $\rightarrow$ iii). Let us show the proof obligations of Definition 2.4.6. Consider $A \leftarrow B_1, \ldots, B_n$ in $\text{ground}_\ell(P)$:

(a) if $I \models B_1, \ldots, B_n$ then by Definition 2.3.7 we have that for every $i \in [1, n]$, $|A| \triangleright |B_i|.$

(b) if $I \not\models B_1, \ldots, B_n$ then let $k \in [1, n]$ such that $I \models B_1, \ldots, B_{k-1}$ and $I \not\models B_k$. By Definition 2.3.7, we conclude $|A| \triangleright |B_k|$. Summarizing, $I \not\models B_k$ and $|A| \triangleright |B_k|.$

(iii $\rightarrow$ iv). Immediate from Definitions 2.4.6 and 2.4.7.

It is also readily checked that the implications of the Theorem are strict. $\square$

**Example 2.7.2 (Reuse of Termination Proofs)** In Example 2.3.8 we showed that the NAIVE REVERSE program is acceptable by a level mapping $||$ and a model $I$. The proof obligations of acceptability should be shown for every clause of the program.

However, we note that APPEND is a sub-program of NAIVE REVERSE. Since we have already proved in Example 2.2.7 that APPEND is recurrent by a level mapping that coincides with $||$ on append atoms, by Theorem 2.7.1, we conclude that the proof obligations for clauses defining append are satisfied for every Herbrand model of APPEND, and in particular for the Herbrand interpretation $I$ of Example 2.3.8.

We refer the reader to Apt and Pedreschi [7] for a collection of results on reuse of proofs of recurrence to show acceptability, and on proving acceptability of $P \cup P'$ by reusing separate proofs for $P$ and $P'$. $\square$
2.8 Inferring Termination

On a theoretical level, the problem of deciding whether a program belongs to one of the classes studied in this Chapter is undecidable. For recurrent programs, this result was proved by Bezem [23]. Undecidability of checking bounded nondeterminism was shown by Devienne et al. [69]. The next theorem sums up the undecidability results.

**Theorem 2.8.1** It is undecidable whether there exist $\|\|$ and $I$ such that a program $P$ and a query $Q$ are both acceptable, fair-bounded, or bounded by $\|\|$ and $I$.

**Proof.** First, we consider acceptability (resp., fair-boundedness). By The Soundness and Completeness Theorems 2.3.15, 2.3.20 (resp., 2.4.15, 2.4.19) the problem whether there exist $\|\|$ and $I$ such that $P$ and $Q$ are acceptable (resp., fair-bounded) by $\|\|$ and $I$ is decidable iff it is decidable whether $P$ and $Q$ left terminate (resp., $\exists$-universally terminate). Devienne et al. [69, Theorem 8] show that it is undecidable whether given a program consisting of only one clause of the form:

$$p(T_1, \ldots, T_n) \leftarrow p(S_1, \ldots, S_n). \quad (2.17)$$

and a query $p(V_1, \ldots, V_n)$, the SLD-resolution stops. The particular form (2.17) implies that there is only one SLD-derivation for the program and the goal. As a consequence, it is undecidable whether they left terminate or $\exists$-universally terminate, and *a fortiori* whether they are acceptable or fair-bounded.

Consider now for every program $P$ of the form (2.17), the program $P'$ obtained by adding the fact $p(X_1, \ldots, X_n)$, where $X_1, \ldots, X_n$ are distinct variables. Due to their particular form, we observe that $P$ and the query $p(V_1, \ldots, V_n)$ are acceptable by $\|\|$ and $I$ iff $P'$ and $p(V_1, \ldots, V_n)$ are bounded by $\|\|$ and some $J$. Therefore, if it were decidable whether a given program and query are bounded by some extended level mapping and Herbrand interpretation, then it would be decidable whether programs and queries of the form above are acceptable by some $\|\|$ and $I$, which has been shown to be undecidable. \qed

On a practical level, however, many (sufficient) approaches are currently available to automatically infer (usually, left) termination. This research area is nowadays very active, as witnessed by the body of research cited in the Related Work Section, with some efficient tools already integrated within existing compilers. In general, we argue that most approaches can be directly adapted for proving the proof obligations of acceptability (Definitions 2.3.7 and 2.3.9) and boundedness (Definitions 2.5.5 and 2.5.6). This claim is not substantiated by a dedicated section, since this is somewhat beyond the scope of the thesis. However, the following example makes it clear how one of the existing approaches can be adapted to infer boundedness and acceptability.

**Example 2.8.2 (Inferring Boundedness)** Let us consider again the program `PERMUTATION`. 

---

**Example 2.8.2 (Inferring Boundedness)** Let us consider again the program `PERMUTATION`. 

---
perm(Xs, Ys) ← Ys is a permutation of the list Xs

\[(p1)\quad perm([], []).\]
\[(p2)\quad perm([X|Xs], Ys) \leftarrow\]
\quad delete(X, Ys, Zs),
\quad perm(Xs, Zs).\]

\[(d1)\quad delete(X, [X|Y], Y).\]
\[(d2)\quad delete(X, [H|Y], [H|Z]) \leftarrow\]
\quad delete(X, Y, Z).\]  

and the query \(perm([a, b], Ys)\). Consider now to have to show that \texttt{PERMUTATION} and the query above are bounded by a same extended level mapping \(| |\) and Herbrand interpretation \(I\). Here, we make the following assumptions:

(Assumption A1). Every \(n\)-ary predicate symbol \(p\) is annotated with a \textit{mode}, namely a function \(d_p\) from \{1, \ldots, \(n\)\} in \{+, \(-\)\}. If \(d_p(i) = ' +'\) we call \(i\) an \textit{input} position. If \(d_p(i) = ' -'\) then \(i\) is called an \textit{output} position. We write \(d_p\) in the form \(p(d_p(1), \ldots, d_p(n))\). Intuitively, a mode specifies the use of predicate arguments of \(p\), with the intended meaning that terms occurring in output positions are determined from the computation of the terms occurring in input positions. As an example, the following are intuitive modes for the predicates of \texttt{PERMUTATION}:

\[
\begin{align*}
\text{perm}(+, -) & \quad \text{delete}(+, -, +)
\end{align*}
\]

Modes are usually specified directly by the programmer. Alternatively, they can be automatically inferred for a given program \(P\), query \(Q\) and selection rule \(s\) in such a way that every atom selected in a SLD-derivation of \(P\) and \(Q\) via \(s\) satisfy some expected property (usually, that computation is \textit{data-driven}).

(Assumption A2). \(| |\) is defined as a linear combination of the list-length of the predicate arguments which occur in input positions, i.e.:

\[
\begin{align*}
|perm(Xs, Ys)| &= p_0 + p_1 |xs| \\
|delete(X, Xs, Ys)| &= d_0 + d_1 |x| + d_2 |ys|,
\end{align*}
\]

where \(p_0, p_1, d_0, d_1, d_2\) denote natural numbers that need to be determined.

This assumption is a special case of the largely employed assumption that level mappings are defined as linear combinations of semi-linear norms.

(Assumption A3) \(I\) is characterized as the set of atoms whose predicate arguments satisfy a linear disequation, such as:

\[
\begin{align*}
I &= \{ \text{perm}(Xs, Ys) \mid p'_0 + p'_1 |xs| \geq p'_2 |ys| \} \cup \\
&\quad \{ \text{delete}(X, Xs, Ys) \mid d'_0 + d'_1 |x| + d'_2 |ys| \geq d'_3 |xs| \}
\end{align*}
\]

where \(p'_0, p'_1, p'_2, d'_0, d'_1, d'_2, d'_3\) denote natural numbers that need to be determined. Observe that the linear disequations are syntactically derived by fixing the arguments
2.8. Inferring Termination

occurring in input positions to the left of the disequation, and those occurring in output positions to the right. This approach is an heuristic due to Decorte et al. [61], based on the intuition that modes specify data-driven computations.

Assumptions (A1-A3) allows us to derive proof obligations of a particular form, which can be solved with efficient methods. A challenging topic of the research in automatic termination analysis consists of finding out standard forms of level mappings and models for which the solution of the resulting proof obligations can be reconducted to known problems for which efficient algorithms exists.

Let us see now the proof obligations of Definition 2.5.6 for the symbolic level mapping and Herbrand interpretation above. Consider a ground instance \( \text{perm}([a, b], ys) \) of the query. We have to find out a natural \( k \) such that:

\[
I \models \text{perm}([a, b], ys) \Rightarrow k > \text{perm}([a, b], ys).
\]

which by definition of \( \models \) and \( I \), can be rewritten as:

\[
2p'_1 + p'_0 \geq p'_2|ys| \Rightarrow k > p_0 + 2p_1.
\]

Therefore, we have the symbolic constraint:

\[
c0. \quad \forall ys \quad -p'_2|ys| + (2p'_1 + p'_0) \geq 0 \Rightarrow k - p_0 - 2p_1 > 0.
\]

Let us write the proof obligations of Definition 2.5.5 for the clauses of \text{PERMUTATION}.

(p1) We have only to show that \( I \) is a model of it, i.e. \( p'_1||| + p'_0 \geq p'_2||| \). By definition of \( ||| \), we have the symbolic constraint:

\[
c1. \quad p'_0 \geq 0.
\]

(p2) Consider a ground instance of (p2):

\[
\text{perm}([x|xs], ys) \leftarrow \text{delete}(x, ys, zs), \text{perm}(xs, zs).
\]

The body is true in \( I \) if \( \phi \) holds, where:

\[
\phi \equiv d'_0 + d'_1|x| + d'_2|zs| - d'_2|ys| \geq 0 \land p'_0 + p'_1|xs| - p'_2|zs| \geq 0.
\]

Therefore, the decreasing of the level mapping from the head to the leftmost atom in the body imposes the constraint:

\[
\forall x, xs, ys, zs \quad \phi \Rightarrow p_0 + p_1|xs| + p_1 > d_0 + d_1|x| + d_2|zs|
\]

which after some rearrangement, can be written as:

\[
c2. \quad \forall x, xs, ys, zs \quad \phi \Rightarrow p_1|xs| - d_1|x| - d_2|zs| + (p_0 + p_1 - d_0) > 0.
\]
Chapter 2. Termination of Logic Programs

The decreasing from the head to the second body atom imposes the conditional constraint:

\( c_3. \forall x, xs, ys, zs \phi \Rightarrow p_1 > 0. \)

Finally, the requirement imposed by proof obligation that \( I \) must be a model of the clause produces the constraint:

\( c_4. \forall x, xs, ys, zs \phi \Rightarrow p'_1|xs| - p'_2|ys| + (p'_1 + p'_0) \geq 0. \)

\( (d1) \) We have only to show that \( I \) is a model of it, i.e.

\[ \forall x, y, \ d'_0 + d'_1|x| + d'_3|y| \geq d'_2|[x|y|]. \]

By definition of \( |\cdot| \), we have the symbolic constraint:

\( c_5. \forall x, y, \ (d'_2 - d'_0)|y| + d'_1|x| + (d'_0 - d'_2) \geq 0. \)

\( (d2) \) The decreasing from the head to the body atom yields:

\[ \forall x, y, z \ d'_0 + d'_1|x| + d'_3|z| \geq d'_2|y| \Rightarrow \]

\[ d_0 + d_1|x| + d_2|z| + d_2 > d_0 + d_1|x| + d_2|z|, \]

which produces the symbolic constraint:

\( c_6. \forall x, y, z \ d'_1|x| + d'_3|z| - d'_2|y| + d'_0 \geq 0 \Rightarrow d_2 > 0. \)

Finally, the constraint imposed by the requirement that \( I \) is a model of \( (d2) \) is the following:

\( c_7. \forall x, y, z \ d'_1|x| + d'_3|z| - d'_2|y| + d'_0 \geq 0 \Rightarrow d'_1|x| + d'_3|z| - d'_2|y| + (d'_2 + d'_0 - d'_2) \geq 0. \)

A general method to solve conditional constraints \( c_0 - c_7 \) of the form above is not known. However, Decorte and De Schreye [60] propose a method that reduces those constraints into a set of linear constraints over the variables \( p_0, p_1, p_2, p'_0, p'_1, \ldots \). The basic idea consists of observing that \( c_0 - c_7 \) are solvable by imposing that all the coefficients appearing in the disequations at the right of the implications are non-negative (and at least one is positive in the case that the disequation is strict). However, since this sufficient condition is in many cases too strong, they propose to apply first a weakening of the original constraint by nondeterministically rewriting a constraint of the form \( \forall \ldots e \geq 0 \land \ldots \Rightarrow e' \geq 0 \) into

\[ \forall \ldots e \geq 0 \land \ldots \Rightarrow e - e' \geq 0. \]

For instance, when applied to constraint \( c2 \) and the second conjunct in \( \phi \), this rule produces:
2.8. Inferring Termination

\( c^2' : \forall x, xs, ys, zs. \phi \Rightarrow \)
\[-d_1|x| + (p_1 - p'_1)|xs| + (-d_2 + p'_2)|zs| + d'_3|ys| + (p_0 + p_1 - d_0 - p'_0) > 0.\]

A sufficient condition to satisfy this constraint is then to require:
\[-d_1 \geq 0\]
\[p_1 - p'_1 \geq 0\]
\[-d_2 + p'_2 \geq 0\]
\[p_0 + p_1 - d_0 - p'_0 > 0.\]

With the same approach, we derive the following constraints from \( c_0, c_1, c_3 - c_7 \):
\[k - p_0 - 2p_1 > 0\]
\[p_1 > 0\]
\[-d'_1 \geq 0\]
\[p'_2 - d'_3 \geq 0\]
\[d'_2 - p'_2 \geq 0\]
\[p'_1 - d'_0 \geq 0\]
\[d'_3 - d'_2 \geq 0\]
\[d'_0 - d'_2 \geq 0\]
\[d_2 > 0,\]

where variables range over naturals. Such constraints are directly solvable by a constraint solver over finite domains – and often over boolean suffices. A solution of those constraints is the following:
\[p_1 = d_2 = p'_1 = p'_2 = d'_0 = d'_2 = d'_3 = 1\]
\[p_0 = d_0 = d_1 = p'_1 = p'_0 = d'_0 = 0\]
\[k = 3,\]

which lead to the level mapping and the interpretation:
\[|\text{perm}(xs, ys)| = |xs|\]
\[|\text{delete}(x, xs, ys)| = |ys|,\]
\[I = \{ \text{perm}(xs, ys) \mid |xs| \geq |ys| \} \cup\]
\[\cup \{ \text{delete}(x, xs, ys) \mid 1 + |ys| \geq |xs| \}.\]

Notice how \( |\mid \) and \( I \) closely resemble the level mapping and the interpretation of Example 2.4.9.

Finally, we observe that symbolic constraints of the same form of \( c_0 - c_7 \) are derived in the case of acceptability. Hence, the approach of Decorte and De Schreye can be readily applied also to infer acceptability – and actually, it has been proposed for inferring left termination. \( \square \)
Example 2.8.3 (Permutation Ctd) Reconsider the Example 2.5.22 at page 51, where the transformation \( Terr \) was shown for \texttt{PERMUTATION} and the query \texttt{perm([a,b], Ys)}. In that Example, we fixed \( k = 3 \). The previous Example shows as the value \( k = 3 \) could have been automatically inferred by an automatic termination analysis compiler module.

\[\square\]

Summarizing, the examples above highlight how the characterizations of the classes of programs and queries that left terminate or have bounded nondeterminism support the termination analysis, by providing the basic proof obligations that a (general purpose) constraint solver must check for satisfiability.

Consider now the class of fair-bounded programs. In general, by the Termination Completeness Theorem 2.4.20, fair-bounded programs and queries include every class of programs and queries that universally terminate via some selection rule. Therefore, any existing automatic tool for proving universal termination of \( P \) and \( Q \) is sufficient for proving that \( P \) and \( Q \) are fair-bounded. This fact allows us to reuse all existing automatic tools and termination proofs to the purpose of showing fair-boundedness.

Apart from this consideration, inferring extended level mappings and Herbrand models (such as in Example 2.8.2) in the case of fair-boundedness seems more difficult than in the case of acceptability and boundedness. In this sense, the problem of automatic inference of fair-boundedness is then substantially open.

### 2.9 Related Work

A comprehensive survey on termination of logic programs can be found in the paper by De Schreye and Decorte [58]. They classify three types of approaches: techniques that express necessary and sufficient conditions for termination, techniques that provide decidable \textit{sufficient} conditions, and techniques that prove decidability or undecidability for subclasses of programs and queries. Under this classification, this Chapter falls in the first type.

We refer the reader to the cited survey for a complete reference to works before 1994. Below, we recall those that are strongly related to the issues treated in this Chapter. Also, we mention some works appeared after 1994. In Chapter 7, we will reference works on termination of constraint logic programs, which, as a special case, can be employed for proving termination of logic programs.

Universal and existential termination

The distinction between \textit{universal} and \textit{existential} termination was identified by Vasak and Potter [158]. They characterized the class of universal terminating queries for a given program with fixed selection rules. Their characterization is expressed in terms of the complement of an inductively defined set. However, it cannot be easily used to prove termination.
Norms and level mappings

Level mappings were used first by Bezem [22] and Cavedon [39]. The list-length norm was introduced by Ullman and van Gelder [155], while the size norms by van Gelder [157]. Norms (on possibly non-ground terms) and their use in proving termination of logic programs are studied, among the others, by Bossi et al. [29], De Schreye et al [59], and Plümer [135].

Recurrent and acceptable programs

Recurrent programs were introduced by Bezem [22, 23] and further generalized to general logic programs by Apt and Bezem [3].

Acceptability (also for general programs and arithmetic built-in's) was introduced by Apt and Pedreschi [16]. In [7], the authors present refinements of the method that deal with modular proofs. Notice, however, that they use a different definition of left termination. In their sense, a program is left terminating if every LD-derivation starting with a ground query is finite. Their definition of acceptability is complete w.r.t. this notion of left termination, but, as discussed, it is not w.r.t. Definition 2.3.2.

Local selection rules, coroutining and 3-universal termination

The vast majority of logic programming systems adopt fixed selection rules (such as the leftmost one), or local selection rules. The study of termination via selection rules other than the leftmost one is a subject which is being increasingly investigated.

Marchiori and Teusink [112] propose a sufficient termination method for so-called local selection rules, i.e. rules that resolve completely an atom in a goal before starting resolution of the other atoms. Contrarily to fair selection rules, local selection rules do not encompass coroutining executions, such as in the producer-consumer program PRODCONS of Example 2.4.4. Also, Example 2.4.22 points how locality of termination is lost in the case of fair selection rules.

We refer the reader to Naish [122] for a discussion about subtleties and problems of termination in presence of coroutining executions. He showed a method for composing terminating programs by disjunction, conjunction and recursion to form terminating programs in presence of coroutining.

In recent years, systems adopting dynamic selection rules are being proposed, i.e. systems where the choice of the selected atom is made at run-time. In Gödel [88], for instance, programs are enriched with annotations, called delay declarations, specifying restrictions on the admissible selection rules. The Mercury system [147] automatically (even tough statically) reorders clause body atoms in order to guarantee left termination of the transformed program and query. The implementation of fair selection rules has been announced for future releases of Mercury.
The properties of fair selection rules in SLD-resolution were studied in the fundamental works of Apt and van Emden [17], Lassez and Maher [103] and van Emden and Nait Abdallah [156].

**Bounded nondeterminism, transformational approaches and loop detection**

Martin and King [115] showed a transformation for Gödel programs, that shares with the transformation of Definition 2.5.20 for bounded programs, the idea of not following derivations longer than a certain length. However, they rely on sufficient conditions for inferring the length of refutations, namely termination via a class of selection rules called semilocal. Their transformation adds run-time overhead, since the maximum length is computed at run time. On the other hand, a run-time analysis is potentially able to generate more precise upper bounds than our static transformation, and then to cut more unsuccessful branches.

Among the transformational methodologies to the termination analysis, we recall the approach of Baudinet [19], transforming logic programs into functional programs, and Krishna Rao et al. [102], transforming logic programs into term rewriting systems.

A well-known approach that prunes SLD-derivations (possibly including refutations) when some patterns are repeated is called loop checking. The advantages and the limitations of the approach are reported by Bol et al. [25, 26].

**The termination problem**

The study of classes of programs and queries for which the termination problem is undecidable has been largely investigated in the literature [27, 69, 58]. In this Chapter, we used a result by Devienne et al. [69] that states undecidability for the class of programs consisting of a single clause of the type \( p(T_1, \ldots, T_n) \rightarrow p(S_1, \ldots, S_n) \). The computations they entail are referred to as cycle unification.

**Automatic termination analysis**

In Example 2.8.2, we have made the assumption that the model \( I \) is of a particular form. In automatic termination analysis, the role of models is played by interargument relations.

An interargument relations for a predicate \( p \) w.r.t. a norm \( \| \| \) is a relation \( R \subseteq N^* \) such that for every \( p(t_1, \ldots, t_n) \) in \( M^I_p \), \( p(\|t_1\|, \ldots, \|t_n\|) \) is in \( R \). Several formulations in the literature [135, 58, 155] assume interargument relations of a certain form (e.g. we assumed in Example 2.8.2 linear disequations) in such a way the termination proof obligations can be stated in a tractable form. Recently, constraint logic programming languages are being widely used as a means to infer interargument relations [118, 21].
Example 2.8.2 is inspired by the constraint solving approach of Decorte and De Schreye [60]. Some other recent proposals concentrate on automatic inference of termination.

Lindenstrauss and Sagiv [105] have developed a system called TermiLog for checking termination. They used linear norms, (monotonicity and equality) constraint inference and the termination test of Sagiv [142], originally designed for Datalog programs.

Speirs et al. reported in [148] the implementation of the static termination analysis algorithm of the Mercury system, which ensures termination with respect to the leftmost selection rule for a program obtained by permuting the body atoms of the original program. They claim a better performance than the system TermiLog in the average case.

Codish and Taboch [43] proposed a formal semantics basis that facilitates abstract interpretation for inferring left termination.

Modes were first considered in Mellish [116]. Decorte et al. [62] investigate the advantages of integrating modes and types in automatic termination analysis. Recently, Etalle et al. [75] proposed a refinement of acceptability in presence of modes.

Finally, Bossi et al. [29], Bronsard et al. [37], and Deransart and Maluszynski [66] propose termination methods that benefit from typing or partial correctness information. This issue will be discussed in Chapter 4.

2.10 Conclusion

Termination is a desirable property in its own, and it turns to be essential for declarative programming, where a crucial point is to associate a complete control strategy to declarative programs.

In this Chapter, we have recalled the definition of recurrent programs, which are designed to characterize universal termination w.r.t. all selection rules, highlighting some incompleteness problems.

Then we have proposed a modification of the well-known class of acceptable programs and queries, that is a sound and complete characterization of left termination.

Also, we introduced the notion of 3-universal termination, and proved that it coincides with universal termination via fair selection rules. Then, we have offered a sound and complete characterization of 3-universal termination by defining fair-bounded programs and queries. They precisely characterize the class of programs and queries for which there exists a complete control. In this sense, Kowalski’s motto can be rephrased as:

\[
\text{fair-bounded programs} = \text{logic + control.}
\]

Moreover, we have introduced the notion of bounded nondeterminism, by providing a characterization of it in terms of bounded programs and queries. A direct
application of the framework presented consists of a source-to-source transformation that, while preserving refutations, provides a terminating control for bounded programs and queries.

The characterizations of acceptable, fair-bounded and bounded programs are purely declarative, modular, simple and practical in paper & pencil proofs, and directly extensible to programs with arithmetic.

The issue of automatic inference of level mappings and models is not treated in this thesis. However, on the one hand, we have sketched how automated (sufficient) methodologies can be designed to infer acceptability and boundedness of programs and queries by adapting existing approaches. On the other hand, we observe that the characterizations introduced in this Chapter provide a valid support for the design of automatic tools.

The problem of automatic inference of fair-boundedness remains, instead, substantially open.
Chapter 3

From Termination to Verification

The main motivation of this thesis lies in the lack of proof methods able to reason on several logic program properties within a unifying declarative framework. The starting point of our approach has been the introduction of various notions of termination, and the design of sound and complete declarative characterizations of terminating programs and queries. In this Chapter, we first introduce concepts and properties that are of interest in program verification, such as (weak) partial correctness, (weak) total correctness, persistence of the proof relation and patterns characterization. Then, we combine the termination proof methods of Chapter 2 with a well-known correctness proof method, called the inductive method, in order to obtain a unifying framework able to reason on all the mentioned properties. The thesis of this Chapter is that the termination methods presented in Chapter 2 can be systematically lifted to proof methods for weak total correctness. Our objective is to design methods that are a trade-off between expressive power (i.e., the set of programs and properties they are able to reason about) and ease of use in paper & pencil proofs.

3.1 Logic Program Verification

3.1.1 Which Semantics for Program Verification?

Of course, when verifying a logic program $P$, it would be helpful to use its declarative semantics. However, several declarative semantics have been proposed as promising alternatives to the least Herbrand model $M_P$ (also known as $M$-semantics) for supporting program verification. Among the others, we mainly focus on two of them, namely the C-semantics of Falaschi et al. [77] (also known as the least term model of Clark [42], or the proof theoretic semantics of Deransart [64]) and the $S$-semantics.
of Falaschi et al. [31, 76].

Definition 3.1.1 For a logic program $P$ we define:

$$
\mathcal{M}(P) = \{ A \in B_L \mid P \models A \},
\mathcal{C}(P) = \{ A \in \text{Atom}_L \mid P \models A \},
\mathcal{S}(P) = \{ A \in \text{Atom}_L \mid A \text{ is a computed instance of a pure atom} \}.
$$

The $\mathcal{C}$-semantics of a program is the set of atomic logical consequence of the program, or, equivalently, the set of roots of proof trees of the program.

While the definition of $\mathcal{S}$-semantics may not seem declarative, we observe that a declarative definition was given by Falaschi et al. Also, we recall from Bossi et al. [31] a result that clarifies the relevance of the $\mathcal{S}$-semantics. Let us denote with $\text{mgi}(Q, \mathcal{Q})$ the set of most general instances of a query $Q$ and any query whose atoms are renamed apart atoms from $Q$. The next theorem states that it is possible to reconstruct the set of computed instances of $P$ and $Q$ starting from $\mathcal{S}(P)$.

Theorem 3.1.2 The set of computed instances of a program $P$ and a query $Q$ coincides with $\text{mgi}(Q, \mathcal{S}(P))$. 

In Apt et al. [11], the relative information ordering of the three semantics is studied and the semantics are related to each other with the claim that for a large class of programs and properties one can restrict to consider only the simple $\mathcal{M}$-semantics. Under certain conditions, the $\mathcal{C}$- and $\mathcal{S}$-semantics can be reconstructed starting from the least Herbrand model.

Therefore, $\mathcal{M}$-semantics seems to be a good trade-off between expressive power, abstraction and easy of use in paper & pencil proof methods. Then, a natural approach consists of considering $\mathcal{M}_P$ as the intended interpretation — therefore, the verification of a program is viewed as checking that the intended interpretation of a program and its least Herbrand model do coincide. This approach, however, turns out to be inadequate: strangely enough, the least Herbrand models semantics is not sufficiently abstract. In fact, the absence of types implies that the least Herbrand model is generally polluted with unintended atoms.

Example 3.1.3 (APPEND) Consider the APPEND program:

```prolog
append(Xs, Ys, Zs) ←
Zs is the result of concatenating the lists Xs and Ys.
append([], Xs, Xs).
append([X|Xs], Ys, [X|Zs]) ←
append(Xs, Ys, Zs).
```

APPEND is intuitively correct with respect to its specification but (if there are sufficiently many symbols in the language) its *intended interpretation* is not a model of the program. In fact, in the least Herbrand model unintended atoms appear, such as \texttt{append([], foo, foo)}.

As a consequence, reasoning about the whole least Herbrand model implies having to take into account unintended atoms, thus making the specification complex and counter-intuitive. This problem becomes much harder in modular program development, since adding more symbols to the language in the upper modules entails changing the least Herbrand model of lower modules, and hence their correctness properties. A clear point emerges from the previous discussion: a semantics for verification should take the intended or well-typed queries into account.

### 3.1.2 Specifications and Correctness

Following a Hoare's logic style of defining partial and total correctness, we stipulate that a specification consists of two parts.

**Definition 3.1.4** A specification is a pair \((\text{Pre}, \text{Post})\) of Herbrand interpretations, i.e., subsets of \(B_L\).

The rationale under this choice is the following. The first interpretation, \(\text{Pre}\), specifies the intended, or well-typed one-atom queries, i.e., those queries for which we designed the program under consideration. The second interpretation, \(\text{Post}\), specifies some desired property of successful one-atom queries. In this sense, a specification \((\text{Pre}, \text{Post})\) describes the input-output behavior of a logic program, in a way that closely resembles that in Hoare's logic, where preconditions specify the admissible input, and postconditions specify (properties of) the expected output. Here, preconditions specify the admissible input queries, and postconditions specify the expected output, namely properties of the correct instances of the input queries.

According to this choice, the well-typed fragment of the least Herbrand model is \(M^L_P \cap \text{Pre}\). We are now ready to define our notions of (weak) partial and (weak) total correctness.

**Definition 3.1.5** Let \(P\) be a logic program.

- \(P\) is **partially correct** w.r.t. a specification \((\text{Pre}, \text{Post})\) iff \(M^L_P \cap \text{Pre} = \text{Post}\).
- \(P\) is **totally correct** w.r.t. a specification \((\text{Pre}, \text{Post})\) iff \(M^L_P \cap \text{Pre} = \text{Post}\) and \(\text{Pre} \subseteq M^L_P \cup FF^L_P\), where \(FF^L_P\) is the finite failure set of \(P\).

In addition, \(P\) is **weak partially** or **weak totally** correct if the weaker requirement \(M^L_P \cap \text{Pre} \subseteq \text{Post}\) holds instead of \(M^L_P \cap \text{Pre} = \text{Post}\).
It should be noted that both partial and total correctness are defined in purely declarative terms, since the sets $M^p_L$ and $FF^p_L$ can be constructed without reference to the procedural interpretation of logic programming [10, 106]. As usual, the difference between partial and total correctness is that, in the latter case, we also require a weak form of termination, namely that every query in $Pre$ either succeeds or (finitely) fails.

Although $Pre$ is a set of ground atoms, it should be stressed that admissible queries are not required to be necessarily ground. We shall devise proof methods to reason about any atomic query $Q$ such that $Pre \models Q$, namely any query true in $Pre$, or equivalently, any query whose instances are included in $Pre$. In this sense, the mechanism of pre- and postconditions is suitable to deal with properties at non-ground level, such as correct instances characterization and the operational notion of call patterns characterization.

**Definition 3.1.6** Let $P$ be a logic program, $(Pre, Post)$ a specification and $Q$ a query.

$Post$ characterizes the correct instances of $P$ and $Q$ if for every correct instance $Q'$ of $P$ and $Q$, $Post \models Q'$.

$Pre$ characterizes the call patterns of $P$ and $Q$ via a selection rule $s$ if for every atom $A$ selected in a SLD-derivation of $P$ and $Q$ via $s$, $Pre \models A$.

In the case of call patterns characterization, $Pre$ can be used to specify certain desired run-time properties, ranging from persistency of types up to absence of run-time errors, safe omission of the occur-check and, for general programs, non-flourdering.

**Example 3.1.7 (APPEND CTD)** As an example, the APPEND program is intuitively totally correct w.r.t. the specification:

$$Pre_{APPEND} = \{ \text{append}(xs, ys, zs) \mid xs, ys \in GList \}$$
$$Post_{APPEND} = \{ \text{append}(xs, ys, zs) \mid xs, ys \in GList \land zs = xs \ast ys \}$$

where $\ast$ is the list concatenation operator and $GList$ the set of ground lists. Moreover, $Post_{APPEND}$ characterizes the success patterns of queries $\text{append}(Xs, Ys, Z)$, where $Xs$ and $Ys$ are (not necessarily ground) lists. Also $Pre_{APPEND}$ characterizes the call patterns of the same class of queries.

Notice that the weak version of either notions of correctness entails that $Post$ specifies some property of $M^p_L \cap Pre$. For instance, the APPEND program is weak totally correct w.r.t. $(Pre_{APPEND}, Post)$, where:

$$Post = \{ \text{append}(xs, ys, zs) \mid |zs| = |xs| + |ys| \}$$

and $|.|$ is the list-length function. Therefore, a postcondition in the sense of the weak correctness describes a property of every correct instance of an intended query. □
3.1.3 Proof Relations

The notions of specification and program correctness w.r.t. a specification are purely declarative, in the sense they express the desired program properties, while not providing us with a method for establishing those properties – apart from the rough construction of the least Herbrand model.

Therefore, (weak) partial/total correctness of a program w.r.t. a specification must be shown by means of some proof method. The core of a proof method consists of a proof relation \( \vdash \{\text{Pre}\} P \{\text{Post}\} \), relating a program \( P \) with the specification \((\text{Pre}, \text{Post})\), and similarly for queries \( Q \).

In this section, we introduce some properties of proof relations that are desirable in the verification of logic programs.

**Soundness.** \( \vdash \) is sound w.r.t. partial correctness (or another notion of correctness from Definition 3.1.1), when \( \vdash \{\text{Pre}\} P \{\text{Post}\} \) implies that \( P \) is partially correct w.r.t. \((\text{Pre}, \text{Post})\) (and equivalently for the other notions of correctness). Soundness is the essential property of a proof relation, stating that the proof method defines sufficient conditions for proving partial correctness (and equivalently for the other notions).

**Completeness.** \( \vdash \) is complete w.r.t. partial correctness (or another notion of correctness from Definition 3.1.1), when \( P \) partially correct w.r.t. \((\text{Pre}, \text{Post})\) implies that \( \vdash \{\text{Pre}\} P \{\text{Post}\} \) holds (and equivalently for the other notions of correctness). Completeness concerns whether the proof relation is strong enough for the problem at hand. Completeness (in presence of soundness) is difficult to obtain, and, in general, it has to be traded-off with practical usefulness for the intended user of the proof method, and with automation/efficiency for automatic implementations of the proof method.

**Persistency.** We say that \( \vdash \) is persistent if \( \vdash \{\text{Pre}\} P \{\text{Post}\} \) and \( \vdash \{\text{Pre}\} Q \{\text{Post}\} \) imply that for every SLD-resolvent \( Q' \) of \( P \) and \( Q \), \( \vdash \{\text{Pre}\} Q' \{\text{Post}\} \) holds. As a consequence, if the root of a SLD-tree is in the proof relation, every query in the tree is. Persistency of proof relations is a natural requirement, satisfied by many of the existing proof relations.

**Call and Success Patterns Characterization.** Consider \( P \) and \( Q \) such that \( \vdash \{\text{Pre}\} P \{\text{Post}\} \) and \( \vdash \{\text{Pre}\} Q \{\text{Post}\} \). We say that \( \vdash \) characterizes the call patterns of \( P \) and \( Q \) if \( \text{Pre} \) characterizes the call patterns of \( P \) and \( Q \). Analogously, we say that \( \vdash \) characterizes the success patterns of \( P \) and \( Q \) if \( \text{Post} \) characterizes the success patterns of \( P \) and \( Q \).

**Modularity w.r.t. programs.** Proof relations are usually required to support modular proofs. As an example, the following rule:

\[
\vdash \{\text{Pre}\} P \{\text{Post}\} \quad \vdash \{\text{Pre}\} P' \{\text{Post}\}
\]

\[
\vdash \{\text{Pre}\} P \cup P' \{\text{Post}\}
\]
expresses the fact that $\vdash$ is modular w.r.t. union of programs. In general, several modularity rules such as the one above hold for a given relation, which allow for modular proofs of program correctness.

**Modularity w.r.t. preconditions and postconditions.** Similarly, modularity may concern preconditions and postconditions. The rule:

$$\vdash \{\text{Pre}\} P \{\text{Post}\} \quad \vdash \{\text{Pre}\} P \{\text{Post}'\}$$

$$\vdash \{\text{Pre}\} P \{\text{Post} \cap \text{Post}'\}$$

expresses the fact that $\vdash$ is modular w.r.t. intersection of postconditions. Modularity of $\vdash$ w.r.t. union of preconditions is another interesting property.

**Example 3.1.8** Consider the proof relation defined as follows: $\vdash \{\text{Pre}\} P \{\text{Post}\}$ holds iff $\text{Pre} = B_L$ and there exist a level mapping $\|\|$ such that the program $P$ is acceptable by $\|\|$ and $\text{Post}$. $\vdash$ is defined similarly for queries.

The proof relation is obviously sound w.r.t. weak partial correctness, since $\text{Post}$ is required to be a model of $P$. By the Termination Soundness Theorem 2.3.15, it is also sound w.r.t. weak total correctness. In contrast, the proof relation is not sound, in general, w.r.t. partial and total correctness. Neither it is complete w.r.t. any correctness notion.

Also, we observe that by the Persistency Theorem 2.3.12, the proof relation is persistent if $\vdash \{\text{Pre}\} P \{\text{Post}\}$ and $\vdash \{\text{Pre}\} Q \{\text{Post}\}$ hold by the same level mapping.

Finally, directly from Definition 2.3.7 we have that the proof relation is modular w.r.t. intersection of postconditions.  

### 3.2 From Termination to Weak Total Correctness

The general approach we propose for deriving proof relations for weak total correctness consists of merging the termination proof methods of Chapter 2 with a well known weak partial correctness method. From a theoretical perspective, the resulting method is at most powerful as the sum of the separated methods. From a practical perspective, instead, the combined method should support shorter and simpler proofs, with respect to the separated methods, i.e. fewer proof obligations should be required.

First, let us introduce the weak partial correctness proof method. We note that the requirement of Definition 3.1.5:

$$M_p^L \cap \text{Pre} \subseteq \text{Post}$$

can be rewritten in the equivalent form:

$$M_p^L \subseteq \text{Pre} \rightarrow \text{Post}$$
where \( \text{Pre} \rightarrow \text{Post} \) is the set \((B_L \setminus \text{Pre}) \cup \text{Post}\). In particular, the latter form is usually known as correctness of the program \( P \) w.r.t. the set \( \text{Pre} \rightarrow \text{Post} \). A well-known method for proving this form of correctness consists of showing that \( \text{Pre} \rightarrow \text{Post} \) is a model of \( P \). This method is called the inductive method, and can be traced back to Clark [42]. When considering sets of the form \( \text{Pre} \rightarrow \text{Post} \), the method is not complete w.r.t. weak partial correctness.

**Example 3.2.1** Consider the program:

\[
p \leftarrow q, r.
\]

and the specification \( \text{Pre} = \{ p, r \} \), \( \text{Post} = \{ r \} \). It is immediate to observe that the program is weak partially correct w.r.t. the specification \((\text{Pre}, \text{Post})\), whilst \( \text{Pre} \rightarrow \text{Post} = \{ q, r \} \) is not a model of it. \( \square \)

For our purposes, however, here we restrict to observe that the inductive method is sound w.r.t. weak partial correctness, i.e. that if \( \text{Pre} \rightarrow \text{Post} \) is a Herbrand model of \( P \) then \( P \) is weak partially correct w.r.t. \((\text{Pre}, \text{Post})\).

Then, we note that all of the termination methods of Chapter 2, apart from recurrence, use some model \( I \) of the program under consideration (We will not deal with recurrence here, since, as already discussed, the class of recurrent programs and queries is rather small).

**(Step 1)** The first step in integrating the inductive method with a termination proof method is then to consider the relation resulting from the proof obligations of the termination method when using an extended level mapping \( | | \) and the Herbrand interpretation \( \text{Pre} \rightarrow \text{Post} \).

This first combination turns out to be sound for weak partial correctness of \( P \) w.r.t. \((\text{Pre}, \text{Post})\).

Moreover, the combined method inherits the persistency property of the original termination method. For example, since acceptability is persistent along SLD-derivations, the relation obtained from acceptability as described above is also persistent along SLD-derivations.

**(Step 2)** By imposing the further requirement:

\[
\text{for every } A \in \text{Pre} \quad |A| \neq \infty,
\]

we have that every query \( A \), with \( A \in \text{Pre} \), is acceptable, fair-bounded and bounded by \( | | \) and \( \text{Pre} \rightarrow \text{Post} \).

By the Termination Soundness Theorems of Chapter 2 and the outcomes of **(Step 1)**, this means that the resulting relation is sound w.r.t. weak total correctness in the case of acceptable and fair-bounded programs. In the case of bounded programs, instead, a restricted result will be shown.

**(Step 3)** At this point, however, there may be some unwanted consequences with this approach.
Example 3.2.2 Consider the program:

\[
\begin{align*}
p & \leftarrow q, \\
q & \leftarrow r, \\
r & \leftarrow r,
\end{align*}
\]

and the specification \(Pre = \{p\}, Post = \{p\}\). We have that \(Pre \rightarrow Post = B_L\), and then it is a model of \(P\). Assuming to show acceptability, we must find out an extended level mapping such that:

\[|p| \gg |q| \gg |r|\]

The problem here is that our analysis has to specify values of the extended level mapping for atoms which are not in \(Pre\), namely \(q\) and \(r\). Atoms not in \(Pre\) are, accordingly to the intended meaning of a specification, uninteresting or ill-typed, and, as such, should not be “involved” in the (verification) analysis.

But, what “involved” exactly means? Let us concentrate on acceptability. The reasoning is similar for fair-boundedness and boundedness.

Let \(P\) a program and \(Q\) a query both acceptable by \(\|\|\) and \(Pre \rightarrow Post\). By looking into the proof of the Termination Soundness Theorem 2.3.15, we realize that termination is established by the decreasing of the multiset of naturals (see Definition 2.3.13) associated to queries. The naturals in the multiset are levels of certain atoms appearing in ground instances of the query. In particular, the levels of these atoms for a query \(Q\) are:

\[
\bigcup_{i=1}^{n_Q} a|Q|_{Pre \rightarrow Post}^{i} = \{ |A_i| \mid A_1, \ldots, A_{n_Q} \in ground_L(Q), i \in [1, n_Q], \text{ and } Pre \rightarrow Post \models A_1, \ldots, A_{i-1} \}
\]

where \(n_Q\) is the number of atoms in \(Q\). A sufficient condition to ensure that every atom with level in \(a|Q|_{Pre \rightarrow Post}^{i}\) is in in \(Pre\), is to require that:

\[
\text{for every } A \in B_L \quad |A| \neq \infty \text{ iff } A \in Pre.
\]  

(3.1)

In fact, by Theorem 2.3.14, the levels in

\[
\bigcup_{i=1}^{n_Q} |Q|_{Pre \rightarrow Post}^{i}
\]

are bounded by a fixed natural number \(k\). Under the hypothesis (3.1), this implies \(A \in Pre\) for every \(A\) whose level in the set above, i.e. for every atom involved in the acceptability analysis.

Example 3.2.3 Consider the program \(P\) and the specification \((Pre, Post)\) of Example 3.2.2. By assuming (3.1), we have that the requirement:

\[|p| \gg |q| \gg |r|\]
cannot be satisfied, since \(|p| \in N\) whilst \(|q| = |r| = \infty\). Therefore, the triple \(\{\text{Pre}\} \ P \{\text{Post}\}\) is not in the proof relation derived from acceptability. And, actually, it was not intended to be.

On the contrary, consider the specification \((\text{Pre}', \text{Post}')\) where \(\text{Pre}' = \text{Post}' = \{p, q, r\}\). In this case, \(\text{Pre}' \rightarrow \text{Post}' = B_L\), hence it is a model of \(P\). By defining:

\[|p| = 2 \quad |q| = 1 \quad |r| = 0\]

we have that the requirement:

\[|p| \gg |q| \gg |r|\]

is satisfied, and, in addition, the extended level mapping has been specified only for interesting atoms, namely those belonging to \(\text{Pre}\). \(\Box\)

Summarizing, the general approach for systematically deriving proof relations for weak total correctness starting from the termination proof methods of Chapter 2 consists of considering:

- the set \(\text{Pre} \rightarrow \text{Post}\) as the candidate Herbrand model of the program under consideration, and
- an extended level mapping satisfying (3.1).

The properties of the resulting relations include at least the following:

- **soundness w.r.t. weak partial correctness** since \(\text{Pre} \rightarrow \text{Post}\) is required to be a model of the program under consideration;
- **soundness w.r.t. weak total correctness** since termination holds for atoms in \(\text{Pre}\);
- **persistence** since every SLD-resolvent \(Q'\) of \(P\) and \(Q\) is in the proof relation, if \(P\) and \(Q\) are;
- **patterns characterization** in the sense that only atoms in \(\text{Pre}\) are involved in the termination analysis.

Finally, we observe that the resulting proof relation is not complete w.r.t. any notion of correctness.

On the one hand, as shown in Example 3.2.1, \(\text{Pre} \rightarrow \text{Post}\) may not be a model of \(P\), in general.

On the other hand, the requirement \(\text{Pre} \subseteq M_P^L \cup FF_P^L\) of Definition 3.1.5 is equivalent to state existential termination via fair selection rules and breadth-first search strategies, while the methods of Chapter 2 address universal termination.

Once again, however, we claim that our objective is to design methods that are a trade-off between expressiveness (i.e., the class of programs and properties it is able to reason about) and ease of use in \textit{paper \& pencil} proofs.
3.3 Acceptable Programs

In this section, we apply the general strategy outlined above to acceptable programs and queries, by formally deriving a proof relation $\vdash_t$ which is sound w.r.t. weak total correctness of Prolog programs.

The definition of acceptable programs, namely Definition 2.3.7, can be concisely written in the following form.

Let $P$ be a program, $\|$ an extended level mapping, and $I$ a Herbrand interpretation. $P$ is acceptable by $\|$ and $I$ iff for every $A \leftarrow B_1, \ldots, B_n$ in $\text{ground}_t(P)$:

(a) for $i \in [1,n]$ $I \models B_1, \ldots, B_{i-1}$ implies $|A| \succ |B_i|$

(b) $I \models B_1, \ldots, B_n$ implies $I \models A$.

Assume now that $I$ is $\text{Pre} \rightarrow \text{Post}$ and that (3.1) holds. The proof obligations $(a,b)$ become:

(a) for $i \in [1,n]$ $\text{Pre} \rightarrow \text{Post} \models B_1, \ldots, B_{i-1}$ implies $|A| \succ |B_i|$

(b) $\text{Pre} \rightarrow \text{Post} \models B_1, \ldots, B_n$ implies $\text{Pre} \rightarrow \text{Post} \models A$.

We observe that, when $\text{Pre} \not\models A$ the conclusions of the implications $(a, b)$ hold. In fact, $A \notin \text{Pre}$ implies $|A| = \infty \succ |B_i|$ for any $B_i$, and $\text{Pre} \rightarrow \text{Post} \models A$. Therefore, the proof obligations are equivalent to:

(a) for $i \in [1,n]$ $\text{Pre} \models A \land \text{Pre} \rightarrow \text{Post} \models B_1, \ldots, B_{i-1}$ implies $|A| \succ |B_i|$

(b) $\text{Pre} \models A \land \text{Pre} \rightarrow \text{Post} \models B_1, \ldots, B_n$ implies $\text{Pre} \rightarrow \text{Post} \models A$.

Further, we note the following fact.

**Lemma 3.3.1** Let $(\text{Pre}, \text{Post})$ be a specification and $\|$ an extended level mapping such that (3.1) holds. Then for every ground atom $A$, if $\text{Pre} \models A$ then:

- $\text{Pre} \rightarrow \text{Post} \models A$ iff $\text{Post} \models A$, and

- $|A| \succ |B_i|$ iff $|A| > |B_i| \land \text{Pre} \models B_i$.

From this Lemma, we derive the following proof obligations:

(a) for $i \in [1,n]$ $\text{Pre} \models A \land \text{Pre} \rightarrow \text{Post} \models B_1, \ldots, B_{i-1}$ implies $|A| > |B_i| \land \text{Pre} \models B_i$,

(b) $\text{Pre} \models A \land \text{Pre} \rightarrow \text{Post} \models B_1, \ldots, B_n$ implies $\text{Post} \models A$.

We further simplify these proof obligations in order to avoid any reference to the set $\text{Pre} \rightarrow \text{Post}$, which is not basic in the notion of weak total correctness. To this
end, we replace $Pre \rightarrow Post$ by $Post$. Also, observing that the proof obligations involving the extended level mapping reason only about atoms that are in $Pre$, we can assume without lack of generality that $||$ is a level mapping. In conclusion, we are led the following definition.

**Definition 3.3.2** Consider a program $P$, and a specification $(Pre, Post)$. We write $\vdash_I \{Pre\} \ P \ \{Post\}$ iff there exists a level mapping $||$ such that for every $A \leftarrow B_1, \ldots, B_n \in \text{ground}_L(P)$:

1. for $i \in [1, n]$:
   - $Pre \models A \land Post \models B_1, \ldots, B_{i-1}$
   - (a) $Pre \models B_i$ and
   - (b) $|A| > |B_i|$

2. $Pre \models A \land Post \models B_1, \ldots, B_n \Rightarrow Post \models A$.

The next lemma shows that the proof obligations of this definition are equivalent to (3.3), hence the proof relation $\vdash_I$ is formally derived by following the general strategy of Section 3.2.

**Lemma 3.3.3** Let $P$ be a program, $(Pre, Post)$ a specification, and $||$ an extended level mapping such that (3.1) holds. Then $P$ is acceptable by $||$ and $Pre \rightarrow Post$ iff $\vdash_I \{Pre\} \ P \ \{Post\}$ holds by using a level mapping $||'$ that coincides with $||$ on $Pre$.

**Proof.** We have shown that, under the hypothesis of the Theorem, $P$ is acceptable by $||$ and $Pre \rightarrow Post$ iff for every $A \leftarrow B_1, \ldots, B_n \in \text{ground}_L(P)$, the proof obligations (3.3) hold.

**Only-if part** We define the level mapping $||'$ as follows: $|A|' = |A|$ for $A \in Pre$, and $|A|' = 0$ for $A \notin Pre$. Since (3.1) holds, $||'$ is well-defined. We claim that $\vdash_I \{Pre\} \ P \ \{Post\}$ holds by using $||'$. Consider, in fact, $A \leftarrow B_1, \ldots, B_n \in \text{ground}_L(P)$.

1. If for some $i \in [1, n]$, $Pre \models A \land Post \models B_1, \ldots, B_{i-1}$ then, since $Post \subseteq Pre \rightarrow Post$:
   
   $Pre \models A \land Pre \rightarrow Post \models B_1, \ldots, B_{i-1}$.

By (3.3 (a)), we have $Pre \models B_i \land |A|' = |A| > |B_i| = |B_i|'$, i.e. Definition 3.3.2 (a,b).

2. If $Pre \models A \land Post \models B_1, \ldots, B_n$ then, since $Post \subseteq Pre \rightarrow Post$:

   $Pre \models A \land Pre \rightarrow Post \models B_1, \ldots, B_n$.
By (3.3 (b)), we conclude \( \text{Post} \models A \).

If part) The proof of (3.3 (a)) proceeds by induction on \( n \).

Case \( n = 0 \). Immediate.

Case \( n > 0 \). By inductive hypothesis, we have:

\[
\text{for } i \in [1, n - 1] \quad \text{Pre} \models A \wedge \text{Pre} \rightarrow \text{Post} \models B_1, \ldots, B_{i-1}
\]

implies \( |A| > |B_i| \wedge \text{Pre} \models B_i \).

Therefore, we have to show that:

\[
\text{Pre} \models A \wedge \text{Pre} \rightarrow \text{Post} \models B_1, \ldots, B_{n-1}
\]

implies \( |A| > |B_{n}| \wedge \text{Pre} \models B_{n} \).

Suppose that \( \text{Pre} \models A \wedge \text{Pre} \rightarrow \text{Post} \models B_1, \ldots, B_{n-1} \). We observe that this implies the antecedent of the induction hypothesis. By the induction hypothesis, \( \text{Pre} \models B_1, \ldots, B_{n-1} \) holds. This and

\[
\text{Pre} \models A \wedge \text{Pre} \rightarrow \text{Post} \models B_1, \ldots, B_{n-1}
\]

imply

\[
\text{Pre} \models A \wedge \text{Post} \models B_1, \ldots, B_{n-1}.
\]

By Definition 3.3.2 (1), we conclude \( \text{Pre} \models B_n \wedge |A| = |A'| > |B_n'| = |B_n| \). This shows (3.3 (a)). Suppose now that

\[
\text{Pre} \models A \wedge \text{Pre} \rightarrow \text{Post} \models B_1, \ldots, B_{n}.
\]

By reasoning as above, we conclude that \( \text{Pre} \models A \wedge \text{Post} \models B_1, \ldots, B_{n} \). By Definition 3.3.2 (2), we conclude \( \text{Post} \models A \). This shows (3.3 (b)). \( \square \)

With same reasonings, we derive the proof relation \( \vdash_t \) in the case of queries.

DEFINITION 3.3.4 Consider a query \( Q \), and a specification \( \langle \text{Pre}, \text{Post} \rangle \). We write:

\[
\vdash_t \{ \text{Pre} \} \langle Q \} \{ \text{Post} \rangle \iff \text{there exist a level mapping } \| \text{ and } k \in \mathbb{N} \text{ such that for every } A_1, \ldots, A_n \in \text{ground}_t(Q) :
\]

for \( i \in [1, n] \quad \text{Post} \models A_1, \ldots, A_{i-1} \Rightarrow \text{Pre} \models A_i \wedge k > |A_i| \).

\( \square \)

As a result of our general approach, we are in the position to state that the proof relation \( \vdash_t \) is sound w.r.t. weak total correctness, and that it is persistent along SLD-derivations.

Consider now the property of patterns characterization. Let \( Q \) be a query such that \( \vdash_t \{ \text{Pre} \} \langle Q \} \{ \text{Post} \rangle \) holds. By assuming (3.1), for every atom \( A' \) in a ground instance of \( Q \) such that \( |A'| \) is in the set

\[
\bigcup_{i=1}^{n} |Q|^\text{Pre} \rightarrow \text{Post}_i
\]
we have \( A' \in \text{Pre} \). In particular, consider the leftmost atom \( A \) of \( Q \). We observe that \(|A'|\) belongs to the set above for every ground instance \( A' \) of \( A \). Therefore \( \text{Pre} \models A \).

Summarizing, \( \text{Pre} \) characterizes the call patterns w.r.t. the leftmost selection rule of queries \( Q \) such that \( \text{P} \{ \text{Pre} \} \{ \text{Post} \} \) holds. In Chapter 4, we will show also that \( \text{Post} \) characterizes the success patterns.

### 3.4 Fair-bounded Programs

Let us apply the general strategy of Section 3.2 to fair-bounded programs and queries. The definition of fair-bounded programs can be concisely written in the following form.

Let \( P \) be a program, \( \mid \mid \) an extended level mapping, and \( I \) a Herbrand interpretation. \( P \) is *fair-bounded by \( \mid \mid \) and \( I \) if and only if for every \( A \leftarrow B_1, \ldots, B_n \) in \( \text{ground}_L(P) \):

\[
\begin{align*}
(a) \quad & I \models B_1, \ldots, B_n & & \text{implies} & & \text{for } i \in [1,n] \quad |A| \triangleright |B_i| \land I \models A \\
(b) \quad & I \not\models B_1, \ldots, B_n & & \text{implies} & & \text{there exists } i \in [1,n] \quad |A| \triangleright |B_i| \land I \not\models B_i.
\end{align*}
\]

Assume now that \( I \) is \( \text{Pre} \rightarrow \text{Post} \) and that (3.1) holds. The proof obligations \((a,b)\) become:

\[
\begin{align*}
(a) \quad & \text{Pre} \rightarrow \text{Post} \models B_1, \ldots, B_n & & \text{implies} & & \text{for } i \in [1,n] \quad |A| \triangleright |B_i| \land \text{Pre} \rightarrow \text{Post} \models A \\
(b) \quad & \text{Pre} \rightarrow \text{Post} \not\models B_1, \ldots, B_n & & \text{implies} & & \text{there exists } i \in [1,n] \quad |A| \triangleright |B_i| \land \text{Pre} \rightarrow \text{Post} \not\models B_i.
\end{align*}
\]

As for acceptable programs, in the case \( \text{Pre} \not\models A \), the conclusions of the implications hold. Therefore, the proof obligations are equivalent to:

\[
\begin{align*}
& \text{Pre} \models A \land \text{Pre} \rightarrow \text{Post} \models B_1, \ldots, B_n & & \text{implies} & & \text{for } i \in [1,n] \quad |A| \triangleright |B_i| \land \text{Pre} \rightarrow \text{Post} \models A \\
& \text{Pre} \models A \land \text{Pre} \rightarrow \text{Post} \not\models B_1, \ldots, B_n & & \text{implies} & & \text{there exists } i \in [1,n] \quad |A| \triangleright |B_i| \land \text{Pre} \rightarrow \text{Post} \not\models B_i.
\end{align*}
\]

Moreover, Lemma 3.3.1 allows us to write the proof obligations in the following form:

\[
\begin{align*}
(a) \quad & \text{Pre} \models A \land \text{Pre} \rightarrow \text{Post} \models B_1, \ldots, B_n & & \text{implies} & & \text{for } i \in [1,n] \quad |A| > |B_i| \land \text{Pre} \models B_i \land \text{Post} \models A \\
(b) \quad & \text{Pre} \models A \land \text{Pre} \rightarrow \text{Post} \not\models B_1, \ldots, B_n & & \text{implies} & & \text{there exists } i \in [1,n] \quad |A| > |B_i| \land \text{Pre} \models B_i \land \text{Post} \not\models B_i.
\end{align*}
\]

We need to further simplify these proof obligations in order to avoid the reference to \( \text{Pre} \rightarrow \text{Post} \), which is not basic in the notion of weak total correctness. To this
end, we distinguish two cases when proving (3.4 (a,b)), depending whether or not $Post \models B_1, \ldots, B_n$.

**Assume** $Pre \models A \land Post \models B_1, \ldots, B_n$. Then the hypothesis in (3.4 (a)) holds. Thus, we have to show the conclusion, i.e.:

$$\text{for } i \in [1, n] \quad |A| > |B_i| \land Pre \models B_i \land Post \models A.$$

Concerning (3.4 (b)), we observe that its hypothesis is not satisfied, thus we have no proof obligation.

**Assume**, on the contrary, $Pre \models A \land Post \not\models B_1, \ldots, B_n$. We distinguish two further cases.

If $Pre \rightarrow Post \models B_1, \ldots, B_n$, then we have that for some $i \in [1, n]$, $Post \not\models B_i \land Pre \rightarrow Post \models B_i$, which implies $Pre \not\models B_i$. Thus, the hypothesis of (3.4 (a)) holds while its conclusion does not. This implies that (3.4 (a,b)) do not hold.

If $Pre \rightarrow Post \not\models B_1, \ldots, B_n$, then the hypothesis of (3.4 (a)) is not satisfied, thus we have no proof obligation from it. Concerning (3.4 (b)), we observe that its hypothesis is satisfied. Thus we have to show its conclusion, namely:

$$\text{there exists } i \in [1, n] \text{ s.t. } |A| > |B_i| \land Pre \models B_i \land Post \not\models B_i. \quad (3.5)$$

Summarizing, under the hypothesis $Pre \models A \land Post \not\models B_1, \ldots, B_n$, (3.4 (a,b)) hold iff $Pre \rightarrow Post \not\models B_1, \ldots, B_n$ and (3.5) hold, i.e. iff (3.5) holds.

We report below the relation $\vdash^f$ obtained by combining the inductive method and fair-boundedness. The definition of $\vdash^f$ for queries is derived analogously to the derivation of the relation for programs. As in the case of the proof relation $\vdash$, we consider level mappings instead of extended level mappings, without lack of generality.

**Definition 3.4.1** Consider a program $P$, a query $Q$, and a specification $(Pre, Post)$. We write:

$$\vdash^f \{Pre\} P \{Post\} \text{ iff there exists a level mapping } | \mid \text{ such that for every } A \leftarrow B_1, \ldots, B_n \in \text{ground}_k(P):$$

1. $Pre \models A \land Post \models B_1, \ldots, B_n \Rightarrow$
   
   (a) for $i \in [1, n]$, $Pre \models B_i \land |A| > |B_i|$, and
   
   (b) $Post \models A$.

2. $Pre \models A \land Post \not\models B_1, \ldots, B_n \Rightarrow$
   
   \[ \exists i \in [1, n] \text{ s.t. } Pre \models B_i \land Post \not\models B_i \land |A| > |B_i|. \]

$$\vdash^f \{Pre\} Q \{Post\} \text{ iff there exists } k \in N \text{ and a level mapping } | \mid \text{ such that for every } B_1, \ldots, B_n \in \text{ground}_k(Q):$$

3. $Post \models B_1, \ldots, B_n \Rightarrow$
for $i \in [1, n]$, $\text{Pre} \models B_i \land k > |B_i|$.

\[ \exists i \in [1, n] \text{ s.t. } \text{Pre} \models B_i \land \text{Post} \not\models B_i \land k > |B_i|. \]

The proof relation $\vdash^f$ is sound w.r.t. weak total correctness. Also, it is persistent along SLD-derivations. Finally, let us consider the property of patterns characterization. Let $Q$ be a query such that $\vdash^f \{\text{Pre}\} Q \{\text{Post}\}$ holds. Symmetrically to the case of acceptability, the set

\[ \bigcup_{i=1}^n |Q|^{\text{Pre} \rightarrow \text{Post}} \]

(see Definition 2.4.12) defines the levels of the ground atoms involved in the fair-boundedness analysis.

Due to the assumption of (3.1), we have that for every atom $A$ in a ground instance of $Q$ such that $|A|$ belongs to (3.6), $A$ is included in $\text{Pre}$.

In particular, consider a correct instance $Q'$ of $Q$. Since $\text{Pre} \rightarrow \text{Post}$ is a model of $P$, then $\text{Pre} \rightarrow \text{Post} \models Q'$. Therefore, every atom in a ground instance of $Q'$ is such that its level is in (3.6). Hence, hence $\text{Pre} \models Q'$. This and $\text{Pre} \rightarrow \text{Post} \models Q'$ imply $\text{Post} \models Q'$.

Summarizing, $\text{Post}$ characterizes the success patterns of queries $Q$ such that $\vdash^f \{\text{Pre}\} Q \{\text{Post}\}$ holds.

**Example 3.4.2 (ProdCons)** Reconsider the PRODCONS program of Example 2.4.8 at page 32.

\[ (s) \quad \text{system}(N) \leftarrow \]
\[ \quad \text{prod}(\text{Bs}), \text{cons}(\text{Bs}, N). \]

\[ (p1) \quad \text{prod}(\text{[s(0) | Bs]})) \leftarrow \]
\[ \quad \text{prod}(\text{Bs}). \]

\[ (p2) \quad \text{prod}(\text{[s(s(0)) | Bs]})) \leftarrow \]
\[ \quad \text{prod}(\text{Bs}). \]
\[ \quad \text{prod}([]). \]

\[ (c) \quad \text{cons}([D | Bs], s(N)) \leftarrow \]
\[ \quad \text{cons}(\text{Bs}, N), \text{wait}(D). \]
\[ \quad \text{cons}([], 0). \]
\[ \quad \text{wait}(0). \]

\[ (w) \quad \text{wait}(s(D)) \leftarrow \]
\[ \quad \text{wait}(D). \]

We have observed that termination of PRODCONS and a query system(n) holds via fair-selection rules. Here, we show that

\[ \vdash^f \{\text{Pre}\} \text{PRODCONS} \{\text{Post}\} \quad \vdash^f \{\text{Pre}\} \text{system}(n) \{\text{Post}\} \]
hold by the same level mapping \(||\). We define:

\[
Pre = \{ \text{system}(n) \mid n \in N \}
\]
\[
\cup \{ \text{prod}(bs) \mid bs \text{ list of 1’s and 2’s} \}
\]
\[
\cup \{ \text{cons}(bs, n) \mid n \in N \}
\]
\[
\cup \{ \text{wait}(n) \mid n \in N \}
\]

\[
Post = \{ \text{system}(n) \mid n \in N \}
\]
\[
\cup \{ \text{prod}(bs) \mid bs \in GList, n \in N, |bs| = size(n) \}
\]
\[
\cup \{ \text{wait}(n) \mid n \in N \}
\]

\[
|\text{system}(n)| = size(n) + 3
\]

\[
|\text{prod}(bs)| = |bs|
\]

\[
|\text{cons}(bs, n)| = \begin{cases} 
  size(n) + \text{lmax}(bs) & \text{if } \text{cons}(bs, n) \in Post \\
  size(n) & \text{if } \text{cons}(bs, n) \notin Post
\end{cases}
\]

\[
|\text{wait}(t)| = size(t).
\]

The proof obligations for the clauses defining \text{prod} and \text{wait}, and for the unit clause of \text{cons} are readily checked. Consider now clauses (c) and (s).

(c) Consider a ground instance

\[
\text{cons}([d \mid bs], s(n)) \leftarrow \text{cons}(bs, n), \text{wait}(d)
\]

of (c).

(1) Suppose that \(Pre \models \text{cons}([d \mid bs], s(n))\), and that:

\[
Post \models \text{cons}(bs, n), \text{wait}(d).
\]

Then, |bs| = size(n) implies |[d \mid bs]| = size(s(n)), hence Post \(\models \text{cons}([d \mid bs], s(n))\). Also, it is readily checked that \(Pre \models \text{cons}(bs, n), \text{wait}(d)\). Finally, we calculate:

\[
|\text{cons}([d \mid bs], s(n))| = size(n) + 1 + \text{max}\{\text{lmax}(bs), size(d)\}
\]
\[
> size(n) + \text{max}(\text{lmax}(bs))
\]
\[
= |\text{cons}(bs, n)|.
\]

and

\[
|\text{cons}([d \mid bs], s(n))| = size(n) + 1 + \text{max}\{\text{lmax}(bs), size(d)\}
\]
\[
> size(d)
\]
\[
= |\text{wait}(d)|.
\]

These two inequalities complete the proof that (1) holds.
Suppose that \( \text{Pre} \models \text{cons}( [d \mid bs], s(n)) \), and that \( \text{Post} \not\models \text{cons}(bs, n), \text{wait}(d) \). Then, it necessarily happens that \( \text{Post} \not\models \text{cons}(bs, n) \). Also, we observe that \( \text{Pre} \models \text{cons}(bs, n) \). To conclude the proof of (2), we calculate:

\[
|\text{cons}( [d \mid bs], s(n))| \\
\geq \text{size}(n) + 1 \\
> \text{size}(n) \\
= \{ \text{Post} \not\models \text{cons}(bs, n) \} \\
|\text{cons}(bs, n)|.
\]

(1) Consider a ground instance

\[
\text{system}(n) \leftarrow \text{prod}(bs), \text{cons}(bs, n)
\]

of (s).

(2) Suppose that \( \text{Pre} \models \text{system}(n) \) and that \( \text{Post} \not\models \text{prod}(bs), \text{cons}(bs, n) \). Then \( bs \) is a list of 1’s and 2’s. This implies \( \text{lmax}(bs) \leq 2 \). Moreover, we have that \( |bs| = \text{size}(n) \).

Obviously, \( \text{Post} \not\models \text{system}(n) \). Also, it is readily checked that \( \text{Pre} \models \text{prod}(bs), \text{cons}(bs, n) \). Finally, we calculate:

\[
|\text{system}(n)| = \text{size}(n) + 3 \\
> \{ |bs| = \text{size}(n) \} \\
|bs| = |\text{prod}(bs)|
\]

and

\[
|\text{system}(n)| = \text{size}(n) + 3 \\
> \{ \text{lmax}(bs) \leq 2 \} \\
\text{size}(n) + \text{lmax}(bs) \\
= |\text{cons}(bs, n)|.
\]

These two inequalities complete the proof of (1).

(2) Suppose now that \( \text{Pre} \models \text{system}(n) \) and that \( \text{Post} \not\models \text{prod}(bs), \text{cons}(bs, n) \). We distinguish two cases.

If \( \text{Post} \not\models \text{cons}(bs, n) \), we observe that, in any case, \( \text{Pre} \models \text{cons}(bs, n) \) and that:

\[
|\text{system}(n)| = \text{size}(n) + 3 > \text{size}(n) = |\text{cons}(bs, n)|.
\]

Hence (2) holds.
Assume now that $Post \models \text{cons}(bs, n)$ and $Post \not\models \text{prod}(bs)$. In any case, $Pre \models \text{prod}(bs)$. Moreover, we calculate:

$$|\text{system}(n)| = size(n) + 3$$
$$> \begin{cases} 
Post \models \text{cons}(bs, n) \implies |bs| = size(n) \\
|bs| 
\end{cases}$$
$$= |\text{prod}(bs)|.$$ 

We conclude by noting that for every $n \in N$:

$$\{Pre\} \text{system}(n) \{Post\}\{ \}$$

holds by the same level mapping $| \ |$.

Summarizing, PRODCONS is weak totally correct w.r.t. $(Pre, Post)$. Moreover, every SLD-derivation via a fair-selection rule for the program and the query $\text{system}(n)$, with $n \in N$, is finite. $\square$

### 3.5 Bounded Programs

Definition 2.5.5 of bounded programs is a simplification of the definition of fair-boundedness. Therefore, by following the same reasonings of the previous section, we are led to the following proof relation.

**Definition 3.5.1** Consider a program $P$, a query $Q$, and a specification $(Pre, Post)$. We write:

$$\vdash^b \{Pre\} P \{Post\}$$

iff there exists a level mapping $| |$ such that for every $A \leftarrow B_1, \ldots, B_n \in \text{ground}_k(P)$:

1. $Pre \models A \land Post \models B_1, \ldots, B_n$ \Rightarrow
   - for $i \in [1,n]$, $Pre \models B_i \land |A| > |B_i|$, and
   - $Post \models A$
2. $Pre \models A \land Post \not\models B_1, \ldots, B_n$ \Rightarrow
   - $\exists i \in [1,n]$ s.t. $Pre \models B_i \land Post \not\models B_i$.

- $\vdash^b \{Pre\} Q \{Post\}$

iff there exists $k \in N$ and a level mapping $| |$ such that for every $B_1, \ldots, B_n \in \text{ground}_k(Q)$:

3. $Post \models B_1, \ldots, B_n$ \Rightarrow
   - for $i \in [1,n]$, $Pre \models B_i \land k > |B_i|$, and
4. $Post \not\models B_1, \ldots, B_n$ \Rightarrow
   - $\exists i \in [1,n]$ s.t. $Pre \models B_i \land Post \not\models B_i$. $\square$
3.5. Bounded Programs

The proof relation $\vdash^b$ is not sound w.r.t. weak total correctness, since boundedness is not a termination proof method. The following restricted result can be given instead.

**Theorem 3.5.2** Assume that $\vdash^b \{\text{Pre}\} P \{\text{Post}\}$ and $\vdash^b \{\text{Pre}\} Q \{\text{Post}\}$ hold by the same level mapping $| |$.

Let the function $\text{Ter}$ be as in Definition 2.5.20. Then:

(i) $P$ is weak partially correct w.r.t. $(\text{Pre}, \text{Post})$.

(ii) Called $k$ a natural satisfying Definition 3.5.1 (3,4), $\text{Ter}(P)$ and the query $\text{Ter}(Q, k - 1)$ universally terminate w.r.t. all selection rules. Moreover, there is a bijection between SLD-refutations of $P$ and $Q$ via a selection rule $s$ and SLD-refutations of $\text{Ter}(P)$ and $\text{Ter}(Q, k - 1)$ via $s$.

**Proof.** (i) is immediate by the fact that $\text{Pre} \rightarrow \text{Post}$ is a model of $P$. (ii) follows by Theorem 2.5.21 by noting that $\vdash^b$ is formally derived from boundedness. □

It is worth noting that in the case that the transformation $\text{Ter}$ is hidden to the programmer, i.e. it is transparently implemented by the underlying system, the programmer observes that $P$ is weak totally correct w.r.t the specification $(\text{Pre}, \text{Post})$. Thus

Finally, we point out that relation $\vdash^b$ is persistent, since boundedness is. Moreover, by reasoning as in the case of fair-bounded programs, it is readily checked that $\text{Post}$ characterizes the success patterns of queries $Q$ such that $\vdash^b \{\text{Pre}\} Q \{\text{Post}\}$ holds.

**Example 3.5.3 (Permutation)** Reconsider the PERMUTATION program:

\[
\text{perm}(Xs, Ys) \leftarrow Ys \text{ is a permutation of the list } Xs.
\]

\[
\text{perm}([], []).\]

\[(p) \quad \text{perm}([X|Xs], Ys) \leftarrow
\]

\[
\quad \text{delete}(X, Ys, Zs),
\]

\[
\quad \text{perm}(Xs, Zs).
\]

\[
\quad \text{delete}(X, [X|Y], Y).
\]

\[
\text{(d) } \quad \text{delete}(X, [H|Y], [H|Z]) \leftarrow
\]

\[
\quad \text{delete}(X, Y, Z).
\]

and the query $\text{perm([a, b], Ys)}$. We show that:

\[
\vdash^b \{\text{Pre}\} \text{PERMUTATION} \{\text{Post}\} \quad \vdash^b \{\text{Pre}\} \text{perm([a, b], Ys)} \{\text{Post}\}
\]

hold by the same level mapping $| |$. We define:

\[
\text{Pre} = \{ \text{perm}(xs, ys) \mid xs \in GList \}
\]

\[
\cup \{ \text{delete}(x, y, z) \mid z \in GList \}
\]

\[
\text{Post} = \{ \text{perm}(xs, ys) \mid ys \text{ is a permutation of } xs \}
\]

\[
\cup \{ \text{delete}(x, y, z) \mid z \text{ is obtained by removing } x \text{ from } y \}
\]
\[ \text{perm}(xs, ys) = |xs| \]
\[ \text{delete}(x, y, z) = |z|. \]

The proof obligations for the unit clauses of \textsc{Permutation} are trivial. Consider, instead, clauses \((p)\) and \((d)\).

\((d)\) Consider a ground instance

\[ \text{delete}(x, [h \mid y], [h \mid z]) \leftarrow \text{delete}(x, y, z) \]

of \((d)\).

(1) Suppose that \(\text{Pre} \models \text{delete}(x, [h \mid y], [h \mid z])\) and \(\text{Post} \models \text{delete}(x, y, z)\). From \([h \mid z] \in GList\), we have \(z \in GList\), and then \(\text{Pre} \models \text{delete}(x, y, z)\). Moreover, we calculate:

\[
|\text{delete}(x, [h \mid y], [h \mid z])| = |z| + 1 \\
> |z| = |\text{delete}(x, y, z)|.
\]

Finally, we observe that if \(z\) is obtained by removing \(x\) from \(y\), then \([h \mid z]\) is obtained by removing \(x\) from \([h \mid y]\), i.e. \(\text{Post} \models \text{delete}(x, [h \mid y], [h \mid z])\). This concludes the proof of (1).

(2) Suppose that \(\text{Pre} \models \text{delete}(x, [h \mid y], [h \mid z])\) and that \(\text{Post} \not\models \text{delete}(x, y, z)\).

From \([h \mid z] \in GList\), we have \(z \in GList\), and then \(\text{Pre} \models \text{delete}(x, y, z)\). Therefore, (2) holds.

\((p)\) Consider a ground instance

\[ \text{perm}([x \mid xs], ys) \leftarrow \text{delete}(x, ys, zs), \text{perm}(xs, zs) \]

of \((p)\).

(1) Suppose that \(\text{Pre} \models \text{perm}([x \mid xs], ys)\) and \(\text{Post} \models \text{delete}(x, ys, zs), \text{perm}(xs, zs)\). From \([x \mid xs] \in GList\), we have \(xs \in GList\), hence \(\text{Pre} \models \text{perm}(xs, zs)\). This and the fact that \(zs\) is a permutation of \(xs\) imply that \(zs \in GList\), hence \(\text{Pre} \models \text{delete}(x, y, z)\). Moreover, we calculate:

\[
|\text{perm}([x \mid xs], ys)| \\
= |xs| + 1 \\
> |xs| \\
= |\text{perm}(xs, zs)|
\]
and
\[
|\text{perm}([x \mid xs], ys)| = |zs| + 1 \\
> |zs| \\
= \{ xs \text{ is a permutation of } zs \Rightarrow |xs| = |zs| \} \\
|zs| \\
= |\text{delete}(x, ys, zs)|.
\]

Finally, we observe that if \( xs \) is a permutation of \( zs \), and \( zs \) is obtained by removing \( x \) from \( ys \), then \( [x \mid xs] \) is a permutation of \( ys \), i.e. \( \text{Post} \models \text{perm}([x \mid xs], ys) \).

(2) Suppose that \( \text{Pre} \models \text{perm}([x \mid xs], ys) \) and that
\[
\text{Post} \nvdash \text{delete}(x, ys, zs), \text{perm}(xs, zs).
\]

We distinguish two cases.

If \( \text{Post} \not\models \text{perm}(xs, zs) \) then (2) follows by noting that, in any case, from \( [x \mid xs] \in \text{GList} \), we have \( xs \in \text{GList} \), hence \( \text{Pre} \models \text{perm}(xs, zs) \).

If \( \text{Post} \not\models \text{delete}(x, ys, zs) \) and \( \text{Post} \models \text{perm}(xs, zs) \), then the fact that \( zs \) is a permutation of \( xs \) implies that \( zs \in \text{GList} \), hence \( \text{Pre} \models \text{delete}(x, ys, zs) \). Thus, (2) holds.

We conclude by noting that \( \vdash^k \{ \text{Pre} \} \text{perm}([a,b], Ys) \{ \text{Post} \} \) holds by the same level mapping \( \dagger \) by fixing \( k = 3 \) in Definition 3.5.1.

Summarizing, from Theorem 3.5.2, we have that \textsc{Permutation} is weak partially correct w.r.t. \( (\text{Pre}, \text{Post}) \).

Moreover, \( \text{Ter}(\textsc{Permutation}) \) and the query \( \text{Ter}(\text{perm}([a,b], Ys), 2) \) universally terminate w.r.t. all selection rules, and there is a bijection between their SLD-refutations and the SLD-refutation of \( P \) and \( \text{perm}([a,b], Ys) \).

\[\square\]

### 3.6 On Proof Methods and Proof Relations

In this Chapter, we have derived proof relations that address weak total correctness of logic programs. The proof relations are defined in terms of a number of proof obligations that must be satisfied for every ground instance of program clauses. These proof obligations define the proof method that allows us to prove weak total correctness, i.e. a proof method consists of proving the validity of a collection of proof obligations.

When specifications are pairs of finite sets, provability of proof obligations can be established by enumeration methods, at least while the computational cost is affordable. In general, however, provability of proof obligations is established within a formal system (e.g., an axiomatization of first order logic). Specifications are expressed
intensionally as a family of formulas in the formal system, and proof obligations assume the form of a finite set of formulas.

In this thesis, we abstract away from the formal system in which the validity of proof obligations has to be proved. Instead, we concentrate on the core of proof methods, namely proof relations (with an abuse of notation, we confuse proof methods and proof relations).

On the one hand, the study of proof relations is not bound to the choice of a particular formal system, provided that it includes arithmetic (which is required in proving decreasing of level mappings), and the first order logic connectives and quantifiers. We observe that the choice affects, instead, implementations issues.

On the other hand, the natural choice of first order logic would lead us to a well-known form of incompleteness, namely the fact that the well-typed fragment of the least Herbrand model cannot be always expressed as a first order formula (see Deransart [64]).

It is worth noting that similar arguments apply to Hoare’s logic. Even though pre- and postconditions are assumed to be first order logic formulas, we observe that axioms and inference rules of Hoare’s logic could be stated in terms of any formal system (that includes arithmetic and first order connectives and quantifiers).

On the one hand, the choice does not affect the core of Hoare’s logic, which is a relation over triples \( \{\phi\} P \{\psi\} \), where \( \phi \) and \( \psi \) are sets of program states. Such a proof relation was defined by Dijkstra [72], which called \( \phi \) the weakest precondition of \( P \) and \( \psi \).

On the other hand, completeness of Hoare’s logic proof method holds only for programs such that first order logic is sufficiently expressive to define weakest preconditions (see Apt and Olderog [14]). On the contrary, completeness of the theory of the weakest preconditions holds in general terms.

### 3.7 Related Work

#### Semantics and Correctness

Many declarative semantics have been proposed for logic programs. We refer the reader to Maher [110] and Apt et al. [11] for an introduction and a comparison of several of them.

We observe here that the notions of specification and correctness can be given in a general framework. Consider, in fact, a semantics \( \mathcal{F} \) as a function from programs into a set \( \mathcal{D} \). A specification is then an element \( \mathcal{I} \in \mathcal{D} \). In the case that \( \mathcal{D} \) is a set of sets, \( \mathcal{I} \) may take the form \( \text{pre} \rightarrow \text{post} \), where \( \text{pre} \) specifies the domain of interest and \( \text{post} \) specifies a property of elements in \( \mathcal{F}(P) \). We say that \( P \) is weak partially correct w.r.t \( \mathcal{I} \) if

\[
\mathcal{F}(P) \subseteq \mathcal{I}.
\]
In the case that \( \mathcal{D} \) is a set of sets and \( \mathcal{I} = \text{pre} \rightarrow \text{post} \), the inclusion above can be rewritten as \( \mathcal{F}(P) \cap \text{pre} \subseteq \text{post} \).

Notice that in the literature, weak partial correctness is usually called \emph{partial correctness}. We adopt a different terminology in order to take into account the converse inclusion, which is rarely addressed.

In the case of \( \mathcal{C} \) and \( \mathcal{S} \)-semantics, a specification \((\text{pre}, \text{post})\) is a pair of sets of (not necessarily ground) atoms. Weak partial correctness w.r.t. the \( \mathcal{C} \)-semantics was called \emph{success correctness} by Boye and Maluszyński [34], and \emph{validity} by Deransart [64]. Note that if \( P \) is weak partially correct w.r.t. the \( \mathcal{C} \)-semantics and the specification \((\text{pre}, \text{post})\) then for an atomic query \( Q \in \text{pre} \), \( \text{post} \) characterizes the success patterns of \( P \) and \( Q \).

Boye and Maluszyński [34] also introduced the \emph{input-output} \( \mathcal{IO} \)-semantics. \( \mathcal{IO}(P) \) is a function mapping a set of atoms \( \mathcal{I} \) into the set of computed instances of \( P \) and atoms in \( \mathcal{I} \). By Theorem 3.1.2,

\[
\mathcal{IO}(P)(\mathcal{I}) = \{ \text{mg}I(A, \mathcal{S}(P)) \mid A \in \mathcal{I} \}.
\]

A specification is a pair of sets of atoms \((\text{pre}, \text{post})\). \( P \) is weak partially correct w.r.t \( \mathcal{IO} \)-semantics if

\[
\mathcal{IO}(P)(\text{pre}) \subseteq \text{post}.
\]

Boye and Maluszyński showed that given a specification \((\text{pre}, \text{post})\) closed under instantiation, weak partial correctness w.r.t. the \( \mathcal{C} \)-semantics and w.r.t. the \( \mathcal{IO} \)-semantics coincide.

Here, we relate weak partial correctness w.r.t. the \( \mathcal{C} \)-semantics with weak partial correctness w.r.t. the \( \mathcal{M} \)-semantics.

**Definition 3.7.1** Let \((\text{pre}, \text{post})\) be a specification w.r.t. the \( \mathcal{C} \)-semantics.

We say that \((\text{pre}, \text{post})\) is \emph{strongly closed under instantiation} if an atom \( A \) is in \( \text{pre} \) (resp., \( \text{post} \)) iff every ground instance of \( A \) is in \( \text{pre} \) (resp., \( \text{post} \)).

If we restrict to consider specifications that are strongly closed under instantiation, correctness w.r.t. the \( \mathcal{C} \) and \( \mathcal{M} \)-semantics coincide.

**Lemma 3.7.2** Let \( P \) be a logic program and \((\text{pre}, \text{post})\) be a strongly closed under instantiation specification w.r.t. the \( \mathcal{C} \)-semantics.

Then \( P \) is weak partially correct w.r.t. the \( \mathcal{C} \)-semantics and the specification \((\text{pre}, \text{post})\) iff it is weak partially correct w.r.t. the \( \mathcal{M} \)-semantics and the specification \((\text{pre} \cap B_L, \text{post} \cap B_L)\).

**Proof.** The only-if part is immediate. Consider the if part. Let \( A \) be in \( \mathcal{C}(P) \cap \text{pre} \). Then, every ground instance \( A' \) of \( A \) is in \( \mathcal{M}(P) \cap \text{pre} \cap B_L \). Since \( P \) is weak partially correct w.r.t \( \mathcal{M} \)-semantics, \( A' \) is in \( \text{post} \cap B_L \). Since \((\text{pre}, \text{post})\) is strongly closed under instantiation, \( A \) is in \( \text{post} \), i.e. \( \mathcal{C}(P) \cap \text{pre} \subseteq \text{post} \). \( \square \)
Call Correctness

The property of call patterns characterization can be seen as semantics correctness, by defining the call patterns semantics:

\[ \mathcal{CP}(P)(\mathcal{I}) = \{ B \mid B \text{ is selected in a LD-derivation of } P \text{ and } A, \text{ with } A \in \mathcal{I} \} \]

and the notion of call correctness w.r.t. a set of atoms \( \text{pre} \):

\[ \mathcal{CP}(P)(\text{pre}) \subseteq \text{pre}. \]

Therefore, if \( P \) is call correct w.r.t \( \text{pre} \), then \( \text{pre} \) characterizes the call patterns of \( P \) and any atomic query \( Q \in \text{pre} \).

We observe that a proof method for call correctness can be directly adapted to prove weak partial correctness w.r.t. the C-semantics.

**Example 3.7.3 (From Call Correctness to Weak Partial Correctness)**

Let \( P \) be a program and \( (\text{pre}, \text{post}) \) a specification. For every predicate \( p \) (for simplicity, assume it is unary) in \( P \) add to \( P \) the clause:

\[ \text{pre}_p(X) \leftarrow p(X), \text{post}_p(X). \]

where \( \text{pre}_p \) and \( \text{post}_p \) are fresh predicates. Let \( P' \) be the resulting program. Consider now:

\[ \text{pre}' = \{ \text{pre}_p(T) \mid p(T) \in \text{pre} \} \]

\[ \cup \{ \text{post}_p(T) \mid p(T) \in \text{post} \} \]

\[ \cup \text{pre}. \]

Assume that the proof method is able to show that \( P' \) is call correct w.r.t. \( \text{pre}' \). Consider now \( p(T) \in \mathcal{C}(P) \cap \text{pre} \). Then \( \text{pre}_p(T) \) is in \( \text{pre}' \), and \( p(T) \) has a LD-refutation. This implies that \( \text{post}_p(T) \) is selected in a LD-derivation of \( P' \) and \( \text{pre}_p(T) \). By call correctness and the definition of \( \text{pre}' \), we conclude \( p(T) \) is in \( \text{post} \).

Therefore, the proof method for call correctness (of \( P' \)) is a sufficient method for weak partial correctness (of \( P \)) w.r.t. the C-semantics. \( \square \)

As noted by Boye and Maluszynski [33, 34], the converse does not hold, i.e. proof methods for weak partial correctness do not necessarily address call correctness. They pointed out that directional types and, more generally, correctness can be viewed under two different aspects, depending whether one is interested in declarative properties (i.e., weak partial correctness), or in the run-time behavior (i.e., call correctness) of programs.
Proof Methods

Formal methods for reasoning about logic programs have been studied for a long time.


Apt’s book [10] presents several results on verification of Prolog programs.

Jacquet’s book [93] collects contributions on program synthesis, derivation, and analysis.

Finally, we cite a forthcoming special issue of the Journal of Logic Programming on analysis, synthesis and transformation of logic programs edited by Bossi and Deville [30].

It is apparent in the above references that the state-of-the-art in this area is that of a wide collection of separated methods and techniques, whose common issues are not properly recognized, and synthesized in a few unifying principles. Ours is an attempt towards this direction.

Below we recall some verification methods proposed in the literature. In the related works of the next Chapter, the expressiveness of the proof method based on \( \leftrightarrow_r \) relation is compared with some of them.

Weak Partial Correctness

The inductive method can be traced back to Clark [42], where it was presented in an axiomatic context. Clark remarked that the “pure” version of the method, i.e. the proof obligation \( TP(I) \subseteq I \), is a sound rule to prove correctness w.r.t. the \( \mathcal{M} \)-semantics. He called this rule the consequence verification method.

Deransart [64] proposed a generalization the inductive method (w.r.t. the \( \mathcal{C} \)-semantics) to any kind of domain of interpretation (vs Herbrand or term interpretations only). Also, the method was refined taking into account pre- and postconditions in order to facilitate correctness proofs. A partial order on the atoms in the body of clauses has to be chosen depending on a given program. The validity of proof obligations associated to the partial order implies correctness of the program. The refined method, called the annotation method, was shown to be complete in the following sense: if \( P \) is weak partially correct w.r.t. \( (pre, post) \) then there exist \( post' \supseteq pre \) and \( post' \subseteq post \) such that the annotation method can show correctness of \( P \) w.r.t. \( (pre', post') \).

Another early work is due to Hogger [90], which observed as correctness of a program \( P \) w.r.t. an axiomatized specification \( S \) can be shown by proving that \( P \) is a logical
consequence of $S$, i.e. $S \models P$.

Apt and Marchiori [12] reviewed and compared several methods, by pointing out that many proposals in the literature adopt a Hoare’s logic proof style [72], where specifications are given in terms of pre- and postconditions.

Among the others, the method of Bossi and Cocco [28] is a trade-off between expressiveness and ease of use, being able to reason on declarative and run-time properties of Prolog programs. By allowing only specifications closed under instantiation, they strictly extend the methods of:

- well-typed programs by Bronsard et al. [37], where directional types are considered to model the input/output behavior of programs,
- well-moded programs by Dembinski and Maluszynski [63].

On the other hand, the method of Bossi and Cocco is a special case of:

- the inductive assertion method of Drabent and Maluszynski [73], which allows for the use of specifications not closed under instantiation,
- the method of Colussi and Marchiori [47], where assertions are associated to control points, rather than to the relations defined in programs,
- the annotation method of Deransart [64], who points out that the method of Bossi and Cocco is obtained by considering the partial order in the annotation method as the (Prolog) left-to-right (total) order.

Other recent approaches investigate extensions of pure logic programming including:

- declarative extensions of first-order built-in’s of Prolog (Apt et al. [13]),
- methods to prove correctness with respect to the C- and S-semantics (Apt et al. [11]),
- general logic program, (Ferrand and Deransart [80], Malfon [111]),
- constraint logic programs (Mesnard [118]),
- concurrent constraint logic programs (de Boer et al. [55]),
- meta-programming (Pedreschi and Ruggieri [131]), and
- dynamic scheduling systems (Apt and Luitjes [6], de Boer et al. [56]).

Naish [124] investigated the parallel between verification of logic and imperative programs. He observed that the inductive method for logic programs yields the same proof obligations of Hoare’s logic when considering logic programming versions of imperative programs.

Naish also [125] discussed the notion of types as supersets of the least Herbrand model, by arguing that purely declarative information can actually express the essence of types and modes. He proposed [123, 125] a definition of declarative typing of programs and applied it to program verification and type checking.
Call Patterns

Apart from the annotation method, all the other proposals deal with run-time properties, and in particular with call patterns. As noted by Boye and Maluszyński [33, 34], this makes them incomplete w.r.t. weak partial correctness.

Also, we mention that some approaches in the literature have been specifically developed to infer approximations of the call patterns semantics. They are mainly based on abstract interpretation techniques (see e.g., [38, 53]).

Partial Correctness

Among the cited approaches to weak partial correctness, only Malfon [111] proposed a method for proving partial correctness (of general logic programs w.r.t. Fitting and well-founded semantics [4]).

(Weak) Total Correctness

Some methods for weak partial correctness have been extended to reason on weak total correctness. The extension consist of considering a level mapping (defined on possibly non-ground atoms) and additional proof obligations that impose decreasing over a well-founded ordering.

In particular, termination conditions were proposed for the method of Bossi and Cocco by Bossi et al. [29], for well-typed programs by Bronsard et al. [37], and for the annotation method by Deransart and Maluszynski [66]. We will relate weak total correctness methods to the $\mathcal F_t$ proof relation in Chapter 4.

3.8 Conclusion

In this Chapter, we have introduced the notions of program semantics, specifications, proof relations and several definitions of correctness. With reference to those notions, we have developed a general strategy for systematically deriving weak total correctness proof methods. The general strategy consists of merging the termination proof methods of Chapter 2 with a well-known weak partial correctness method.

From a theoretical perspective, the resulting method is at most powerful as the sum of the separated methods. From a practical perspective, instead, the combined method should support shorter and simpler proofs, with respect to the separated methods, i.e. fewer proof obligations should be required.

We have applied the general strategy to the class of acceptable, fair-bounded and bounded programs, obtaining the proof relations $\mathcal T_t$, $\mathcal T_f$ and $\mathcal T_b$, respectively. In general, those proof relations are sound w.r.t. weak total correctness, persistent along SLD-derivations, and allow for characterizing success patterns.
Proof relation $\vdash_1$ will be extensively investigated in the next Chapter. In particular, additional proof obligations will be devised there in order to obtain a method which is sound w.r.t. total correctness. We mention here that those proof obligations can be readily generalized to a systematic approach for deriving total correctness proof methods.
Chapter 4

Verification of Prolog programs

This Chapter introduces a proof theory – formally derived in Chapter 3 – designed as a candidate unifying framework for the verification of Prolog programs. The starting point of the research reported has been the recognition of a few core principles, common to several existing proof methods for logic programs. On this basis, a thorough proof theory has been developed, capable of addressing a reasonably large spectrum of properties for a reasonably large class of programs. The spectrum of properties includes

(i) weak partial correctness,
(ii) weak partial correctness,
(iii) characterization of call and success patterns,
(iv) characterization of correct and computed instances,
(v) absence of type and run-time errors,
(vi) safe omission of the occur-check,
(vii) modular program development,

whereas the class of programs is that of logic programs, possibly with negation and arithmetic built-in’s, which are designed to be executed according to a fixed selection rule. As a consequence, the proposed proof theory is general enough to encompass verification of Prolog programs.

4.1 Proof Theory

The proof theory is based on Hoare’s style triples \{Pre\} P \{Post\} which, for a logic program P, specify the admissible input and expected output by means of pre-
and postconditions. Although the logic programming version of a triple is defined on purely logical terms, it can be readily applied to reason about operational and run-time properties, thus abstracting away from the subtleties of the procedural interpretation of logic programming—unification and the logical variable, the search strategy, to mention a few. In this sense, the proposed verification method is carefully designed as a compromise between generality and expressiveness from one side, and ease of use from the other side.

On the basis of the above discussion, the contribution of this Chapter is the introduction of a proof relation $\vdash_t \{\text{Pre}\} P \{\text{Post}\}$ for total correctness, capable of addressing properties (i–vi) for logic programs with negation and arithmetic built-in’s. For obvious reasons of presentation, the $\vdash_t$ proof relation is introduced in an incremental way, by a stepwise definition of increasingly higher levels of verification, from weak partial correctness up to full-fledged total correctness. This is a standard presentation style adopted in many textbooks on Hoare’s logic for imperative programming, such as [14].

### 4.1.1 The Proof Method

The proof method essentially consists in the definition of a proof relation $\vdash_t$ for triples, which, as we will show later, provides a tool for reasoning about (weak) total correctness. As already mentioned, we shall study $\vdash_t$ in an incremental way, by considering first a sub-relation $\vdash$ on triples, which provides a tool for reasoning about (weak) partial correctness.

The next key definition introduces both proof relations, $\vdash_t$ and $\vdash$, by explaining the proof obligations needed to prove a triple in either sense.

**Definition 4.1.1** Consider a program $P$, a query $Q$, and a specification $(\text{Pre}, \text{Post})$. We write:

- $\vdash_t \{\text{Pre}\} P \{\text{Post}\}$ iff there exists a level mapping $|$ such that for every $A \leftarrow B_1, \ldots, B_n \in \text{ground}_L(P)$:

  1. for $i \in [1, n]$:
     
     $\text{Pre} \models A \land \text{Post} \models B_1, \ldots, B_{i-1} \Rightarrow$
     
     (a) $\text{Pre} \models B_i$ and
     
     (b) $|A| > |B_i|$

  2. $\text{Pre} \models A \land \text{Post} \models B_1, \ldots, B_n \Rightarrow \text{Post} \models A$.

We write $\vdash \{\text{Pre}\} P \{\text{Post}\}$ when (1a) and (2) hold. $\text{Pre}$ is called a precondition and $\text{Post}$ a postcondition.

- $\vdash \{\text{Pre}\} Q \{\text{Post}\}$ iff for every ground instance $A_1, \ldots, A_n$ of $Q$:

  3. for $i \in [1, n]$ $\text{Post} \models A_1, \ldots, A_{i-1} \Rightarrow \text{Pre} \models A_i$. 


4.1. Proof Theory

- $\vdash_i \{\text{Pre}\} \{\text{Post}\}$ iff there exist a level mapping $\mid \mid$ and $k \in N$ such that for every ground instance $A_1, \ldots, A_n$ of $Q$:

  \[ j. \text{for } i \in [1,n] \quad \text{Post} \models A_1, \ldots, A_{i-1} \Rightarrow \text{Pre} \models A_i \land k > |A_i| \].

Intuitively, for a clause $C$ in $\text{ground}_L(P)$ there are $n+1$ proof obligations to conclude that the relation $\vdash \{\text{Pre}\} \{\text{Post}\}$ holds:

1. each atom $B$ in the body of $C$ is in $\text{Pre}$ when the head of $C$ is in $\text{Pre}$ and all the atoms to the left of $B$ in the body of $C$ are in $\text{Post}$;

2. the head of $C$ is in $\text{Post}$ when it is in $\text{Pre}$ and all the atoms in the body of $C$ are in $\text{Post}$.

In the case of $\vdash_i \{\text{Pre}\} \{\text{Post}\}$ the decreasing of the level mapping is also required, i.e. the level mapping plays the role of a termination function. Strictly speaking, the level mapping has to be defined only on the precondition $\text{Pre}$.

The left-to-right propagation of assumptions in proof obligations is biased by the left-to-right evaluation strategy of Prolog, in the sense that we require that a body atom is ready to be executed (i.e., it is in $\text{Pre}$) when the atoms to its left have been already executed (i.e., they are in $\text{Post}$). However, the presented notion is purely declarative, and no procedural intuition is needed to carry on the proof obligations. Moreover, we observe that the proof method applies to arbitrary fixed selection rules other than leftmost’s by simply considering permutations of the body atoms.

Proving that a triple is in the relation $\vdash$ or $\vdash_i$ for a given program or query involves reasoning on their ground instances only. Basically, the definition provides a standard way for lifting up the results to non-ground queries. The advantage is that this lifting is made a posteriori.

Finally, we point out that Definitions 4.1.1 (3,4) for a query $Q$ are derived from Definitions 4.1.1 (1,2) by considering the program $p \leftarrow Q$, where $p$ is a fresh predicate. This follows from the following useful relation, which is immediate from Definition 4.1.1 (1,2); for $\vdash \{\text{Pre}\} \{\text{Post}\}$ and $A \leftarrow B_1, \ldots, B_n \in \text{ground}_L(P)$:

$A \in \text{Pre} \quad \text{implies} \quad \vdash \{\text{Pre}\} B_1, \ldots, B_n \{\text{Post}\}$.  \hspace{1cm} (4.1)

The study of relation $\vdash_i$ is performed by means of some intermediate steps. First, we study how the subrelation $\vdash$ allows us to reason on (weak) partial correctness and to characterize call patterns; second, we study how $\vdash$ allows us to characterize correct and computed instances of intended queries; and finally, we study how the $\vdash_i$ relation completes the picture by dealing with termination.

**Example 4.1.2** (Preorder tree traversal) As an example to clarify the form of the needed proof obligations, consider the program $\text{PREORDER}$:

\[ \vdash_i \{\text{Pre}\} \{\text{Post}\} \]
Chapter 4. Verification of Prolog programs

preorder(T, Ls) ←
   Ls is a preorder traversal of the binary tree T

(p1) preorder(nil, []).
(p2) preorder(leaf(X), [X]).
(p3) preorder(tree(X, Left, Right), Ls) ←
   preorder(Left, As),
   preorder(Right, Bs),
   append([X|As], Bs, Ls).

augmented by the APPEND program.

The set of ground binary trees Tree(α, β) whose leaves belong to α ⊆ U_L and intermediate nodes belong to β ⊆ U_L is defined by the grammar:

Tree ::=: nil | leaf(α) | tree(β, Tree, Tree)

For instance, if α = {0,1,2,...} and β = {+,−,*,...}, we have that Tree(α, β) is the set of syntax trees defining arithmetic expressions on natural numbers. We denote by ||t|| the number of nodes of a tree t, defined as follows:

||leaf(x)|| = 1
||tree(x, l, r)|| = ||l|| + ||r|| + 1
||f(t_1,...,t_n)|| = 0 otherwise.

Intuitively, an intended use of PREORDER is to compute the preorder traversal of a given tree. This is formally expressed by defining:

Pre_PREORDER = { preorder(t, ls) | t ∈ Tree(α, β) } ∪ Pre_APPEND

or, in a more intuitive representation:

Pre_PREORDER = preorder(Tree(α, β) × U_L) ∪ append(ulist × ulist × U_L).

This precondition allows us to concentrate on the relevant input queries, abstracting away from ill-typed information which is present in the least Herbrand model of PREORDER, such as the unintended atom

preorder(tree(0,leaf([]),leaf(nil)), [0,[],nil]).

Indeed, a complete characterization of M_E PREORDER is much more laborious than simply reasoning on correctness and termination of atoms in Pre_PREORDER.

A candidate level mapping is:

||preorder(t, ls)|| = ||t|| + 1
||append(xs, ys, zs)|| = ||xs||.
The +1 adjustment is needed to satisfy the required proof obligations, but, as shown later in Section 4.2.2, the method can be refined to avoid this complication. Finally, we define the postcondition, which reflects the intended interpretation of \textsc{Preorder}:

\[
\text{Post}_{\text{PREORDER}} = \{ \text{preorder}(t, ls) \mid t \in \text{Tree}(v, \beta) \land \\
ls \text{ is a preorder traversal of } t \} \cup \text{Post}_{\text{APPEND}}.
\]

We are now ready to prove \( \vdash \{ \text{Pre}_{\text{PREORDER}} \} \text{PREORDER} \{ \text{Post}_{\text{PREORDER}} \} \), by showing the proof obligations of Definition 4.1.1.

For clause (p1), we have to show that if \( \text{preorder}(\text{nil}, []) \) is in the precondition then it is in the postconditions as well, which is obvious.

The reasoning on (p2) and on the clauses of \textsc{Append} is also immediate.

Let us concentrate on clause (p3), and consider a ground instance:

\[
\text{preorder}(\text{tree}(x, \text{left}, \text{right}), ls) \leftarrow \\
\text{preorder}(\text{left}, as), \\
\text{preorder}(\text{right}, bs),
\text{append}([x|as], bs, ls).
\]

Assume that the head is in the precondition, i.e. that \( x \in \beta \) and \( \text{left}, \text{right} \in \text{Tree}(v, \beta) \). By definition of \( \text{Pre}_{\text{PREORDER}} \), we have that \( \text{preorder}(\text{left}, as) \) and \( \text{preorder}(\text{right}, bs) \) are also in the precondition. Moreover,

\[
\begin{align*}
|\text{preorder}(\text{tree}(x, \text{left}, \text{right}), ls)| &= |\text{left}| + |\text{right}| + 2 \\
&> |\text{left}| + 1 \\
&= \text{preorder}(\text{left}, as)
\end{align*}
\]

and analogously for \( \text{preorder}(\text{right}, bs) \). This shows the proof obligation (1) of Definition 4.1.1 in the cases \( i = 1, 2 \).

In the case \( i = 3 \), we have to show that \( \text{append}([x|as], bs, ls) \) is in \( \text{Pre}_{\text{PREORDER}} \), i.e. that \([x|as]\) and \( bs \) are ground lists, under the assumption that \( \text{preorder}(\text{left}, as) \) and \( \text{preorder}(\text{right}, bs) \) are in \( \text{Post}_{\text{PREORDER}} \), which is obvious. Moreover,

\[
\begin{align*}
|\text{preorder}(\text{tree}(x, \text{left}, \text{right}), ls)| &= |\text{left}| + |\text{right}| + 2 \\
&> |\text{left}| + 1 \\
&= |as| + 1 \\
&= |\text{append}([x \mid as], bs, ls)|
\end{align*}
\]

since \( \text{preorder}(\text{left}, as) \in \text{Post}_{\text{PREORDER}} \) implies that the number of nodes of \( \text{left} \) is equal to the length of \( as \).

Finally, consider the proof obligation (2). Under the hypothesis that:
tree(x, left, right) ∈ Tree(α, β), and
as is a preorder traversal of left, and
bs is a preorder traversal of right, and
ls = [x | as] * bs,
we have to show that ls is a preorder traversal of tree(x, left, right), which is immediate.

In conclusion, \( \vdash_t \{ \text{Pre}_\text{PREORDER} \} \text{PREORDER} \{ \text{Post}_\text{PREORDER} \} \) holds. Lifting up to non-ground queries, we observe that:

\( \vdash_t \{ \text{Pre}_\text{PREORDER} \} \text{preorder}(T, L) \{ \text{Post}_\text{PREORDER} \} \)

holds, where T is a (possibly non-ground) term with every ground instance in Tree(α, β), and L is a variable. The query preorder(T, L) is intended to calculate the preorder traversal of T. In particular, if α = β = U_L then the nodes of the tree T can be any term. Analogously, if α = GList the leaves can be any list. □

**Proof Outlines**

The proof that \( \vdash_t \{ \text{Pre} \} \text{P} \{ \text{Post} \} \) holds can be presented in a suggestive way by means of *proof outlines*, a well-known tool in verification of imperative programs.

**Definition 4.1.3** A proof outline for a clause \( A \leftarrow A_1, \ldots, A_n \), a level mapping \( \| \| \), and a specification \( \langle \text{Pre}, \text{Post} \rangle \) is a labeled clause of the form:

\[
A_0 \{ g_0 \} \leftarrow \{ t_0 \}
\]
\[
\begin{array}{l}
A_0 \{ g_0 \} \\
A_1 \{ f_1 \} \{ t_1 \} \\
\vdots \\
A_{n-1} \{ f_{n-1} \} \{ t_{n-1} \} \\
A_n \{ f_n \} \{ t_n \}
\end{array}
\]

where \( t_i \) and \( f_i, g_i \), for \( i \in [0, n] \) are respectively integer expressions and assertions (in some formal logic), such that every ground instance of the following *proof obligations* is satisfied:

(i) for \( i \in [0, n] \): \( t_i = |A_i| \),
(ii) for \( i \in [0, n] \): \( g_i \Rightarrow A_i \in Pre \),
(iii) for \( i \in [0, n] \): \( f_i \Rightarrow A_i \in Post \),
(iv) for \( i \in [1, n] \): \( g_0 \land f_1 \land \ldots \land f_{i-1} \Rightarrow g_i \land t_0 > t_i \),
(v) \( g_0 \land f_1 \land \ldots \land f_n \Rightarrow f_0 \).  

The intuition is that the assertion \( g_i \) specifies the conditions under which the atom \( A_i \) of the clause is in \( Pre \), the assertion \( f_i \) specifies the conditions under which \( A_i \) is in \( Post \), and the expressions \( t_i \) represent the level of \( A_i \). Under this assumption, the proof obligations (iv) and (v) directly reflect respectively Definition 4.1.1 (1) and (2).

By construction, \( \vdash \{ Pre \} P \{ Post \} \) holds if and only if there exists a level mapping \( \| \) and a proof outline for each clause of \( P \) and \( \| \), \( \{ Pre, Post \} \).

**Example 4.1.4** Let us see the proof outlines for \( PREORDER \).

```plaintext
{ nil \in Tree(\alpha, \beta) } 
preorder(nil, \[] ).  { 1 }
    { nil \in Tree(\alpha, \beta) \land \[] \ \text{is a preorder traversal of it} }

{ leaf(X) \in Tree(\alpha, \beta) } 
preorder(leaf(X), [X] ).  { 2 }
    { leaf(X) \in Tree(\alpha, \beta) \land [X] \ \text{is a preorder traversal of it} }

{ tree(X, Left, Right) \in Tree(\alpha, \beta) } 
preorder(tree(X, Left, Right), Ls) \rightarrow \|Left\| + \|Right\| + 2
    { Left \in Tree(\alpha, \beta) }
preorder(Left, As), \quad \|Left\| + 1
    { Left \in Tree(\alpha, \beta) \land As \ \text{is a preorder traversal of it} }
preorder(Right, Bs), \quad \|Right\| + 1
    { Right \in Tree(\alpha, \beta) \land Bs \ \text{is a preorder traversal of it} }
    { [X|As], Bs \in GList } 
append([X|As], Bs, Ls). \quad |As| + 1
    { [X|As], Bs \in GList \land Ls = [X|As]*Bs } 
{ tree(X, Left, Right) \in Tree(\alpha, \beta) \land Ls \ \text{is a preorder traversal of it} }
```

The related proofs have been shown in the previous section.

Proof outlines become simpler when considering the relation \( \vdash \). In the labeled clause no \( t_i \) appears and the proof obligations are (ii-iii-v) plus the following simplified version of (iv):

(iv') for \( i \in [1, n] \): \( g_0 \land f_1 \land \ldots \land f_{i-1} \Rightarrow g_i \).
Again, by construction \( \vdash \{ \text{Pre} \} P \{ \text{Post} \} \) holds iff there exists a proof outline for each clause of \( P \) and \( \text{Pre}, \text{Post} \). It should be observed how proof outlines represent in a concise way the domino-style propagation of premises in the proof obligations of Definition 4.1.1.

We also present a more general form of proof outlines, which enable us to simplify proofs by using assertions \( f_i \) and \( g_i \) which do not coincide with post and preconditions, but rather represent strengthenings and weakenings of \( \text{Post} \) and \( \text{Pre} \), according to the following reformulation of proof outlines. We denote with \( h_i \) for \( i \in [0, n+1] \) the assertion \( g_0 \land f_1 \land \ldots \land f_{i-1} \). Thus, in particular, \( h_0 = h_1 = g_0 \).

(i) for \( i \in [0, n] \): \( h_i \land g_i \Rightarrow t_i = |A_i| \).
(ii) \( g_0 \iff A_0 \in \text{Pre} \), and for \( i \in [1, n] \): \( h_i \land g_i \Rightarrow A_i \in \text{Pre} \),
(iii) for \( i \in [1, n] \): \( f_i \iff A_i \in \text{Post} \land h_i \), and \( h_{i+1} \land f_0 \Rightarrow A_0 \in \text{Post} \),
(iv) for \( i \in [1, n] \): \( h_i \Rightarrow g_i \land t_0 > t_i \),
(v) \( h_{n+1} \Rightarrow f_0 \).

The construction of a proof outline satisfying the proof obligations (i) through (v) for every clause in a program \( P \) amounts to showing that \( \vdash \{ \text{Pre} \} P \{ \text{Post} \} \) holds. In the following figure it is depicted how (i) through (iv) imply conditions 1(a) and 1(b) of Definition 4.1.1.

\[
\begin{array}{c}
g_0 \land f_1 \land \ldots \land f_{i-1} \downarrow \vdash g_i \land t_0 > t_i \\
\uparrow A_0 \in \text{Pre} \land A_1 \in \text{Post} \land \ldots \land A_{i-1} \in \text{Post} \downarrow \\
\downarrow A_i \in \text{Pre} \land |A_0| > |A_i|
\end{array}
\]

Similarly, the proof obligations (ii,iii,v) imply condition 2 of Definition 4.1.1. Finally, we observe that proof obligations for the \( \vdash \) relation can be similarly relaxed by considering (ii,iii,v) and
\[ iv' \text{ for } i \in [1, n] : h_i \Rightarrow g_i. \]

### 4.1.2 Modularity

Proving \( \vdash \{ \text{Pre} \} P \{ \text{Post} \} \) and \( \vdash \{ \text{Pre} \} P \{ \text{Post} \} \) can be a difficult task when considering either large programs or complex specifications. We will discuss the former case in Section 4.2.2, by providing some results in order to show that

\[
\vdash \{ \text{Pre} \} P \cup Q \{ \text{Post} \}
\]

holds, starting from triples for \( P \) and \( Q \). In this section, we investigate how to prove \( \vdash \{ \text{Pre} \} P \{ \text{Post} \} \) starting from triples involving simpler specifications. First, we show that the following (Hoare’s logic style) rule is valid:

**Theorem 4.1.5**

\[
\vdash \{ \text{Pre} \} P \{ \text{Post} \} \quad \vdash \{ \text{Pre} \} P \{ \text{Post}' \}
\]

\[
\vdash \{ \text{Pre} \} P \{ \text{Post} \cap \text{Post}' \}
\]
Proof. Let $A \leftarrow B_1, \ldots, B_n$ be a ground instance of a clause from $P$:

- for $i \in [1, n]$, if $\text{Pre} \models A$ and $\text{Post} \cap \text{Post}' \models B_1, \ldots, B_{i-1}$ then $\vdash \{\text{Pre}\} P \{\text{Post}\} \implies \text{Pre} \models B_i$;
- if $\text{Pre} \models A$ and $\text{Post} \cap \text{Post}' \models B_1, \ldots, B_n$ then $\vdash \{\text{Pre}\} P \{\text{Post}\}$ implies $\text{Post} \models A$ and $\vdash \{\text{Pre}\} P \{\text{Post}'\}$ implies $\text{Post}' \models A$. Therefore $\text{Post} \cap \text{Post}' \models A$. □

The importance of this rule is twofold. On the one hand, it is relevant from a practical point of view, since the proof of a triple is split into two simpler proofs. For instance, if we have to show the correctness of a sorting program, with postcondition:

\[
\{ \text{sort}(xs, ys) \mid ys \text{ is an ordered permutation of } xs \}
\]

then we can split it into:

\[
\{ \text{sort}(xs, ys) \mid ys \text{ is a permutation of } xs \}
\]

and

\[
\{ \text{sort}(xs, ys) \mid ys \text{ is an ordered list } \}.
\]

Proving separately the correctness of the two simpler postconditions generally involves less effort than proving the correctness of their conjunction. On the other hand, Theorem 4.1.5 allows us to define the notion of strongest postcondition.

**Definition 4.1.6** Assume that $\vdash \{\text{Pre}\} P \{\text{Post}\}$ holds. We denote by $\text{sp}(P, \text{Pre})$ the intersection of every $\text{Post}'$ such that $\vdash \{\text{Pre}\} P \{\text{Post}'\}$, and call it the strongest postcondition of $P$ and $\text{Pre}$. □

It is worth noting that the strongest postcondition is defined only for programs for which there exists at least one $\text{Post}$ such that $\vdash \{\text{Pre}\} P \{\text{Post}\}$. In fact, there exist programs and preconditions for which $\vdash \{\text{Pre}\} P \{\text{Post}\}$ holds for no $\text{Post}$, such as $p \leftarrow q$ and $\text{Pre} = \{p\}$.

Symmetrically, the following rule holds, which allows us to simplify preconditions.

**Theorem 4.1.7**

\[
\vdash \{\text{Pre}\} P \{\text{Post}\} \quad \vdash \{\text{Pre}'\} P \{\text{Post}\} \quad \vdash \{\text{Pre} \cup \text{Pre}'\} P \{\text{Post}\}.
\]

We introduce now the notion of weakest liberal precondition, defined as the union of all preconditions.

**Definition 4.1.8** We denote by $\text{wlp}(P, \text{Post})$ the union of every $\text{Pre}'$ such that $\vdash \{\text{Pre}'\} P \{\text{Post}\}$, and call it the weakest liberal precondition of $P$ and $\text{Post}$. □
We point out that it is not necessary to define a notion of strongest terminating postcondition and a precondition.

**Theorem 4.1.9**

\[
\vdash \{ \text{Pre} \} P \{ \text{Post} \} \quad \vdash \{ \text{Pre} \} P \{ \text{sp}(P, \text{Pre}) \} \quad \vdash \{ \text{wl}(P, \text{Post}) \} P \{ \text{Post} \}.
\]

\[\]

We next extend Theorems 4.1.5 and 4.1.7 to relation \( \vdash_i \).

**Theorem 4.1.10**

\[
\vdash_i \{ \text{Pre} \} P \{ \text{Post} \} \quad \vdash_i \{ \text{Pre} \} P \{ \text{Post}' \} \quad \vdash_i \{ \text{Pre} \} P \{ \text{Post} \cap \text{Post}' \}.
\]

**Proof.** Suppose that \( \vdash_i \{ \text{Pre} \} P \{ \text{Post} \} \) holds using the level mapping \( |_1 \), and \( \vdash_i \{ \text{Pre} \} P \{ \text{Post}' \} \) holds using \( |_2 \). Then we show that \( \vdash_i \{ \text{Pre} \} P \{ \text{Post} \cap \text{Post}' \} \) holds using \( |_1 + |_2 \) (or equivalently \( \min\{|_1, |_2\} \)). By Theorem 4.1.5, we have only to show the decreasing of the level mapping. Consider \( A \leftarrow B_1, \ldots, B_n \in \text{ground}_L(P) \) and \( i \in [1, n] \). If \( \text{Pre} \models A \land \text{Post} \cap \text{Post}' \models B_1, \ldots, B_{i-1} \) then, by hypothesis, \( |A|_1 > |B|_1 \) and \( |A|_2 > |B|_2 \). Therefore \( |A|_1 + |A|_2 > |B|_1 + |B|_2 \), and analogously for \( \min \).

We point out that it is not necessary to define a notion of strongest terminating postcondition as the intersection of all \( \text{Post} \) such that \( \vdash_i \{ \text{Pre} \} P \{ \text{Post} \} \). In fact, \( \vdash_i \{ \text{Pre} \} P \{ \text{Post} \} \) trivially implies \( \vdash_i \{ \text{Pre} \} P \{ \text{sp}(P, \text{Pre}) \} \), and therefore \( \text{sp}(P, \text{Pre}) \) is the strongest terminating postcondition.

On the other hand, it is important to define a notion of weakest precondition of \( P \) and \( \text{Post} \) as the union of every precondition \( \text{Pre} \) such that \( \vdash_i \{ \text{Pre} \} P \{ \text{Post} \} \). We start by stating the following rule:

**Theorem 4.1.11**

\[
\vdash_i \{ \text{Pre} \} P \{ \text{Post} \} \quad \vdash_i \{ \text{Pre}' \} P \{ \text{Post} \} \quad \vdash_i \{ \text{Pre} \cup \text{Pre}' \} P \{ \text{Post} \}.
\]

**Proof.** Assume that \( \vdash_i \{ \text{Pre} \} P \{ \text{Post} \} \) holds using the level mapping \( |_1 \), and that \( \vdash_i \{ \text{Pre}' \} P \{ \text{Post} \} \) holds using the level mapping \( |_2 \). By defining \( || \) as follows:

\[
||A|| = \begin{cases} 
\min\{|A|_1, |A|_2| & \text{if } A \in \text{Pre} \cap \text{Pre}' \\
|A|_1 & \text{if } A \in \text{Pre} \setminus \text{Pre}' \\
|A|_2 & \text{if } A \in \text{Pre}' \setminus \text{Pre} 
\end{cases}
\]

and \( ||A|| = 0 \) otherwise, it is readily checked that \( \vdash_i \{ \text{Pre} \cup \text{Pre}' \} P \{ \text{Post} \} \) holds using the level mapping \( || \).

\[\]
Example 4.1.12 (Append Ctd) As an example, consider again the APPEND program. One of its uses is to compute the list obtained by concatenating two given lists. In fact, we have that

$\forall t \{ Pre \} APPEND \{ Post \} \text{ holds by the level mapping } | |_1$, where:

$$|\text{append}(xs, ys, zs)|_1 = |xs|.$$  

In addition to this use (or directionality), APPEND is also employed to extract prefixes or suffixes of a given list. For instance, for a list $zs$, a computed instance of $\text{append}(Xs, Ys, zs)$ binds $Xs$ to a prefix and $Ys$ to a suffix of $zs$. Formally, one can show that $\forall t \{ Pre \} APPEND \{ Post \} \text{ holds by the level mapping } | |_2$, where:

$$Pre = \text{append}(U_L \times U_L \times GList)$$

$$|\text{append}(xs, ys, zs)|_2 = |zs|.$$  

By Theorem 4.1.11, we conclude that $\forall t \{ Pre \} APPEND \{ Post \} \text{ holds by the level mapping } | |_2$.

$$|\text{append}(xs, ys, zs)|_2 = \begin{cases} \min\{|xs|,|zs|\} & \text{if } xs, ys, zs \in GList \\ |xs| & \text{if } xs, ys \in GList, zs \notin GList \\ |zs| & \text{if } xs, ys \notin GList, zs \in GList \\ 0 & \text{otherwise.} \end{cases}$$

The next definition introduces the notion of the weakest precondition.

Definition 4.1.13 We denote by $wp(P, Post)$ the union of every $Pre'$ such that $\forall t \{ Pre' \} P \{ Post \}$, and call it the weakest precondition of $P$ and $Post$. 

In general, the weakest precondition and the weakest liberal precondition do not coincide.

Example 4.1.14 Consider the following program $P$:

$$p \leftarrow p.$$  

and $Post = \{ p \}$. We have that:

$$wp(P, Post) = \{ p \} \neq \emptyset = wp(P, Post).$$

By generalizing the proof of Theorem 4.1.11, we obtain that $wp(P, Post)$ is indeed a precondition for $P$ and $Post$. 

2. Verification of Prolog programs

Theorem 4.1.15

\[ \vdash_t \{ \text{wp}(P, \text{Post}) \} P \{ \text{Post} \}. \]

Proof. Consider the sequence \( \text{Pre}_1, \text{Pre}_2, \ldots \) of all preconditions \( \text{Pre}_i \) such that \( \vdash_t \{ \text{Pre}_i \} P \{ \text{Post} \} \) holds using \( || \). We define \( \|A\| = \min \{ \|A_i\| : \text{Pre}_i \vdash \text{Post}, i \geq 1 \} \), for \( A \in \text{wp}(P, \text{Post}) \), and \( \|A\| = 0 \) otherwise. We show that \( \vdash_t \{ \text{wp}(P, \text{Post}) \} P \{ \text{Post} \} \) holds using \( || \). Consider \( A \leftarrow B_1, \ldots, B_n \in \text{ground}_E(P) \).

- Assume that, for \( j \in [1, n] \), \( \text{wp}(P, \text{Post}) \models A \) and \( \text{Post} \models B_1, \ldots, B_{j-1} \). For every \( \text{Pre}_i \) such that \( A \in \text{Pre}_i \), we have that \( \text{Pre}_i \models B_j \) and \( |A| > |B_j| \). Therefore, \( \text{wp}(P, \text{Post}) \models B_j \) and \( \|A\| = \min \{ |A_i| : \text{Pre}_i \vdash \text{Post}, i \geq 1 \} > \min \{ |B_j| : B_j \in \text{Pre}_i, i \geq 1 \} = \|B_j\| \).
- If \( \text{wp}(P, \text{Post}) \models A \) and \( \text{Post} \models B_1, \ldots, B_n \), then there exists \( \text{Pre}_i \) such that \( \text{Pre}_i \models A \) and \( \text{Post} \models B_1, \ldots, B_n \). Therefore, \( \text{Post} \models A \).

We conclude this section by introducing two further rules.

Theorem 4.1.16

\[ \vdash \{ \text{Pre} \} P \{ \text{Post} \} \quad \vdash_t \{ \text{Pre} \} P \{ \text{Post} \} \quad \vdash \{ \text{Pre} \} P \{ \text{Post} \cap \text{Pre} \} \quad \vdash_t \{ \text{Pre} \} P \{ \text{Post} \cap \text{Pre} \}. \]

We already observed that in Definition 4.1.1, the values assumed by a level mapping are relevant only on the set \( \text{Pre} \). For postconditions, the above theorem shows that \( \text{Post} \cap \text{Pre} \) is a postcondition if \( \text{Post} \) is so. Intuitively, this fact is justified by the intuition that a postcondition describes a property of atoms in \( \text{Pre} \) which are consequences of the program.

The next sections are devoted to study the relation of \( \vdash \) and \( \vdash_t \) with the four introduced notions of correctness, namely (weak) partial and (weak) total, and with the properties of persistency and call patterns characterization.

4.1.3 Persistency, Call Patterns and Success Patterns

A key result is the following theorem, stating the persistency of the relation \( \vdash \) along SLD-derivations of programs \( P \) and queries \( Q \).

Theorem 4.1.17 (Persistency) Assume that

\[ \vdash \{ \text{Pre} \} P \{ \text{Post} \} \quad \text{and} \quad \vdash \{ \text{Pre} \} Q \{ \text{Post} \}. \]

Then for every SLD-resolvent \( Q' \) of \( P \) and \( Q \), \( \vdash \{ \text{Pre} \} Q' \{ \text{Post} \} \) holds.
Proof. First, we show that if \( Q = A_1, \ldots, A_n \) is a ground query and 
\( A_i \leftarrow B_1, \ldots, B_l \in \text{ground}_i(P) \) then the ground resolvent:

\[
Q' = A_1, \ldots, A_{i-1}, B_1, \ldots, B_l, A_i, A_{i+1}, \ldots, A_n
\]
is such that \( \vdash \{\text{Pre}\} Q \{\text{Post}\} \) holds. Let us show the proof obligations.

- For \( k \in [1, i - 1] \), if \( \text{Post} \models A_1, \ldots, A_{k-1} \) then, since \( \vdash \{\text{Pre}\} Q \{\text{Post}\}, \) \( \text{Pre} \models A_k \).
- For \( k \in [1, l] \), if \( \text{Post} \models A_1, \ldots, A_{k-1}, B_1, \ldots, B_{k-1} \) then \( \text{Pre} \models A_k \), since \( \text{Post} \models A_1, \ldots, A_{k-1} \) and \( \vdash \{\text{Pre}\} Q \{\text{Post}\} \) holds. In addition, \( \vdash \{\text{Pre}\} P \{\text{Post}\} \) and \( \text{Pre} \models A \wedge \text{Post} \models B_1, \ldots, B_{k-1} \) imply \( \text{Pre} \models B_k \).
- For \( k \in [i+1, n] \), if \( \text{Post} \models A_1, \ldots, A_{i-1}, B_1, \ldots, B_l, A_i+1, \ldots, A_{k-1} \) then \( \text{Pre} \models A_k \), since \( \text{Post} \models A_1, \ldots, A_{i-1} \) and \( \vdash \{\text{Pre}\} Q \{\text{Post}\} \) holds. In addition, \( \vdash \{\text{Pre}\} P \{\text{Post}\} \) and \( \text{Pre} \models A \wedge \text{Post} \models B_1, \ldots, B_l \) imply \( \text{Post} \models A_i \), and then

\[
\text{Post} \models A_1, \ldots, A_{i-1}, A_i, A_i, A_{i+1}, \ldots, A_{k-1}.
\]

Since \( \vdash \{\text{Pre}\} Q \{\text{Post}\} \), we conclude \( \text{Pre} \models A_k \).

Consider now a not necessarily ground query \( Q = A_1, \ldots, A_n \), and let

\[
Q' = (A_1, \ldots, A_{i-1}, B_1, \ldots, B_l, A_i, A_{i+1}, \ldots, A_n) \theta
\]
be the SLD-resolvent of \( Q \) and a variant \( B \leftarrow B_1, \ldots, B_l \) of a clause from \( P \), where \( \theta \) is the mgu of \( A_i \) and \( B \). We point out that every ground instance of \( Q' \) is a ground resolvent of a ground instance of \( Q \theta \) and a ground instance of \( (B \leftarrow B_1, \ldots, B_l) \theta \). Since \( \vdash \{\text{Pre}\} Q \{\text{Post}\} \) holds, then \( \vdash \{\text{Pre}\} Q \theta \{\text{Post}\} \) holds. From the first part of this proof, we conclude that for every ground instance \( Q'' \) of \( Q' \), \( \vdash \{\text{Pre}\} Q'' \{\text{Post}\} \) holds. By Definition 4.1.1, this is equivalent to say that \( \vdash \{\text{Pre}\} Q' \{\text{Post}\} \) holds, hence the conclusion of the theorem. \( \square \)

Directly from persistency and Definition 4.1.1 (3), we have that the leftmost atom \( A \) of any query in a LD-derivation for \( P \) and \( Q \) is true in \( \text{Pre} \), i.e. \( \text{Pre} \models A \). In other words, \( \text{Pre} \) declaratively characterizes call patterns w.r.t. LD-resolution.

Another immediate consequence is that every computed instance of \( P \) and \( Q \) is true in the interpretation \( \text{Post} \), i.e. success patterns characterization. This is a formal counterpart to the intuitive notion that a postcondition is a description of correct instances of intended queries.

**Corollary 4.1.18** Assume that \( \vdash \{\text{Pre}\} P \{\text{Post}\} \) and \( \vdash \{\text{Pre}\} Q \{\text{Post}\} \) holds. Then

(i) (Call Patterns) for every atom \( A \) selected in a LD-derivation for \( P \) and \( Q \), \( \text{Pre} \models A \).

(ii) (Success Patterns) for every computed instance \( Q' \) of \( P \) and \( Q \), \( \text{Post} \models Q' \).
Proof.  
(i) Let \( A, Q' \) be a query in a LD-derivation for \( P \) and \( Q \). By Theorem 4.1.17, we have that \( \vdash \{ \text{Pre} \} A, Q' \{ \text{Post} \} \). By Definition 4.1.1, \( \text{Pre} \models A' \) for every instance \( A' \) of \( A \). Therefore \( \text{Pre} \models A \).

(ii) Let \( x_1, \ldots, x_n \) be the variables of \( Q \), and \( p \) be a fresh predicate symbol of arity \( n \). We define:

\[
\text{Pre}' = \text{Pre} \cup \{ p(t_1, \ldots, t_n) \mid \text{Post} \models Q\{x_i/t_i \mid i \in [1,n]\} \}
\]

\[
\text{Post}' = \text{Post} \cup p(U_L \times \ldots \times U_L).
\]

With this assumptions, it is readily checked that \( \vdash \{ \text{Pre}' \} P \{ \text{Post}' \} \), since \( p \) does not appear in \( P \), and \( \vdash \{ \text{Pre}' \} Q, p(x_1, \ldots, x_n) \{ \text{Post}' \} \), by definition of \( \text{Post}' \).

By Strong Completeness of SLD-resolution, there exists a LD-refutation for \( P \) and \( Q \) with computed instance (a variant of) \( Q' \). As a consequence, there exists a LD-derivation for \( P \) and \( Q, p(x_1, \ldots, x_n) \) where \( p(T_1, \ldots, T_n) \) is selected, with \( Q\{x_i/T_i \mid i \in [1,n]\} \) variant of \( Q' \). By (i), \( \text{Pre}' \models p(T_1, \ldots, T_n) \). By definition of \( \text{Pre}' \), we conclude that \( \text{Post} \models Q\{x_i/T_i \mid i \in [1,n]\} \), and then \( \text{Post} \models Q' \). \( \square \)

Corollary 4.1.18 is the key result to reason about call patterns and computed/correct instances, as, under the hypothesis of the corollary, \( \text{Pre} \) describes the shapes of every atom selected during a LD-derivation, and \( \text{Post} \) describes a property of computed/correct instances.

**Example 4.1.19 (Preorder Ctd)** Consider a term \( T \) such that every ground instance of it is in \( \text{Tree}(\alpha, \beta) \).

We have that \( \vdash \{ \text{Pre}_\text{PREORDER} \} \text{preorder}(T, X) \{ \text{Post}_\text{PREORDER} \} \) holds. By Corollary 4.1.18, every atom \( \text{preorder}(T', X) \) selected in a LD-derivation of \( \text{PREORDER} \) and \( \text{preorder}(T, X) \) is true in \( \text{Pre}_{\text{PREORDER}} \), which implies that every ground instance of \( T' \) is in \( \text{Tree}(\alpha, \beta) \).

In addition, every computed instance \( \text{preorder}(T, X) \) of \( \text{PREORDER} \) and the query \( \text{preorder}(T, X) \) is true in \( \text{Post}_{\text{PREORDER}} \). In other words, for every ground instance \( \text{preorder}(t, x) \) of \( \text{preorder}(T, X) \), \( x \) is a preorder traversal of \( t \).

\( \square \)

**4.1.4 Weak Partial Correctness**

In general, proof relation \( \vdash \) is sound for proving weak partial correctness.

**Theorem 4.1.20 (Weak Partial Correctness)** If \( \vdash \{ \text{Pre} \} P \{ \text{Post} \} \) holds then \( P \) is weak partially correct w.r.t. the specification \( (\text{Pre}, \text{Post}) \), i.e.

\[
M^L_P \cap \text{Pre} \subseteq \text{Post}.
\]
4.1. Proof Theory

Proof. Consider \( A \in M^L_P \cap \text{Pre} \). Then \( \vdash \{ \text{Pre} \} \ A \{ \text{Post} \} \) holds, and by Strong Completeness of SLD-resolution there exists a LD-refutation for \( P \) and \( A \). Therefore \( A \in \text{Post} \) by Corollary 4.1.18 (\( ii \)). □

Notice, however, that the proof method is not complete with respect to the proposed definition of weak partial correctness.

Example 4.1.21 If we consider the program \( P \)

\[
\begin{align*}
p & \leftarrow q, \\
p. 
\end{align*}
\]

and \( \text{Pre} = \text{Post} = \{p\} \), we have that \( P \) is weak partially correct w.r.t. the specification \((\text{Pre}, \text{Post})\) but \( \vdash \{ \text{Pre} \} \ P \{ \text{Post}' \} \) does not hold for any \( \text{Post}' \). □

The reason of the incompleteness lies in the fact that the proof method based on relation \( \vdash \) addresses call patterns characterization as well as correctness, in the sense clarified by Corollary 4.1.18 (\( i \)). In the example, \( \text{Pre} \) does not characterize call patterns of \( P \) and \( p \) since \( \text{Pre} \not= q \).

4.1.5 Partial Correctness

The Weak Partial Correctness Theorem 4.1.20 suggests to investigate further the relation between postconditions and the well-typed fragment of the least Herbrand model, i.e. \( M^L_P \cap \text{Pre} \). Since every postcondition is a superset of \( M^L_P \cap \text{Pre} \), if we prove that \( \vdash \{ \text{Pre} \} \ P \{ M^L_P \cap \text{Pre} \} \), then we conclude that \( M^L_P \cap \text{Pre} \) is the strongest postcondition. This fact is made precise in the following theorem.

Theorem 4.1.22 Assume that \( \vdash \{ \text{Pre} \} \ P \{ \text{Post} \} \) holds.
Then \( \vdash \{ \text{Pre} \} \ P \{ M^L_P \cap \text{Pre} \} \) holds, and then

\[
sp(P, \text{Pre}) = M^L_P \cap \text{Pre}.
\]

Proof. Consider \( A \leftarrow B_1, \ldots, B_n \in \text{ground}_L(P) \).

- For \( i \in [1, n] \), if \( \text{Pre} \models A \wedge M^L_P \cap \text{Pre} \models B_1, \ldots, B_{i-1} \), then by Theorem 4.1.20, \( \text{Post} \models B_1, \ldots, B_{i-1} \), and, by the hypothesis, \( \text{Pre} \models B_i \).
- Suppose that \( \text{Pre} \models A \wedge M^L_P \cap \text{Pre} \models B_1, \ldots, B_n \). As \( M^L_P \) is a model of \( P \), we conclude that \( M^L_P \models A \), and then \( M^L_P \cap \text{Pre} \models A \).

Therefore \( \vdash \{ \text{Pre} \} \ P \{ M^L_P \cap \text{Pre} \} \) holds.

By Definition 4.1.6 we have that \( sp(P, \text{Pre}) \subseteq M^L_P \cap \text{Pre} \). The converse inclusion \( sp(P, \text{Pre}) \supseteq M^L_P \cap \text{Pre} \) follows from Theorems 4.1.9 and 4.1.20. □

As a consequence, if \( \vdash \{ \text{Pre} \} \ P \{ \text{Post} \} \) holds, then \( P \) is partially correct w.r.t. the specification \((\text{Pre}, sp(P, \text{Pre}))\).
**Theorem 4.1.23 (Partial Correctness)** If \( \vdash \{\text{Pre}\} P \{\text{Post}\} \) holds then \( P \) is partially correct w.r.t. the specification \((\text{Pre}, sp(P, \text{Pre}))\).\!

The problem is now how to characterize the strongest post condition directly, without first constructing the least Herbrand model \( M^L_P \). To this end, we define the notion of *well-supported interpretation*, introduced in the context of semantics for general logic programs (see [4]). A similar notion has been employed for program verification w.r.t. three-valued semantics by Malvone ([11]).

**Definition 4.1.24** Assume that \( \vdash \{\text{Pre}\} P \{\text{Post}\} \) holds. 

\textit{Post} is a well-supported interpretation (w.r.t. \( P \) and \( \text{Pre} \)) iff there exist:

- a well-founded poset \((W, >)\), and
- a function \(|| : B_L \rightarrow W\)

such that for every \( A \in \text{Post} \cap \text{Pre} \) there exists

\[ A \leftarrow B_1, \ldots, B_n \in \text{ground}_L(P) \]

such that: \( \forall i \in [1, n] : \text{Post} \models B_i \land |A| > |B_i| \).

Observe that the condition \( \text{Post} \models B_i \) for \( i \in [1, n] \) is equivalent to require \( \text{Post} \cap \text{Pre} \models B_i \) for \( i \in [1, n] \) under the assumption that \( \vdash \{\text{Pre}\} P \{\text{Post}\} \) holds. The idea underlying this definition is to require that every atom in \( \text{Post} \cap \text{Pre} \) has a successful ground derivation. In fact, for any such atom there exists a ground derivation which is finite, because the poset is well-founded, and successful, because the last selected atom unifies with some clause head.

**Theorem 4.1.25** Let \( P \) be a program such that \( \vdash \{\text{Pre}\} P \{\text{Post}\} \). Then

\[ \text{Post} \cap \text{Pre} = sp(P, \text{Pre}) \iff \text{Post} \text{ is well-supported (w.r.t. } P \text{ and } \text{Pre}). \]

**Proof.**

\textit{only-if} We consider natural numbers with the usual ordering relation \(<\) as the well-founded poset. Let \(|| : B_L \rightarrow N\) be the function:

\[ |A| = \begin{cases} \min\{i : A \in T_P \uparrow i\} & \text{if } A \in M^L_P \cap \text{Pre} \\ 0 & \text{otherwise.} \end{cases} \]

\(||\) is well-defined since \( M^L_P = T_P \uparrow \omega \). By hypothesis, \( \text{Post} \cap \text{Pre} = sp(P, \text{Pre}) \), and then, by Theorem 4.1.22, \( \text{Post} \cap \text{Pre} = M^L_P \cap \text{Pre} = T_P \uparrow \omega \cap \text{Pre} \). To show the conclusion, we prove by induction on \( i \geq 0 \) that for every \( A \in T_P \uparrow i \cap \text{Pre} \) there exists \( A \leftarrow B_1, \ldots, B_n \in \text{ground}_L(P) \) such that:

\[ \forall i \in [1, n] : \text{Post} \models B_i \land |A| > |B_i|. \]
The base case is trivial, since $T_P \uparrow 0 = \emptyset$. Let $A$ be in $T_P \uparrow i \cap Pre$. If $A \in T_P \uparrow (i-1)$ then the conclusion follows from the inductive hypothesis. On the contrary, let $A$ be in $T_P \uparrow i \setminus T_P \uparrow (i-1) \cap Pre$. By definition of $T_P$, there exists $A \leftarrow B_1, \ldots, B_n$ in $\text{ground}_L(P)$ such that:

$$T_P \uparrow (i-1) = B_1, \ldots, B_n.$$ \hfill (4.2)

We now observe that, by Theorem 4.1.20,

$$\text{Post} \supseteq M_P^L \cap Pre \supseteq T_P \uparrow (i-1) \cap Pre.$$ 

Since $\vdash \{Pre\} \{Post\}$ hold, by a simple induction on $n$, this and (4.2) imply:

$$\text{Post} \models B_1, \ldots, B_n.$$ 

In addition, $A \in T_P \uparrow i \setminus T_P \uparrow (i-1)$ implies that for $k \in [1, n]$:

$$|A| = i > i - 1 \geq |B_k|.$$ 

\textbf{if)} We only show that $\text{Post} \cap Pre \subseteq M_P^L$, as the converse inclusion follows directly from Theorem 4.1.20. Consider $A \in \text{Post} \cap Pre$ and a maximal tree $T$ such that:

- $A$ is the root;
- if $B$ is a node such that $B \in \text{Post} \cap Pre$ and $B_1, \ldots, B_n$ are its children then $B \leftarrow B_1, \ldots, B_n$ in $\text{ground}_L(P)$ and

$$\text{Post} \models B_1, \ldots, B_n \quad \land \quad |B| > \max_{i=1}^n |B_i|.$$ 

Since $W$ is well-founded, there is no infinite branch: by the König lemma, the tree is finite. Since $\vdash \{Pre\} \{Post\}$ and $A \in Pre$, it is readily checked by induction that every atom in $T$ is in $\text{Post} \cap Pre$. If a leaf $B$ is not a ground instance of a fact in $P$, then, by hypothesis, there exists $B \leftarrow B_1, \ldots, B_n$ in $\text{ground}_L(P)$ such that $\text{Post} \models B_1, \ldots, B_n$ and $|B| > \max_{i=1}^n |B_i|$, thus $T$ is not maximal. In conclusion, $T$ is a proof tree for $A$ (see [42]), which implies $A \in M_P^L$.

Therefore, a well-supported postcondition is a subset of the interesting fragment of the least Herbrand model, i.e. well-supportedness combined with the proof relation $\vdash$ is a sound and complete proof method for partial correctness.

\textbf{Corollary 4.1.26} Assume that $\vdash \{Pre\} \{Post\}$ holds. Then $P$ is partially correct w.r.t. the specification $(Pre, Post \cap Pre)$ iff Post is well-supported w.r.t. $P$ and Pre.

\textbf{Proof}. The \textit{if}-part is immediate by Theorems 4.1.25 and 4.1.23. Consider the \textit{onlyif}-part. If $P$ is partially correct then $Post \cap Pre = M_P^L \cap Pre = sp(P, Pre)$, by Theorem 4.1.22. This and Theorem 4.1.25 imply that Post is well-supported.
As a final observation, we point out that when $\vdash \{\text{Pre}\} \ \{\text{Post}\}$ holds, then $sp(P, Pre)$ is well-supported.

In fact, by Theorem 4.1.22, $\vdash \{\text{Pre}\} \ \{sp(P, Pre)\}$ holds, and

$$sp(P, Pre) \cap Pre = sp(P, Pre).$$

Thus, we are in the hypothesis of Theorem 4.1.25.

**Proof Outlines**

Proving weak partial correctness is handy, as the task can be carried out using the proof outlines. In contrast, Definition 4.1.24 may seem intricate and difficult to handle. Fortunately, it has a straightforward interpretation in terms of proof outlines.

**Definition 4.1.27** A proof outline for a clause $A \leftarrow A_1, \ldots, A_n$, a function $\mid \mid : B_L \rightarrow W$ into a well-founded poset $(W, <)$ and $Pre, Post$, is a labeled clause of the form:

$$
\begin{align*}
\{g\} & \leftarrow \{t_0\} \\
A_0 & \leftarrow \{t_1\} \\
A_1 & \leftarrow \{f_1\} \\
\vdots & \leftarrow \\
A_{n-1} & \leftarrow \{t_{n-1}\} \\
A_n & \leftarrow \{f_{n-1}\} \\
A_n & \leftarrow \{t_n\} \\
A_n & \leftarrow \{f_n\}
\end{align*}
$$

where $t_i$ for $i \in [0, n]$ and $f_i, g$, for $i \in [1, n]$ are respectively expressions over $W$ and assertions (in some formal logic), such that every ground instance of the following proof obligations holds:

(i) for $i \in [0, n]$: $g \Rightarrow t_i = \mid A_i\mid$,
(ii) for $i \in [1, n]$: $g \land f_i \Rightarrow A_i \in Post$,
(iii) for $i \in [1, n]$: $g \Rightarrow f_i \land t_0 > t_i$.

The assertion $g$ expresses a relation among the variables of the clause in such a way that for every ground instance of the proof outline if the instance of $g$ holds then $A_i \in Post \land \mid A_0\mid > \mid A_i\mid$. The use of the auxiliary assertions $f_i$’s is not strictly necessary, albeit useful in constructing proof outlines.

Next, in order to prove that $Post$ is well-supported w.r.t. $P$ and $Pre$ we have to show that there exist proof outlines for (instances of) clauses from $P$ and a function $\mid \mid : B_L \rightarrow W$ such that every atom $A$ in $Post \cap Pre$ is a ground instance of some head atom in a proof outline and the assertion associated with the head holds.
Example 4.1.28 (Lexicographic Ordering) Consider the following program LEXORD, specifying a lexicographic ordering relation over pairs of natural numbers.

\begin{align*}
(l1) & \quad \preceq ( [X, Y], [X, Y] ). \\
(l2) & \quad \preceq ( [X, Y], [s(U), V] ) \Leftarrow \\
& \quad \quad \preceq ( [X, Y], [U, Y] ). \\
(l3) & \quad \preceq ( [X, Y], [X, s(V)] ) \Leftarrow \\
& \quad \quad \preceq ( [X, Y], [X, V] ).
\end{align*}

For notational convenience, we shall identify the natural number \( a \) with the term \( s^a(0) \). It is readily checked that \( \vdash \{ \text{Pre} \} \text{LEXORD} \{ \text{Post} \} \) holds, where

\begin{align*}
\text{Pre} &= \{ \preceq ([x, y], [u, v]) \mid x, y, u, v \in \mathbb{N} \} \\
\text{Post} &= \{ \preceq ([x, y], [u, v]) \mid x, y, u, v \in \mathbb{N}, (x, y) \leq_{lex} (u, v) \}
\end{align*}

where \( \leq_{lex} \) is the lexicographic ordering relation.

We observe that one would have expected clauses \((l2, l3)\) to be more general. In fact, a natural version of the program is the following:

\begin{align*}
(l1) & \quad \preceq ( [X, Y], [X, Y] ). \\
(l4) & \quad \preceq ( [X, Y], [s(U), V] ) \Leftarrow \\
& \quad \quad \preceq ( [X, Y], [U, Z] ). \\
(l5) & \quad \preceq ( [X, Y], [U, s(V)] ) \Leftarrow \\
& \quad \quad \preceq ( [X, Y], [U, V] ).
\end{align*}

Although \((l2, l3)\) are instances of \((l4, l5)\), we are still in the position to show that LEXORD is partially correct w.r.t. the specification \((\text{Pre}, \text{Post})\), by means of Theorem 4.1.25. We fix \( W = (\mathbb{N} \times \mathbb{N}, \leq_{lex}) \), and define \( | \) as follows:

\[
| \preceq ([x, y], [u, v])| = (u - x, v - y)
\]

where difference over natural numbers is interpreted as a total function, by letting for \( a, b \in \mathbb{N}, a - b = 0 \) if \( a < b \). The proof outlines for well-supportedness follow.

\begin{align*}
(p1) & \quad \{ X, Y \in \mathbb{N} \} \\
& \quad \leq ( [X, Y], [X, Y] ). \quad \{ (0, 0) \}
\end{align*}

\begin{align*}
(p2) & \quad \{ X, Y, U, V \in \mathbb{N} \land X < s(U) \} \\
& \quad \leq ( [X, Y], [s(U), V] ) \Leftarrow (U - X + 1, V - Y) \\
& \quad \quad \leq ( [X, Y], [U, Y] ). \quad \{ (U - X, 0) \} \\
& \quad \quad \{ (X, Y) \leq_{lex} (U, Y) \}
\end{align*}

\begin{align*}
(p3) & \quad \{ X, Y, V \in \mathbb{N} \land Y < s(V) \} \\
& \quad \leq ( [X, Y], [X, s(V)] ) \Leftarrow (0, V - Y + 1) \\
& \quad \quad \leq ( [X, Y], [X, V] ). \quad \{ (0, V - Y) \} \\
& \quad \quad \{ (X, Y) \leq_{lex} (X, V) \}
\end{align*}
Since the decreasing of $\| \|$ is obvious, we need only to show that every atom in $\text{Pre} \cap \text{Post}$ is an instance of some head, and that the associated assertion holds. Let $\leq ((x, y), [u, v])$ in $\text{Pre} \cap \text{Post}$. Then $(x, y) \leq_{\text{lex}} (u, v)$.

By definition of $\leq_{\text{lex}}$, we have that $(x, y) \leq_{\text{lex}} (u, v)$ iff $(x = u \wedge y = v) \vee (x < u) \vee (x = u \wedge y < v)$. Proof outlines $(p1)$, $(p2)$ and $(p3)$ cover respectively $(x = u \wedge y = v)$, $(x < u)$ and $(x = u \wedge y < v)$. Summarizing, $\text{Post}$ is well-supported, and, then, partial correctness follows from Theorems 4.1.25 and 4.1.23, i.e. $\text{Post} = \text{Post} \cap \text{Pre} = sp(\text{LEXORD}, \text{Pre}) = M^L_{\text{LEXORD}} \cap \text{Pre}$.

$\square$

## 4.1.6 Correct and Computed Instances

So far, we stressed the fact that postconditions describe correct and computed instances of the intended queries, as formally stated by Corollary 4.1.18 (ii). However, under certain rather general assumptions, the proposed proof method can be also employed to achieve a full characterization of correct and computed instances of queries. In this section, we present two methods, applicable in different situations.

Given a set of atoms $I$, we define $\text{Min}(I) = \{ A \in I \mid \exists B \in I : B < A \}$, where $<$ is the instantiation ordering. We write $B \leq A$ if $A$ is an instance of $B$; and $B < A$ iff $B \leq A$ and not $A \leq B$, i.e., if $A$ is an instance of $B$ which is not a variant of $B$. We recall that $\Sigma_L$ is the set of function symbols of the underlying language $L$.

### Theorem 4.1.29

Assume that $\vdash \{ \text{Pre} \} P \{ \text{Post} \}$ and $\vdash \{ \text{Pre} \} Q \{ \text{Post} \}$ hold with $\text{Post}$ well-supported, and $\Sigma_L$ contains infinitely many symbols. Called $Q = \{ Q \theta \mid \text{Post} \vdash Q \theta \}$, we have that:

(i) $Q$ is the set of correct instances of $P$ and $Q$, and

(ii) each of the following conditions is sufficient to state that $Q$ is the set of computed instance of $P$ and $Q$:

(a) $\text{Min}(Q) = Q$,

(b) $Q$ is a set of ground queries,

(c) $Q$ is a finite set.

### Proof

(i) Since $\text{Post}$ is well-supported, $\text{Post} \models Q \theta$ iff $sp(P, \text{Pre}) \models Q \theta$ iff $M^P_P \cap \text{Pre} \models Q \theta$. If $P \models Q \theta$ then $M^P_P \cap \text{Pre} \models Q \theta$, and by the equivalences above, $Q \theta \in Q$. On the other hand, if $\text{Post} \models Q \theta$ then $M^P_P \models Q \theta$. Since $\Sigma_L$ is infinite, $M^P_P \models Q \theta \gamma$, where $\gamma$ maps every variable of $Q \theta$ into a distinct ground term with functor not appearing in $P$ or $Q \theta$. $M^P_P \models Q \theta \gamma$ implies $P \models Q \theta \gamma$, and by the Theorem on Constants (see [146]), $P \models Q \theta$.

(ii)/(a) Let $Q'$ be the set of computed instances of $P$ and $Q$. By Theorem 4.1.18(ii) $Q' \subseteq Q$. Conversely, consider $Q' \in Q$. By (i) $P \models Q'$. Therefore, by completeness of LD-resolution there exists $Q'' \in Q'$ more general than $Q'$. As pointed out above $Q'' \in Q$. This and the hypothesis $\text{Min}(Q) = Q$ imply that $Q'$ is a variant of $Q''$. As $Q'$ is closed under the variant-relation, we conclude $Q' \in Q$, and then $Q' = Q$. 

(ii)(b) We observe that the hypothesis (ii)(a) holds. Consider $Q'$ and $Q''$ in $\mathcal{Q}$ such that $Q' \leq Q''$. Since $\mathcal{Q}$ is a set of ground queries, $Q'$ and $Q''$ are ground and then $Q' \leq Q''$ implies $Q' = Q''$. Therefore, for no $Q', Q'' \in \mathcal{Q}$ we have $Q' < Q''$, and then $\text{Min}(\mathcal{Q}) = \mathcal{Q}$.

(ii)(c) Let us show by contrapositive that (ii)(c) $\Rightarrow$ (ii)(b). If $Q' \in \mathcal{Q}$ is not ground then $\text{Post} \models Q'$ implies $\text{Post} \models Q''$ for every ground instance $Q''$ of $Q'$. Since $\Sigma_L$ is infinite, there are infinitely many ground instances of $Q'$, and then $\mathcal{Q}$ is infinite. □

In particular, for atomic queries $Q$, hypothesis (ii)(b) can be rewritten as $Q \subseteq \text{Post}$.

**Example 4.1.30 (Even)** The simple program **EVEN**


even(0).
even(s(s(X))) $\leftarrow$ even(X).

(defined on a language with $\Sigma_L$ infinite) and the query **even(X)** with $\text{Pre} = B_L$, and $\text{Post} = \{\text{even}(s^k(0)) \mid k \geq 0\}$. It is easy to show that $\models \{\text{Pre}\} \text{EVEN} \{\text{Post}\}$ holds, and that $\text{Post}$ is well-supported. Also, we have that $\mathcal{Q} = \{\text{even}(T) \mid \text{Post} \models \text{even}(T)\}$ coincides with $\text{Post}$, and then it is a set of ground queries. By Theorem 4.1.29(ii)(b), we have that $\mathcal{Q}$ is the set of computed instances of **EVEN** and **even(X)**. □

Recently, Apt et al. [11] introduced a method for characterizing the computed instances of queries on the basis of the declarative semantics. The method is developed with reference to the notion of partial correctness presented in Chapter 3. The key notion is that of $(\text{Pre}, \text{Post})$-redundancy-free programs. Roughly, such programs have the property of delivering non-redundant computed instances of the intended queries.

The next result, a slight improvement over [11], shows a sufficient condition that allows us to retrieve the computed instances of the intended queries from $\text{Post}$.

**Theorem 4.1.31** Assume that $\models \{\text{Pre}\} P \{\text{Post}\}$ and $\models \{\text{Pre}\} Q \{\text{Post}\}$ hold with $\text{Post}$ well-supported, and $\Sigma_L$ contains infinitely many symbols. If the following conditions hold:

**SEM1.** If $H \leftarrow B_1, \ldots, B_n$ and $H \leftarrow C_1, \ldots, C_k$ are ground instances of two different clauses in $P$ then:

$$\text{Post} \cap \text{Pre} \not\models H \land B_1, \ldots, B_n \land C_1, \ldots, C_k.$$ 

**SEM2.** If $H \leftarrow B_1, \ldots, B_n$ and $H \leftarrow C_1, \ldots, C_n$ are distinct ground instances of the same clause in $P$ then:

$$\text{Post} \cap \text{Pre} \not\models H \land B_1, \ldots, B_n \land C_1, \ldots, C_k.$$
then the set of computed instances of $P$ and $Q$ is $\text{Min}\{Q\emptyset \mid \text{Post} \models Q\emptyset\}$.

Proof. By Theorem 4.1.20, $M_P^f \cap \text{Pre} \subseteq \text{Post} \cap \text{Pre}$.
Therefore $\text{Post} \cap \text{Pre} \not\models H \land B_1, \ldots, B_n \land C_1, \ldots, C_k$ implies

$$M_P^f \cap \text{Pre} \not\models H \land B_1, \ldots, B_n \land C_1, \ldots, C_k.$$  

Therefore [SEM1] and [SEM2] imply respectively the two conditions assumed in [11, Theorem 7.4] to prove that $P$ is $(\text{Pre, Post})$-redundant-free. Corollary 7.2(iii) from [11] states that under the hypothesis that $\Sigma_L$ contains infinitely many symbols, and $P$ is $(\text{Pre, Post})$-redundant-free, the set of computed instances of $P$ and $Q$ is $\text{Min}\{Q\emptyset \mid M_P^f \cap \text{Pre} \models Q\emptyset\}$. Our conclusion follows by considering that $M_P^f \cap \text{Pre} \models Q\emptyset$ iff $\text{Post} \cap \text{Pre} \models Q\emptyset$, since $\text{Post}$ is well-supported (see Theorems 4.1.25 and 4.1.22), and $\text{Post} \cap \text{Pre} \models Q\emptyset$ iff $\text{Post} \models Q\emptyset$, since $\vdash \{\text{Pre}\} Q \{\text{Post}\}$ holds.

In certain situations the conditions of Theorem 4.1.31 can be ensured by means of syntactic restrictions. Namely, condition SEM1 is obviously implied by condition:

SYN1. If $H_1 \leftarrow B_1, \ldots, B_n$ and $H_2 \leftarrow C_1, \ldots, C_k$ are variable disjoint variants of different clauses in $P$, then $H_1$ and $H_2$ do not unify,

and condition SEM2 is automatically satisfied when:

SYN2. If $H \leftarrow B_1, \ldots, B_n \in P$, then $\text{Var}(B_1, \ldots, B_n) \subseteq \text{Var}(H)$.

where $\text{Var}(X)$ is the set of logic variables appearing in $X$. As an example, the program $\text{EVEN}$ satisfies both SYN1 and SYN2. We refer the reader to [11] for several examples and a comparison of the approach with others related to $S$-semantics.

### 4.1.7 Weak Total Correctness

While proof relation $\vdash$ allows us to reason on (weak) partial correctness, the additional proof obligations of relation $\vdash_t$ are intended to address universal termination via the leftmost selection rule, and then (weak) total correctness. On the other hand, the approach of restricting attention to the atoms in preconditions may facilitate the termination proofs, as will be pointed out in the Related Work section.

Our commitment to the study of the leftmost selection rule is made apparent in the left-to-right propagation of assumptions in proof obligations of Definition 4.1.1. However, it should be observed that no assumption is made on the search strategy. These remarks are formalized by the following results on persistency of $\vdash_t$ and termination.

**Theorem 4.1.32 (Persistency and Termination)**

Assume that $\vdash_t \{\text{Pre}\} P \{\text{Post}\}$ and $\vdash_t \{\text{Pre}\} Q \{\text{Post}\}$ hold by the same level mapping. Then for every LD-resolvent $Q'$ of $P$ and $Q$,

$$\vdash_t \{\text{Pre}\} Q' \{\text{Post}\}$$

holds by the same level mapping.

Moreover, every LD-tree of $P$ and $Q$ is finite.
4.1. Proof Theory

**Proof.** By the formal derivation of relation $\vdash_1$ presented in Section 3.3, we have that $\vdash_1 \{ \text{Pre} \} P \{ \text{Post} \}$ and $\vdash_1 \{ \text{Pre} \} Q \{ \text{Post} \}$ hold by the level mapping $\vdash'$ iff $P$ and $Q$ are acceptable by the extended level mapping $\vdash'$ and $\text{Pre} \rightarrow \text{Post}$, where $\vdash'$ is defined as $\vdash$ on $\text{Pre}$, and $\infty$ otherwise. Therefore, persistency follows by the Persistency Theorem 2.3.12. Moreover, by the Termination Soundness Theorem 2.3.15, every LD-derivation for $P$ and $Q$ is finite. Since the LD-tree is finitely branching, by König’s Lemma, it is are finite. \hfill \Box

An immediate consequence of this result and Theorem 4.1.20 is the Weak Total Correctness Theorem.

**Theorem 4.1.33 (Weak Total Correctness)** If $\vdash_1 \{ \text{Pre} \} P \{ \text{Post} \}$ holds then $P$ is weak totally correct w.r.t. the specification $(\text{Pre}, \text{Post})$. \hfill \Box

**Example 4.1.34 (Preorder Ctd)** Since $\vdash_1 \{ \text{Pre} \, \text{PREORDER} \} \text{PREORDER} \{ \text{Post} \, \text{PREORDER} \}$ holds, then $\text{PREORDER}$ is weak totally correct w.r.t. $(\text{Pre} \, \text{PREORDER}, \text{Post} \, \text{PREORDER})$.

In addition, for every term $T$ whose ground instances are in $\text{Tree}(\alpha, \beta)$, the LD-tree of $\text{PREORDER}$ and $\text{preorder}(T, L)$ is finite. \hfill \Box

In addition, we show a form of completeness of the termination proof method associated with relation $\vdash_1$. First, we adapt the notion of universal left termination proposed in Chapter 2, by taking into account preconditions.

**Definition 4.1.35** For a program $P$ and a set $\text{Pre} \subseteq B_1$, we say that $P$ is Pre-terminating iff for every $A \in \text{Pre}$ the LD-tree for $P$ and $A$ is finite. \hfill \Box

From the next Theorem, we have that, if $\vdash \{ \text{Pre} \} P \{ \text{Post} \}$ holds then $P$ is Pre-terminating iff $\vdash_1 \{ \text{Pre} \} P \{ \text{sp}(P, \text{Pre}) \}$ holds. In other words, the relation $\vdash_1$ is a complete termination proof method, restricted to the triples in $\vdash$.

**Theorem 4.1.36 (Termination Completeness I)** If $\vdash_1 \{ \text{Pre} \} P \{ \text{Post} \}$ holds then $P$ is Pre-terminating. Conversely, $\vdash \{ \text{Pre} \} P \{ \text{Post} \}$ and $P$ Pre-terminating imply $\vdash_1 \{ \text{Pre} \} P \{ \text{sp}(P, \text{Pre}) \}$.

**Proof.** The if part is stated in Theorem 4.1.33. Consider now the only-if part. By Theorem 4.1.22, we have that

$$\vdash \{ \text{Pre} \} P \{ \text{sp}(P, \text{Pre}) \}$$

(4.3)
This result shows that the proof of relation we conclude that total correctness proof method specifically targeted to reason on left termination.

B holds, by using an appropriate level mapping \( | | \). By Lemma 2.3.19, \( P \) is acceptable by an extended level mapping \( | ' | \) and \( M^L_P \). Moreover, if every LD-derivation of \( A \in B_L \) and \( P \) is finite, then \( |A'| \in N \). This allows us to define the level mapping \( | | \) as \( |A| = |A'| \) for \( A \in Pre \) and \( |A| = 0 \) otherwise.

Since (4.3) holds, the conclusion follows by proving that for every \( A \leftarrow B_1, \ldots, B_n \in \text{ground}_L(P) \) and \( i \in [1, n] \), if

\[
\vdash_A \{ Pre \} \; P \; \{ sp(P, Pre) \}
\]

then \( |A| > |B_i| \). Suppose that \( Pre \models A \) and \( sp(P, Pre) \models B_1, \ldots, B_{i-1} \). Since (4.3) holds, \( Pre \models B_i \). Therefore \( |A| = |A'| \) and \( |B_i| = |B_i'| \). Moreover, \( sp(P, Pre) \models B_1, \ldots, B_{i-1} \) implies \( M^L_P \models B_1, \ldots, B_{i-1} \). Therefore, since \( P \) is acceptable by \( | ' | \) and \( M^L_P \), we conclude \( |A| = |A'| > |B_i'| = |B_i| \).

A stronger form of completeness can be derived by the formal derivation of relation \( \vdash \) pursued in Chapter 3.

**Theorem 4.1.37 (Termination Completeness II)** If \( P \) is weak partially correct w.r.t. the specification \((Pre, Post)\) and Pre-terminating, then \( \vdash \{ Pre' \} P \{ Post' \} \) holds for some \( Pre' \supseteq Pre \) and \( Post' \cap Pre \subseteq Post \).

**Proof.** By Lemma 2.3.19, there exists an extended level mapping \( | | \) such that \( P \) is acceptable by \( | | \) and \( M^L_P \). Moreover, for every \( A \in B_L \), \( |A| \in N \) iff every LD-derivation of \( P \) and \( A \) is finite. We now define:

\[
Pre' = \{ A \mid |A| \neq \infty \} \quad Post' = M^L_P \cap Pre'.
\]

Since \( P \) is Pre-terminating, we have \( Pre' \supseteq Pre \). This and the fact that \( P \) is weak partially correct imply:

\[
Post' \cap Pre = M^L_P \cap Pre' \cap Pre = M^L_P \cap Pre \subseteq Post.
\]

We claim that \( \vdash \{ Pre' \} P \{ Post' \} \) holds by using the level mapping \( | ' | \) defined as \( | | \) on \( Pre' \), and 0 elsewhere. Consider now \( A \leftarrow B_1, \ldots, B_n \in \text{ground}_L(P) \).

Let \( i \) be in \([1, n]\). If \( Pre' \models A \land Post' \models B_1, \ldots, B_{i-1} \) then \( |A| \neq \infty \) and \( M^L_P \models B_1, \ldots, B_{i-1} \). Since \( P \) is acceptable by \( | | \) and \( M^L_P \), this implies \( |A'| = |A| > |B_i'| = |B_i| \) and \( Pre' \models B_i \).

Suppose now that \( Pre' \models A \land Post' \models B_1, \ldots, B_n \). Since \( M^L_P \) is a model of \( P \), we conclude that \( M^L_P \cap Pre' \models A \), i.e. \( Post' \models A \).

This result shows that the proof relation \( \vdash \) is at least as expressive as any weak total correctness proof method specifically targeted to reason on left termination.
The intuitive reading of the Theorem is as follows. Let $P$ be any program weak partially correct w.r.t a specification $(Pre, Post)$. Then there exist a specification $(Pre', Post')$ such that $Pre'$ is a larger precondition than $Pre$, and $Post'$ is a finer postcondition than $Post$ relatively to the original precondition $Pre$. In other words, $(Pre', Post')$ is more expressive than $(Pre, Post)$. More importantly, we can prove weak total correctness of $P$ w.r.t. $(Pre', Post')$ by means of the proof relation $\vdash_i$.

### 4.1.8 Total Correctness

First, we state the analogous of Theorem 4.1.22 for relation $\vdash_i$. This fact allows us to use the results of Section 4.1.5, in a simplified form, as tools for proving total correctness.

**Theorem 4.1.38**

\[
\vdash_i \{Pre\} P \{Post\} \quad \vdash_i \{Pre\} P \{M_{Pre}^P \cap Pre\}.
\]

The proof is analogous to the one of Theorem 4.1.22. An immediate consequence of Theorem 4.1.38 is that $\vdash_i \{Pre\} P \{Post\}$ implies the total correctness of $P$ w.r.t. the specification $(Pre, sp(P, Pre))$.

**Theorem 4.1.39 (Total Correctness)** If $\vdash_i \{Pre\} P \{Post\}$ holds, then $P$ is totally correct w.r.t. the specification $(Pre, sp(P, Pre))$.

**Proof.** By Theorem 4.1.38, $\vdash_i \{Pre\} P \{sp(P, Pre)\}$ holds, since $sp(P, Pre) = M_{Pre}^P \cap Pre$. From this and Theorem 4.1.33, we have that $P$ is weak totally correct w.r.t. $(Pre, sp(P, Pre))$. Again by the relation $sp(P, Pre) = M_{Pre}^P \cap Pre$, we conclude that $P$ is totally correct w.r.t. $(Pre, sp(P, Pre))$. 

**Well-supported interpretations**

The notion of well-supported interpretation becomes simpler when considering relation $\vdash_i$. In the particular case that $Post \subseteq Pre$, it boils down to the well-known notion of supported interpretation (see e.g. [9]). As the next Theorem shows, the proof obligations on the level mapping used to prove $\vdash_i \{Pre\} P \{Post\}$ implicitly satisfy part of the requirements of Definition 4.1.24. As a consequence, we obtain a simpler proof method.

**Theorem 4.1.40** Assume that $\vdash_i \{Pre\} P \{Post\}$ holds. Then the following statements are equivalent:

(i) $Post \cap Pre = sp(P, Pre)$,
(ii) \( T_P(\text{Post}) \supseteq \text{Post} \cap \text{Pre}, \)

(iii) \( \text{Post} \) is well-supported (w.r.t. \( P \) and \( \text{Pre} \)).

**Proof.** (i) \( \rightarrow \) (ii) and (iii) \( \rightarrow \) (i) follow from Theorem 4.1.25. Let us prove (ii) \( \rightarrow \) (iii). We show that \( \text{Post} \) is a well-supported interpretation by considering the level mapping \( \| \| \) used to prove \( \vdash_1 \{\text{Pre}\} P \{\text{Post}\} \). Consider \( A \in \text{Post} \cap \text{Pre} \). Since \( T_P(\text{Post}) \supseteq \text{Post} \cap \text{Pre}, \) there exists a clause \( A \leftarrow B_1, \ldots, B_n \in \text{ground}_P(P) \) such that \( \text{Post} \models B_1, \ldots, B_n \). Moreover, since \( \vdash_1 \{\text{Pre}\} P \{\text{Post}\} \), we have that for every \( i \in [1, n] \) \( |A| > |B_i| \). Therefore \( \text{Post} \) is a well-supported interpretation w.r.t. \( P \) and \( \text{Pre} \).

\[
\square
\]

Proof outlines for well-supportedness in the case of total correctness are simpler than those in the case of partial correctness. In practice, proof outlines are now obtained from Definition 4.1.27 simply by not considering the expressions \( t_i \) for \( i \in [0, n] \) and the related proof obligations.

**Example 4.1.41 (Preorder Ctd)** As an example, we report the proof outlines for \( \text{PREORDER} \), omitting those for \( \text{APPEND} \).

\[
\{ \text{true} \}
\]

\[
\text{preorder}(\text{nil}, []) .
\]

\[
\{ \text{true} \}
\]

\[
\text{preorder}(\text{leaf}(X), [X]) .
\]

\[
\{ \text{tree}(X, \text{Left}, \text{Right}) \in \text{Tree}(\alpha, \beta) \land \\
\text{Ls} = [X|\text{As}|]* \text{Bs} \text{ is a preorder traversal of it } \land |\text{As}| = ||\text{Left}|| \}
\]

\[
\text{preorder}(\text{tree}(X, \text{Left}, \text{Right}), \text{Ls}) \leftarrow \\
\quad \text{preorder}(\text{Left}, \text{As}), \\
\quad \{ \text{Left} \in \text{Tree}(\alpha, \beta) \land \text{As} \text{ is a preorder traversal of it } \}
\]

\[
\quad \text{preorder}(\text{Right}, \text{Bs}), \\
\quad \{ \text{Right} \in \text{Tree}(\alpha, \beta) \land \text{Bs} \text{ is a preorder traversal of it } \}
\]

\[
\quad \text{append}([X|\text{As}|], \text{Bs}, \text{Ls}). \\
\quad \{ [X|\text{As}|], \text{Bs} \in \text{GList} \land \text{Ls} = [X|\text{As}|]* \text{Bs} \}
\]

The proof obligations of the last proof outline are fulfilled by noting that the preorder traversal of \( \text{Left} \) coincides with the sublist of \( \text{Ls} \) from position 2 to position \( ||\text{Left}|| + 1 \), i.e. with \( \text{As} \), and similarly for \( \text{Right} \) and \( \text{Bs} \). We now check that every \( \text{preorder} \)-atom in \( \text{Post}_{\text{PREORDER}} \cap \text{Pre}_{\text{PREORDER}} \) is an instance of the head of some clause for which the assertion associated with the clause holds. Assume that \( \text{preorder}(t, \text{ls}) \) is in \( \text{Post}_{\text{PREORDER}} \cap \text{Pre}_{\text{PREORDER}} \). Then:

- either \( t \) is \( \text{nil} \) and \( \text{ls} \) is \([\text{\}]\). This case is covered by the first proof outline;
- or \( t \) is \( \text{leaf}(x) \) for some \( x \) and \( \text{ls} \) is \([x]\). This case is covered by the second proof outline;
• or \( t \) is \( \text{tree}(x, \text{left}, \text{right}) \) and \( ls \) is \([x \mid as \] * \( bs \) where \( as \) is a preorder traversal of \( \text{left} \) and \( bs \) is a preorder traversal of \( \text{right} \). In particular, \([as] \) do coincide with \( \| \text{left} \| \), and then this case is covered by the third proof outline.

In conclusion, \( \text{PREORDER} \) is totally correct w.r.t. the specification

\[
(\text{Pre}_{\text{PREORDER}}, \text{Post}_{\text{PREORDER}} \cap \text{Pre}_{\text{PREORDER}}),
\]

and since \( \text{Post}_{\text{PREORDER}} \cap \text{Pre}_{\text{PREORDER}} = \text{Post}_{\text{PREORDER}} \), w.r.t. the specification

\[
(\text{Pre}_{\text{PREORDER}}, \text{Post}_{\text{PREORDER}}).
\]

\[\Box\]

### 4.1.9 Weakest (Liberal) Preconditions

The theory of the weakest the weakest (liberal) preconditions was originally introduced by Dijkstra [71], as an alternative, yet equivalent formulation of Hoare’s logic, more geared to the calculation of assertions and programs. The theory of weakest preconditions was the basis for the systematic development of correct programs first described in [71, 72], and further explained in [84].

Differently from the case of the strongest postconditions, we were unable to provide a simple direct characterization of the weakest (liberal) preconditions. We provide here a characterization of the weakest (liberal) preconditions as ordinal closures of a function \( \vartheta_{P,P_{\text{post}}} \) defined over the lattice of Herbrand interpretations. The results of this work show how tight is the parallel between logic and imperative programming.

**Definition 4.1.42** Let \( P \) be a program, and \( \text{Post} \subseteq B_L \). We define the function \( \vartheta_{P,P_{\text{post}}} : 2^{B_L} \rightarrow 2^{B_L} \) as follows:

\[
\vartheta_{P,P_{\text{post}}}(I) = \{ A \in B_L \mid \forall A \leftarrow B_1, \ldots, B_n \in \text{ground}_i(P) : \\
\forall i \in [1,n]: \text{Post} \models B_1, \ldots, B_{i-1} \Rightarrow I \models B_i \\
\land \text{Post} \models B_1, \ldots, B_n \Rightarrow \text{Post} \models A \}.
\]

The definition of \( \vartheta_{P,P_{\text{post}}} \) is readily derived from the proof relations \( \vdash \) and \( \vdash_t \). In particular, the following fundamental relation holds.

**Lemma 4.1.43** Let \( P \) be a program, and \( (\text{Pre}, \text{Post}) \) a specification. Then

\[
\vdash \{ \text{Pre} \} \ P \ \{ \text{Post} \} \ \text{holds \ \iff \ \text{Pre} \subseteq \vartheta_{P,P_{\text{post}}}(\text{Pre}).}
\]
Proof. We calculate:
\[ \vdash \{ \text{Pre} \} \ P \ \{ \text{Post} \} \]
\[ \Leftrightarrow \quad \forall A \leftarrow B_1, \ldots , B_n \in \text{ground}_L(P) : \]
\[ \forall i \in [1, n] : \text{Pre} \models A \wedge \text{Post} \models B_1, \ldots , B_{i-1} \Rightarrow \text{Pre} \models B_i \]
\[ \wedge \ (\text{Pre} \models A \wedge \text{Post} \models B_1, \ldots , B_n \Rightarrow \text{Post} \models A) \]
\[ \Leftrightarrow \quad \forall A \in \text{Pre} \ \forall A \leftarrow B_1, \ldots , B_n \in \text{ground}_L(P) : \]
\[ \forall i \in [1, n] : \text{Post} \models B_1, \ldots , B_{i-1} \Rightarrow \text{Post} \models B_i \]
\[ \wedge \ \text{Post} \models B_1, \ldots , B_n \Rightarrow \text{Post} \models A \]
\[ \Leftrightarrow \quad \text{Pre} \subseteq \vartheta_{P, \text{Post}}(\text{Pre}). \] 

Let us now study the properties of \( \vartheta_{P, \text{Post}} \).

Lemma 4.1.44 Let \( P \) be a program, and \( \text{Post} \subseteq B_L \). The function \( \vartheta_{P, \text{Post}} \) is monotonic and downward continuous over the lattice \( (2^{B_L}, \subseteq) \). Moreover, there exist \( P \) and \( \text{Post} \) such that \( \vartheta_{P, \text{Post}} \) is not continuous.

Proof. Monotonicity is immediate from Definition 4.1.42. Let us show now that \( \vartheta_{P, \text{Post}} \) is downward continuous. Consider a chain \( \{ I_k \}_{k \geq 0} \) of subsets of \( B_L \). We have to show that \( \vartheta_{P, \text{Post}}(\bigcap_{k \geq 0} I_k) = \bigcap_{k \geq 0} \vartheta_{P, \text{Post}}(I_k) \). Consider now any \( A \in B_L \). We calculate:

\[ A \in \bigcap_{k \geq 0} \vartheta_{P, \text{Post}}(I_k) \]
\[ \Leftrightarrow \quad \forall k \geq 0 : \ A \in \vartheta_{P, \text{Post}}(I_k) \]
\[ \Leftrightarrow \quad \forall k \geq 0 : \]
\[ \forall A \leftarrow B_1, \ldots , B_n \in \text{ground}_L(P) : \]
\[ \forall i \in [1, n] : \text{Post} \models B_1, \ldots , B_{i-1} \Rightarrow I_k \models B_i \]
\[ \wedge \ \text{Post} \models B_1, \ldots , B_n \Rightarrow \text{Post} \models A \]
\[ \Leftrightarrow \quad \forall A \leftarrow B_1, \ldots , B_n \in \text{ground}_L(P) : \]
\[ \forall i \in [1, n] : \text{Post} \models B_1, \ldots , B_{i-1} \Rightarrow \forall k \geq 0 : I_k \models B_i \]
\[ \wedge \ \text{Post} \models B_1, \ldots , B_n \Rightarrow \text{Post} \models A \]
\[ \Leftrightarrow \quad A \in \vartheta_{P, \text{Post}}(\bigcap_{k \geq 0} I_k) \].

Finally, we exhibit a program \( P \) and a set \( \text{Post} \) such that \( \vartheta_{P, \text{Post}} \) is not continuous.

Let \( L \) be the program consisting of the unique clause \( q \leftarrow p(X) \), defined on the language \( L = \langle \{ 0^0, s^1 \}, \{ q^0, p^1 \} \rangle \), and \( \text{Post} = [p(X)]_L \cup \{ q \} \). Consider now the chain \( \{ I_k \}_{k \geq 0} \) where \( I_k = \{ p^j s^{j-1}(0) \} \mid 0 \leq j < k \). We have that \( q \in \vartheta_{P, \text{Post}}(\bigcup_{k \geq 0} I_k) \) albeit \( q \not\in \vartheta_{P, \text{Post}}(I_k) \) for every \( k \). Therefore, \( \vartheta_{P, \text{Post}}(\bigcup_{k \geq 0} I_k) \nsubseteq \bigcup_{k \geq 0} \vartheta_{P, \text{Post}}(I_k) \), and then \( \vartheta_{P, \text{Post}} \) is not continuous. \( \Box \)

An interesting consequence of the monotonicity of \( \vartheta_{P, \text{Post}} \) is that, by applying \( \vartheta_{P, \text{Post}} \) to a set \( \text{Pre} \) such that \( \vdash \{ \text{Pre} \} \ P \ \{ \text{Post} \} \) holds yields a precondition weaker than \( \text{Pre} \).
Corollary 4.1.45 If $\vdash \{Pre\} P \{Post\}$ holds then $\vdash \{\vartheta_{P,Post}(Pre)\} P \{Post\}$ holds.

Proof. By Lemma 4.1.43, $Pre \subseteq \vartheta_{P,Post}(Pre)$. By monotonicity of $\vartheta_{P,Post}$, this implies $\vartheta_{P,Post}(Pre) \subseteq \vartheta_{P,Post}(\vartheta_{P,Post}(Pre))$. Again by Lemma 4.1.43, we get the conclusion. □

We recall the following classical results, which are weak forms of theorems due to Kleene and Tarski [152].

Theorem 4.1.46 Let $f$ be a monotonic function over the lattice $(2^{B_L}, \subseteq)$. Then the greatest fixpoint $\text{gfp}(f)$ and the least fixpoint $\text{lfp}(f)$ exist. Moreover:

(i) if $f$ is downward continuous then $\text{gfp}(f) = f \downarrow \omega = \bigcup I \subseteq f(I)$;
(ii) for every ordinal $\alpha$, $f \uparrow \alpha \subseteq f(f \uparrow \alpha)$;
(iii) for some ordinal $\alpha$, $\text{lfp}(f) = f \uparrow \alpha$.

From the fact that $\vartheta_{P,Post}$ is downward continuous, we can conclude that the greatest fixpoint $\text{gfp}(\vartheta_{P,Post})$ coincides with $\vartheta_{P,Post} \downarrow \omega$, i.e., the downward ordinal closure of $\vartheta_{P,Post}$, and with $\text{wlP}(P, Post)$, i.e., the weakest liberal precondition of $P$ and $Post$.

Theorem 4.1.47 Let $P$ be a program, and $Post \subseteq B_L$. Then

$$\text{wlP}(P, Post) = \text{gfp}(\vartheta_{P,Post}) = \vartheta_{P,Post} \downarrow \omega.$$ 

Proof. We calculate:

$$\text{wlP}(P, Post)$$
$$= \quad \{ \text{Definition 4.1.8 } \}$$
$$\cup_{(P')^c \in Post} \{P\} \{Post\} \ Pre'$$
$$= \quad \{ \text{Lemma 4.1.43 } \}$$
$$\cup_{P' \in Post} \subseteq \vartheta_{P,Post}(P') \ Pre'$$
$$= \quad \{ \text{Theorem 4.1.46 (i) and Lemma 4.1.44 } \}$$
$$\vartheta_{P,Post} \downarrow \omega$$
$$= \quad \{ \text{Theorem 4.1.46 (i) and Lemma 4.1.44 } \}$$
$$\text{gfp}(\vartheta_{P,Post}).$$

It is now legitimate to ask oneself whether there is a relation between the set $\vartheta_{P,Post} \uparrow \omega$, and the proof method based on relations $\vdash$ and $\vdash_t$. A generalization of Corollary 4.1.45 to arbitrary ordinals holds.

Corollary 4.1.48 For every ordinal $\alpha$, $\vdash \{\vartheta_{P,Post} \uparrow \alpha\} P \{Post\}$ holds.
Proof. Since \( \wp_{P, \text{post}} \) is monotonic, by Theorem 4.1.46 (ii) we have that \( \wp_{P, \text{post}} \uparrow \alpha \subseteq \wp_{P, \text{post}}(\wp_{P, \text{post}} \uparrow \alpha) \). By Lemma 4.1.43, this implies that \( \vdash \{ \wp_{P, \text{post}} \uparrow \alpha \} P \{ \text{Post} \} \) holds.

In addition, when \( \alpha = \omega \), a stronger conclusion can be shown.

**Theorem 4.1.49** Let \( P \) be a program, and \( \text{Post} \subseteq B_L \).

Then \( \vdash \{ \wp_{P, \text{post}} \uparrow \omega \} P \{ \text{Post} \} \) holds.

**Proof.** By Corollary 4.1.48 \( \vdash \{ \wp_{P, \text{post}} \uparrow \omega \} P \{ \text{Post} \} \) holds. Let us now show the decreasing of the level mapping defined as follows:

\[
|A| = \min\{i \mid A \in \wp_{P, \text{post}} \uparrow (i+1)\}
\]

for \( A \in \wp_{P, \text{post}} \uparrow \omega \), and \( |A| = 0 \) otherwise. Consider \( A \leftarrow B_1, \ldots, B_n \in \text{ground}_L(P) \) and \( i \in [1, n] \). If \( \wp_{P, \text{post}} \uparrow \omega \models A \land \text{Post} \models B_1, \ldots, B_i-1 \) then \( \wp_{P, \text{post}} \uparrow (|A|+1) \models A \land \text{Post} \models B_1, \ldots, B_i-1 \).

By Definition 4.1.42, \( \wp_{P, \text{post}} \uparrow |A| \models B_i \). This implies \( |A| > |A| - 1 \geq \min\{j : B_i \in \wp_{P, \text{post}} \uparrow (j+1)\} = |B_i| \).

The following result provides a characterization of \( \wp_{P, \text{post}} \uparrow \omega \), by showing that it coincides with the weakest precondition of \( P \) and \( \text{Post} \).

**Theorem 4.1.50** Let \( P \) be a program, and \( \text{Post} \subseteq B_L \). Then

\[
\wp(P, \text{Post}) = \wp_{P, \text{post}} \uparrow \omega.
\]

**Proof.** The inclusion \( \wp(P, \text{Post}) \supseteq \wp_{P, \text{post}} \uparrow \omega \) is an immediate consequence of the definition of \( \wp(P, \text{Post}) \) and Theorem 4.1.49. To prove the converse inclusion, we show that for every \( \text{Pre} \) such that \( \vdash \{ \text{Pre} \} P \{ \text{Post} \} \) holds, we have \( \text{Pre} \subseteq \wp_{P, \text{post}} \uparrow \omega \), hence \( \wp(P, \text{Post}) \subseteq \wp_{P, \text{post}} \uparrow \omega \).

Consider now \( \text{Pre} \) such that \( \vdash \{ \text{Pre} \} P \{ \text{Post} \} \) holds by means of a level mapping \( |.\). We show by induction on \( k \geq 0 \) that:

\[
\{ A \in \text{Pre} \mid |A| = k \} \subseteq \wp_{P, \text{post}} \uparrow (k+1).
\]

Case \( k = 0 \). By Definition 4.1.1, for every \( A \leftarrow B_1, \ldots, B_n \in \text{ground}_L(P) \), we have that the body is empty, i.e. \( n = 0 \) and that \( \text{Post} \models A \). By Definition of \( \wp_{P, \text{post}} \), this implies \( A \in \wp_{P, \text{post}} \uparrow 1 \).

Case \( k > 0 \). Assume \( |A| = k + 1 \) with \( k \geq 0 \). Consider \( A \leftarrow B_1, \ldots, B_n \in \text{ground}_L(P) \). For \( i \in [1, n] \), if \( \text{Post} \models B_1, \ldots, B_i-1 \) then \( B_i \in \text{Pre} \land |A| > |B_i| \), since \( \vdash \{ \text{Pre} \} P \{ \text{Post} \} \) holds.

By inductive hypothesis \( B_i \in \wp_{P, \text{post}} \uparrow (|B_i| + 1) \). By monotonicity of \( \wp_{P, \text{post}} \) and \( |A| > |B_i| \), we have that \( B_i \in \wp_{P, \text{post}} \uparrow |A| \). Finally, if \( \text{Post} \models B_1, \ldots, B_n \) then \( \text{Post} \models A \), as \( A \in \text{Pre} \) and \( \vdash \{ \text{Pre} \} P \{ \text{Post} \} \) holds. Therefore, by Definition
we conclude $A \in \vartheta_{P,\text{post}} \uparrow (|A| + 1)$.

Finally, from (4.5), we have that

$$\text{Pre} = \bigcup_{k \geq 0} \{ A \in \text{Pre} \mid |A| = k \} \subseteq \bigcup_{k \geq 0} \vartheta_{P,\text{post}} \uparrow (k + 1) = \vartheta_{P,\text{post}} \uparrow \omega,$$

and hence the conclusion. $\square$

As an immediate consequence, we have that a minimal level mapping can be characterized in terms of ordinal powers of $\vartheta_{P,\text{post}}$.

**Corollary 4.1.51** Let $P$ be a program such that $\vdash_t \{ \text{Pre} \} P \{ \text{Post} \}$ holds by means of a level mapping $\parallel$. Consider now the level mapping $\parallel$ defined as follows:

$$\|A\| = \min \{ i \mid A \in \vartheta_{P,\text{post}} \uparrow (i + 1) \},$$

for $A \in \text{Pre}$, and $\|A\| = 0$ otherwise. Then $\vdash_t \{ \text{Pre} \} P \{ \text{Post} \}$ holds using $\parallel$, and for every $A \in \text{Pre}$, $|A| \geq \|A\|$.

**Proof.** By Theorem 4.1.50, $\text{Pre} \subseteq \vartheta_{P,\text{post}} \uparrow \omega$. Thus $\parallel$ is well-defined. The proof that $\vdash_t \{ \text{Pre} \} P \{ \text{Post} \}$ holds using $\parallel$ follows by noting that $\vdash_t \{ \text{Pre} \} P \{ \text{Post} \}$ holds and that $\vdash_t \{ \vartheta_{P,\text{post}} \uparrow \omega \} P \{ \text{Post} \}$ holds using a level mapping that coincides with $\parallel$ on $\text{Pre}$ (see Theorem 4.1.49).

Finally, in the proof of Theorem 4.1.50, it is showed that

$$\{ A \in \text{Pre} \mid |A| = k \} \subseteq \vartheta_{P,\text{post}} \uparrow (k + 1).$$

An immediate consequence of this fact is that, for every $A \in \text{pre}$:

$$|A| \geq \min \{ i : A \in \vartheta_{P,\text{post}} \uparrow (i + 1) \} = \|A\|. \quad \square$$

Let us now turn our attention on the least fixpoint $lfp(\vartheta_{P,\text{post}})$. The following example shows that, in general, $lfp(\vartheta_{P,\text{post}}) \neq \vartheta_{P,\text{post}} \uparrow \omega$.

**Example 4.1.52** Consider the program:

\begin{align*}
q & \leftarrow p(X), \\
p(\emptyset), \\
p(s(X)) & \leftarrow p(X),
\end{align*}

defined on $L = \{ \{0^0, s^1\}, \{ q^0, p^1\} \}$ and let $\text{Post} = [p(X)]_L \cup \{ q \}$. We have that for $i \geq 0$, $\vartheta_{P,\text{post}} \uparrow i = \{ \text{P}(s^j(O)) \mid 0 \leq j < i \}$. Therefore, we conclude that:

$$\vartheta_{P,\text{post}} \uparrow \omega \subseteq \vartheta_{P,\text{post}} \uparrow \omega \cup \{ q \} = \vartheta_{P,\text{post}} \uparrow (\omega + 1) = lfp(\vartheta_{P,\text{post}}). \quad \square$$

Finally, the following result clarifies the status of $lfp(\vartheta_{P,\text{post}})$. We recall that a ground $(S)LD$-derivation of a program $P$ and a query $Q$ is a $(S)LD$-derivation for $\text{ground}_L(P)$ and $\text{ground}_L(Q)$.
Theorem 4.1.53 Let $P$ be a program, and $\text{Post} \subseteq B_L$. Then

$$\vdash \{ lfp(\vartheta_{P,\text{Post}}) \} P \{ \text{Post} \}$$

holds. Moreover, every ground LD-derivation of $P$ and any $A \in lfp(\vartheta_{P,\text{Post}})$ is finite.

Proof. By Theorem 4.1.46 (iii), there exists an ordinal $\alpha$ such that $lfp(\vartheta_{P,\text{Post}}) = \vartheta_{P,\text{Post}} \uparrow \alpha$. By Corollary 4.1.48, $\vdash \{ lfp(\vartheta_{P,\text{Post}}) \} P \{ \text{Post} \}$ holds. Consider $A \in lfp(\vartheta_{P,\text{Post}})$ and let $\xi$ be a ground LD-derivation of $P$ and $A$. We denote by $\alpha(A)$ the minimum ordinal $\alpha$ such that $A \in \vartheta_{P,\text{Post}} \uparrow \alpha$.

Consider now an atom $B_i$ with $i \in [1, n]$ such that $A \leftarrow B_1, \ldots, B_n \in \text{ground}_e(P)$ and $B_i$ is eventually selected in $\xi$. We show by induction on $i$ that $\vartheta_{P,\text{Post}} \uparrow \alpha(B_i) \subseteq lfp(\vartheta_{P,\text{Post}})$ and that $\alpha(B_i) < \alpha(A)$.

If $i = 1$ then by Definition 4.1.42, $B_1 \in \vartheta_{P,\text{Post}} \uparrow \alpha(B_1) \subseteq lfp(\vartheta_{P,\text{Post}})$ with $\alpha(B_1) < \alpha(A)$.

If $i > 1$ then, by inductive hypothesis, we have that $lfp(\vartheta_{P,\text{Post}}) \models B_1, \ldots, B_{i-1}$. Since $B_1, \ldots, B_{i-1}$ has a ground LD-refutation, and $\vdash \{ lfp(\vartheta_{P,\text{Post}}) \} P \{ \text{Post} \}$ holds, by weak partial correctness of $\vdash$, we have that $\text{Post} \models B_1, \ldots, B_{i-1}$.

Then, by Definition 4.1.42, $B_i \in \vartheta_{P,\text{Post}} \uparrow \alpha(B_i) \subseteq lfp(\vartheta_{P,\text{Post}})$ with $\alpha(B_i) < \alpha(A)$. In conclusion, since the $<$ ordering on ordinals is well-founded, there is no infinite descending chain, i.e. infinitely many selected atoms. Therefore, every ground LD-derivation $\xi$ is finite.

Unfortunately, even for small programs, the direct construction of the weakest (liberal) preconditions may result in a complex task. The origin of this complication lies in the fact that logic programming in an untyped language, so the weakest (liberal) preconditions may contain unintended atoms. We believe, however, that in the context of a typed logic programming language, the theory of weakest preconditions presented in this section may be fruitful for practical calculations (in the style of Hoare’s logic), or for inferring larger preconditions starting from a given one.

4.2 Applications

In this section, we show how the proof methods based on relations $\vdash$ and $\vdash_t$ (or some variant of them) are of practical use for real applications. First, by appropriately defining $\text{Pre}$ and $\text{Post}$ for arithmetic built-in’s, the proof method can be used to show the absence of run-time errors due to the selection of ill-typed arithmetic atoms. Second, we introduce a relation $\text{semi} - \vdash_t$ which, although equivalent to $\vdash_t$, is more suitable for proving correctness in a modular way. Also, we provide a sufficient condition, in the form of restrictions to the admissible $\text{Pre}$ and $\text{Post}$, which ensures the safe omission of the occur-check in the unification algorithm. Finally, we briefly overview some applications of the proof method to the verification of meta-programs, and some results on semantics decidability.
4.2.1 Arithmetic Built-in’s

We have introduced syntax and operational semantics of programs with arithmetic in Section 2.6. In particular, here, we define natural pre- and postconditions for the arithmetic built-in’s. For the predicate $\succ$, we define:

$$\text{Pre}_{\succ} = \{ n > m \mid n, m \in \text{Gae} \}$$

$$\text{Post}_{\succ} = \{ n > m \mid n, m \in \text{Gae} \land \text{value}(n) > \text{value}(m) \},$$

and similarly for $\prec$, $\equiv$, $\neq$, $\neq$, $\geq$. Notice that $\text{Pre}_{\succ} \subseteq \text{Post}_{\succ}$. For the $\equiv$ predicate, we define: index $\text{Pre}_{\equiv}$

$$\text{Pre}_{\equiv} = \{ n \equiv m \mid m \in \text{Gae} \}$$

$$\text{Post}_{\equiv} = \{ \text{value}(m) \equiv m \mid m \in \text{Gae} \}.$$

As discussed by Apt [10], it is not possible to reason in a declarative way on runtime arithmetic errors within the logic programming theory. In particular, the Lifting Lemma does not hold for programs with arithmetic.

Example 4.2.1 (Part) Consider the program $\text{PART}$ from Example 2.6.1:

$$\text{part}(X, [Y|Xs], [Y|Ls], Bs) \leftarrow$$

$$X \succ Y, \text{part}(X, Xs, Ls, Bs).$$

$$\text{part}(X, [Y|Xs], Ls, [Y|Bs]) \leftarrow$$

$$X \equiv Y, \text{part}(X, Xs, Ls, Bs).$$

$$\text{part}(X, [], [], []).$$

for partitioning a list of gae’s. We recall that $\text{PART}$ (implicitly) contains clauses in $P_{\succ}$ and $P_{\equiv}$. The query $\text{part}(3, [2,3,4], Ls, Bs)$ has the computed instance $\text{part}(3, [2,3,4], [2], [3,4])$, while the query $\text{part}(X, [2,3,4], Ls, Bs)$ has only LD-derivations ending in arithmetic errors. □

Suppose now that $\vdash \{ \text{Pre} \} P \{ \text{Post} \}$ and $\vdash \{ \text{Pre} \} Q \{ \text{Post} \}$ hold. By Corollary 4.1.18 (i), for every selected atom $A$, we have that $\text{Pre} \models A$. This suggests us a simple conditions to prevent the selection of ill-typed arithmetic atoms, consisting of imposing that if $\text{Pre} \models A$ holds for an arithmetic atom $A$ then $A$ is correctly typed. For instance, if $n > m$ is selected and $\text{Pre}$ coincides on $\succ$-atoms with the set $\text{Pre}_{\succ}$, then $\text{Pre} \models n > m$ implies that $n, m$ are gae’s, under the weak hypothesis that there exists at least one symbol $f$ in $\Sigma_{\text{Ar}}$ that does not belong to $\Sigma_{\text{Ar}}$. In fact, we notice that $n > m$ is ground, otherwise by instantiating the variables of $n, m$ with a ground term containing $f$ we get two terms that are not gae’s. Since $n > m$ is ground, we have that $n > m \in \text{Pre}$ and then $n, m$ are gae’s.
Example 4.2.2 (Part Ctd) Considering again \textsc{part}, we define the pre- and post-condition as follows:

\[
\begin{align*}
\text{Pre}_{\text{part}} &= \text{Pre}_> \cup \text{Pre}_< \cup \{\text{part}(x, xs, ls, bs) \mid x \in \text{Gae} \land xs \in \text{List}(	ext{Gae})\} \\
\text{Post}_{\text{part}} &= \text{Post}_> \cup \text{Post}_< \cup \{\text{part}(x, xs, ls, bs) \mid x \in \text{Gae} \land xs, ls, bs \in \text{List}(	ext{Gae}) \land ls < x \geq bs\}
\end{align*}
\]

By \(ls < x \geq bs\) we mean that every element in the list \(ls\) (resp., \(bs\)) is smaller (resp., greater or equal) than \(x\). It is readily checked that \(\vdash \{\text{Pre}_{\text{part}}\} \text{ part} \{\text{Post}_{\text{part}}\}\). Therefore, when an arithmetic atom \(n > m\) is selected in a LD-derivation for \textsc{part} and a query \(Q\) such that \(\vdash \{\text{Pre}_{\text{part}}\} Q \{\text{Post}_{\text{part}}\}\), we have:

\[
\text{Pre}_> \models n < m. \tag{4.6}
\]

As discussed above, this implies that \(n, m\) are gae’s and a fortiori that the LD-derivation does not end in an error. \(\square\)

We generalize this reasoning by means of the following definition.

Definition 4.2.3 Let \(P\) be a program with arithmetic, and \(L\) such that \(\Sigma_L \setminus \Sigma_{Ar} \neq \emptyset\). We write \(\vdash \{\text{Pre}\} P \{\text{Post}\}\) if \(\vdash \{\text{Pre}\} P \{\text{Post}\}\) holds for \(P\) as a logic program, and for every arithmetic predicate \(\text{op}\) appearing in \(P\), the sets of \(\text{op}\)-atoms in \(\text{Pre}\) and in \(\text{Post}\) coincide with \(\text{Pre}_{\text{op}}\) and \(\text{Post}_{\text{op}}\), respectively. \(\square\)

Under the hypothesis of Definition 4.2.3, absence of run-time errors can be shown.

Lemma 4.2.4 Assume that \(\vdash \{\text{Pre}\} P \{\text{Post}\}\) and \(\vdash \{\text{Pre}\} Q \{\text{Post}\}\) hold for a program with arithmetic \(P\) and a query \(Q\). Then no LD-derivation for \(P\) and \(Q\) ends in an error. \(\square\)

Since LD-trees of programs with arithmetic are still finitely branching, by the Lemma above the Lifting Lemma and the Strong Completeness Theorem for (S)LD-resolution extend to programs with arithmetic \(P\) such that \(\vdash \{\text{Pre}\} P \{\text{Post}\}\) holds. As a consequence, the proof theory of Section 4.1 and the related results can be generalized to programs with arithmetic.

4.2.2 Modular Verification

The definition of relation \(\vdash_{t}\) has a major drawback, due to the lack of expressiveness of level mappings in modular correctness proofs. We introduce the problem with an example.

Example 4.2.5 (Sublist) Consider the program \textsc{sublist}. 
sublist(Xs, Ys) ← Xs is a sublist of Ys

sublist(Xs, Ys) ←
  append(\_, Zs, Ys),
  append(Xs, \_, Zs).

augmented by the \texttt{APPEND} program.

A level mapping such that:

\[ |\text{append}(xs, ys, zs)| = |zs| \]

is a natural candidate to show \( \vdash_i \{ \text{Pre}'_{\text{APPEND}} \} \text{APPEND} \{ \text{Post}_{\text{APPEND}} \} \), where:

\[
\text{Pre}'_{\text{APPEND}} = \{ \text{append}(xs, ys, zs) \mid zs \in \text{GList} \}.
\]

Analogously, when considering the specification for \texttt{SUBLIST}:

\[
\begin{align*}
\text{Pre}_{\text{SUBLIST}} &= \{ \text{sublist}(xs, ys) \mid ys \in \text{GList} \} \cup \text{Pre}'_{\text{APPEND}} \\
\text{Post}_{\text{SUBLIST}} &= \{ \text{sublist}(xs, ys) \mid xs \text{ sublist of } ys \in \text{GList} \} \\
&\cup \text{Post}_{\text{APPEND}}
\end{align*}
\]

an intuitively correct level mapping is such that \( |\text{sublist}(xs, ys)| = |ys| \). However, the proof obligations of Definition 4.1.1 require that:

\[ |\text{sublist}(xs, ys)| > |\text{append}(\_, zs, ys)| = |ys|. \]

Therefore, to show \( \vdash_i \{ \text{Pre}_{\text{SUBLIST}} \} \text{SUBLIST} \{ \text{Post}_{\text{SUBLIST}} \} \), we must consider a somewhat unnatural level mapping such as \( |\text{sublist}(xs, ys)| = |ys| + 1 \). Unfortunately, such a phenomena propagates upward when considering programs which use \texttt{SUBLIST}, giving rise to counter-intuitive level mappings and preventing modular program development.

A relation \textit{semi} \(-\vdash\) is introduced in [129] following the approach of [7], which addresses the modularity problems shown above. For two predicate symbols \( p \) and \( q \), we write \( p \sqsupset q \) if \( p \) uses \( q \) in its definition, but \( p \) and \( q \) are not mutually recursive; we write \( p \simeq q \) if \( p \) and \( q \) are mutually recursive. \( \text{rel}(A) \) denotes the predicate symbol of the atom \( A \).

**Definition 4.2.6** Given a program \( P \) and a specification \( (\text{Pre}, \text{Post}) \), we write \( \text{semi} - \vdash_i \{ \text{Pre} \} P \{ \text{Post} \} \) if there exists a level mapping \( \models \) such that for every \( A \leftarrow B_1, \ldots, B_n \in \text{ground}_{L}(P) \)

1. for \( i \in [1, n] \):
   \[
   \text{Pre} \models A \land \text{Post} \models B_1, \ldots, B_{i-1} \Rightarrow
   \]
(a) \( \text{Pre} \models B_i \quad \text{and} \)
(b) \(|A| > |B_i| \) if \( \text{rel}(A) \preceq \text{rel}(B) \)
\(|A| \geq |B_i| \) if \( \text{rel}(A) \sqsubseteq \text{rel}(B) \)

2. \( \text{Pre} \models A \land \text{Post} \models B_1, \ldots, B_n \Rightarrow \text{Post} \models A. \)

In contrast to Definition 4.1.1, in (b) we now distinguish two cases depending whether \( \text{rel}(A) \) and \( \text{rel}(B_i) \) are (or not) mutually recursive predicates. If they are mutually recursive, a strict decreasing is imposed.

**Example 4.2.7 (Sublist Ctd)** Consider the \( \text{SUBLIST} \) program again. By defining:

\[
|\text{append}(xs, ys, zs)| = |zs|
\]
\[
|\text{sublist}(xs, ys)| = |ys|
\]

we have that \( \text{semi} \vdash \{ \text{Pre}_{\text{SUBLIST}} \} \text{SUBLIST} \{ \text{Post}_{\text{SUBLIST}} \} \) holds. In fact, since \( \text{sublist} \) and \( \text{append} \) are not mutually recursive, the decreasing of the level mapping has not to be strict.

In [129], it is shown that relations \( \vdash \) and \( \text{semi} \vdash \) coincide, i.e.:

\( \vdash \{ \text{Pre} \} P \{ \text{Post} \} \) holds \iff \( \vdash \{ \text{Pre} \} P \{ \text{sp}(P, \text{Pre}) \} \) holds.

This result allows us to extend all of the properties and tools (such as proof outlines) of triples in relation \( \vdash \) to triples in \( \text{semi} \vdash \). The wide applicability of relation \( \text{semi} \vdash \) is supported by several results on modular program verification. For reasons of space, we do not included in this thesis those results. Rather, we refer the reader to Pedreschi and Ruggieri [129] for methods to prove that a triple \( \{ \text{Pre} \} P \cup P' \{ \text{Post} \} \) is in a relation \( \vdash \), \( \text{semi} \vdash \) starting from proofs that triples for \( P \) and \( P' \) are in the same relation.

### 4.2.3 The Occur-check Problem

Most of Prolog interpreters omit the occur-check in the unification algorithm for efficiency reasons: unfortunately, this means that correctness of LD-resolution without the occur-check is lost. However, some papers [8, 41, 65] propose sufficient conditions that show how the omission of the occur-check is safe for many practical programs and queries.

First, we introduce some basic definitions, including the notion of \( \text{moding} \), already introduced in an intuitive manner in Example 2.8.2.

**Definition 4.2.8**
Consider a n-ary predicate symbol \( p \). A mode for \( p \) is a function \( d_p \) from \( \{1, \ldots, n\} \) to \( \{+, -\} \). If \( d_p(i) = + \) we call \( i \) an input position. If \( d_p(i) = - \) then \( i \) is called an output position (with respect to \( d_p \)). Usually, \( d_p \) is written in the form \( p(d_p(1), \ldots, d_p(n)) \).

An atom is called output-linear if the family of terms which occur in its output positions is linear, i.e. no variable occurs twice in the family.

A pair of atoms \((A, B)\) is NSTO (not subject to occur-check) if in every computation of the Martelli-Montanari algorithm the occur-check yields false.

We say that the omission of the occur-check is safe for a program \( P \) and an atom \( A \) if for every head \( B \) of a (variable disjoint with \( A \)) clause of \( P \), the pair \((A, B)\) is NSTO.

A sufficient condition is stated in [8] which can be integrated within the proof theory based on relation \( \vdash \) in order to show the safe omission of the occur-check along a LD-derivation.

**Theorem 4.2.9** Assume that \( \vdash \{\text{Pre}\} P \{\text{Post}\} \) and \( \vdash \{\text{Pre}\} Q \{\text{Post}\} \) hold, and consider a set \( \Pi \subseteq \Pi_L \) of predicate symbols such that:

(i) for every atom \( A \) such that \( \text{rel}(A) \in \Pi \), if \( \text{Pre} \models A \) then only ground terms appear in the input positions of \( A \), and

(ii) the head of every clause from \( P \) whose predicate symbol is in \( \Pi \) is output linear.

Then for every atom \( A \) such that \( \text{rel}(A) \in \Pi \) and selected in a LD-derivation for \( P \) and \( Q \), the omission of occur-check in the Martelli-Montanari unification algorithm is safe.

**Proof.** See [133, Theorem 4.9].

**Example 4.2.10** (Preorder) Consider the \texttt{PREORDER} program, where the predicate \texttt{preorder} is modeled \texttt{preorder}(+, -). It is readily checked that the head of every clause defining \texttt{preorder} is output linear.

Moreover, consider \( \text{Pre}_{\text{PREORDER}} \) defined for binary trees \( \text{Tree}(\alpha, \beta) \) such that \( \alpha, \beta \) are sets of constants. Then \( \text{Pre}_{\text{PREORDER}} \models \text{preorder}(t, \text{ls}) \) implies that \( t \) is ground.

Therefore, Theorem 4.2.9 allows us to state that the omission of the occur-check is safe when a \texttt{preorder}-atom is selected in a LD-derivation for \texttt{PREORDER} and a query \texttt{preorder}(t, x) with \( t \in \text{Tree}(\alpha, \beta) \).

In general, however, not all interesting programs can be reasoned about by the method above, and for some the presence of the occur-check is essential.
Example 4.2.11 (Preorder Ctd) In the previous example, we were forced to assume that α, β are sets of constants in order to satisfy hypothesis (ii) of Theorem 4.2.9. As a consequence, we can say nothing about the omission of the occur-check in LD-derivations for the queries:

\[
\begin{align*}
\text{preorder}(\text{tree}(\text{leaf}(X), \text{nil}, \text{nil}), \gamma) \\
\text{preorder}(\text{tree}(\text{leaf}(X), \text{nil}, \text{nil}), \chi).
\end{align*}
\]

In the first case, the omission of the occur-check is safe, while in the second one it is not. □

Apt and Pellegrini [8] showed a program transformation for \(P\) that produces an equivalent program \(P'\) for which the occur-check is necessary only in the unification of atoms with a particular predicate \(\neq\). We refer the reader to Chapter 7 for a general method of reasoning about Prolog programs in absence of the occur-check. The method is based on the observation that Prolog without occur-check can be viewed as an instance of the Constraint Logic Programming Scheme.

### 4.2.4 Meta-interpreters

Meta-circular interpreters have been introduced as a fundamental feature of advanced programming languages. Since the early studies, many meta-interpreters have been proposed and proved correct with respect to their intended behavior. However, the task of proving correctness has been largely performed using ad-hoc techniques, depending case by case on the semantics, the particular meta-program and the range of properties one was interested in verifying.

In Chapter 5, a general criterion is introduced for reasoning about meta-interpreters. The basic idea is to apply the general purpose verification methods based on relations \(\vdash\) and \(\vdash_i\) to the case study of the Vanilla meta-interpreter, and, more generally, to generic meta-interpreters, by relating the pre- and postconditions of the object program to those of the meta-program. Under certain natural assumptions, all interesting verification properties lift up from the object program to Vanilla.

### 4.2.5 Semantics Decidability

The semantics decidability issue has been largely investigated in the literature with respect to the \(M\)-semantics [24]. In our context, as a by-result of Theorem 4.1.32, if \(\vdash_i \{\text{Pre}\} P \{\text{Post}\}\) holds then it is decidable for every \(A \in \text{Pre}\) whether \(A \in M(P)\).

In Chapter 6, the decidability problem of the \(C\), and \(S\)-semantics will be investigated. A Prolog implementation of a decision procedure will be presented for a class of logic programs including programs \(P\) such that \(\vdash_i \{B_L\} P \{\text{Post}\}\) holds for some \(\text{Post}\). Moreover, semantics decidability and program testing are shown to be strongly
related, and, in practice, the proposed decision procedure is the core of an approach to testing and debugging of logic programs. We generalize those decidability results to programs such that $\vdash_t \{Pre\} P \{Post\}$ holds.

**Theorem 4.2.12** Assume that $\vdash_t \{Pre\} P \{Post\}$ holds, and consider an atom $A$ such that $Pre \models \exists A$. Then

(i) it is decidable whether $A \in C(P)$;
(ii) it is decidable whether $A \in S(P)$.

*Proof.* See [133, Theorem 4.13].

### 4.3 General Logic Programs

General programs and queries are introduced by allowing negated atoms in the body of clauses and queries. In this section, we extend the proof theory to reason on general programs and queries. Many results cannot be lifted directly, due to some well-known problems with extending the logic programming theory to handle negation. In particular, a major difficulty is the incompleteness of the negation as failure rule w.r.t. Clark’s completion semantics $comp(P)$ (see [106]). We partly solve this problem by reasoning on the basis of the $\vdash_t$ relation. This section is structured as follows: first, we consider negated atoms only in queries, dealing with the so called LDNF$^-$-resolution. Then we extend the approach to general programs, by providing a method for (weak) total correctness. As an application, we also obtain a rather general completeness result for LDNF-resolution.

#### 4.3.1 LDNF$^-$-resolution

We start by considering negation only in queries. Following Apt [9], we introduce LDNF$^-$-resolution.

**Definition 4.3.1**

- A LDNF$^-$-derivation is a SLDNF-derivation for a program and a general query, by using the *leftmost* selection rule.
- We write $\vdash_t \{Pre\} Q \{Post\}$ for a general query $Q$ iff there exist a level mapping $j$ and $k \in \mathbb{N}$ such that for every ground instance $L_1, \ldots, L_n$ of $Q$:

$$Post \models L_1, \ldots, L_{i-1} \Rightarrow \begin{cases} Pre \models A_i \land k > |A_i| & \text{if } L_i = A_i \\ Pre \models A_i \land k > |A_i| & \text{if } L_i = \neg A_i \end{cases}$$

- Given a program $P$ and a general query $Q$, we say $P \cup \{Q\}$ *does not flounder* if there is no LDNF$^-$-derivation for $P$ and $Q$ in which a non-ground negative literal is selected.
In the following theorem, the completeness of negation as failure rule is shown for (positive) programs $P$ and general queries $Q$ that are in the $\triangleright_1$ relation.

**Theorem 4.3.2** Let $P$ be a program and $Q$ a general query such that $\trianglerighthook_1 \{\text{Pre}\} P \{\text{Post}\}$ and $\trianglerighthook_1 \{\text{Pre}\} Q \{\text{Post}\}$ hold by the same level mapping. If:

- $P \cup \{Q\}$ does not flounder, and
- $\text{comp}(P) \models Q'$ for $Q'$ instance of $Q$

then there exists a LDNF$^-$-refutation for $P$ and $Q$ with computed instance more general than $Q'$.

**Proof.** First, we point out that by Theorem 4.1.22 $\trianglerighthook_1 \{\text{Pre}\} P \{M^f_\text{P} \cap \text{Pre}\}$ and $\trianglerighthook_1 \{\text{Pre}\} Q \{M^f_\text{P} \cap \text{Pre}\}$ hold. Let us prove that if $\text{comp}(P) \models Q'$ then there exists a LDNF$^-$-refutation for $P$ and $Q'$ with computed instance $Q'$. The proof is by induction on the number $n$ of literals in $Q'$.

*(Base)* If $Q'$ consists of one literal then we distinguish two cases. If $Q' = A$ then by Strong Completeness of SLD-resolution there exists a LD(NF$^-$) refutation for $P$ and $A$. If $Q' = \neg A$ then $A$ is ground since $P \cup \{Q'\}$ does not flounder — otherwise $P \cup \{Q\}$ flounders. Since $\trianglerighthook_1 \{\text{Pre}\} A \{M^f_\text{P} \cap \text{Pre}\}$, by Theorem 4.1.32 the LD-tree for $P$ and $A$ is finite. Moreover, it is finitely failed. Otherwise, by correctness of LDNF$^-$-resolution $\text{comp}(P) \models A$: this is in contradiction with the fact that $\text{comp}(P)$ is consistent. Thus there exists a LDNF$^-$-refutation for $P$ and $Q'$.

*(Step)* We distinguish two cases.

- If $Q' = A, Q''$ then by Strong Completeness of SLD-resolution there exists a LD(NF$^-$) refutation for $P$ and $A$. Moreover, by Corollary 4.1.18 (ii) $M^f_\text{P} \cap \text{Pre} \models A$ and then
  $$\trianglerighthook_1 \{\text{Pre}\} Q'' \{M^f_\text{P} \cap \text{Pre}\}.$$

  The result follows by applying the inductive hypothesis on $Q''$.

- If $Q' = \neg A, Q''$ then $A$ is ground since $P \cup \{Q'\}$ does not flounder — otherwise $P \cup \{Q\}$ flounders. Since $\trianglerighthook_1 \{\text{Pre}\} A \{M^f_\text{P} \cap \text{Pre}\}$, the LD-tree for $P$ and $A$ is finite. Moreover, it is finitely failed, i.e. $A \in FF^f_\text{P}$. Otherwise, by correctness of LDNF$^-$-resolution $\text{comp}(P) \models A$: this is in contradiction with the assumption $\text{comp}(P) \models \neg A, Q''$ and the fact that $\text{comp}(P)$ is consistent. Thus $Q''$ is the LDNF$^-$-resolvent of $Q'$. Moreover, $A \in FF^f_\text{P}$ implies $M^f_\text{P} \cap \text{Pre} \models \neg A$, which in turn implies $\trianglerighthook_1 \{\text{Pre}\} Q'' \{M^f_\text{P} \cap \text{Pre}\}$. The conclusion follows by applying the inductive hypothesis on $Q''$.

Since $P \cup \{Q\}$ does not flounder, we can apply the (LDNF$^-$-version of the) Lifting Lemma to the refutation for $P$ and $Q'$, thus obtaining a LDNF$^-$-refutation for $P$ and $Q$ with computed instance more general than $Q'$. □
4.3.2 LDNF-resolution

The exposition of the approach for general programs is organized as follows: first, we extend to general programs the definitions of (weak) total correctness, and of the proof relation \( \vdash t \). Second, we show some verification properties of \( \vdash t \), including persistency, termination and (weak) total correctness. Therefore, we start by extending Definition 3.1.5.

**Definition 4.3.3** Consider a general program \( P \).

(i) LDNF-resolution is SLDNF-resolution together with the leftmost selection rule.

(ii) We denote with \( M^L_P \) the set of \( A \in B_L \) such that there exists a LDNF-refutation for \( P \) and \( A \), and with \( FF^L_P \) the set of \( A \in B_L \) such that there exists a finitely failed LDNF-tree for \( P \) and \( A \). \( FF^L_P \) is the set \( B_L \setminus FF^L_P \).

(iii) Given a general program \( P \) and a general query \( Q \), we say \( P \cup \{ Q \} \) does not flounder if there is no LDNF-derivation for \( P \) and \( Q \) where a non-ground negative literal is selected.

(iv) \( P \) is weak totally correct w.r.t. a specification \((Pre, Post)\) iff \( M^L_P \cap Pre \subseteq Post \) and \( Pre \subseteq M^L_P \cup FF^L_P \).

(v) \( P \) is totally correct w.r.t. a specification \((Pre, Post)\) iff \( M^L_P \cap Pre = Post \) and \( Pre \subseteq M^L_P \cup FF^L_P \). \( \square \)

Although \( M^L_P \) and \( FF^L_P \) now are not declaratively defined, we will show later that for the class of programs we are interested in they have the expected declarative interpretation. Regarding queries, the definition of relation \( \vdash t \) remains the same as Definition 4.3.1. Relation \( \vdash t \) is extended to general programs as follows.

**Definition 4.3.4** Let \( P \) be a general program, and \((Pre, Post)\) a specification.

- We write \( \vdash t \{Pre\} P \{Post\} \) if there exists a level mapping \( \mid \mid \) such that:
  1. for every \( A \leftarrow L_1, \ldots, L_n \in ground_L(P) \):
     - for \( i \in [1, n] \):
       \[
       Pre \models A \land Post \models L_1, \ldots, L_{i-1} \Rightarrow \begin{cases} 
       Pre \models B_i \land |A| > |B_i| \quad \text{if } L_i = B_i \\
       Pre \models B_i \land |A| > |B_i| \quad \text{if } L_i = \neg B_i 
       \end{cases}
       \]
  2. \( Pre \models A \land Post \models L_1, \ldots, L_n \Rightarrow Post \models A \)

- We say that \( P \) is non-floundering (w.r.t. \( Pre \)) iff for every \( A \in Pre, P \cup \{ A \} \) does not flounder. \( \square \)
It is worth noting that relation $\vdash_t$ for general programs is not a conservative extension of that for logic programs. In general, $\vdash_t \{\text{Pre}\} P \{\text{Post}\}$ for a logic program $P$ does not necessarily imply that $\vdash_t \{\text{Pre}\} P \{\text{Post}\}$ holds for $P$ as a general program. This is due to the additional requirement (ii) of Definition 4.3.4:

$$T_P(\text{Post}) \supseteq \text{Post} \cap \text{Pre}.$$ 

However, we point out that we already met a similar relation: by Theorem 4.1.40, it is equivalent for logic programs to require that $\text{Post}$ is well-supported w.r.t. $P$ and $\text{Pre}$, i.e., that $\text{Post} \cap \text{Pre} = M^L_P \cap \text{Pre}$. Therefore, by Theorem 4.1.38, if $\vdash_t \{\text{Pre}\} P \{\text{Post}\}$ holds for a logic program then

$$\vdash_t \{\text{Pre}\} P \{M^L_P \cap \text{Pre}\}$$

holds considering $P$ as a general program. In other words, when dealing with triples for general programs, we are forced to consider the strongest postconditions. If condition (ii) of Definition 4.3.4 is omitted, then we are not able to show the basic properties of correctness, such as the equivalent of Corollary 4.1.18 (ii) for general programs.

**Example 4.3.5** Consider, as an example the program $P$:

$$p \leftarrow \neg q.$$ 

and the query $\neg q$. Condition (i) holds for $P$ and $\text{Pre} = \text{Post} = \{p, q\}$, and $\vdash_t \{\text{Pre}\} \neg q \{\text{Post}\}$. However, $\text{Post} \not\models \neg q$ even though $\neg q$ is a computed instance of $P$ and $\neg q$. $\square$

**Lemma 4.3.6** Assume that $\vdash_t \{\text{Pre}\} P \{\text{Post}\}$ holds. For every $A \in \text{Pre}$:

(i) If the LDNF-tree for $P$ and $A$ is finitely failed then $\text{Post} \models \neg A$,

(ii) If there is a LDNF-refutation for $P$ and $A$ then $\text{Post} \models A$.

**Proof.** The proof is by a double induction on the rank ($\geq 0$) and on the depth ($\geq 1$) of the finitely failed LDNF-tree in the case (i). In the case (ii), induction is on the rank ($\geq 0$) and on the length ($\geq 1$) of the LDNF-refutation.

rank = 0

$\text{depth/length} = 1$ (i) If $\text{Post} \models A$ then by Definition 4.3.4 (ii) there exists $A \leftarrow L_1, \ldots, L_n \in \text{ground}_L(P)$ such that:

$$\text{Post} \models L_1, \ldots, L_n.$$ 

However, this is impossible since $\text{depth} = 1$ implies that $A$ does not unify with any clause head.
(ii) Since \( \text{length} = 1 \), the hypothesis implies that \( A \) is an instance of the head of a unit clause. By Definition 4.3.4 (i2), we conclude \( \text{Post} \models A \).

\[ \text{Post} \models A \]

Since the resolvent of \( A \) and \( C \) has a finitely failed \( \text{LD(NF)} \)-tree, every instance of its, and in particular \( L_1, \ldots, L_n \), has a finitely failed \( \text{LD(NF)} \)-tree. Therefore there exists \( i \in [1, n] \) such that \( L_1, \ldots, L_{i-1} \) have a refutation and \( L_i \) has a finitely failed \( \text{LD(NF)} \)-tree. Since \( \text{rank} = 0 \), \( L_1, \ldots, L_i \) are positive literals. By inductive hypothesis (ii) on the length of refutations:

\[ \text{Post} \models L_1, \ldots, L_{i-1} \]

and then, since \( \models \{ \text{Pre} \} P \{ \text{Post} \}, \text{Pre} \models L_i \). By inductive hypothesis (i) on the depth \( \text{Post} \not\models L_i \). This contradicts (4.7), thus we conclude \( \text{Post} \models \neg A \).

(ii) Consider the \( \text{LDNF} \)-resolvent of \( A \). Since \( \text{rank} = 0 \), every literal in it is positive. Moreover, some ground instance \( B_1, \ldots, B_n \) of it has a \( \text{LD(NF)} \)-refutation. By inductive hypothesis on the depth, we have:

\[ \text{Post} \models B_1, \ldots, B_n \]

Since \( \models \{ \text{Pre} \} P \{ \text{Post} \} \) holds, by Definition 4.3.4 (i2) this implies \( \text{Post} \models A \).

(i) Since \( \text{rank} = 1 \), then by Definition 4.3.4 (ii) there exists \( C \theta = A \leftarrow L_1, \ldots, L_n \in \text{ground}_L(P) \), with \( C \) clause from \( P \), such that:

\[ \text{Post} \models L_1, \ldots, L_n. \] (4.8)

Since the resolvent of \( A \) and \( C \) has a finitely failed \( \text{LD(NF)} \)-tree, every instance of its, and in particular \( L_1, \ldots, L_n \), has a finitely failed \( \text{LD(NF)} \)-tree. Therefore there exists \( i \in [1, n] \) such that \( L_1, \ldots, L_{i-1} \) have a refutation and \( L_i \) has a finitely failed \( \text{LD(NF)} \)-tree. By inductive hypothesis (ii) on the length of refutations and (i) on the rank:

\[ \text{Post} \models L_1, \ldots, L_{i-1}. \]

Since \( \models \{ \text{Pre} \} P \{ \text{Post} \} \), we have \( \text{Pre} \models A_i \), where \( L_i = A_i \) or \( L_i = \neg A_i \). We distinguish now two cases.

If \( L_i = A_i \), then by inductive hypothesis (i) on the depth \( \text{Post} \not\models L_i \). This contradicts (4.8), thus we conclude \( \text{Post} \models \neg A \).

If \( L_i = \neg A_i \), then \( A_i \) has a \( \text{LD(NF)} \)-refutation with lower rank. By inductive hypothesis (ii) on the rank, we have \( \text{Post} \models A_i \), and then \( \text{Post} \not\models L_i \). This contradicts (4.8), thus we conclude \( \text{Post} \models \neg A \).
(ii) Consider the resolvent of $A$. We observe that some ground instance $L_1, \ldots, L_n$ of it has a LDNF-refutation. By inductive hypothesis (ii) on the length of refutations, every positive literal in $L_1, \ldots, L_n$ is in $Post$. By inductive hypothesis (i) on the rank, we have that every negative literal is true in $Post$, i.e.

$$Post \models L_1, \ldots, L_n.$$ 

Since $\vdash \{\text{Pre}\} P \{\text{Post}\}$ holds, by Definition 4.3.4 (i2), this implies $Post \models A$.  

We associate a finite multiset over $N$ to general queries by extending Definition 2.3.13.

**Definition 4.3.7** Let $Q = L_1, \ldots, L_n$ be a general query such that $\vdash \{\text{Pre}\} Q \{\text{Post}\}$ holds by the level mapping $\mid \mid$. We define the sets $\mathcal{Q}^{|Post|}_P$ for $i \in [1, n]$ as follows:

$$\mathcal{Q}^{|Post|}_P = \{ A^i \mid L'_1, \ldots, L'_n \in \text{ground}_L(Q) \mid Post \models L'_1, \ldots, L'_{i-1}, \text{ and } L'_i = A^i \lor L'_i = \neg A^i \}.$$ 

We define $\mathcal{Q}^{|Post|}_P$ as the finite multiset

$$\mathcal{Q}^{|Post|}_P = \max\{\mathcal{Q}^{|Post|}_P, \ldots, \max^{n}\} \mathcal{Q}^{|Post|}_P.$$ 

where $k$ is the maximum in $[1, n]$ such that $Post \models \exists (L_1, \ldots, L_{k-1})$.  

**Lemma 4.3.8** Assume that $\vdash \{\text{Pre}\} P \{\text{Post}\}$ and $\vdash \{\text{Pre}\} Q \{\text{Post}\}$ hold by the same level mapping $\mid \mid$. For every LDNF-resolvent $Q'$ of $P$ and $Q$:

(i) $\vdash \{\text{Pre}\} Q' \{\text{Post}\}$ holds by $\mid \mid$, and

(ii) $\mathcal{Q}^{|Post|}_P \subseteq_m \mathcal{Q}^{|Post|}_P$.

**Proof.** In the case that a positive literal is selected, the reasoning is similar to the Persistency Lemmas 4.1.17 and 2.3.12 for (i), and to Lemma 2.3.14 for (ii). Therefore, we have only to consider $Q = \neg A, Q'$. In this case, $A$ is ground, there exists a finitely failed LDNF-tree for $P$ and $A$, and $Q'$ is the LDNF-resolvent of $P$ and $Q$. By Lemma 4.3.6 (i), $Post \models \neg A$. From this, (i-iii) readily follow.  

In the following theorem, we extend the properties of persistency and call patterns characterization to general programs. As we will see in an example, call patterns characterization is essential for establishing non-floundering of general programs.

**Theorem 4.3.9 (Persistency, Call and Success Patterns)** Assume that $\vdash \{\text{Pre}\} P \{\text{Post}\}$ and $\vdash \{\text{Pre}\} Q \{\text{Post}\}$ hold by the same level mapping $\mid \mid$. Then:
4.3. General Logic Programs

(i) for every LDNF-resolvent \( Q' \) of \( P \) and \( Q \), \( \vdash \{ \text{Pre} \} Q' \{ \text{Post} \} \) holds by the same level mapping.

(ii) for every literal \( A \) or \( \neg A \) selected in a LDNF-derivation for \( P \) and \( Q \), \( \text{Pre} \models A \);

(iii) for every computed instance \( Q' \) of \( P \) and \( Q \), \( \text{Post} \models Q' \).

Proof. (i) See Lemma 4.3.8 (i).

(ii) Immediate by (i) and Definition 4.3.1.

(iii) Let \( x_1, \ldots, x_n \) be the variables of \( Q \), and \( p \) a fresh predicate symbol of arity \( n \). We define \( \text{Pre}' \) as \( \text{Pre} \cup \{ p(t_1, \ldots, t_n) \mid \text{Post} \models Q(x_i/t_i \mid i \in [1, n]) \} \) and \( \text{Post}' \) as \( \text{Post} \cup p(U_L \times \ldots \times U_L) \). With this assumptions, it is readily checked that

\[
\vdash \{ \text{Pre}' \} P \cup \{ p(x_1, \ldots, x_n) \leftarrow Q \} \{ \text{Post}' \}, \quad \text{and} \quad \vdash \{ \text{Pre}' \} Q, p(x_1, \ldots, x_n) \{ \text{Post}' \}, \quad \text{by fixing the level of any } p(t_1, \ldots, t_n) \text{ to the natural number } k \text{ provided by Definition 4.3.1. By hypothesis, there exists a LDNF-derivation for } P \text{ and } Q, p(x_1, \ldots, x_n) \text{ where } p(t_1, \ldots, t_n) \text{ is selected, with } Q(x_i/T_i \mid i \in [1, n]) \text{ variant of } Q'. \]

By (ii) and the definition of \( \text{Pre}' \), we conclude that \( \text{Post} \models Q(x_i/T_i \mid i \in [1, n]) \), and then \( \text{Post} \models Q' \).

As a consequence, the Termination Theorem 4.1.32 extends to general programs.

**Theorem 4.3.10 (Termination)**

Assume that \( \vdash \{ \text{Pre} \} P \{ \text{Post} \} \) and \( \vdash \{ \text{Pre} \} Q \{ \text{Post} \} \) hold by the same level mapping. Then the LDNF-tree for \( P \) and \( Q \) is finite.

Proof. The proof is by induction on the rank of the LDNF-tree. If the rank is 0, then by Lemma 4.3.8(ii) there cannot be an infinite branch. Since the LDNF-tree is finitely branching, by K"onig's Lemma, it is finite. If the rank is greater than 0, then by inductive hypothesis the rank of a subsidiary tree used in a LDNF-derivation is lower and then, by inductive hypothesis, finite. Moreover, by Lemma 4.3.8(ii) there cannot be an infinite branch. Since the LDNF-tree is finitely branching, by K"onig's Lemma, it is finite.

However, stating that every LDNF-tree of \( P \) and \( Q \) is finite does not necessarily mean that Prolog computation for \( P \) and \( Q \) eventually terminates.

**Example 4.3.11 (Nil)** For the program \textsc{Nil}:

\[
p \leftarrow \neg p.
\]

and the query \( p \), we have that there exists no LDNF-tree. Therefore, every LDNF-tree is finite. In contrast, the Prolog computation runs forever, by trying to build a subsidiary tree for \( p \), each time \( \neg p \) is selected.

Another difference between LDNF-resolution and Prolog is that the latter does not check for floundering. We refer the reader to the paper of Apt and Doets [5] for a discussion on the differences between LDNF-resolution and Prolog.
Here, we restrict ourselves to observe that with the assumptions of Theorem 4.3.10, termination w.r.t. the Prolog computation can be shown. This follows by observing that each time a subsidiary tree for \( \neg A \) is being computed, its root is \( A \), i.e. it is a positive literal. Therefore, there is at least one child, due to the resolution of \( A \) with some clause. Therefore, if there is an infinite sequence \( \{ A_i \}_{i \geq 0} \) of atoms such that \( A_{i+1} \) is the root of a subsidiary tree being computed during the resolution of \( A_i \), then, by Lemma 4.3.8 (ii), \( \beta \{ A_i \} \; \exists_{\alpha} \; \beta \{ A_{i+1} \} \), where \( \beta \) maps queries into bags of natural numbers. We conclude that the sequence above cannot be infinite, since bags of natural numbers is a well-founded order.

**Example 4.3.12 (Nil Ctd)** In particular, \( \vdash_t \{ \text{Pre} \} \; \text{NIL} \; \{ \text{Post} \} \) does not hold for \( (\text{Pre}, \text{Post}) \) such that \( p \in \text{Pre} \), since we would have to show \( |p| > |p| \). \( \square \)

Some relevant properties of relation \( \vdash_t \) are summarized in the following Lemma.

**Lemma 4.3.13** Assume that \( \vdash_t \{ \text{Pre} \} \; P \; \{ \text{Post} \} \) holds and \( P \) is non-floundering. The following statements hold:

(i) \( \text{Pre} \subseteq M^P \cup F F^P \),

(ii) \( M^P \cap \text{Pre} = F F^P \cap \text{Pre} \),

(iii) \( M^P \cap \text{Pre} \subseteq \text{Post} \),

(iv) \( F F^P \cap \text{Pre} \subseteq \text{Post}^c \),

(v) \( \vdash_t \{ \text{Pre} \} \; P \; \{ \text{Post} \cap \text{Pre} \} \).

**Proof.**

(i) For every \( A \in \text{Pre} \), by Theorem 4.3.10 the LDNF-tree for \( P \) and \( A \) is finite. Since \( P \) is non-floundering, then either there exists a LDNF-refutation or the LDNF-tree is finitely failed, i.e. \( A \in M^P \cup F F^P \). Therefore, \( \text{Pre} \subseteq M^P \cup F F^P \).

(ii). The \( \subseteq \) inclusion holds since \( M^P \subseteq F F^P \) by definition of \( M^P \) and \( F F^P \). On the other hand, by (i), \( F F^P \cap \text{Pre} \) is included in \( M^P \).

Therefore \( M^P \cap \text{Pre} = F F^P \cap \text{Pre} \).

(iii) Consider \( A \in M^P \cap \text{Pre} \). By definition of \( M^P \) there exists a LDNF-refutation for \( P \) and \( A \). By Lemma 4.3.6 (ii), \( A \in \text{Post} \). Thus, \( M^P \cap \text{Pre} \subseteq \text{Post} \).

(iv) Consider \( A \in F F^P \cap \text{Pre} \). By definition of \( F F^P \) there exists a LDNF-refutation for \( P \) and \( \neg A \). By Lemma 4.3.6 (ii), \( \text{Post} \vdash \neg A \).

Therefore, \( F F^P \cap \text{Pre} \subseteq \text{Post}^c \).

(v) Definition 4.3.4 (i) holds by reasoning in the same way of Theorem 4.1.16. Let us verify Definition 4.3.4 (ii). We have to show that \( T_p(\text{Post} \cap \text{Pre}) \equiv \text{Post} \cap \text{Pre} \).

Consider \( A \in \text{Post} \cap \text{Pre} \).

Since \( \vdash_t \{ \text{Pre} \} \; P \; \{ \text{Post} \} \) holds, there exists

\[ A \leftarrow L_1, \ldots, L_n \in \text{ground}_t(P) \]

such that \( \text{Post} \vdash L_1, \ldots, L_n \). We show that for \( i \in [1, n] \) \( \text{Post} \cap \text{Pre} \vdash L_i \). Consider two cases.
4.3. General Logic Programs

- If \( L_i \) is a positive literal, say \( B \), then by Definition 4.3.4 (i), \( \text{Pre} \models B \). Therefore \( \text{Post} \cap \text{Pre} \models L_i \).
- If \( L_i \) is a negative literal then \( \text{Post} \models L_i \) implies \( \text{Post} \cap \text{Pre} \models L_i \).

The first consequence of the Lemma is the Weak Total Correctness Theorem for general programs.

**Theorem 4.3.14 (Weak Total Correctness)** If \( \models \{ \text{Pre} \} P \{ \text{Post} \} \) holds and \( P \) is non-floundering then \( P \) is weak totally correct w.r.t. the specification \( (\text{Pre}, \text{Post}) \).

**Proof.** An immediate consequence of Lemma 4.3.13 (i,iii).

Similarly to positive programs, we are in the position to define the notion of strongest postcondition for \( P \) and \( \text{Pre} \) as the intersection of all postconditions \( \text{Post} \) such that \( \models \{ \text{Pre} \} P \{ \text{Post} \} \). We still denote it by \( sp(P, \text{Pre}) \). However, in the case of general programs, the notion of strongest postcondition does not result in a relevant concept. As showed in Theorem 4.1.40, the inclusion \( T_P(\text{Post}) \supseteq \text{Post} \cap \text{Pre} \) required in Definition 4.3.4 is equivalent for positive programs to force \( \text{Post} \cap \text{Pre} = M^P_{\text{Post}} \cap \text{Pre} = sp(P, \text{Pre}) \). This fact extends to general programs, as pointed out by the following theorem.

**Theorem 4.3.15** Assume that \( \models \{ \text{Pre} \} P \{ \text{Post} \} \) holds and \( P \) is non-floundering. Then:

(i) \( M^P_\text{Post} \cap \text{Pre} = \text{Post} \cap \text{Pre} \),
(ii) \( FF^P_\text{Post} \cap \text{Pre} = \text{Post}^* \cap \text{Pre} \),
(iii) \( \models \{ \text{Pre} \} P \{ M^P_\text{Post} \cap \text{Pre} \} \).

**Proof.**
(i) The \( \subseteq \) inclusion is shown in Lemma 4.3.13 (iii). Moreover:

\[
\text{Post} \cap \text{Pre} \\
\{ \text{Lemma 4.3.13 (i)} \} \\
\subseteq \text{Post} \cap ((M^P_{\text{Post}} \cap \text{Pre}) \cup (FF^P_{\text{Post}} \cap \text{Pre})) \\
\{ \text{Distributivity} \} \\
= (\text{Post} \cap M^P_{\text{Post}} \cap \text{Pre}) \cup (\text{Post} \cap FF^P_{\text{Post}} \cap \text{Pre}) \\
\{ \text{Lemma 4.3.13 (iii), (iv)} \} \\
= M^P_{\text{Post}} \cap \text{Pre}.
\]

(ii) This is a direct consequence of (i) and of Lemma 4.3.13 (ii).
(iii) By Lemma 4.3.13 (v), \( \models \{ \text{Pre} \} P \{ \text{Post} \cap \text{Pre} \} \) holds. By (i), \( \text{Post} \cap \text{Pre} \) coincides with \( M^P_{\text{Post}} \cap \text{Pre} \). Therefore, \( \models \{ \text{Pre} \} P \{ M^P_{\text{Post}} \cap \text{Pre} \} \) holds. \( \square \)

We are now in the position to state the Total Correctness Theorem for general programs.
Chapter 4. Verification of Prolog programs

Theorem 4.3.16 (Total Correctness)
If \( \vdash \{ \text{Pre} \} P \{ \text{Post} \} \) holds and \( P \) is non-floundering, then \( P \) is totally correct w.r.t. the specification \((\text{Pre}, \text{Post} \cap \text{Pre})\).

Proof. By Lemma 4.3.15(i, iii), \( \vdash \{ \text{Pre} \} P \{ M^L_P \cap \text{Pre} \} \) holds, with \( M^L_P \cap \text{Pre} = \text{Post} \cap \text{Pre} \). By Theorem 4.3.14, \( P \) is weak totally correct w.r.t. the specification \((\text{Pre}, \text{Post} \cap \text{Pre})\). Noting that \( M^L_P \cap \text{Pre} = \text{Post} \cap \text{Pre} \), by Definition 4.3.3, we conclude that \( P \) is totally correct w.r.t. the specification \((\text{Pre}, \text{Post} \cap \text{Pre})\). \( \square \)

We conclude by pointing out that notions and results such as proof outlines, reasoning on arithmetic built-ins, and modular proofs, directly extend to general programs.

Example 4.3.17 (Transitive closure) The following program TRANS is used to calculate the transitive closure of a given relation.

\[
\begin{align*}
\text{trans}(x, y, e, v) & \leftarrow x \sim_{e \backslash v} y \\
\text{trans}(X, Y, E, V) & \leftarrow \\
& \text{member}([X, Y], E), \\
& \neg \text{member}(X, V).
\end{align*}
\]

\[
\begin{align*}
\text{trans}(X, Z, E, V) & \leftarrow \\
& \text{member}([X, Y], E), \\
& \neg \text{member}(X, V), \\
& \text{trans}(Y, Z, E, [X| V]).
\end{align*}
\]

\[
\begin{align*}
\text{member}(X, [X| T]). \\
\text{member}(X, [Y| T]) & \leftarrow \\
& \text{member}(X, T).
\end{align*}
\]

\[
\begin{align*}
\text{member}(X, [X| T]). \\
\text{member}(X, [Y| T]) & \leftarrow \\
& \text{member}(X, T).
\end{align*}
\]

The definitions of \text{member} and \text{member}' coincide, and, in practice, they are not replicated. However, the uses (or \textit{directionalities}) highlighted by that distinction will be useful to shown absence of floundering. Let \( e \) be a binary relation on a set of constants \( \alpha \), represented as an element in List([\alpha, \alpha]), i.e. a list of pairs \([x, y]\) with \( x, y \in \alpha \). For \( x \in \alpha \), the intended meaning of a query such as \text{trans}(x, y, e, []) is to find out all \([x, y]\) in the transitive closure of \( e \). To define suitable pre- and postconditions, we write \( x \sim_{e \backslash v} y \) when there is a path from \( x \) to \( y \) in \( e \) that does not traverse pairs \([a, b]\) such that \( a \) is in the list \( v \). We define:

\[
\text{Pre} = \{ \text{trans}(x, y, e, v) \mid x \in \alpha \land v \text{ list of distinct elements in } \alpha \\
\land e \in \text{List}([\alpha, \alpha]) \land \forall a \in v \exists [a, b] \text{ in } e \} \cup
\]
Next, we define the following level mapping:

\[
\begin{align*}
|\text{member}(x, e)| &= |\text{member}'(x, e)| = |e| \\
|\text{trans}(x, y, e, v)| &= 2 \cdot |e| - |v| + 1
\end{align*}
\]

for atoms in \(\text{Pre}\), and 0 elsewhere. We note that the level mapping is well-defined, since for \(\text{trans}(x, y, e, v) \in \text{Pre}\), we have that \(|e| \geq |v|\) and then:

\[
|\text{trans}(x, y, e, v)| \geq 0.
\]

We observe that in the case \(\text{Pre} = B_L\), we would have needed a more complicated level mapping. Such a case is shown in [16], where the focus was on termination. Here, our precondition simplifies the definition of the level mapping considerably, albeit large enough to reason on the interesting queries, such as \(\text{trans}(x, Y, e, [])\). Proving that \(\Gamma \vdash \{\text{Pre}\} \\text{TRANS} \{\text{Post}\}\) holds is handy. As an example, we show a proof obligation relatively to the second clause. Consider a ground instance:

\[
\text{trans}(x, z, e, v) \leftarrow \\
\text{member}([x, y], e), \\
\neg \text{member}'(x, v), \\
\text{trans}(y, z, e, [x|v]).
\]

such that \(\text{Pre} \models \text{trans}(x, z, e, v)\) and

\[
\text{Post} \models \text{member}([x, y], e), \neg \text{member}'(x, v).
\]

We have to show that:

\[
\begin{align*}
\text{Pre} \models \text{trans}(y, z, e, [x|v]) & \text{ and } \quad \text{(4.9)} \\
|\text{trans}(x, z, e, v)| & > |\text{trans}(y, z, e, [x|v])| \quad \text{(4.10)}
\end{align*}
\]

For (4.9), we have that since \([x, y]\) is in \(e\) and \(e\) is a list of pairs of elements in \(\alpha\), then \(y\) is in \(\alpha\). In addition, \([x \mid v]\) is a list of distinct elements in \(\alpha\), since \(v\) is a list of distinct elements and \(x\) is not in \(v\). Finally, for \(\forall a \in v \exists a, b\) in \(e\) and \([x, y]\) is in \(e\) imply \(\forall a \in [x \mid v] \exists a, b\) in \(e\). Therefore, (4.9) holds.

For (4.10), we note that \(|\text{trans}(x, z, e, v)| = 2 \cdot |e| - |v| + 1 > 2 \cdot |e| - (|v| + 1) + 1 = |\text{trans}(y, z, e, [x|v])|\).
Another useful observation in showing the proof obligations relatively to the decreasing of the level mapping from the head to the two first body atoms is to note that $\text{Pre} \models \text{trans}(x, z, e, v)$ implies $|e| \geq |v|$, and then:

$$2 \cdot |e| - |v| + 1 > |e|, |v|.$$  

Let us see how the Call Patterns Theorem 4.3.9 (ii) helps us in showing that $\text{TRANS}$ is non-floundering.

Consider a negative literal $\neg \text{member}'(X, V)$ selected along a LD-derivation for $\text{TRANS}$ and any query $Q$ such that $\vdash \{ \text{Pre} \} \ldots \{ \text{Post} \}$ holds by $|\cdot|$. By Theorem 4.3.9 (ii), $\text{Pre} \models \text{member}'(X, V)$. Due to the form of $\text{Pre}$, this implies that $X \in \alpha$ and $V \in \text{List}(\alpha)$. This and the fact that $\alpha$ is a set of constants imply that $\neg \text{member}'(X, V)$ is ground. In particular, we have that $\text{TRANS}$ is non-floundering w.r.t. $\text{Pre}$. By Theorem 4.3.16, we conclude that $\text{TRANS}$ is totally correct w.r.t. the specification $\text{Pre}, \text{Post} \cap \text{Pre}$.

4.3.3 Completeness of LDNF-resolution

In the following, we present some results on completeness of Negation as Failure for LDNF-resolution, as a by-result of the verification method developed so far. Also, this allows us to provide a declarative interpretation of the sets $M^P_\text{L}$ and $FF^P_\text{L}$, which were defined operationally.

**Theorem 4.3.18 (Completeness of Negation as Failure)**

Assume that $\vdash \{ \text{Pre} \} \ldots \{ \text{Post} \}$ holds and $\text{comp}(P)$ is consistent. If for $A \in \text{Pre}$, $P \cup \{ A \}$ does not flounder, then

(i) if $\text{comp}(P) \models \neg A$ then there exists a finitely failed LDNF-tree for $P$ and $A$.

(ii) if $\text{comp}(P) \models A$ then there exists a LDNF-refutation for $P$ and $A$.

**Proof.** By the Termination Theorem 4.3.10 the LDNF-tree for $P$ and $A$ is finite. Since $P \cup \{ A \}$ is non-floundering either the LDNF-tree is finitely failed or there is a refutation.

(i). The latter case is not possible, otherwise by Soundness of SLDNF-resolution [106, Theorem 15.6] $\text{comp}(P) \models A$. This is in contradiction with the hypothesis $\text{comp}(P) \models \neg A$ and the assumption that $\text{comp}(P)$ is consistent.

(ii). The former case is not possible, otherwise by Soundness of Negation as Failure [106, Theorem 15.4] $\text{comp}(P) \models \neg A$. This is in contradiction with the hypothesis $\text{comp}(P) \models A$ and the assumption that $\text{comp}(P)$ is consistent.

We are now in the position to give a declarative interpretation of $M^P_\text{L}$ and $FF^P_\text{L}$.

**Theorem 4.3.19** Assume that $\vdash \{ \text{Pre} \} \ldots \{ \text{Post} \}$ holds, $P$ is non-floundering and $\text{comp}(P)$ is consistent. Then

(i) $M^P_\text{L} \cap \text{Pre} = \{ A \in \text{Pre} \mid \text{comp}(P) \models A \}$

(ii) $FF^P_\text{L} \cap \text{Pre} = \{ A \in \text{Pre} \mid \text{comp}(P) \models \neg A \}$
(ii) $FF^L_P \cap Pre = \{ A \in Pre \mid \text{comp}_P(P) \models \neg A \}$.

Proof. The $\subseteq$ inclusions follow from Soundness of SLDNF-resolution and of the Negation as Failure rule. The $\supseteq$ inclusions follow from the Completeness Theorem 4.3.18.

The next result extends completeness to ground general queries. Proof relation $\vdash$ naturally extends to general queries by discarding the $k > |A_i|$ requirements in Definition 4.3.1.

Theorem 4.3.20 (Completeness of LDNF-resolution I)
Assume that $\vdash \{ Pre \} P \{ Post \}$ and $\vdash \{ Pre \} Q \{ Post \}$ hold, where $Q$ is a ground general query. Moreover, assume that $P \cup \{ Q \}$ does not flounder and $\text{comp}(P)$ is consistent. If $\text{comp}(P) \models Q$ then there exists a LDNF-refutation for $P$ and $Q$.

Proof. The proof is by induction on the number of literals in $Q$.

(Base) If $Q$ consists of only one literal then the result follows by Theorem 4.3.18.

(Step) If $Q = L, Q'$ then by Theorem 4.3.18 there exists a LDNF-refutation for $L$. By Lemma 4.3.13 (iii, iv) we have $\text{Post} \models L$ and then, by Definition 4.3.4, $\vdash \{ Pre \} Q' \{ Post \}$ holds. Therefore we can apply the inductive hypothesis on $Q'$ to reach the desired conclusion.

The final result of this section is concerned with a further extension of completeness of LDNF-resolution. In this case, assuming an underlying language with infinitely many function symbols (i.e., $\Sigma_L$ infinite), we can state a completeness result that extends a well-known theorem by Cavedon [40].

Theorem 4.3.21 (Completeness of LDNF-resolution II)
Assume that $\vdash \{ Pre \} P \{ Post \}$ and $\vdash \{ Pre \} Q \{ Post \}$ hold. Moreover, assume that $P \cup \{ Q \}$ does not flounder, $\Sigma_L$ is infinite and $\text{comp}(P)$ is consistent. If $\text{comp}(P) \models Q'$ for an instance $Q'$ of $Q$, then there exists a LDNF-refutation for $P$ and $Q$ with computed instance more general than $Q'$.

Proof. Let $Q''$ be the query obtained by substituting every variable $x_i$ in $Q'$ by a term $t_i$ with principal functor not appearing in $P$ or $Q'$, and distinct from that of the others $t_j$, for $j \neq i$. Such terms exist since $\Sigma_L$ is infinite. $P \cup \{ Q'' \}$ cannot flounder, otherwise by substituting the $t_i$'s with the $x_i$'s in the derivation, we would conclude that $P \cup \{ Q' \}$ flounders, and a fortiori that $P \cup \{ Q \}$ flounders. Therefore, by Theorem 4.3.20, there exists a LDNF-refutation for $P$ and $Q''$. By substituting the $t_i$'s with the $x_i$'s along that refutation we obtain a LDNF-refutation for $P$ and $Q'$. Since $P \cup \{ Q \}$ does not flounder, we can lift that refutation to a LDNF-refutation for $P$ and $Q$ with computed instance more general than $Q'$.

As a special case we find again the results of Cavedon [40] on acyclic programs. We recall that a program is acyclic if there exists a level mapping $\| \|$ such that:

for every $A \leftarrow L_1, \ldots, L_n \in \text{ground}_{L_P}(P)$ for $i \in [1, n]$ $|A| > |L_i|$
where for a negative literal \( \neg B_i \) is set to \(|B_i|\).

It can be shown that if a program is acyclic with respect to a language \( L \), then it is acyclic with respect to every extension of \( L \). Therefore, we can assume, without loss of generality, that \( \Sigma_L \) is infinite.

Apt and Bezem [2, Theorem 2.5] showed that for an acyclic program \( P \), \( M^P_L \) is a model of \( comp(P) \), i.e. that \( T_P(M^P_L) = M^P_L \). By Definition 4.3.4, we conclude that if \( P \) is acyclic then \( \vdash \{ B_L \} P \{ M^P_L \} \) holds in some language \( L \) with \( \Sigma_L \) infinite. In addition, \( comp(P) \) is consistent and \( \vdash \{ B_L \} Q \{ M^P_L \} \) holds for any query.

Summarizing, the only hypothesis needed to apply Theorem 4.3.1 is that \( P \cup \{ Q \} \) does not flounder, which is implied by the only hypothesis of the Completeness Theorem [40, Theorem 4.5] that \( P \) and \( Q \) are allowed.

### 4.4 Related Work

In the following, we discuss the relations of our approach with other proof methods for reasoning on (weak) partial correctness, termination, (weak) total correctness and general programs. Once again, we recall that our intended objective is to show that the proposed method – based on the \( \vdash_t \) relation – is a trade-off between expressiveness (i.e., the class of programs and properties it is able to reason about) and ease of use in paper \& pencil verification proofs.

#### Weak Partial Correctness

We now show that the method based on the relation \( \vdash_t \) is equivalent with the one of Bossi and Cocco (see Related Works of Chapter 3), thus precisely classifying the expressiveness of \( \vdash_t \). We follow the presentation of Apt and Marchiori [12].

**Definition 4.4.1** A type \( I \) is a set of atoms such that if an atom \( A \) is in \( I \) then every instance of \( A \) is in \( I \). Let \( \pre \), \( \post \) be types. A program is well-m-asserted by \( \pre, \post \) if for every \( A \leftarrow B_1, \ldots, B_n \) instance of a clause from it, for \( i \in [1, n] \):

\[
A \in \pre \land B_1, \ldots, B_{i-1} \in \post \Rightarrow B_i \in \pre
\]

and

\[
A \in \pre \land B_1, \ldots, B_n \in \post \Rightarrow A \in \post.
\]

A query is well-m-asserted by \( \pre, \post \) if for every \( B_1, \ldots, B_n \) instance of it, for \( i \in [1, n] \):

\[
A \in \pre \land B_1, \ldots, B_{i-1} \in \post \Rightarrow B_i \in \pre.
\]

A type \( I \) is called **strongly closed under substitution** if an atom is in \( I \) iff every ground instance of it is in \( I \). \( \Box \)
As an example of type which is not strongly closed under substitution is the set of ground atoms. The method of Bossi and Cocco consists of proving a program well-m-asserted. It is evident that relation ⊢ is a simplification of well-m-assertedness. Intuitively, ⊢ can be seen as well-m-assertedness restricted to types strongly closed under instantiation.

**Lemma 4.4.2** Let \( P \) be a program and \( \text{pre}, \text{post} \) be types strongly closed under instantiation. Then \( P \) is well-m-asserted by \( \text{pre}, \text{post} \) iff
\[
\vdash \text{pre} \cap B_L \mid P \mid \text{post} \cap B_L.
\]

Under a rather general hypothesis, the two methods exhibit the same expressiveness, in a sense clarified by the following definition.

**Definition 4.4.3** Consider two sets of types, \( \mathcal{I} \) and \( \mathcal{J} \). We say that \( \mathcal{I} \) is at least as expressive as \( \mathcal{J} \) if every program well-m-asserted by two types \( \text{pre}, \text{post} \) in \( \mathcal{J} \) is well-m-asserted by \( \text{pre}', \text{post}' \) types in \( \mathcal{I} \) such that:
\[
\text{pre} \subseteq \text{pre}' \quad \text{and} \quad \text{post}' \cap \text{pre} \subseteq \text{post}.
\]

We say that \( \mathcal{I} \) is as expressive as \( \mathcal{J} \) if \( \mathcal{I} \) is at least as expressive as \( \mathcal{J} \) and vice-versa.

In other words, \( \mathcal{I} \) is at least as expressive as \( \mathcal{J} \) if whenever we can reason on \( P \) and \( Q \) using types from \( \mathcal{J} \), then we are able to reason on \( P \) using types from \( \mathcal{I} \) that allow for reasoning on a class of queries containing \( Q \), since \( \text{pre} \subseteq \text{pre}' \), and on finer properties, since for \( \text{post}' \cap \text{pre} \subseteq \text{post} \).

**Theorem 4.4.4** Assume that \( \Sigma_L \) is infinite. Then types strongly closed under instantiation are as expressive as types.

**Proof.** Obviously, types are at least as expressive as strongly monotonic types. Conversely, consider types \( \text{pre} \) and \( \text{post} \), and a program \( P \) well-m-asserted by \( \text{pre}, \text{post} \). We define the types (strongly closed under instantiation):
\[
\text{pre}' = \text{True}(\text{pre} \cap B_L) \quad \text{post}' = \text{True}(M^L_P \cap \text{post}),
\]
where \( \text{True}(I) = \{ A \in \text{Atom}_L \mid I \models A \} \). It is readily checked that \( P \) is well-m-asserted by \( \text{pre}', \text{post}' \) as well. Moreover, consider \( A \in \text{pre} \). Since every ground instance of \( A \) is in \( \text{pre} \cap B_L \), then \( A \) is in \( \text{pre}' \). Therefore, \( \text{pre} \subseteq \text{pre}' \). In addition, if \( A \in \text{post}' \cap \text{pre} \) then every ground instance of \( A \) is in \( M^L_P \). Let \( A' \) be a ground instance of \( A \) obtained by instantiating every variable of \( A \) with ground terms whose
principal function symbol is distinct and does not appear in \( A \) or \( P \). \( A' \) exists since \( \Sigma_L \) is infinite. Then

\[
A' \in M_P^L \\
\Leftrightarrow \{ \text{\( A' \) ground } \}
\]

\[
P \models A'
\]

\[
\Leftrightarrow \{ \text{Theorem on Constants (see e.g. \[146]) } \}
\]

\[
P \models A.
\]

By \[12, \text{Corollary 4.8}\], \( A \in \text{pre} \) and \( P \models A \) imply \( A \in \text{post} \), hence \( \text{post'} \cap \text{pre} \subseteq \text{post} \). \( \square \)

**Call Patterns Characterization**

In Section 4.1.4, the method based on the proof relation \( \models \) was shown to be incomplete w.r.t. the notion of (weak) partial correctness, in the sense that there are programs \( P \) weak partially correct w.r.t. a specification \((\text{Pre}, \text{Post})\) for which \( \models \{\text{Pre}\} \ P \{\text{Post'}\} \) does not hold for any \( \text{Post'} \).

As observed in the Related Works of Chapter 3, incompleteness is due to the fact that the proof method addresses call patterns characterization (see Corollary 4.1.18) as well as declarative properties. In other words, we had to trade completeness of the method for the possibility of reasoning on call patterns with respect to the leftmost selection rule.

**Partial Correctness and Weakest (Liberal) Preconditions**

Malfon \[111\] showed a method for proving partial correctness. It is worth noting that the notion of well-supported interpretation results to be a simplification of a similar notion introduced by Malfon in the context of Fitting and well-founded semantics for general logic programs.

To the best of our knowledge, no approach discusses methods for characterizing the weakest (liberal) preconditions, in the sense of Theorem 4.1.44.

**Termination**

In Chapter 3, we have formally derived the proof relation \( \models_t \) from acceptability.

From a theoretical point of view, proof relation \( \models_t \) is a special case of acceptability, where the Herbrand model is of the particular form \( \text{Pre} \rightarrow \text{Post} \) and the level of a ground atom is finite iff it belongs to \( \text{Pre} \). On the other hand, in Section 4.1.7 we have seen that \( \models_t \) is a complete termination proof method for triples in \( \models \).

From a practical point of view, adapting acceptability to take into account the intended queries facilitates in many examples the required reasoning, in that uninteresting input queries are not to be considered.
Example 4.4.5 (Flat) Consider the following program FLAT:

\[
\begin{align*}
\text{flat}([], []). \\
\text{flat}([X \mid Xs], [f(X) \mid FXs]) & \leftarrow \\
\quad \text{flat}(Xs, FXs). \\
\text{flat}(nil, []). \\
\text{flat}(tree(X, Ls, Rs), [f(X) \mid Fs]) & \leftarrow \\
\quad \text{flat}(Ls, FLs), \\
\quad \text{flat}(Rs, FRs), \\
\quad \text{append}(FLs, FRs, Fs). \\
\end{align*}
\]

augmented by the APPEND program.

FLAT applies \( f(\cdot) \) to every element of a given list, or of a preorder traversal of a given binary tree. We denote by \( BTree \) the set of binary trees, and for \( bt \in BTree, \|bt\| \) denotes the number of nodes of \( bt \). Given:

\[
\begin{align*}
\text{Pre} &= \text{flat}(GList \times U_L) \cup \text{flat}(BTree \times U_L) \cup \text{Pre}_{\text{APPEND}} \\
\text{Post} &= \{ \text{flat}(ls, rs) \mid ls, rs \in GList \land \|ls\| = |rs| \} \\
& \quad \cup \{ \text{flat}(bt, rs) \mid bt \in BTree, rs \in GList \land \|bt\| = |rs| \}
\end{align*}
\]

it is straightforward to exhibit proof outlines to show that \( \vdash_t \{ \text{Pre} \} \text{FLAT} \{ \text{Post} \} \) holds by using a level mapping \( \| \) such that:

\[
\|\text{flat}(ls, rs)\| = \begin{cases} 
\|ls\| + 1 & \text{if } ls \in GList \\
\|ls\| + 1 & \text{if } ls \in BTree
\end{cases}
\]

On the contrary, proving acceptability is awkward, due to the fact that badly-typed atoms have to be considered in the definition of a model of the program, e.g. the atom \( \text{tree}(a, [a,b,c], \text{tree}(a, [], \text{nil})) \).

\( \square \)

(Weak) Total Correctness

Technically speaking, the proof theory is obtained by the formal derivation of Section 3.3, and it turns out to be a combination of (modifications of) existing proposals: the proof method for partial correctness of Bossi and Coccolo [28], and the proof method for termination of Apt and Pedreschi [16]. The expressiveness of the combined method strictly exceeds the expressiveness of the separated methods both from a theoretical and a practical perspective.

On the theoretical level, the Termination Completeness II Theorem 4.1.37 states that the proof relation \( \vdash_t \) is at least as expressive as any weak total correctness proof method specifically targeted to reason on left termination. On the contrary, we have shown in Chapter 2, that acceptability as defined by Apt and Pedreschi is not a complete method for left termination.
On the practical level, we claim that the sum of a weak partial correctness method, such as well-m-assertedness, and of a termination method, such as acceptability is not as expressive as the method based on proof relation $\vdash$. On the one hand, by simply applying in turn well-m-assertedness and acceptability involves considering more proof obligations than establishing $\vdash$. On the other hand, the complications with proving acceptability highlighted in the example program FLAT still continue to hold. Furthermore, consider $P$ well-m-asserted by $pre, post$. Since in general $post$ is not a model of the program (see APPEND for an example), acceptability must be shown by considering a further set $Post'$ — a model of $P$ — which is not present in our approach. In addition, confusion can arise due to the fact that acceptability analysis acts at a ground level, whilst well-m-assertedness acts at a non-ground level.

Also, we mention that well-m-assertedness was extended by Bossi et al. [29] to reason on weak total correctness. They defined level mappings $\|\|$ on non-ground atoms as well, and required that for every $A \leftarrow B_1, \ldots, B_n$ instance of a clause of $P$, for every $i \in [1, n]$:

$$pre \models A \land post \models B_1, \ldots, B_{i-1} \implies |A| > |B_i|.$$  \hfill (4.11)

However, this leads to complications, since termination can be proved only using rigid level mappings, and then a further proof obligation has to be satisfied. $\|\|$ is called rigid if whenever $pre \models A$ then $|A| = |A'|$ for every instance $A'$ of $A$. Moreover, the resulting proof method is not complete in the sense of Theorem 4.1.36. In fact, consider the program $P$:

$$p(0),$$
$$p(1) \leftarrow p(0).$$

and $pre = Atom_L$, and any $post$. For every level mapping $\|$, we have that (4.11) requires $|p(1)| > |p(0)|$. Therefore, $\|$ cannot be rigid, since $pre \models p\infty$ (we believe, however, that a completeness result should follow by weakening the definition of rigid level mappings by considering a non-strict ordering). On the contrary, it is straightforward to show that $\vdash \{B_L\} P \{B_L\}$ and $\vdash \{B_L\} p(X) \{B_L\}$ hold by a same level mapping. A similar argument applies to the termination proof methods proposed for well-typed programs by Bronsard et al. [37], and for the annotation method by Deransart and Maluszyński [66].

**General programs**

Ferrand and Deransart [80] extended the proof method of Deransart [64] to prove declarative properties of general logic programs. Differently from our approach, they do not discuss termination issues and adopt the well-founded semantics [4]. As in the case of definite programs, their method is more general for proving declarative properties, albeit ours is also able reason on call patterns characterization and termination as well as ensuring completeness of LDNF-resolution.
4.5. Conclusion

The same arguments apply to the proposal of Malfon [111], which presented a sound and complete method to prove declarative properties with respect to Fitting and well-founded semantics.

Integrated approaches

There are a few attempts to present in a uniform way methods dealing with correctness, termination, call patterns, occur-check freedom, modular proofs, and other program properties. A valuable approach is due to Apt [10]. However, his book presents several separated results, which in many cases are instantiations of the proof method presented in this Chapter.

Also, Deville [70] proposes an approach for systematically deriving terminating programs from specifications provided in a Clark’s completion-like format. However, the method is not applicable to check correctness of existing programs.

Recently, Stärk [149] proposed a logic program theorem prover in which termination and correctness can be formally proved for programs containing negation and arithmetic built-in’s. The formal theory underlying the theorem prover is an extension of pure Prolog including induction principles and axioms for built-in’s.

4.5 Conclusion

The starting point of the research reported in this Chapter has been the recognition of a few core principles, common to several existing proof methods for logic programs. On this basis, a thorough proof theory has been developed as a candidate unifying framework capable of addressing a reasonably large spectrum of properties for a reasonably large class of programs.

The original contribution of this Chapter is the introduction of a proof relation $\vdash_t$ for total correctness of logic programs, possibly containing negation and arithmetic built-in’s, which are designed to be executed according to a fixed selection rule. In particular, the proposed proof theory concentrates on the (Prolog’s) leftmost selection rule. For reasons of presentation, the $\vdash_t$ proof method has been introduced in an incremental way, by a stepwise definition of increasingly higher levels of verification, from a weak form of partial correctness up to full-fledged total correctness. Also, we presented a novel characterization of weakest (liberal) preconditions of a program $P$ and a postcondition $Post$ in terms of ordinal closure of a monotonic operator $\vartheta_{P,P,Post}$.

Some applications of the method have been surveyed, including proving absence of run-time errors, modular program development, safe omission of the occur-check, verification of meta-programs, semantics decidability. By lack of space, we could not include the presentation of case studies of significant dimension. However, we refer the reader to [141] for a collection of case studies.
Finally, we compared the expressiveness of the proposed approach with existing proposals. Technically speaking, the proof theory has been formally derived in Section 3.3, and it turns out to be a combination of the proof method of Bossi and Cocco [28] for weak partial correctness, and the proof method of Apt and Pedreschi [16] for termination. The advantage of the combined method is that its expressiveness strictly exceeds the expressiveness of the separated methods both from a theoretical and a practical perspective. In particular, the proof relation $\Gamma \vdash_t$ is at least as expressive as any weak total correctness proof method specifically targeted to reason on left termination.
Chapter 5

Verification of Meta-interpreters

The intended goal of this Chapter is to illustrate the broad applicability of the adopted verification principles on the case study of the well-known \textit{Vanilla} meta-interpreter. Interestingly, as a by-result of the case study, we introduce a rather general criterion for reasoning about meta-interpreters and demonstrate its applicability. We found that many known properties of meta-interpreters can be proved in a direct way using the proposed method, often in a generalized sense. Therefore, another relevant contribution of this Chapter can be summarized as follows: under certain natural assumptions, all interesting verification properties lift up from the object program to the meta-program, including

- (weak) partial correctness,
- (weak) total correctness,
- absence of run-time errors,
- call and success patterns characterization,
- correct and computed instances characterization.

Interestingly, it is possible to establish these results on the basis of purely declarative reasoning, using the proof method of Chapter 4.

5.1 Meta-Interpreters

Meta-programming refers to any kind of computer programming where the input or output represents programs [18]. In the computational logic paradigm, meta-programming is a natural and powerful tool. Since the early studies, it has become clear the broad scope of this research area, whose applications now include expert systems, semantics and implementation, prototyping and debugging, among others.
Meta-circular interpreters (i.e., interpreters of a language written in the language itself) have been introduced as a fundamental feature of advanced programming languages. A number of meta-interpreters of logic programming have been introduced and proved correct with respect to their intended behavior. However, the task of proving correctness has been largely performed using ad-hoc techniques, depending case by case on the semantics, the particular meta-program and the range of properties one is interested in verifying. No uniform and general method has been proposed for tackling the problem by simple and powerful tools. In addition, the proofs of correctness are typically based on operational reasonings, without any chance to exploit the declarative reading of logic programs.

### 5.1.1 Correctness

What is a meta-program? Generally speaking, the answer is not obvious, as in computational logic programs and data are of the “same” nature. In this Chapter, we make the liberal assumption that meta-programs are just programs over the ambivalent language $L$. Ambivalence means that the sets of predicate and function symbols in $L$ may overlap.

It is however necessary to agree when a given meta-program can be regarded as a meta-interpreter in a meaningful sense, i.e., to define a notion of correctness of a meta-interpreter with respect to the object program under consideration. We stipulate that a meta-interpreter is a meta-program defining a specific relation called demo, which is expected to represent provability of object-level queries. This is the motivation for the following definition of correct meta-interpreters. For the sake of generality, the definition is parametric with respect to a set $Pre$ of object-level atoms, which represents the intended (one-atom) object-level queries. We recall that $L$ is the language in which programs and queries are written.

**Definition 5.1.1** A program $V$ is declaratively correct with respect to a program $P$ and a set $\text{Pre} \subseteq B_L$ iff for every $A \in \text{Pre}$:

$$A \in M_P \iff \text{demo}(A) \in M_V.$$

When the if part only is considered, we have the notion of soundness. In this case, we say that $V$ is declaratively sound w.r.t. $P$ and Pre.

The intuition is that, when considering intended queries only, the demo-predicate of a correct meta-interpreter behaves in the same way as the object-level program. In the literature, the case $\text{Pre} = B_P$ is usually considered, i.e., all object-level queries are considered in the above definition of correctness.

We introduce next a weakening of the notion of soundness, given with reference to a specification of the object program.
Definition 5.1.2 A program $V$ is \textit{weak declaratively sound} with respect to a program $P$ and a specification $(\text{Pre}, \text{Post})$ iff for every $A \in \text{Pre}$:

$$A \in \text{Post} \quad \text{if} \quad \text{demo}(A) \in M^P_v.$$

In the special case $\text{Post} = M^P_v$ we find again declarative soundness. Definition 5.1.2 only requires that if $\text{demo}(A) \in M^P_v$ then $A$ satisfies some property denoted by $\text{Post}$. This may be sufficient in some interesting cases, i.e., when some weak properties of the meta-interpreter are needed.

5.1.2 Vanilla

A \textit{jewel} of logic programming is the elegant meta-program which specifies the meta-circular interpreter for logic programs. This program, referred to as the \textit{Vanilla} meta-interpreter, and first introduced by Bowen and Kowalski [32], is denoted by $\text{Van}(P)$ when instantiated with an object program $P$, and consists of the following clauses:

\begin{itemize}
  \item \textbf{(d1)} \quad \text{demo( true ).}
  \item \textbf{(d2)} \quad \text{demo( A \& B ) } \leftarrow \text{demo( A ), demo( B ).}
  \item \textbf{(d3)} \quad \text{demo( A ) } \leftarrow \text{clause( A, B ), demo( B ).}
  \item \textbf{(e)} \quad \text{clause( A, B \& \ldots \& B_n ).}
\end{itemize}

for every $A \leftarrow B_1, \ldots, B_n \in P$.

where $B_1 \& \ldots \& B_n$ is an abbreviation for

- $B_1 \& ( B_2 \& \ldots ( B_{n-1} \& B_n ) \ldots )$, if $n > 1$,
- $B_1$, if $n = 1$
- true, if $n = 0$.

Example 5.1.3 For instance, for $P = \text{NAIVE REVERSE}$ (see Example 2.3.3) the definition of \text{clause} is:

\begin{itemize}
  \item \text{clause(\text{reverse}([X | Xs], Ys), \text{reverse}(Xs, Zs) \& \text{append}(Zs, [X], Ys)).}
  \item \text{clause(\text{reverse}([], [], true)).}
  \item \text{clause(\text{append}([X|Xs], Ys, [X|Zs]), \text{append}(Xs, Ys, Zs)).}
  \item \text{clause(\text{append}([], Xs, Xs), true)).}
\end{itemize}
Specializing the above notion of declarative correctness, we have that $Van(P)$ is declaratively correct w.r.t. $P$ and $Pre$ iff for every $A \in Pre$ we have $A \in M_{P}^{L}$ iff $demo(A) \in M_{Van(P)}^{L}$.

**Assumption 5.1.4** When reasoning on Vanilla (or any of its extensions), we assume that the symbols $\&$ and $\text{true}$ are not predicate symbols of $L$. □

Indeed, without this assumption Vanilla is not correct.

**Example 5.1.5** Consider the program
\[
q(a).
p(b).
\& (p(c),p(c)).
\]
Then $demo(\& (p(b),q(a)))$ is in $M_{Van(P)}^{L}$ whereas the atom $\& (p(b),q(a))$ is not in $M_{P}^{L}$. A similar argument applies to the program $\text{true} \leftarrow \text{true}$. □

The relevance of Vanilla lies in the fact that it provides a terse account of the provability relation of logic programming, and a natural basis for extending the computation strategy. Due to its unique nature, the Vanilla meta-interpreter has raised non-trivial questions concerning its correctness, and a vast literature on the subject is available (see Related Work).

**Amalgamation and Reflection Down/Up**

The ambivalent syntax allows us to combine the object program with the meta-program - a technique known as amalgamation [32]. We will show that, under certain assumptions, Vanilla can be amalgamated with the object program without compromising its declarative correctness.

According to the treatment of amalgamation reported in [32], we will consider several types of amalgamation. A simple form of amalgamation consists of joining object and meta-program preventing any interference. A more liberal approach allows the object program to make a call to the meta-level. This possibility is usually referred to as reflection up. As an example, we consider Vanilla augmented with the clause:
\[
demo( demo(A)) \leftarrow demo(A).
\]
which allows the object program to make a call at the meta-level.

On the other hand, the meta-program which makes a call to the object-level is in the form of amalgamation known as reflection down. The simplest reflective meta-interpreter is the standard Prolog implementation of the meta-predicate call (see e.g., [150]) by means of meta-variables:
call( X ) ← X.

It should be noted that a meta-interpreter such as Vanilla, which basically relies on the ambivalent syntax, cannot be defined in strongly typed languages, such as Gödel [87], as in every type discipline functors and relations cannot be confused.

Parameterization and Extensions

Meta-interpreters such as Vanilla have been enhanced with various functionalities in a simple way. For example, the demo relation can be augmented with new arguments which represent proof trees and other information about proofs (see [150] for a collection of enhanced meta-interpreters). Also, extra arguments may be used to deal with theories or modules, thus supporting various forms of modular programming and program composition. Later we discuss how the proposed approach can be extended to deal with enhanced meta-interpreters.

Ambivalent Syntax

In contrast to the usual definition of first order logic, we assume that ambivalent syntax is allowed, in the sense that function and predicate symbols of $L$ may overlap [98, 99]. This form of ambivalence is called atoms-as-terms, since it allows atoms to occur in term positions.

Ambivalent syntax is necessary in order Definition 5.1.1 to make sense. In fact, since we assume a superlanguage [18] containing all symbols appearing in object- and meta-programs, such a superlanguage must necessarily admit overlapping of symbols. Also, ambivalence is necessary in presence of amalgamation.

Moreover, in order to admit the Call($P$) meta-program, we will adopt a more complex form of ambivalent syntax, called terms-as-atoms, where terms may occur in atom positions. Note that the atoms-as-terms ambivalent syntax and meta-variables (a special case of the terms-as-atoms ambivalent syntax) are allowed in Prolog implementations.

A general form of ambivalent logic, including the two forms above of ambivalent syntax, has been investigated by Kalsbeek and Jang [98, 99]. Their theoretical results justify the current practice of using ambivalent syntax in Prolog and in meta-programming.

For our purposes, we restrict to observe that the basic results of the theory of logic programming holds in presence of overlapping of function and predicate symbols. This fact has been pointed out by Martens and De Schreye [114] and Kalsbeek [98]. Also, we observe that the definitions and the results on proof relations $\vdash$ and $\vdash_i$ readily apply in presence of overlapping of function and predicate symbols.
5.2 Vanilla

We propose in this Chapter a systematic method for assessing the correctness of Vanilla based on its declarative semantics, and improve on existing results in various directions. The advantages of our method can be summed up as follows. First, our method yields declarative correctness proofs, which are simpler of the operational proofs provided in the literature. Second, new results are established as a by-product. Third, a correctness criterion for arbitrary meta-interpreters is introduced, as a result of the analysis of Vanilla.

5.2.1 Specifying Pre and Post

As a first exercise, let us show that the proof relation \( \vdash \) lifts from the object program up to Vanilla, in the sense that the rule:

\[
\vdash \{\text{Pre}\} \; P \; \{\text{Post}\} \\
\vdash \{\text{Pre}_{\text{Van}}(P)\} \; \text{Van}(P) \; \{\text{Post}_{\text{Van}}(P)\}
\]

holds for certain \( \text{Pre}_{\text{Van}}(P) \), \( \text{Post}_{\text{Van}}(P) \) defined starting from \( \text{Pre} \) and \( \text{Post} \) in a natural way.

Let \( A, B_1, \ldots, B_n \) be ground atoms, with \( n \geq 0 \). We recall that \( B_1 \land \ldots \land B_n \) is an abbreviation for

- \( (B_1 \land \ldots \land (B_{n-1} \land B_n)) \), if \( n > 1 \),
- \( B_1 \), if \( n = 1 \)
- true, if \( n = 0 \).

Conversely, by writing \( (B_1 \land \ldots \land B_n) \) we denote the query \( B_1, \ldots, B_n \).

We define:

\[
\text{demo}(B_1 \land \ldots \land B_n) \in \text{Pre}_{\text{Van}}(P) \iff
\vdash \{\text{Pre}\} \; B_1, \ldots, B_n \; \{\text{Post}\}
\]

\[
\text{clause}(A, B) \in \text{Pre}_{\text{Van}}(P) \iff \text{true}
\]

\[
\text{demo}(B_1 \land \ldots \land B_n) \in \text{Post}_{\text{Van}}(P) \iff
\text{Post} \vdash B_1, \ldots, B_n
\]

\[
\text{clause}(A, B_1 \land \ldots \land B_n) \in \text{Post}_{\text{Van}}(P) \iff
A \leftarrow B_1, \ldots, B_n \in \text{ground}_L(P).
\]

No other atom is in \( \text{Pre}_{\text{Van}}(P) \) or in \( \text{Post}_{\text{Van}}(P) \). Formally, \( \text{Pre}_{\text{Van}}(P) \) and \( \text{Post}_{\text{Van}}(P) \) are defined in terms of \( \text{Pre} \) and \( \text{Post} \), and their definition reflects at the meta-level the notions of intended queries and intended correct instances of queries at the object-level. The next proof outlines establish that \( \vdash \{\text{Pre}_{\text{Van}}(P)\} \; \text{Van}(P) \; \{\text{Post}_{\text{Van}}(P)\} \) holds under the assumption that \( \vdash \{\text{Pre}\} \; P \; \{\text{Post}\} \) holds.
Proof outlines (a,b) are self-explanatory, by simply observing that for a ground query $B_1,\ldots,B_n$:

(i) $\vdash \{\text{Pre}\} B_1,\ldots,B_n \{\text{Post}\} \Rightarrow \text{Pre} \models B_1$

(ii) $\vdash \{\text{Pre}\} B_1,\ldots,B_n \{\text{Post}\} \wedge \text{Post} \models B_1 \Rightarrow$

\quad $\vdash \{\text{Pre}\} B_2,\ldots,B_n \{\text{Post}\}$

The proofs of (i,ii) are immediate by the Definition 4.1.1. Proof outline (d) is actually a proof outline schema (one for each clause from $P$), and it is of direct verification.

Consider now the proof outline (c). First, we observe that, as a direct consequence of Definition 4.1.1 (3), we have for an atom $A$:

\[
\vdash \{\text{Pre}\} A \{\text{Post}\} \Leftrightarrow \text{Pre} \models A
\]  

Let us prove the proof obligations of page 104. (ii,iii) are immediate from definition of $\text{Pre}_{\text{VAN}(P)}$ and $\text{Post}_{\text{VAN}(P)}$. Consider now (iv'). The case $i=1$ is trivial. In the case
i = 2, we assume \( \vdash \{ \text{Pre} \} A \{ \text{Post} \} \) and \( A \leftarrow B^- \in \text{ground}_L(P) \). By (5.1), we have \( \text{Pre} \models A \) and, as observed in (4.1) (see page 99), we conclude \( \vdash \{ \text{Pre} \} B^- \{ \text{Post} \} \).

To establish the proof obligation (v), assume \( \vdash \{ \text{Pre} \} A \{ \text{Post} \} \) and \( A \leftarrow B^- \in \text{ground}_L(P) \) and \( \text{Post} \models B^- \). By (5.1), \( \text{Pre} \models A \). By Definition 4.1.1 (2), we conclude \( \text{Post} \models A \).

Finally, consider a (not necessarily ground) query \( B_1, \ldots, B_n \) such that

\[ \vdash \{ \text{Pre} \} B_1, \ldots, B_n \{ \text{Post} \} \]

We have directly from the definition of \( \text{Pre}_{\text{Van}(P)} \) that

\[ \vdash \{ \text{Pre}_{\text{Van}(P)} \} \text{demo}(B_1 \& \ldots \& B_n) \{ \text{Post}_{\text{Van}(P)} \} \]

(5.2)

holds, i.e. \( \vdash \) lifts up for queries as well.

It is worth noting that the assumption that the symbols \& and \texttt{true} are not predicate symbols of \( L \) is used in proof outlines (b) and (a), respectively. In fact, any ground instance of the assertion

\[ \vdash \{ \text{Pre} \} A, B^- \{ \text{Post} \} \]

at the head of clause (b) is equivalent to state \( \text{demo}(A \& B^-) \in \text{Pre}_{\text{Van}(P)} \) in the case that \& is not a predicate symbol of \( L \).

### 5.2.2 Persistency, Call Patterns and Success Patterns

From now on, we exploit the results of Chapter 4 in order to lift properties of the object program up to \text{Vanilla}. Let us start considering the simple Corollary 4.1.18, providing call patterns and success patterns characterizations. It allows us to prove two natural properties of \text{Vanilla}.

The first relevant consequence of Corollary 4.1.18 is the persistency of call patterns from the object program to \text{Vanilla}, under the leftmost selection rule of Prolog. To the best of our knowledge, this property is not reported anywhere in the literature.

**Theorem 5.2.1** Assume that \( \vdash \{ \text{Pre} \} P \{ \text{Post} \} \) and \( \vdash \{ \text{Pre} \} Q^- \{ \text{Post} \} \) hold. Then for every selected atom \( \text{demo}(Q_1) \) in a LD-derivation for \( \text{Van}(P) \) and \( \text{demo}(Q) \), we have \( \vdash \{ \text{Pre} \} Q^- \{ \text{Post} \} \).

**Proof.** Since \( \vdash \{ \text{Pre} \} P \{ \text{Post} \} \) holds, we have that

\[ \vdash \{ \text{Pre}_{\text{Van}(P)} \} \text{Van}(P) \{ \text{Post}_{\text{Van}(P)} \} \]

holds. Moreover \( \vdash \{ \text{Pre} \} Q^- \{ \text{Post} \} \), implies by (5.2) that

\[ \vdash \{ \text{Pre}_{\text{Van}(P)} \} \text{demo}(Q) \{ \text{Post}_{\text{Van}(P)} \} \]
holds. Therefore, by Corollary 4.1.18 (i), \( \text{Pre}_{\text{Van}(P)} \models \text{demo}(Q_1) \). The conclusion follows from the definition of \( \text{Pre}_{\text{Van}(P)} \).

In the special case that \( Q_1 \) is an atom, the above result states that \( \text{Pre} \models Q_1 \). In this sense, call patterns of the object program lift up to \textit{Vanilla}.

**Example 5.2.2 (Arithmetic Built-in's)** Assume that \( \vdash \{ \text{Pre} \} P \{ \text{Post} \} \) and \( \vdash \{ \text{Pre} \} Q^{-} \{ \text{Post} \} \) hold. Let us consider an extension of Vanilla which includes the treatment of arithmetic built-in's, such as \( > \), obtained by adding the clause

\[
\text{demo}(X > Y) \leftarrow X > Y.
\]

According to the general approach of Section 4.2.1, we assume that \( \text{Pre} \) is such that:

\[
n > m \in \text{Pre} \quad \text{iff} \quad n, m \in \text{Gae}.
\]

By Lemma 4.2.4 no LD-derivation for \( P \) and \( Q^{-} \) ends in an error.

We observe that (5.3) lifts to \( \text{Pre}_{\text{Van}(P)} \):

\[
\text{demo}(n > m) \in \text{Pre}_{\text{Van}(P)} \quad \text{iff} \quad n, m \in \text{Gae}.
\]

Since \( \vdash \{ \text{Pre} \} P \{ \text{Post} \} \) holds, we have that

\[
\vdash \{ \text{Pre}_{\text{Van}(P)} \} \text{Van}(P) \{ \text{Post}_{\text{Van}(P)} \}
\]

holds. Moreover \( \vdash \{ \text{Pre} \} Q^{-} \{ \text{Post} \} \), implies by (5.2) that

\[
\vdash \{ \text{Pre}_{\text{Van}(P)} \} \text{demo}(Q) \{ \text{Post}_{\text{Van}(P)} \}
\]

holds. Therefore, by Lemma 4.2.4, no LD-derivation for \( \text{Van}(P) \) and \( \text{demo}(Q) \) ends in an error.

In this sense, we conclude that \textit{Vanilla} extended with arithmetic built-in's, preserves the property of absence of arithmetic errors, in the sense that the meta-program is error-free if the object program is. \( \square \)

The second consequence of Corollary 4.1.18 is the characterizations of the success patterns of \textit{Vanilla}.

**Theorem 5.2.3** Assume that \( \vdash \{ \text{Pre} \} P \{ \text{Post} \} \) and \( \vdash \{ \text{Pre} \} Q^{-} \{ \text{Post} \} \) hold. Then for every computed instance \( \text{demo}(Q_1) \) of \( \text{Van}(P) \) and \( \text{demo}(Q) \), we have \( \text{Post} \models Q_1^{-} \).

**Proof.** Since \( \vdash \{ \text{Pre} \} P \{ \text{Post} \} \) holds, we have that

\[
\vdash \{ \text{Pre}_{\text{Van}(P)} \} \text{Van}(P) \{ \text{Post}_{\text{Van}(P)} \}
\]
holds. Moreover $\vdash \{\text{Pre}\} Q^- \{\text{Post}\}$, implies by (5.2) that

$$\vdash \{\text{Pre}_{\text{Van}(P)}\} \text{demo}(Q) \{\text{Post}_{\text{Van}(P)}\}$$

holds. Therefore, by Corollary 4.1.18 (ii), $\text{Post}_{\text{Van}(P)} \models \text{demo}(Q_1)$. The conclusion follows from the definition of $\text{Post}_{\text{Van}(P)}$.

As a consequence of this simple result we have that Vanilla preserves the declarative properties of object-level computed and correct instances of intended queries.

### 5.2.3 From (Weak) Partial Correctness to (Weak) Declarative Soundness

By the Weak Partial Correctness Theorem 4.1.20, $\text{Van}(P)$ is weak partially correct w.r.t. the specification

$$(\text{Pre}_{\text{Van}(P)}, \text{Post}_{\text{Van}(P)})$$

when $\vdash \{\text{Pre}\} P \{\text{Post}\}$ holds. This result allows us to show weak declarative soundness of Vanilla. We recall that we assume that the symbols & and true are not predicate symbols of $L$.

**Theorem 5.2.4** Assume that $\vdash \{\text{Pre}\} P \{\text{Post}\}$ holds. Then $\text{Van}(P)$ is weak declaratively sound with respect to $P$ and $(\text{Pre}, \text{Post})$.

**Proof.** Since $\vdash \{\text{Pre}\} P \{\text{Post}\}$ holds, we have that

$$\vdash \{\text{Pre}_{\text{Van}(P)}\} \text{Van}(P) \{\text{Post}_{\text{Van}(P)}\}$$

holds. Consider now $A \in \text{Pre}$. By definition of $\text{Pre}_{\text{Van}(P)}$, $\text{demo}(A)$ is in $\text{Pre}_{\text{Van}(P)}$. Assume now that $\text{demo}(A) \in M^L_{\text{Van}(P)}$. By Theorem 4.1.20, we have $\text{demo}(A) \in \text{Post}_{\text{Van}(P)}$, which implies $A \in \text{Post}$. □

Thanks to Theorem 4.1.22, we know that $\vdash \{\text{Pre}\} P \{M^L_P \cap \text{Pre}\}$ holds when $\vdash \{\text{Pre}\} P \{\text{Post}\}$ holds. Thus, by the above Theorem, Vanilla is weak declarative sound w.r.t. $P$ and the specification $(\text{Pre}, M^L_P \cap \text{Pre})$. But this exactly means that Vanilla is declarative sound w.r.t. $P$.

**Theorem 5.2.5** Assume that $\vdash \{\text{Pre}\} P \{\text{Post}\}$ holds. Then $\text{Van}(P)$ is weak declaratively sound with respect to $P$ and $\text{Pre}$. □

### 5.2.4 From Partial Correctness to Declarative Correctness

Here, we intend to show that $\text{Post}_{\text{Van}(P)}$ is a well-supported interpretation (w.r.t. Vanilla and $\text{Pre}_{\text{Van}(P)}$) when the postcondition $\text{Post}$ of the object program is well-supported w.r.t. $P$ and $\text{Pre}$.
Suppose $Post$ is well-supported. Then there exist a well-founded poset $(W, >)$ and a function $\| : B \rightarrow W$ such that for any $A \in Post \cap Pre$ there exists $A \leftarrow B_1, \ldots, B_n \in ground_L(P)$ such that

$$\forall i \in [1,n] : Post \models B_i \land |A| > |B_i|.$$  \hfill (5.3)

In order to construct the relevant proof outlines, consider the well-founded poset $(bag(W), \succ_m)$ over finite multiset of $W$ induced by $(W, >)$, and the function:

$$\| \text{demo}(B_1 \& \ldots \& B_n) \| = bag(|B_1|, \ldots, |B_n|)$$

if $\text{demo}(B_1 \& \ldots \& B_n) \in Pre_{Van(P)}$, and $bag()$ otherwise.

Notice that for $\text{demo}(A \& B) \in Pre_{Van(P)}$:

$$\| \text{demo}(A \& B) \| \succ_m \| \text{demo}(A) \|, \| \text{demo}(B) \|.$$  \hfill (5.4)

The next proof outlines show that $Post_{Van(P)}$ is a well-supported interpretation.

(a) \hspace{1cm} \{ \text{true} \}

\hspace{1cm} \{ bag() \}

(b) \hspace{1cm} \{ demo(A \& B) \in Pre_{Van(P)} \land Post \models A, B^- \}

\hspace{1cm} demo(A \& B) \leftarrow \{ \| \text{demo}(A \& B) \| \}

\hspace{2cm} demo(A), \{ \| \text{demo}(A) \| \}

\hspace{2cm} demo(B), \{ \| \text{demo}(B) \| \}

\hspace{2cm} \{ Post \models B^- \}

(c) \hspace{1cm} \{ A \in Post \cap Pre \land Post \models B^- \land A \leftarrow B^- \in ground_L(P) \land \| \text{demo}(A) \| \succ_m \| \text{demo}(B) \| \}

\hspace{1cm} demo(A) \leftarrow \{ \| \text{demo}(A) \| \}

\hspace{2cm} clause(A, B), \{ bag() \}

\hspace{2cm} \{ A \leftarrow B^- \in ground_L(P) \}

\hspace{2cm} demo(B), \{ \| \text{demo}(B) \| \}

\hspace{2cm} \{ Post \models B^- \}

(d) \hspace{1cm} \{ A \leftarrow B^- \in ground_L(P) \}

\hspace{1cm} clause(A, B), \{ bag() \}

The proof outlines are of immediate verification by using the definitions of $Pre_{Van(P)}$ and $Post_{Van(P)}$.

To conclude that $Post_{Van(P)}$ is a well-supported interpretation, we have to show that every $A \in Post_{Van(P)} \cap Pre_{Van(P)}$ is a ground instance of some head clause in a proof outline and the assertion associated with the head holds.
For the atoms whose predicate symbol is \texttt{clause} the conclusion is trivial. Assume now that \texttt{demo}(B_1 \& \ldots \& B_n) \in \texttt{Pre}_{\text{Van}(P)} \cap \texttt{Post}_{\text{Van}(P)}.

If \( n \neq 1 \) we use the proof outline (a) or (b), and observe that the assertion of the head is satisfied by the hypothesis \texttt{demo}(B_1 \& \ldots \& B_n) \in \texttt{Pre}_{\text{Van}(P)} \cap \texttt{Post}_{\text{Van}(P)}.

If \( n = 1 \) then we consider proof outline (c). We have to prove that for \( A \in \texttt{Post} \cap \texttt{Pre} \) there exists \( B^- \) such that

\[
\text{Post} \models B^- \land A \leftrightarrow B^- \in \text{ground}_{\mathbb{L}}(P) \land \|\text{demo}(A)\| \succ_m \|\text{demo}(B)\|.
\]

Such a \( B^- \) clearly exists as a consequence of (5.3).

Starting from this result, we are now in position to show declarative correctness of \texttt{Vanilla}.

**Theorem 5.2.6** Assume that \( \vdash \{\text{Pre}\} P \{\text{Post}\} \) holds. Then \( \text{Van}(P) \) is declaratively correct with respect to \( P \) and \( \text{Pre} \).

**Proof.** By Theorem 4.1.22, \( \vdash \{\text{Pre}\} P \{\text{Post}\} \) implies

\[
\vdash \{\text{Pre}\} P \{\text{sp}(P, \text{Pre})\}.
\]

We now define \( \text{Pre}_{\text{Van}(P)}, \text{Post}_{\text{Van}(P)} \) starting from \( \text{Pre}, \text{sp}(P, \text{Pre}) \). We have that

\[
\vdash \{\text{Pre}_{\text{Van}(P)}\} \text{Van}(P) \{\text{Post}_{\text{Van}(P)}\}
\]

holds. Moreover, since by Theorem 4.1.25 \( \text{sp}(P, \text{Pre}) \) is well-supported, \( \text{Post}_{\text{Van}(P)} \) is well-supported as well. By Theorems 4.1.25 and 4.1.22:

\[
\begin{align*}
\text{sp}(\text{Van}, \text{Pre}_{\text{Van}(P)}) &= \text{Post}_{\text{Van}(P)} \cap \text{Pre}_{\text{Van}(P)} \\
&= M^L_{\text{Van}(P)} \cap \text{Pre}_{\text{Van}(P)}.
\end{align*}
\]

Consider now \( A \in \text{Pre} \). We have \( \text{demo}(A) \in \text{Pre}_{\text{Van}(P)} \). We show the requirements of Definition 5.1.1:

\[
\begin{align*}
\text{demo}(A) &\in M^L_{\text{Van}(P)} \\
\iff &\ \{ (5.5) \land \text{demo}(A) \in \text{Pre}_{\text{Van}(P)} \} \\
\text{demo}(A) &\in \text{Post}_{\text{Van}(P)} \\
\iff &\ \{ \text{Definition of Post}_{\text{Van}(P)} \} \\
A &\in \text{sp}(P, \text{Pre}) \\
\iff &\ \{ \text{Theorem 4.1.22 and } A \in \text{Pre} \} \\
A &\in M^L_P.
\end{align*}
\]

\[\square\]

As a special case consider any logic program \( P \). Since \( \vdash \{B_L\} P \{B_L\} \) holds, the above Theorem implies that for every \( A \in B_P \subseteq B_L \), \( A \in M^L_P \iff \text{demo}(A) \in M^L_{\text{Van}(P)} \), which implies the notion of correctness usually considered in the literature.
5.3. Proving declarative correctness

Can we generalize the method used to verify Vanilla to arbitrary meta-interpreters? The answer to this question is affirmative, and this section is devoted to illustrate a general method to tackle the proof of correctness of a generic meta-interpreter. We stress that this method was literally extracted from the verification of the Vanilla in a systematic way. As an illustration, we apply it to several modifications and extensions of Vanilla, supporting amalgamation and non-standard mechanisms.

5.3.1 A General Method

The reasoning followed in the proof of correctness of the Vanilla can be easily generalized. The next definition provides us with a general criterion of correctness.

**Definition 5.3.1** Suppose that ⊢ \{Pre\} P \{Post\} holds with Post well-supported w.r.t. P and Pre. A program V satisfies the General Criterion [GC] with respect to P and Pre iff for some Pre’, Post’, the following properties hold:
1. ⊢ \{Pre’\} V \{Post’\},
2. Post’ well-supported w.r.t. V and Pre’,
3. ∀A ∈ Pre. demo(A) ∈ Pre’,
4. ∀A ∈ Pre. A ∈ Post ⇔ demo(A) ∈ Post’.

It is worth noting that neither $M_L^P$ nor $M_L^V$ appear in the formulation of the General Criterion. This fact makes it of practical interest. On the contrary, we would not need any criterion. We could just compare them to check declarative correctness.

Conditions (3,4) relate Pre’, Post’ with Pre, Post. However, in general, it is not required that Pre, Post are given. In the case of Vanilla, for instance, we assumed generic Pre, Post, with Post supposed to be well-supported. In particular, (3-4) imply for A ∈ Pre:

$$A ∈ Post \cap Pre ⇔ demo(A) ∈ Post’ ∩ Pre’. \quad \square$$

Since Post and Post’ are well-supported, by Theorems 4.1.25 and 4.1.22, the relation above implies:

$$A ∈ M_L^P ⇔ demo(A) ∈ M_L^V,$$

i.e. declarative correctness of V w.r.t. P and Pre.
These intuitions are formalized in the following main theorem, stating that the General Criterion is a necessary and sufficient condition for declarative correctness of meta-interpreters w.r.t. programs $P$ and sets $Pre$ such that $\vdash \{Pre\} P \{Post\}$ holds.

**Theorem 5.3.2** Assume that $\vdash \{Pre\} P \{Post\}$ holds with $Post$ well-supported w.r.t. $P$ and $Pre$. A program $V$ is declaratively correct w.r.t. $P$ and $Pre$ iff it satisfies $[GC]$ w.r.t. $P$ and $Pre$.

**Proof.** (if). Assume $(1-4)$. We now calculate, for $A \in Pre$:

\[
A \in M_P^L \quad \Leftrightarrow \quad \{ \text{Theorem 4.1.22} \} \\
A \in sp(P, Pre) \quad \Leftrightarrow \quad \{ \text{Post well-supported and Theorem 4.1.25} \} \\
A \in Post \cap Pre \quad \Leftrightarrow \quad \{ \text{Definition of Post} \} \\
demo(A) \in Post' \quad \Leftrightarrow \quad \{ \text{Definition of Post} \} \quad \text{and} \quad \{ \text{Definition of Post} \}
\]

(only-if). Let us consider $Pre' = B_L$ and $Post' = M_V^L$. Since $\vdash \{B_L\} V \{B_L\}$ holds, by Theorems 4.1.22, 4.1.25 we have that $\vdash \{B_L\} V \{M_V^L\}$ holds, i.e. $(1)$, and $M_V^L = sp(V, B_L)$ is well-supported, i.e. $(2)$. $(3)$ is obvious.

To prove $[GC](4)$, we calculate, for $A \in Pre$:

\[
A \in Post \quad \Leftrightarrow \quad \{ \text{Post well-supported and Theorems 4.1.22,4.1.25} \} \\
A \in M_P^L \quad \Leftrightarrow \quad \{ \text{Definition of Post} \} \\
demo(A) \in M_V^L \quad \Leftrightarrow \quad \{ \text{Definition of Post} \} \quad \text{and} \quad \{ \text{Definition of Post} \}
\]

\(\Box\)
By weakening conditions (1-4), a variant of the method is able to reason about declarative soundness. Below, the Soundness Criterion [sc] is introduced, together with a result showing that [sc] is a proof method for declarative soundness.

**Definition 5.3.3** Suppose $\vdash \{\text{Pre}\} P \{\text{Post}\}$ with Post well-supported w.r.t. $P$ and Pre. A program $V$ satisfies the Soundness Criterion [sc] with respect to $P$ and Pre if, for some $\text{Pre'}, \text{Post'}$, the following properties hold:

1. $\vdash \{\text{Pre'}\} V \{\text{Post'}\}$,
2. $\forall A \in \text{Pre}. \quad \text{demo}(A) \in \text{Pre'}$,
3. $\forall A \in \text{Pre}. \quad A \in \text{Post} \iff \text{demo}(A) \in \text{Post'}$.

**Theorem 5.3.4** Suppose that $\vdash \{\text{Pre}\} P \{\text{Post}\}$ with Post well-supported w.r.t. $P$ and Pre. A program $V$ is declaratively sound w.r.t. $P$ and Pre if it satisfies [sc] w.r.t. $P$ and Pre.

A further weakening allows us to characterize weak soundness.

**Definition 5.3.5** Suppose $\vdash \{\text{Pre}\} P \{\text{Post}\}$. A program $V$ satisfies the Weak Soundness Criterion [wsc] with respect to $P$ and $(\text{Pre}, \text{Post})$ iff, for some $\text{Pre'}, \text{Post'}$, the following properties hold:

1. $\vdash \{\text{Pre'}\} V \{\text{Post'}\}$,
2. $\forall A \in \text{Pre}. \quad \text{demo}(A) \in \text{Pre'}$,
3. $\forall A \in \text{Pre}. \quad A \in \text{Post} \iff \text{demo}(A) \in \text{Post'}$.

**Theorem 5.3.6** Suppose $\vdash \{\text{Pre}\} P \{\text{Post}\}$. A program $V$ is weak declaratively sound w.r.t. $P$ and $(\text{Pre}, \text{Post})$ iff it satisfies [wsc] w.r.t. $P$ and $(\text{Pre}, \text{Post})$.

### 5.3.2 Amalgamation

Amalgamating an object program with its meta-interpreter means considering their union as a single program. This technique reveals to be a powerful tool for extending the logic programming paradigm with new features, provided by the intertwined use of the meta-program, which may offer new control facilities, and the object program, which may use these facilities. Assuming ambivalent syntax is a basic requirement for amalgamating object and meta-programs.

**Simple amalgamation**

Problems may arise if amalgamation is not used in a disciplined way.

**Example 5.3.7 (Strange)** Consider this STRANGE program:

```prolog
p ← demo(q).
q.
```
and the following meta-interpreter $V$ for STRANGE:

$$\text{demo}(q).$$

It is simple to observe that $V$ is declaratively correct w.r.t. STRANGE and $B_L$. However, amalgamating $V$ and STRANGE leads to the following inconsistency:

$$V \cup \text{STRANGE} \models p \quad \text{while} \quad V \cup \text{STRANGE} \not\models \text{demo}(p).$$

Such a behavior is unsafe, since $V \cup \text{STRANGE}$ should intuitively be declaratively correct with respect to itself. \qed

Next, we provide a formal account of this intuition, and a sufficient condition for safe amalgamation.

**Definition 5.3.8** We say that $V \cup P$ is a safe amalgamation for $V$ and $P, \text{Pre}$ iff for every $A \in \text{Pre}$:

$$A \in M^L_V \cup P \quad \text{iff} \quad \text{demo}(A) \in M^L_V \cup P.$$  

\qed

An alternative way of reading this definition is to require that $V \cup P$ is declaratively correct w.r.t. $V \cup P$ and $\text{Pre}$. The following theorem gives us a sufficient condition for safety of simple amalgamation. The underlying idea is to avoid interference between the object- and the meta-level.

**Theorem 5.3.9** Consider a program $P$ such that $\vdash \{\text{Pre}\} P \{\text{Post}\}$ holds with $\text{Post}$ well-supported w.r.t. $P$ and $\text{Pre}$, and a program $V$ which satisfies $[\text{GC}]$ w.r.t. $P$ and $\text{Pre}$. Assume that:

1. $\vdash \{\text{Pre} \cup \text{Pre}'\} V \cup P \{\text{Post} \cup \text{Post}'\}$, and
2. $\text{Post}' \cap \text{Pre} \subseteq \text{Post}$, and
3. no atom in $\text{Post}$ has $\text{demo}$ as predicate symbol,

where $\text{Pre}', \text{Post}'$ are the pre- and postconditions used in the proof of $[\text{GC}]$.

Then for every $A \in \text{Pre}$ the following statements are equivalent:

(i) $A \in M^L_V \cup P$,
(ii) $\text{demo}(A) \in M^L_V$,
(iii) $\text{demo}(A) \in M^L_V \cup P$,
(iv) $A \in M^L_P$.

In particular, this implies that $V \cup P$ is a safe amalgamation for $V$ and $P, \text{Pre}$. 
5.3. Proving declarative correctness

Proof. Let \( A \) be in \( \text{Pre} \).

\( (i) \rightarrow (ii) \). If \( A \in M^L_{V \cup P} \) then, by \( (i) \) and Theorem 4.1.20, \( A \in \text{Post} \cup \text{Post}' \). This implies that \( A \in \text{Post} \). By [GC] \( (3, 4) \), this implies \( \text{demo}(A) \in \text{Post}' \cap \text{Pre}' \). By [GC] \( (1, 2) \) and Theorems 4.1.22, 4.1.25, this implies \( \text{demo}(A) \in M^L_{V} \).

\( (ii) \rightarrow (iii) \). Immediate by noting that \( M^L_{V} \subseteq M^L_{V \cup P} \).

\( (iii) \rightarrow (iv) \). Let \( \text{demo}(A) \) be in \( M^L_{V \cup P} \). By [GC] \( (3) \), \( A \in \text{Pre} \) implies \( \text{demo}(A) \in \text{Pre}' \). By \( (1) \) and Theorem 4.1.20, \( \text{demo}(A) \in \text{Post} \cup \text{Post}' \). By \( (3) \), \( \text{demo}(A) \in \text{Post}' \). By [GC] \( (4) \), this implies \( A \in \text{Post} \). Since \( \text{Post} \) is well-supported, we conclude that \( A \in M^L_{V} \).

\( (iv) \rightarrow (i) \). Immediate by noting that \( M^L_{V} \subseteq M^L_{V \cup P} \). \( \square \)

Example 5.3.10 Consider the Vanilla meta-interpreter and a program \( \text{Post} \) such that \( \vdash \{ \text{Pre} \} \text{P} \{ \text{Post} \} \) holds. Without lack of generality, we can assume that \( \text{Post} \) is well-supported. Moreover, \( \text{Van}(P) \) satisfies [GC] w.r.t. \( P \) and \( \text{Pre} \), by considering \( \text{Pre}_{\text{Van}(P)} \) and \( \text{Post}_{\text{Van}(P)} \).

The hypothesis of the theorem above are satisfied if we assume that the predicates \( \text{demo} \) and \( \text{clause} \) do not appear in \( P \) or \( \text{Pre} \). Since \( \text{Post} \) is well-supported, we have that \( \text{Post} \subseteq \text{Pre} \), and then \( \text{demo} \) and \( \text{clause} \) do not appear in \( \text{Post} \). Therefore, \( \text{Pre} \) and \( \text{Pre}_{\text{Van}(P)} \) are disjoint sets, and the same is for \( \text{Post} \) and \( \text{Post}_{\text{Van}(P)} \). From this facts, hypothesis \( (1-3) \) are readily checked.

As a result, \( \text{Van}(P) \cup P \) is a safe amalgamation for \( P \) and \( \text{Pre} \), when for some \( \text{Post} \vdash \{ \text{Pre} \} \text{P} \{ \text{Post} \} \) holds. In the case, \( \text{Pre} = B_P \), this is a classical result on safety of the simple amalgamation (see e.g. [98]). \( \square \)

Reflection up

A more liberal way of giving a meaning to the program STRANGE of Example 5.3.7 is to interpret \( \text{demo}(A) \) as a call to the meta-level. This possibility is usually referred to as reflection up [51]. Let us consider the case of Vanilla. We implement a call to the meta-level by adding to Vanilla the clause:

\[
\text{demo}( \text{demo}(A) ) \leftarrow \text{demo}( A ).
\]

We denote by \( \text{VU}(P) \) the extended Vanilla instantiated by the object program \( P \). The next definition introduces \( \text{Pre}_{\text{VU}(P)} \) and \( \text{Post}_{\text{VU}(P)} \).

Definition 5.3.11 For a program \( P \) and a specification \( \{ \text{Pre}, \text{Post} \} \), we define:

\[
\text{demo}(B_1 \& \cdots \& B_n) \in \text{Pre}_{\text{VU}(P)} \text{ iff }
\vdash \{ \text{Pre} \cup \text{Pre}_{\text{VU}(P)} \} B_1, \ldots, B_n \{ \text{Post} \cup \text{Post}_{\text{VU}(P)} \}
\]

\[
\text{clause}(A, B) \in \text{Pre}_{\text{VU}(P)} \text{ iff } \text{true}
\]

\[
\text{demo}(B_1 \& \cdots \& B_n) \in \text{Post}_{\text{VU}(P)} \text{ iff }
\]
\( \text{Post} \cup \text{Post}_{VU}(P) \models B_1, \ldots, B_n \)

\[
\text{clause}(A, B_1 \& \ldots \& B_n) \in \text{Post}_{VU}(P) \quad \text{iff} \quad A \leftarrow B_1, \ldots, B_n \in \text{ground}_L(P)
\]

\( \Box \)

The next theorem shows that \( VU(P) \) is a declaratively correct meta-interpreter w.r.t. \( VU(P) \cup P \) and \( \text{Pre} \), formalizing the intuitive notion of that a call to the \text{demo} predicate is interpreted at meta-level.

**Theorem 5.3.12** Assume that

\[ \vdash \{ \text{Pre} \cup \text{Pre}_{VU}(P) \} P \{ \text{Post} \cup \text{Post}_{VU}(P) \} \]

holds with \( \text{Post} \cup \text{Post}_{VU}(P) \) well-supported w.r.t. \( VU(P) \cup P \) and \( \text{Pre} \cup \text{Pre}_{VU}(P) \).

If no head of a clause from \( P \) and no atom in \( \text{Pre} \) and \( \text{Post} \) has \text{demo} or \text{clause} as predicate symbol, then:

(i) \( VU(P) \) is declaratively correct w.r.t. \( VU(P) \cup P \) and \( \text{Pre} \), and

(ii) \( VU(P) \) is a safe amalgamation for \( VU(P) \) and \( P \), \( \text{Pre} \).

**Proof.** It is readily checked that

\[ \vdash \{ \text{Pre}_{VU}(P) \} VU(P) \{ \text{Post}_{VU}(P) \} \quad (5.6) \]

holds. Moreover, since no \text{demo} or \text{clause} atom is in \( \text{Pre} \) and \( \text{Post} \), this and

\[ \vdash \{ \text{Pre} \cup \text{Pre}_{VU}(P) \} P \{ \text{Post} \cup \text{Post}_{VU}(P) \} \]

imply

\[ \vdash \{ \text{Pre} \cup \text{Pre}_{VU}(P) \} P \cup VU(P) \{ \text{Post} \cup \text{Post}_{VU}(P) \}. \quad (5.7) \]

By showing that \( VU(P) \) satisfies \([\text{gc}]\) with respect to \( VU(P) \cup P \) and \( \text{Pre} \cup \text{Pre}_{VU}(P) \), by Theorem 5.3.2 we can conclude that \( VU(P) \) is \text{declaratively correct} with respect to \( VU(P) \cup P \) and \( \text{Pre} \cup \text{Pre}_{VU}(P) \). As a consequence, it is \text{declaratively correct} with respect to \( VU(P) \cup P \) and \( \text{Pre} \) as well, i.e. (i).

Let us show that \( VU(P) \) satisfies \([\text{gc}]\). We define \( \text{Pre}' \) and \( \text{Post}' \) as \( \text{Pre}_{VU}(P) \) and \( \text{Post}_{VU}(P) \) respectively.

(5.6) shows \([\text{gc}]\) (i). Moreover, by hypothesis \( \text{Post} \cup \text{Post}_{VU}(P) \) is well-supported w.r.t. \( P \cup VU(P) \) and \( \text{Pre} \cup \text{Pre}_{VU}(P) \). Since no head of a clause from \( P \) has \text{demo} or \text{clause} as predicate symbol, we have that \( \text{Post} \cup \text{Post}_{VU}(P) \) is well-supported w.r.t. \( VU(P) \) and \( \text{Pre}_{VU}(P) \). Finally, since no atom in \( \text{Post} \) has \text{demo} or \text{clause} as predicate symbol, we conclude that \( \text{Post}_{VU}(P) \) is well-supported w.r.t. \( VU(P) \) and \( \text{Pre}_{VU}(P) \), i.e. \([\text{gc}]\) (2).

In addition, for \( A \in \text{Pre} \cup \text{Pre}_{VU}(P) \) we have directly from Definition 5.3.11, that:
[gc] (3) \( \text{demo}(A) \in \text{Pre}' \), and
[gc] (4) \( A \in \text{Post} \cup \text{Post}_{\text{VU}(P)} \) iff \( \text{demo}(A) \in \text{Post}' \).

This concludes the proof of (i). Let us show (ii). For \( A \in \text{Pre} \), we calculate:

\[
A \in M^L_P \cup \text{VU}(P)
\]
iff \( \{ A \in \text{Pre}, (5.7) \text{ and Post } \cup \text{Post}_{\text{VU}(P)} \text{ well-supported } \} \)
iff \( A \in \text{Post} \cup \text{Post}_{\text{VU}(P)} \)
iff \( \{ \text{Definition of Post}_{\text{VU}(P)} \} \)
iff \( \text{demo}(A) \in \text{Post}_{\text{VU}(P)} \)
iff \( \{ \text{no atom in Post has demo or clause as predicate symbol} \} \)
iff \( \text{demo}(A) \in \text{Post} \cup \text{Post}_{\text{VU}(P)} \)
iff \( \{ \text{demo}(A) \in \text{Pre}_{\text{VU}(P)}, (5.7) \text{ and Post } \cup \text{Post}_{\text{VU}(P)} \text{ well-sup.} \} \)
iff \( \text{demo}(A) \in M^L_P \cup \text{VU}(P) \).

\[ \square \]

**Example 5.3.13** *(Strange Ctd)* Consider again the program \textsc{Strange} of Example 5.3.7.

\[
p \leftarrow \text{demo}(q).
q.
\]

Given \( \text{Pre} = \text{Post} = \{ p, q \} \), we have:

\[
\text{demo}(q) \in \text{Pre}_{\text{VU(Strange)}} \quad \text{demo}(q) \in \text{Post}_{\text{VU(Strange)}}.
\]

From this, it follows that

\[ \vdash \{ \text{Pre } \cup \text{Pre}_{\text{VU(Strange)}} \} \text{Strange} \{ \text{Post } \cup \text{Post}_{\text{VU(Strange)}} \} \]

holds, and that \( \text{Post } \cup \text{Post}_{\text{VU(Strange)}} \) is a well-supported interpretation w.r.t. \textsc{Strange} \( \cup \text{VU(Strange)} \) and \( \text{Pre } \cup \text{Pre}_{\text{VU(Strange)}} \). Finally, no head of a clause from \textsc{Strange} and no atom in \text{Pre} or \text{Post} has \text{demo or clause} as predicate symbol. Therefore, we are in the hypotheses of Theorem 5.3.12, and can conclude that:

\[
p \in M^L_{\text{Strange} \cup \text{VU(Strange)}}
\]
iff \( \text{demo}(p) \in M^L_{\text{VU(Strange)}} \)
iff \( \text{demo}(p) \in M^L_{\text{Strange} \cup \text{VU(Strange)}} \)

and analogously for \( q \). \[ \square \]
Reflection down

We conclude the overview of the amalgamation techniques, by looking at reflection down, i.e., the possibility of calling the object-level from the meta-level.

The simplest reflective meta-interpreter is the standard Prolog implementation of the predicate call by means of meta-variables [150]. For a program $P$ we denote by $Call(P)$, the meta-program consisting of $P$ together with the clause:

$$\text{demo}(X) \leftarrow X.$$  

Strictly speaking, this is not a Horn clause, since variable $X$ appears in the body in an atom position. Besides, the use of meta-variables may lead to run time errors: a computation ends in an error when a meta-variable is selected.

We tackle the problem by assuming the use of the terms-as-atoms ambivalent syntax. Within our context of ambivalent syntax, we simply assume $\Sigma_L = \Pi_L$, and provide a sufficient condition for avoiding that LD-derivations end in errors. Moreover, we observe that Definition 4.1.1 makes sense even if arbitrary instances of clauses with meta-variables are considered: pre- and postconditions can be used to describe the meaningful instances of clauses.

Let us consider a program $P$ such that $\vdash \{\text{Pre}\} P \{\text{Post}\}$ holds, and assume that no atom in $\text{Pre}$ or $\text{Post}$ or $P$ has $\text{demo}$ as the predicate symbol. We show that $Call(P)$ satisfies $[\text{GC}]$ w.r.t. $P$ and $\text{Pre}$.

First, we point out that by defining:

$$\text{Pre}_{Call(P)} = \text{Pre} \cup \{\text{demo}(A) \mid A \in \text{Pre}\},$$  
$$\text{Post}_{Call(P)} = \text{Post} \cup \{\text{demo}(A) \mid A \in \text{Post}\},$$

we have that

$$\vdash \{\text{Pre}_{Call(P)}\} Call(P) \{\text{Post}_{Call(P)}\}.$$  

The proof obligations for the clauses of $P$ are readily checked, since $\vdash \{\text{Pre}\} P \{\text{Post}\}$ holds, and

$$\text{rel}(A) \neq \text{demo} \land A \in \text{Pre}_{Call(P)} \Leftrightarrow A \in \text{Pre}$$

and analogously for $\text{Post}_{Call(P)}$. The proof outline for the clause of $\text{demo}$ is straightforward.

$$\{X \in \text{Pre}\}$$  
$$\text{demo}(X) \leftarrow$$  
$$\{X \in \text{Pre}\}$$  
$$X.$$  
$$\{X \in \text{Post}\}$$  
$$\{X \in \text{Post}\}$$
This concludes the proof of \([\text{GC}]\) (1). Let us show that \([\text{GC}]\) (2) holds. Assuming \(\text{Post}\) well-supported, we show that \(\text{Post}_{\text{Call}(P)}\) is well-supported as well. By hypothesis, there exists a well-founded poset \((W, >)\) and a function \(\|\) : \(B_L \rightarrow W\) such that for any \(A \in \text{Post} \cap \text{Pre}\) there exists \(A \leftarrow B_1, \ldots, B_n \in \text{ground}_L(P)\) such that:

\[
\forall i \in [1, n] : \text{Post} \models B_i \land |A| > |B_i|.
\]

Without lack of generality, we can assume \(W\) without maximal elements.

Consider now a function \(\| \|_\varepsilon : B_L \rightarrow W\) such that, for \(A \in \text{Pre}\):

\[
\|\text{demo}(A)\|_\varepsilon > |A| \text{ and } \|A\|_\varepsilon = |A|.
\]

We claim that there exist proof outlines for the clauses of \(\text{Call}(P)\) and \(\| \|_\varepsilon\). This is immediate for the clauses from \(P\), by hypothesis. The proof outline for the clause of \(\text{demo}\) is:

\[
\begin{align*}
\{X \in \text{Pre} \cap \text{Post}\} & \quad \text{demo}(X) \leftarrow \quad \{\|\text{demo}(X)\|_\varepsilon\} \\
& \quad X. \quad \{\|X\|_\varepsilon\} \\
& \quad \{X \in \text{Post}\}
\end{align*}
\]

Therefore, we proved \([\text{GC}]\) (2). \([\text{GC}]\) (3-4) hold by definition of \(\text{Pre}_{\text{Call}(P)}\) and \(\text{Post}_{\text{Call}(P)}\). As a consequence of the main Theorem 5.3.2, we are in the position to state declarative correctness of \(\text{Call}(P)\).

**Theorem 5.3.14** Assume that \(\Pi_L = \Sigma_L\), that \(\vdash \{\text{Pre}\} P \{\text{Post}\}\) holds, and that no atom in \(\text{Pre}\) or \(\text{Post}\) or \(P\) has \(\text{demo}\) as predicate symbol. Then \(\text{Call}(P)\) is declaratively correct w.r.t. \(P\) and \(\text{Pre}\).

Moreover, for any query \(Q\) such that \(\vdash \{\text{Pre}_{\text{Call}(P)}\} Q \{\text{Post}_{\text{Call}(P)}\}\), no meta-variable is selected in a LD-derivation for \(\text{Call}(P)\) and \(Q\).

**Proof.** We proved that \(\text{Call}(P)\) satisfies \([\text{GC}]\) w.r.t. \(P\) and \(\text{Pre}\). Then, by Theorem 5.3.2, it is declaratively correct w.r.t. \(P\) and \(\text{Pre}\).

Suppose now that a meta-variable \(X\) is selected in a LD-derivation for \(\text{Call}(P)\) and \(Q\). Then, by Corollary 4.1.18 (i), we have

\[
\text{Pre}_{\text{Call}(P)} \models X
\]

and, since \(\Pi_L = \Sigma_L\), \(\text{Pre}_{\text{Call}(P)} = B_L\). By definition of \(\text{Pre}_{\text{Call}(P)}\), this implies that \(\text{demo}(t)\) is in \(\text{Pre}\) for some \(t\). This contradicts the hypothesis that no atom with predicate symbol \(\text{demo}\) belongs to \(\text{Pre}\). □
5.3.3 Extensions

Several other extensions of Vanilla are reported in the literature (see e.g., [150]). The approach based on the general criterion $[gc]$ can be applied systematically. However, we need to extend the definition of declarative correctness and $[gc]$ to consider meta-predicates $\text{demo}$ with more than one argument. A simple way to do that is to consider the top level predicate $\text{demo}2$ defined as follows:

$$ \text{demo2}(A) \leftarrow \text{demo}(A, B). $$

where $B$ is the collection of the $h$ additional arguments of $\text{demo}$. In this case, however, Definition 5.1.1 of declarative correctness states a weak relation between $A \in \text{Pre}$ and the other arguments of $\text{demo}$, namely that:

$$ A \in M_P^L \text{ iff there exists } B \in U^h_L \text{ such that } \text{demo}(A, B) \in M_P^L. $$

A stronger relation is introduced next. We denote by $\mathcal{R}$ an arbitrary relation.

**Definition 5.3.15** Let $\mathcal{R}$ be a relation on $B_L \times 2^{BL} \times B_L^h$. Given the $h+1$-ary predicate $\text{demo}$, we say that a program $V$ is $\mathcal{R}$-declaratively correct with respect to a program $P$ and a set $\text{Pre} \subseteq B_L$ iff for every $A \in \text{Pre}$ and $B \in U^h_L$:

$$ \mathcal{R}(A, M_P^L \cap \text{Pre}, B) \text{ iff } \text{demo}(A, B) \in M_P^L. $$

\[\square\]

Definition 5.1.1 is a special case of this definition, obtained by considering:

$$ \mathcal{R}(A, M_P^L \cap \text{Pre}, B) \overset{def}{=} A \in M_P^L \cap \text{Pre}. $$

By simple modifications of conditions (3,4) in Definition 5.3.1, Theorem 5.3.2 extends to $\mathcal{R}$-declarative correctness.

**Definition 5.3.16** Let $\mathcal{R}$ a relation on $B_L \times 2^{BL} \times B_L^h$, and $P$ a program such that $\vdash \{\text{Pre}\} P \{\text{Post}\}$ holds with $\text{Post}$ well-supported w.r.t. $P$ and $\text{Pre}$.

A program $V$ satisfies the $\mathcal{R}$-General Criterion $[\mathcal{R}-gc]$ with respect to $P$ and $\text{Pre}$ iff for some $\text{Pre}', \text{Post}'$, the following properties hold:

1. $\vdash \{\text{Pre}'\} V \{\text{Post}'\}$,
2. $\text{Post}'$ well-supported w.r.t. $V$ and $\text{Pre}'$,
3. $\forall B \in U^h_L \forall A \in \text{Pre}. \text{demo}(A, B) \in \text{Pre}'$,
4. $\forall B \in U^h_L \forall A \in \text{Pre}. \mathcal{R}(A, \text{Post} \cap \text{Pre}, B) \leftrightarrow \text{demo}(A, B) \in \text{Post}'$.  \[\square\]
5.3. Proving declarative correctness

**Theorem 5.3.17** Let \( \mathcal{R} \) be a relation on \( B_L \times 2^{B_L} \times B_L^1 \).

Assume that \( \vdash \{ \text{Pre} \} P \{ \text{Post} \} \) holds with \( \text{Post} \) well-supported w.r.t. \( P \) and \( \text{Pre} \). A program \( V \) is \( \mathcal{R} \)-declaratively correct w.r.t. \( P \) and \( \text{Pre} \) if it satisfies \( [\mathcal{R}\text{-GC}] \) w.r.t. \( P \) and \( \text{Pre} \). \( \square \)

In the rest of this section, we briefly sketch the correctness proof of one extension of \textit{Vanilla}, which constructs a proof tree while solving a query. For a program \( P \), we denote by \( PTB(P) \) the program:

\[
\begin{align*}
demo(\text{empty}, \text{true}) & . \\
demo( X \& Y, TX \& TY ) & \leftarrow \\
\quad \quad demo( X, TX ) , \\
\quad \quad demo( Y, TY ) . \\
demo( X, X \text{ if } T) & \leftarrow \\
\quad \quad \text{clause}(X, Y) , \\
\quad \quad demo(Y, T) . \\
\text{clause}( A, B_1, \ldots, B_n ) & . \\
\quad \text{for every } A \leftarrow B_1, \ldots, B_n \in P .
\end{align*}
\]

This meta-interpreter is devised for yielding a justification for the logical consequences of the object program in terms of proof trees.

**Definition 5.3.18** The following rules define the \textit{proof tree terms} for ground queries, and a program \( P \):

1. \( T_1 \& \ldots \& T_n \) is a proof tree term for the ground query \( B_1, \ldots, B_n \), if for \( i \in [1,n] (n \geq 0) \), \( T_i \) is a proof tree term for \( B_i \),

2. \( A \text{ if } T \) is a proof tree term for the ground atom \( A \), if \( T \) is a proof tree term for the query \( B_1, \ldots, B_n \), and \( A \leftarrow B_1, \ldots, B_n \in \text{ground}_L(P) \).

\( \square \)

Notice that the base case is (1) for \( n = 0 \), i.e. \text{true} is a proof tree term for the empty query. Also, the bijection between ground proof trees (see preliminaries) and proof tree terms is clear.

We recall that Clark [42] showed that \( A \in M^p_L \) iff there exists a ground proof tree for \( A \), i.e. iff there exists a proof tree term for \( A \). In this sense, a proof tree (term) is then a justification for \( A \). After these preliminaries, we construct suitable pre- and postconditions for \( PTB(P) \) starting from those for \( P \):

\[
\begin{align*}
demo(B_1 \& \ldots \& B_n, T) \in \text{Pre}_{PTB(P)} & \text{ iff } \\
\vdash \{ \text{Pre} \} B_1, \ldots, B_n \{ \text{Post} \} . \\
\text{clause}(A, B) \in \text{Pre}_{PTB(P)} & \text{ iff } \text{true}
\end{align*}
\]
As a consequence of Theorem 5.3.17 we have the following result.

**Theorem 5.3.19** Assume that \( \vdash \{ \text{Pre} \} P \{ \text{Post} \} \) holds. Then for \( A \in \text{Pre} \) and every ground term \( T \):

\[
T \text{ is a proof tree term for } A \iff \text{demo}(A,T) \in M^L_{PTB(P)}.
\]

**Proof.** It is readily checked that \( PTB(P) \) satisfies the general criterion \([\mathcal{R}-\text{gc}]\) w.r.t. \( P \) and \( \text{Pre} \), where:

\[
\mathcal{R}(A,I,T) \xrightarrow{\text{def}} I \models A \land T \text{ is a proof tree term for } A.
\]

By Theorem 5.3.17, we have that \( PTB(P) \) is \( \mathcal{R} \)-declaratively correct w.r.t. \( P \) and \( \text{Pre} \), i.e. for \( A \in \text{Pre} \):

\[
M^L_P \cap \text{Pre} \models A \land T \text{ is a proof tree term for } A \iff \text{demo}(A,T) \in M^L_{PTB(P)}.
\]

The conclusion follows by noting that \( M^L_P \cap \text{Pre} \models A \land T \) is a proof tree term for \( A \) iff \( T \) is a proof tree for \( A \).

\[\square\]

## 5.4 Vanilla Ctd

In this section, we continue applying the results of Chapter 4 to the case study of *Vanilla*. In particular, we address correct and computed instances characterization, and (weak) total correctness.

### 5.4.1 Correct and Computed Instances

So far, we limited our attention to relations between ground consequences of the object-level and meta-level programs, i.e. to ground correct instances of queries. Here we show how it is possible to study relations between correct and computed instances of the object program and of *Vanilla*. The approach is easily generalizable to arbitrary meta-interpreters.

The next result states an expected relation between correct instances of the object program and of *Vanilla*, under the weak hypothesis that the underlying language \( L \) contains infinitely many function symbols.
Theorem 5.4.1 Assume that $\vdash \{\text{Pre}\} P \{\text{Post}\}$ and $\vdash \{\text{Pre}\} Q^- \{\text{Post}\}$ hold and $\Sigma_L$ contains infinitely many symbols. Then $Q_1^-$ is a correct instance of $P$ and $Q^-$ iff $\text{demo}(Q_1^-)$ is a correct instance of $\text{Van}(P)$ and $\text{demo}(Q)$.

Proof. Obviously, if $Q_1^-$ is not an instance of $Q^-$ the conclusion is true.

Suppose now it is. By Theorems 4.1.22 and 4.1.25, we can assume that $\text{Post}$ is well-supported. Thus we are in the hypotheses of Theorem 4.1.29 (i), and can conclude that $Q_1^-$ is a correct instance of $P$ and $Q^-$ iff $\text{Post} \models Q_1^-$. By the same Theorem, and the fact that $\vdash$ and well-supportedness of postconditions lift from $P$ up to $\text{Van}(P)$, we have that $\text{demo}(Q_1^-)$ is a correct instance of $\text{Van}(P)$ and $\text{demo}(Q)$ iff $\text{Post}_{\text{Van}(P)} \models \text{demo}(Q_1^-)$.

The conclusion of the theorem follows by noting that, by definition of $\text{Post}_{\text{Van}(P)}$:

$$\text{Post} \models Q^- \text{ iff } \text{Post}_{\text{Van}(P)} \models \text{demo}(Q_1^-).$$

Let us turn the focus on computed instances. In Section 4.1.6 at page 116, we have discussed some sufficient conditions that allow us to provide a declarative characterization of computed instances. Unfortunately, not all of them lift from the object program up to $\text{Vanilla}$.

Example 5.4.2 Consider the simple program:

\begin{align*}
p(X) & \leftarrow q(X), r(0). \\
p(X) & \leftarrow q(0), r(X).
\end{align*}

and the specification $\{p(0), q(0)\}, 0$. Obviously, SEM1 is satisfied. On the contrary, consider the definition of the predicate $\text{clause}$ in $\text{Vanilla}$:

\begin{align*}
\text{clause}(p(X), [q(X), r(0)]).
\text{clause}(p(X), [q(0), r(X)]).
\end{align*}

The body of the common ground instance $\text{clause}(p(0), [q(0), r(0)])$ is trivially true in $\emptyset$. Thus SEM1 is not satisfied.

On the contrary, conditions SYN1 and SEM2 lift.

Theorem 5.4.3 Assume that $\vdash \{\text{Pre}\} P \{\text{Post}\}$ holds, and that $P$ satisfies conditions SYN1 and SEM2.

Then $\text{Van}(P)$ also satisfies conditions SYN1 and SEM2.

Proof. By the assumption that $\vdash \{\text{Pre}\} P \{\text{Post}\}$ holds,

$$\vdash \{\text{Pre}_{\text{Van}(P)}\} \text{Van}(P) \{\text{Post}_{\text{Van}(P)}\}$$
holds as well. Also, for any ground query $Q^-:$
\[
Post \models Q^- \text{ iff } Post_{Van(P)} \models demo(Q).
\]

(SYN1). Consider a pair of disjoint variants of clauses from $Van(P).$ If the two clauses define the $demo$ predicate, then they clearly do not unify. If the two clauses define the $clause$ predicate, then they unify iff the object program clauses unify, i.e. $P$ violates SYN1. This shows that $Van(P)$ satisfies SYN1.

(SEM2). The only non trivial case arises considering clause $(d3)$ of Vanilla. Assume by contradiction that for two distinct instances of $(d3),$ say
\[
demo(A) \leftarrow clause(A,B), demo(B) \]
\[
demo(A) \leftarrow clause(A,B'), demo(B'),
\]
we have $Post_{Van(P)} \cap Pre_{Van(P)} \models demo(A), clause(A,B), demo(B), clause(A,B'), demo(B').$ By definition of $Post_{Van(P)}$ we have that $A \leftarrow B^-$ and $A \leftarrow B'^-$ are two distinct clauses from $ground(P).$ Moreover, by (5.8) we have $Post \models A \land B^- \land B'^-.$ If the two distinct clauses are ground instances of the same clause in $P$ then the object program $P$ violates SEM2. Otherwise, $P$ violates SYN1.

As a consequence of this result, we conclude that the computed instances of $P$ and any query $Q^-$ such that $\vdash \{Pre\} Q^- \{Post\}$ coincide with the computed instances of $Van(P)$ and $demo(Q).$ Moreover, those computed instances can be retrieved from the declarative specification.

**Theorem 5.4.4** Assume that $\vdash \{Pre\} P \{Post\}$ and $\vdash \{Pre\} Q^- \{Post\}$ hold, that $P$ satisfies conditions SYN1 and SEM2, and that $\Sigma_L$ contains infinitely many symbols.

Then the following statements are equivalent:
(i) $Q^-$ is a computed instance of $P$ and $Q^-;
(ii) Q^-_1$ is a most general instance of $Q^-$ which is true in $Post;$
(iii) $demo(Q_1)$ is a most general instance of $demo(Q)$ which is true in $Post_{Van(P)};$
(iv) $demo(Q_1)$ is a computed instance of $Van(P)$ and $demo(Q).$

**Proof.** By Theorems 4.1.22 and 4.1.25, we can assume that $Post$ is well-supported.

(i) is equivalent to (ii) by Theorem 4.1.31. The equivalence between (ii) and (iii) comes directly from the definition of $Post_{Van(P)}.$ Finally, (iii) is equivalent to (iv) since, by Theorem 5.4.3, we are in the hypothesis of Theorem 4.1.31.

**5.4.2 (Weak) Total Correctness**

We now show that the proof relation $\vdash_t$ lifts from the object program up to the Vanilla, i.e.:
\[
\vdash_t \{Pre\} P \{Post\} \vdash_t \{Pre_{Van(P)}\} Van(P) \{Post_{Van(P)}\}
\]
Assume that $\| \|$ is the level mapping used to prove $\vdash_{\ell} \{\text{Pre}\} P \{\text{Post}\}$, and define the function $\| \|$:

$$\|\text{demo}(B_1 \& \ldots \& B_n)\| = \text{bag}(|B_1|, \ldots, |B_m|)$$

for $\text{demo}(B_1 \& \ldots \& B_n) \in \text{Pre}_{\text{Van}(P)}$, where $m$ is the maximum $i \in [1,n]$ such that $\text{Post} \models B_1, \ldots, B_{i-1}$, and $\text{bag}()$ otherwise.

It is a simple exercise to prove

$$\vdash_{\ell} \{\text{Pre}_{\text{Van}(P)}\} \text{Van}(P) \{\text{Post}_{\text{Van}(P)}\}$$

using a level mapping $\|_{\text{Van}(P)}$ such that for $A \in B_L$:

$$|A|_{\text{Van}(P)} = \mu(\|A\|)$$

where $\mu : \text{bag}(N) \rightarrow N$ is a partially monotonic function w.r.t. $P$. A monotonic function from $\text{bag}(N)$ to $N$ does not exist, since $\text{bag}(N)$ is not an $\omega$-ordering. However, we need a weaker property. We recall that $n_P$ is the maximum number of atoms in the body of a clause from $P$.

**Definition 5.4.5** A function $\mu : \text{bag}(N) \rightarrow N$ is partially monotonic w.r.t. $P$ if $\mu(b_1) > \mu(b_2)$ for the bag $b_2$ obtained replacing an element of $b_1$ by at most $n_P$ elements lower than it.

This is a well-formed definition and the set of partially monotonic functions is non-empty. The proof outlines for the recursive clauses $(b,c)$ are then:

(b) \quad \begin{align*}
\{ \vdash \{\text{Pre}\} A, B^- \{\text{Post}\} \} \\
\text{demo}(A \& B) \leftarrow \{ \mu(\|\text{demo}(A \& B)\|) \} \\
\{ \text{Pre} \models A \} \\
\text{demo}(A) \leftarrow \{ \mu(\|\text{demo}(A)\|) \} \\
\{ \text{Post} \models A \} \\
\{ \vdash \{\text{Pre}\} B^- \{\text{Post}\} \} \\
\text{demo}(B) \leftarrow \{ \mu(\|\text{demo}(B)\|) \} \\
\{ \text{Post} \models B^- \} \\
\{ \text{Post} \models A, B^- \} \\
\{ \vdash \{\text{Pre}\} A \{\text{Post}\} \}
\end{align*}

(c) \quad \begin{align*}
\text{demo}(A) \leftarrow \{ \mu(\|\text{demo}(A)\|) \} \\
\{ \text{true} \} \\
\text{clause}(A, B) \leftarrow \{ \mu(\text{bag}()) \} \\
\{ A \leftarrow B^- \in \text{ground}_L(P) \} \\
\{ \vdash \{\text{Pre}\} B^- \{\text{Post}\} \} \\
\text{demo}(B) \leftarrow \{ \mu(\|\text{demo}(B)\|) \} \\
\{ \text{Post} \models B^- \} \\
\{ \text{Post} \models A \} \\
\end{align*}
As an example, the decreasing of the level mapping in proof outline (c) is given by the property:
\[ \mu(||\text{demo}(B)||) < \mu(||\text{demo}(A)||) \]
for \( A \leftarrow B^\rightarrow \in \text{ground}_K(P) \), which holds by the hypothesis on \( \mu \).

Moreover, as in the case of relation \( \vdash \), we notice that for a (not necessarily ground) query \( B_1, \ldots, B_n \),
\[ \vdash_1 \{ \text{Pre}_{\text{Van}(P)} \} \, \text{demo}(B_1 \& \ldots \& B_n) \, \{ \text{Post}_{\text{Van}(P)} \} \]
when \( \vdash \{ \text{Pre} \} \, B_1, \ldots, B_n \, \{ \text{Post} \} \) holds by \( | | \). This allows us to state termination of the Vanilla.

**Theorem 5.4.6** Assume that \( \vdash \{ \text{Pre} \} \, P \, \{ \text{Post} \} \) and \( \vdash \{ \text{Pre} \} \, Q^\rightarrow \, \{ \text{Post} \} \) hold by the same level mapping. Then every LD-tree of \( \text{Van}(P) \) and \( \text{demo}(Q) \) is finite.

**Proof.** By (5.9), we have:
\[ \vdash_1 \{ \text{Pre}_{\text{Van}(P)} \} \, \text{Van}(P) \, \{ \text{Post}_{\text{Van}(P)} \} . \]

Moreover, by (5.10),
\[ \vdash_1 \{ \text{Pre}_{\text{Van}(P)} \} \, \text{demo}(Q) \, \{ \text{Post}_{\text{Van}(P)} \} \]
holds. Thus, by Theorem 4.1.32, every LD-tree for \( \text{Van}(P) \) and \( \text{demo}(Q) \) is finite.

More generally, left termination lifts up from the object program to Vanilla.

**Corollary 5.4.7** Assume that \( \vdash \{ \text{Pre} \} \, P \, \{ \text{Post} \} \) holds. If \( P \) is \( \text{Pre} \)-terminating then \( \text{Van}(P) \) is \( \text{Pre}' \)-terminating, where
\[ \text{Pre}' = \{ \text{demo}(A) \mid A \in \text{Pre} \} . \]

**Proof.** By Theorem 4.1.36, \( \vdash \{ \text{Pre} \} \, P \, \{ \text{Post} \} \) and \( P \) \( \text{Pre} \)-terminating imply:
\[ \vdash_1 \{ \text{Pre} \} \, P \, \{ \text{sp}(P, \text{Pre}) \} . \]

By defining \( \text{Pre}_{\text{Van}(P)} \) and \( \text{Post}_{\text{Van}(P)} \) starting from \( \text{Pre} \) and \( \text{sp}(P, \text{Pre}) \), we have that (5.9) holds, i.e.
\[ \vdash_1 \{ \text{Pre} \} \, P \, \{ \text{Post} \} \]
\[ \vdash_1 \{ \text{Pre}_{\text{Van}(P)} \} \, \text{Van}(P) \, \{ \text{Post}_{\text{Van}(P)} \} . \]

By Theorem 4.1.36 again, we conclude that \( \text{Van}(P) \) is \( \text{Pre}_{\text{Van}(P)} \)-terminating. The conclusion that \( \text{Van}(P) \) is \( \text{Pre}' \)-terminating follows by noting that \( \text{Pre}' \subseteq \text{Pre}_{\text{Van}(P)} \).

In conclusion, we observe that the last two results can be generalized to arbitrary meta-interpreters, by rephrasing the general criterion \([gc]\) using the \( \vdash_1 \) relation instead of \( \vdash \). We omit the formal presentation of this generalization due to space limitations.
5.5 Related Work

Meta-programming in logic programming

The first definition of meta-programming in logic programming was proposed by Bowen and Kowalski [32], which also reasoned about amalgamation, reflection up and down. The origin of the Vanilla meta-interpreter traces back to logic programming folklore. Vanilla and the call predicate appeared first in DEC-10 Prolog user’s guide [134]. Also, we refer to the book by Sterling and Shapiro [150] for several enhanced meta-interpreters and some applications using meta-programs. More in general, we refer the reader to the survey of Barklund [18] and to the recent paper of Hill and Gallagher [86] for complete references to the literature and to logic programming languages that support meta-programming.

There are some major topics in the field that we did not discuss in detail. Below, we mention a few of them.

We adhered here to the non-ground representation of object programs (i.e. they are represented by terms having their same syntax), while the ground representation is claimed to have the greater potential [18]. In the ground representation, programs are represented as ground meta-level terms. For example, the Gödel programming language of Hill and Lloyd [88] supports the ground representation.

Within the non-ground representation, functional and identity representations of symbols can be distinguished. A (function or predicate) symbol $f$ is functionally represented if it is represented by a symbol $f'$ in the meta-language. We refer to Martens and De Schreye [114] for a detailed discussion of the functional representation. On the contrary, we adopted the identity representation (i.e. a symbol $f$ is represented by itself), which has the advantage of allowing for amalgamation.

Dynamic meta-programming [86] is a form of meta-programming in which the object program is not static, but rather it is changed or constructed by the meta-program. The simplest form of constructing a program is by combining basic programs. A number of operators (including union, intersection, and encapsulation) and an algebra for them have been defined by Brogi et al. [35]. In [131], we discuss how to reason on declarative correctness of the enhanced meta-interpreter of Brogi et al. by means of the general criterion [GC].

Finally, a large amount of effort has been directed towards reducing the overhead of meta-level computations due to the interpretation of representations of object-programs. In particular, a linear relation between the length of object-level and Vanilla meta-level SLD-derivations has been observed [98]. The technique of partial evaluation [104, 107] has been demonstrated to be a valid solution in many cases.

Language independence

We have assumed that $L$ is the underlying language of the object program and of Vanilla. This assumption is useful in modular proofs of correctness and amalga-
The alternative assumption of considering \( L_P \) as the language of the object program \( P \) has been usually considered in the literature.

On the one hand, we claim that there is no lack of generality. In fact, in [131] we show that the proof relation \( \vdash \) is language independent, in the following sense. Assume that \( \vdash \{ \mathit{Pre} \} P \{ \mathit{Post} \} \) holds in a language \( L \). For any extension \( L' \) of \( L \), there exist \( \mathit{Pre}' \) and \( \mathit{Post}' \) such that \( \mathit{Pre}' \cap B_L = \mathit{Pre} \) and \( \mathit{Post}' \cap B_L = \mathit{Post} \) and such that \( \vdash \{ \mathit{Pre}' \} P \{ \mathit{Post}' \} \) holds.

On the other hand, a stronger correlation between the least Herbrand models \( M_P \) and \( M_{\text{Van}}(P) \) than the one of Corollary 5.2.7 has been considered:

[Sli] For \( A \in B_{\{	ext{Van}(P) \mid P \}} \), if \( \text{demo}(A) \in M_{\text{Van}}(P) \), then \( A \in M_P \). Note that [Sli] does not hold in general. As an example, the least Herbrand model of Vanilla contains atoms such as \( \text{demo}(\mathit{p}(\emptyset)) \), while the least Herbrand model of the object program does not contain \( \mathit{p}(\emptyset) \) in general.

[Sli] states declarative soundness for all atoms with object-level relations and meta-level functors. Martens and De Schreye [113] showed that [Sli] holds for language independent programs, which are characterized by the fact that their least Herbrand models do not change over language extension. A proof of the converse, i.e. that [Sli] holds iff \( P \) is language independent, is shown in [131] with the results presented in this Chapter.

**Verification of meta-programs**

Hill and Lloyd [87] proved declarative and procedural correctness of a typed version of Vanilla extended with a clause dealing with negation. As an instantiation of their results, we find out Corollary 5.2.7.

Kalsbeek [98] observed that a consequence of procedural correctness for the typed Vanilla is the procedural correctness of the untyped Vanilla. More precisely, she showed that a query \( Q^+ \) is a computed instance of \( P \) and \( Q^- \) iff \( \text{demo}(Q^-) \) is a computed instance of \( \text{Van}(P) \) and \( \text{demo}(Q) \). This result is more general than Theorem 5.4.4. However, we point out that Theorem 5.4.4 was obtained by means of a general purpose verification method based on declarative semantics, whilst the proof of Kalsbeek relies on operational reasoning.

Declarative and procedural correctness can be also obtained through the \( S \)-semantics theory. Independently, Levi and Ramundo [104] and Martens and De Schreye [114] showed that \( \text{demo}(\mathit{p}(T_1, \ldots, T_n)) \) is in \( S(\text{Van}(P)) \) iff \( T_1, \ldots, T_n \) are object-level terms and \( \mathit{p}(T_1, \ldots, T_n) \) is in \( S(P) \).

These results are special cases of those reported in Brogi and Turini [36] on the extension of Vanilla including operators for combining programs.
5.6 Conclusion

In this Chapter, we have applied the proof method of Chapter 4 to some well-known meta-programs from the logic programming folklore, such as the Vanilla meta-interpreter, the simple reflective meta-interpreter, and some extensions, including forms of amalgamations and parameterizations.

On the one hand, the broad applicability of the verification principles underlying the proof method has been demonstrated.

On the other hand, as a by-product of the case study of the Vanilla meta-interpreter, we have extended the notions of declarative correctness and soundness and designed a criterion [GC] for declarative correctness of arbitrary meta-interpreters.

The contribution of this Chapter can be summarized as follows: under certain natural assumptions, all interesting verification properties lift up from object programs to meta-programs, including:

- (weak) partial correctness,
- (weak) total correctness,
- absence of run-time errors,
- call and success patterns characterization,
- correct and computed instances characterization.

Interestingly, it is possible to prove all this on the basis of purely declarative reasoning, using the proof relations \( \vdash \) and \( \vdash_1 \).

New results are obtained, which are related to amalgamation, reflection down/up, parameterization and extensions, termination, and absence of errors.
Chapter 6

From Termination to Validation

Starting from the declarative characterizations of terminating programs provided in Chapter 2, we have systematically derived proof methods for the verification of logic programs.

In this Chapter, we derive from the same declarative characterizations methods which are useful for the validation of logic programs. While verification is concerned with proving correctness of programs w.r.t. their formal specification, validation is concerned with showing that programs meet the user requirements, which usually are stated informally (e.g. by natural language statements) and are partially specified. We will consider here the validation issues of testing and debugging.

We start by investigating some natural properties of (sub-)classes of terminating logic programs, namely semantics and observable decidability - with reference to the $C$ and $S$-semantics of Definition 3.1.1 at page 70. While decidability of $M$-semantics is a direct consequence of the termination properties of the classes under consideration, decidability of $C$ and $S$ semantics is not so obvious.

In the first part of the Chapter, we present decision procedures for sub-classes of the terminating programs studied in Chapter 2 and provide an implementation for them in the form of Prolog meta-programs. While semantics decidability may seem a pure theoretical notion, we recognize a tight relation between semantics decidability and testing of logic programs, by showing that the two problems are equivalent. The decision procedures are then recognized to be automatic tools for testing logic programs. Also, we present some preliminary experimental results and an efficient compilation-oriented approach that overcome the overhead due to meta-programming.

In the second part of the Chapter, a specialization of the decision procedures are employed as the basic components for a declarative debugging approach of missing answers, namely of those queries which are valid in the intended meaning of a
program but that (testing shows that) are not in its actual semantics.

Finally, we point out that while we will reason on acceptable programs rather than in the framework of the $\vdash_t$ proof relation, we point out that all the results can be readily rephrased for programs such that $\vdash_t \{\text{Pre}\} P \{\text{Post}\}$ holds, thus producing a single framework for verification and validation.

## 6.1 Semantics Decidability and Testing

In general, the problem of deciding whether an atom belongs to the $\mathcal{M}$-semantics of a logic program $P$ is undecidable – and the same result holds for $\mathcal{C}$ and $\mathcal{S}$-semantics. However, we observe that universal termination of a program $P$ and a ground atom $A$ is a sufficient condition for deciding whether $A \in \mathcal{M}(P)$. Under this view, the characterizations of terminating logic programs of Chapter 2 are natural candidates for the research of classes of programs whose semantics are decidable sets. In this section, we identify large subclasses of acceptable, fair-bounded and bounded programs $P$, such that $\mathcal{C}(P)$ and $\mathcal{S}(P)$ are decidable sets, by providing a procedure for deciding whether an atom $A$ belongs to the semantics of $P$.

Semantics decidability coincide with observable decidability, and actually the proposed decision procedures check whether a query is a *computed instance* or a *correct instance* of another query. Interestingly, the decision procedures have intuitive implementations in the logic programming paradigm itself, in the form of Prolog meta-programs. The meta-programming approach reveals to be successful in modeling extensions to programs with arithmetic, meta-programs, modular programming, general logic programs and other declarative semantics, such as the *finite failure set*, the *closed word assumption set*, and the *computed answers with depth*.

Semantics decidability is directly related to testing. Software testing is an important stage in program development. It covers more than one third of the development time, and requires a high degree of specialization of the developers. Although testing cannot show the absence of errors, but only their presence, it is still a necessary stage, even when a formal proof of correctness is provided. In our terminology the testing problem consists of checking whether or not the formal semantics of a program includes a given *finite* set of atoms. This set represents a collection of test cases provided by the requirement documents (*validation* testing), or the formal specification (*verification* testing), or a previous version of the program (*regression* testing). We relate semantics decidability and testing by showing that the testing problem for a program $P$ and a semantics $\mathcal{F}$ is decidable iff $\mathcal{F}(P)$ is a decidable set. Therefore, the procedure for observable decidability can be used as automatic tools for testing logic programs.
6.1. Relating Semantics Decidability to Testing

Semantics and Testing

We restrict to consider a semantics as a function from programs to sets of atoms.

**Definition 6.1.1** A semantics $\mathcal{F}$ is a function from programs $P$ into sets of atoms $\mathcal{F}(P)$. □

As anticipated in Chapter 3, in addition to $\mathcal{M}$-semantics, we mainly focus on two well-known semantics, namely $\mathcal{C}$ and $\mathcal{S}$-semantics.

We now define the testing problem for logic programs. Consider two finite sets: one of atoms which should belong to the formal semantics of the program, and the other of atoms which should not belong. In practice, these sets are provided by an analysis of the requirement documents (validation testing), of the formal specification (verification testing) or of a previous version of the program (regression testing). Testing a program on this pair of sets means checking that the formal semantics of the program includes every atom in the first set and no atom in the second one.

**Definition 6.1.2** A program $P$ is tested w.r.t. a semantics $\mathcal{F}$ on a pair $(I, S)$ of finite sets of atoms iff:

$$I \subseteq \mathcal{F}(P) \subseteq \text{Atom}_L \setminus S.$$

The testing problem consists of deciding whether a program $P$ is tested w.r.t. $\mathcal{F}$ on a given pair $(I, S)$.

An atom in $I$ that is not in $\mathcal{F}(P)$ is called an *incompleteness symptom*. An atom in $S$ which is also in $\mathcal{F}(P)$ is called an *incorrectness symptom*. □

In this Section, we are concerned with a formal theory and some practical tools to make the testing problem decidable. A further stage in the program development process, called diagnosis problem, consists of determining the program components which are sources of incompleteness or incorrectness symptoms, and will be addressed in Section 6.2.

Observables and Testing

Observables are abstractions of SLD-trees. Formally, we restrict to consider an observable $\mathcal{O}$ as a function from programs into sets of $n_o$-tuples of queries. The next definition introduces the observables $\mathcal{O}_M, \mathcal{O}_C$ and $\mathcal{O}_S$, for which the relative semantics $\mathcal{M}, \mathcal{C}$ and $\mathcal{S}$ are *AND-compositional* (see [31] for a survey).
Definition 6.1.3 For a program $P$ we define:

$$
\mathcal{O}_M(P) = \{ Q \mid P \models Q \}
$$

$$
\mathcal{O}_c(P) = \{ (Q, Q') \mid Q' \text{ correct instance of } Q \}
$$

$$
\mathcal{O}_s(P) = \{ (Q, Q') \mid Q' \text{ computed instance of } Q \}.
$$

In a real development context, testing is performed with respect to observables rather than semantics. However, we will show that for the considered semantics, the testing problems w.r.t. observables and semantics are equivalent.

Definition 6.1.4 Let $\mathcal{O}$ be an observable from programs into sets of $n_\mathcal{O}$-tuples of queries. A program $P$ is tested w.r.t. $\mathcal{O}$ on a pair $(I, S)$ of finite sets of $n_\mathcal{O}$-tuples of queries iff:

$$
I \subseteq \mathcal{O}(P) \subseteq \text{Query}_{n_\mathcal{O}} \setminus S.
$$

The testing problem consists of deciding whether a program $P$ is tested w.r.t. $\mathcal{O}$ on a given pair $(I, S)$.

Relating Them

The next simple result clarifies the relation between the testing problem and the semantics decidability issue.

Theorem 6.1.5 For a program $P$ and a semantics $\mathcal{F}$ (resp., an observable $\mathcal{O}$), the following statements are equivalent:

(i) the testing problem w.r.t. $\mathcal{F}$ (resp., $\mathcal{O}$) is decidable,

(ii) $\mathcal{F}(P)$ (resp., $\mathcal{O}(P)$) is a decidable set.

Proof. We show only the case of semantics. The case of observables is analogous.

(i $\rightarrow$ ii). Consider, for an atom $A$, the sets $I = \{ A \}$ and $S = \emptyset$. By (i), it is decidable whether $I \subseteq \mathcal{F}(P)$, i.e. whether $A \in \mathcal{F}(P)$. Thus, $\mathcal{F}(P)$ is a decidable set.

(ii $\rightarrow$ i). We recall that the complement of a decidable set is decidable as well. Then since $I$ and $S$ are finite sets, it is decidable whether $I \subseteq \mathcal{F}(P)$ and $S \subseteq \text{Atom}_L \setminus \mathcal{F}(P)$, i.e. $I \subseteq \mathcal{F}(P) \subseteq \text{Atom}_L \setminus S$. □

As a consequence, the testing problem is undecidable for the class of logic programs w.r.t. the considered semantics.

Corollary 6.1.6 The testing problem is undecidable for the class of logic programs w.r.t. $\mathcal{M}$, $\mathcal{C}$, and $\mathcal{S}$-semantics.
Proof. In general, for a logic program $P$ the set $M(P)$ is undecidable (see [9] for a proof). Since $C(P) \cap B_{1} = M(P)$, if $C(P)$ were decidable then $M(P)$ would be decidable as well. Finally, if $S(P)$ were decidable then $C(P)$ would be decidable as well. In fact, by Completeness of SLD-resolution, an atom is in $C(P)$ iff any of its anti-instances (that are finite, modulo renaming) is in $S(P)$. □

Analogously, one can show the undecidability of the testing problem w.r.t. the observables of Definition 6.1.3. However, we note that it comes directly from the following theorem, which relates the testing problem w.r.t. semantics to that w.r.t. observables.

**Theorem 6.1.7** The testing problem for a program $P$ w.r.t. the $M$-semantics (resp., $C$ and $S$-semantics) is decidable iff the testing problem for $P$ w.r.t. the observable $O_{M}$ (resp., $O_{C}$ and $O_{S}$) is decidable.

Proof. By Theorem 6.1.5, we have only to show that $M(P)$ (resp., $C(P)$ and $S(P)$) is a decidable set iff $O_{M}(P)$ (resp., $O_{C}(P)$ and $O_{S}(P)$) is a decidable set.

The if parts are trivial since an atom $A$ is in $M(P)$ iff it is in $O_{M}(P)$; $A$ is in $C(P)$ iff ($A, A$) is in $O_{C}(P)$; and $A$ is in $S(P)$ iff ($p(x_{1}, \ldots, x_{n}), A$) is in $O_{S}(P)$, where $p$ is the predicate symbol of $A$ and $x_{1}, \ldots, x_{n}$ distinct fresh variables.

As for the only-if parts, we have:

- $A_{1}, \ldots, A_{n}$ is in $O_{M}(P)$ iff for every $i \in [1, n], A_{i}$ is in $M(P)$,
- ($A_{1}, \ldots, A_{n}, B_{1}, \ldots, B_{n}$) is in $O_{C}(P)$ iff $B_{1}, \ldots, B_{n}$ is an instance of $A_{1}, \ldots, A_{n}$ and for every $i \in [1, n], B_{i}$ is in $C(P)$,
- by Theorem 3.1.2, $(Q, Q')$ is in $O_{S}(P)$ iff there exists $C_{1}, \ldots, C_{n}$ in $S(P)$, renamed apart, such that $Q'$ is the most general instance (modulo renaming) of $Q$ and the query $C_{1}, \ldots, C_{n}$.

In the first and the second case, it is immediate to get the conclusion. In the last case, we note that there are only finitely many queries (modulo renaming) that can be considered as candidates for $C_{1}, \ldots, C_{n}$, namely those queries that are anti-instances of $Q'$.

### 6.1.2 Acceptable, Fair-bound and Bounded Programs

The aim of this section is to identify suitable large classes of logic programs whose declarative semantics are decidable sets, and a fortiori the relative testing problems are decidable.

**Acceptable programs**

In Chapter 2, we have introduced several classes of universal terminating programs and queries. For instance, we have shown that every LD-derivation is finite for a program acceptable by an extended level mapping || and a Herbrand interpretation
I, and a ground atomic query $A$ such that $|A| \neq \infty$. Therefore, a Prolog interpreter is a decision procedure w.r.t. $M$-semantics for the class of programs acceptable by level mappings. It is therefore worth introducing this class of programs.

**Definition 6.1.8** We say that a program is acceptable if there exist a level mapping $||$ and a Herbrand interpretation $I$ such that it is acceptable by $||$ and $I$. □

It turns out that this is the original notion of acceptability proposed by Apt and Pedreschi [16].

**Example 6.1.9 (SAT)** Consider the program SAT of Example 2.2.4.

\[
\text{satisfiable(Formula)} \leftarrow \text{there is a true instance of Formula}
\]

\[
\text{satisfiable(true)}.
\]

\[
\text{satisfiable(X \land Y)} \leftarrow \text{satisfiable(X), satisfiable(Y)}.
\]

\[
\text{satisfiable(not X)} \leftarrow \text{invalid(X)}.
\]

\[
\text{invalid(false)}.
\]

\[
\text{invalid(X \land Y)} \leftarrow \text{invalid(X)}.
\]

\[
\text{invalid(X \land Y)} \leftarrow \text{invalid(Y)}.
\]

\[
\text{invalid(not X)} \leftarrow \text{satisfiable(X)}.
\]

We have shown that it is recurrent by the level mapping:

\[|\text{satisfiable}(t)| = |\text{invalid}(t)| = \text{size}(t).\]

By Theorem 2.7.1, SAT is acceptable by $||$ and $B_L$, hence it is acceptable. □

Let us summarize the termination property of acceptable programs we are interested in this Chapter.

**Theorem 6.1.10** A program $P$ is acceptable iff the LD-tree of $P$ and any ground query (written in any language) is finite. The only-if part holds also for programs with arithmetic.

**Proof.** Pedreschi and Ruggieri [127] showed that a program $P$ (with arithmetic) is acceptable w.r.t. the language $L_P$ iff it is acceptable w.r.t. any extension of $L_P$. Consider now a fixed language $L$. If a program (with arithmetic) is acceptable then by Theorem 2.3.15 (resp., 2.6.3), every LD-derivation for it and a ground query is finite. Since LD-trees are finitely branching, they are finite. Conversely, the if part is a simple consequence of Lemma 2.3.19 (ii). □
It is worth noting that no restriction is assumed on the (first order) language in which the ground query is written, i.e. acceptable logic programs are language-independent [127].

A Prolog interpreter is then a decision procedure w.r.t. $M$-semantics of acceptable programs. However, as pointed out in Chapter 2, termination does not lift up to all non-ground queries. Therefore, a Prolog interpreter is not a decision procedure w.r.t. $C$- and $S$-semantics, or their observables.

**Example 6.1.11 (Plain Prolog Procedure)** Suppose to be interested in checking whether the query $\text{satisfiable}(\text{not}(X \land \text{false}))$ is in the $C$-semantics of SAT (which is indeed the case). Running the query on a Prolog interpreter produces an infinite sequence of answers $X = \text{false}, X = \text{false} \land Y, X = \text{false} \land \text{false} \land Y, \ldots$. None of the answers, however, allows us to conclude that the query is or is not in the $C$-semantics of SAT.

The same example applies to the $S$-semantics. In this case, we have to run the Prolog interpreter on the query $\text{satisfiable}(Z)$ looking for the computed instance $\text{satisfiable}(\text{not}(X \land \text{false}))$. Even worst than in the case of $C$-semantics, the query $\text{satisfiable}(Z)$ produces an infinite sequence of answer in which the function symbol $\text{not}$ never appears at all.

Also, a similar example can be stated in the context of the $O_S$ observable. For instance, testing whether $\text{satisfiable}(\text{not}(X \land \text{false}))$ is a computed instance of $\text{satisfiable}(\text{not}(X \land Y))$ by running the latter query on a Prolog interpreter, produces an infinite sequence of answers instantiating only $X$. \hfill $\square$

**Example 6.1.12 (SAT-ERROR)** Assume that in writing the SAT program, a programmer forget to write the unit clause of invalid. Call the resulting program SAT-ERROR. It is readily checked that SAT-ERROR is acceptable.

Suppose now to test SAT-ERROR checking whether $\text{satisfiable}(\text{not}(\text{false} \land X))$ is a computed instance of $\text{satisfiable}(\text{not}(Y \land X))$, as one could expect. Running $\text{satisfiable}(\text{not}(Y \land X))$ on a Prolog interpreter causes the interpreter to enter an infinite loop, producing no answer at all. \hfill $\square$

**Fair-bounded programs**

Following the same reasoning, we generalize the approach above to the larger class of fair-bounded programs.

**Definition 6.1.13** We say that a program is fair-bounded if there exist a level mapping $| |$ and a Herbrand interpretation $I$ such that it is fair-bounded by $| |$ and $I$. \hfill $\square$

The next Lemma shows that also fair-bounded programs are language-independent.
Lemma 6.1.14 A program (with arithmetic) $P$ is fair-bounded w.r.t. a language $L$ extending $L_P$ iff it is fair-bounded w.r.t. $L_P$.

Proof. We consider only the case of logic programs. The conclusion holds for program with arithmetic as well, since we never make the assumption that programs are finite sets of clauses.

(Only-if). It is straightforward. It is sufficient to consider the restrictions of the level mapping and of the model to the language $L_P$.

(If). Conversely, consider $\models : B_P \to N$ and $I \subseteq B_P$, such that $I$ is a model of $P$ and for every $A \leftarrow B_1, \ldots, B_n$ in $\text{ground}_{L_P}(P)$:

(a) $I \models B_1, \ldots, B_n$ implies for every $i \in [1, n]$ $|A| > |B_i|$, and

(b) $I \not\models B_1, \ldots, B_n$ implies there exists $i \in [1, n]$ $I \not\models B_i \land |A| > |B_i|$.

We recall that $B_P$ is the Herbrand base on $L_P$, and $U_P$ is the set of ground terms on $L_P$. Let $H : B_L \to B_L$ be a function such that $H(A)$ is obtained by replacing every maximal subterm in $A$ whose principal functor $f$ is not in $L_P$ with a ground term $t_f \in U_P$.

We show that $P$ is fair-bounded w.r.t. $L$ by considering $\models' : B_L \to N$ and $A \subseteq B_L$ such that for $A \in B_{(\Sigma_{L,n_P})}$$\models' = |H(A)|$ and $A \models' \iff H(A) \in I$.

Consider $A \leftarrow B_1, \ldots, B_n$ in $\text{ground}_{L}(P)$. We point out that

$$H(A) \leftarrow H(B_1), \ldots, H(B_n)$$

is in $\text{ground}_{L_P}(P)$. Suppose that $A \models B_1, \ldots, B_n$. By definition of $\models'$

$$I \models H(B_1), \ldots, H(B_n).$$

Since $I$ is a model of $P$, we conclude $I \models H(A)$ and then $A \models A$, i.e. $A$ is a model of $P$. Moreover, by (a) for $i \in [1, n]$ $|H(A)| > |H(B_i)|$. By definition of $\models'$ this implies for $i \in [1, n]$ $|A| > |B_i|$$\models'$.

Suppose now that $A \not\models B_1, \ldots, B_n$. By definition of $\models'$

$$I \not\models H(B_1), \ldots, H(B_n).$$

By (b) there exists $i \in [1, n]$ such that $I \not\models H(B_i) \land |H(A)| > |H(B_i)|$. By definition of $\models'$ and $\models'\models$ this implies that there exists $i \in [1, n]$ such that $I \not\models B_i \land |A| > |B_i|$$\models'$. Therefore, we conclude that $P$ is fair-bounded w.r.t. $L$ under the hypothesis that it is bounded w.r.t. $L_P$. 

The following result provides an operational characterization of the class of fair-bounded programs.

Theorem 6.1.15 A program $P$ is fair-bounded iff the SLD-tree via any fair selection rule of $P$ and any ground query (written in any language) is finite. The only-if part holds also for programs with arithmetic.
Proof. Lemma 6.1.14 shows that a program \( P \) (with arithmetic) is fair-bounded on the language \( L_P \) iff it is fair-bounded on any extension of \( L_P \). Consider now a fixed language \( L \). If a program (with arithmetic) is fair-bounded then by Theorem 2.4.14 (resp., 2.6.3), every SLD-derivation via a fair selection rule for it and a ground query is finite. Since SLD-trees are finitely branching, they are finite. Conversely, the if part is a simple consequence of Lemma 2.4.19 (ii).

Example 6.1.16 (ProdCons) Consider the \texttt{PRODCONS} program of Example 2.4.4 on page 30.

\begin{verbatim}
  system(N) ← prod(Bs), cons(Bs, N).
  prod([s(0) | Bs]) ← prod(Bs).
  prod([s(s(0)) | Bs]) ← prod(Bs).
  prod([]).
  cons([D | Bs], s(N)) ← cons(Bs, N), wait(D).
  cons([], 0).
  wait(0).
  wait(s(D)) :- wait(D).
\end{verbatim}

We have shown that it is fair-bounded by \( \mid \mid \) and \( I \), where:

\[
I = [\text{system}(N)]_L \\
\cup \{ \text{prod}(bs) \mid \text{bs list of 1's and 2's} \} \\
\cup \{ \text{cons}(bs, n) \mid \mid bs \mid = \text{size}(n) \} \\
\cup \{ \text{wait}(X) \}_{L_n} \\
\mid \text{system}(n) \mid = \text{size}(n) + 3 \\
\mid \text{prod}(bs) \mid = \mid bs \mid \\
\mid \text{cons}(bs, n) \mid = \begin{cases} \text{size}(n) + \text{lmax}(bs) & \text{if } \text{cons}(bs, n) \in I \\
\text{size}(n) & \text{if } \text{cons}(bs, n) \notin I \end{cases} \\
\mid \text{wait}(t) \mid = \text{size}(t). \]

Since \( \mid \mid \) is a level mapping, \texttt{PRODCONS} is fair-bounded.

As a consequence of Theorem 6.1.15, any system adopting a fair selection rule is a decision procedure for \texttt{PRODCONS} w.r.t. \( \mathcal{M} \)-semantics.

As in the case of acceptable programs, however, the procedure is not effective w.r.t. \( \mathcal{C} \) and \( \mathcal{S} \)-semantics.

Example 6.1.17 Suppose to be interested in checking whether \texttt{system(s(X))} is in the \( \mathcal{C} \)-semantics of \texttt{PRODCONS} (which is not indeed the case). Running the query on a systems adopting a fair selection rule and a breadth-first search strategy produces
an infinite sequence of answers \( X = 0, X = s(0), X = s(s(0)), \ldots \). None of the answers, however, allows us to conclude that the query is or is not in the \( \mathcal{S} \)-semantics of \textsc{Prodccons}. Fair selection rules and depth-first search strategy cause the system to enter an infinite loop.

The same example applies to \( \mathcal{S} \)-semantics.

### Bounded programs

Finally, consider the class of bounded programs. We have shown in Chapter 2, that it characterizes bounded nondeterminism. In particular, for a program bounded by a given and computable level mapping, we are in the position to compute an upper bound to the depth of proof trees. It is therefore worth introducing such a class of programs.

**Definition 6.1.18** We say that a program is bounded if there exist a level mapping \( \mid \mid \) and a Herbrand interpretation \( I \) such that it is bounded by \( \mid \mid \) and \( I \).

We say that a program is bounded by a computable level mapping if there exists a computable level mapping \( \mid \mid \) and a Herbrand interpretation \( I \) such that it is bounded by \( \mid \mid \) and \( I \).

The next Lemma shows that bounded programs are language-independent.

**Lemma 6.1.19** A program (with arithmetic) \( P \) is bounded w.r.t. a language \( L \) extending \( L_P \) iff it is bounded w.r.t. \( L_P \).

**Proof.** A simplification of Lemma 6.1.14.

In the following theorem, we characterize operationally bounded programs as the class of programs with finitely many refutations starting with a ground query.

**Theorem 6.1.20** A program \( P \) is bounded iff for every selection rule \( s \) and every ground query \( Q \) (written in any language), the number of SLD-refutations for \( P \) and \( Q \) via \( s \) is finite. The only-if part holds also for programs with arithmetic.

**Proof.** Lemma 6.1.19 shows that a program \( P \) (with arithmetic) is bounded on the language \( L_P \) iff it is fair-bounded on any extension of \( L_P \). Consider now a fixed language \( L \). If a program (with arithmetic) is bounded then by Theorem 2.5.14 (resp., 2.6.3), the number of SLD-refutations via any selection rule for it and a ground query is finite. Conversely, the if part is a simple consequence of Lemma 2.5.18 (ii).

**Example 6.1.21.** (Circular Queues) The following program implements a set of operations on circular queues whose elements belong to a set of ground terms \( \mathcal{E} \). 
Circular queues are represented as lists. Insertion of an element is done at the head of the list. The search for an element is implemented by means of member. Removing an existing element is done by shifting the list until the element is in the first position, and then taking the rest of the list.

Let us call RQUEUE the definition of remove. We observe that for a ground atom \( \text{remove}(a, [b, c], [b, c]) \) there are infinitely many SLD-derivations via any selection rule. In fact, termination holds only when the element to be searched is in the queue. This is sufficient in many cases, and actually non-termination of unintended queries is not an error.

Therefore, RQUEUE is not acceptable or fair-bounded. On the contrary, it is bounded by the level mapping \( \| \) and the Herbrand interpretation \( I \), where:

\[
I = \{ \text{remove}(x, ys, xs) \mid ys \in GList \Rightarrow x \text{ is in } ys \} \\
\cup \{ \text{append}(xs, ys, zs) \mid xs \in GList \land \\
(yb \in GList \Rightarrow zs = xs \ast yb) \} \\
\cup \{ x \neq y \mid x, y \in E \land x \neq y \}
\]

\[
|\text{remove}(x, ys, xs)| = \begin{cases} |ys| + \min \{ i \in [1, n] \mid x = t_i \} \\
0 \quad \text{if } ys = [t_1, \ldots, t_n] \text{ and } x \text{ in } ys \\
\text{otherwise}
\end{cases}
\]

\[
|\text{append}(xs, ys, zs)| = |xs| \\
|x \neq y| = 0
\]

By Theorem 6.1.20 we conclude that for any selection rule \( s \) there are finitely many SLD-refutations of RQUEUE and any ground query. By means of the transformation of Section 2.5.4, we are in the position to find out all of them. \( \square \)

Once again, however, the procedure is not effective w.r.t. \( \mathcal{C} \) and \( \mathcal{S} \)-semantics
Example 6.1.22 Suppose to be interested in checking whether \( \text{remove}(X, [Y,X], [Y]) \) is in the \( S \)-semantics of \( \text{QUEUE} \) (which is not indeed the case). First, we observe that if the set \( E \) contains at least two elements \( a \) and \( b \), then there are infinitely many SLD-refutations for the pure atom \( \text{remove}(A, B, C) \). This implies that it is not bounded. So, the transformational approach of Section 2.5.6 cannot be applied.

6.1.3 Decision Procedures

As an immediate consequence of Theorems 6.1.10 and 6.1.15, the set \( M(P) \) is decidable when \( P \) is acceptable or fair-bounded. In the case of bounded programs, we observe that since there are finitely many refutations for a ground atom \( A \), their lengths are bounded. If we restrict to computable level mappings, we are in the position to compute an upper bound.

**Theorem 6.1.23** For a program \( P \) acceptable, fair-bounded or bounded by a given computable level mapping, \( M(P) \) is a decidable set.

**Proof.** Consider first \( P \) acceptable (resp., fair-bounded). By Theorem 6.1.10 (resp., Theorem 6.1.15) the LD-tree of \( P \) and any ground atom \( A \) is finite. By Soundness and Strong Completeness of SLD-resolution, there is a LD-refutation iff \( A \in M(P) \).

Consider now \( P \) bounded by \( | | \) and \( I \), with \( | | \) given and computable. By Theorem 2.5.21, the LD-tree of \( \text{Ter}(P) \) and any ground atom \( \text{Ter}(A, |A|) \) is finite, and contains a LD-refutation iff there exists a LD-refutation for \( P \) and \( A \), i.e., iff \( A \in M(P) \).

Programs bounded by computable level mappings strictly include the class of acceptable and fair-bounded programs.

**Example 6.1.24** An example of a program bounded by a computable level mapping, but which is not acceptable or fair-bounded is the one-clause program \( p \leftarrow p \).

**Theorem 6.1.25** Every acceptable or fair-bounded program is bounded by a computable level mapping.

**Proof.** Let \( P \) be acceptable (resp., fair-bounded). By Lemma 6.1.10 (resp., 6.1.15), the LD-tree (resp., the SLD-tree via a fair selection rule \( f \)) of \( P \) and any ground atomic query \( A \) is finite. By Lemma 2.3.19 (resp., 2.4.19) \( P \) is acceptable by the level mapping defined as follows:

\[ |A| = \text{length}_{LD}(A) \]

(resp., \( |A| = \text{length}_{r}(A) \)) and some \( I \). Therefore, \( | | \) is computable by any Prolog interpreter (resp., any system adopting the round-robin rule \( rr \)). By Theorem 2.7.1, \( P \) is bounded by \( | | \) and \( I \), hence \( P \) is bounded by a computable level mapping. \( \square \)
Although acceptable and fair-bounded programs are a subclass of bounded programs, the decision procedures for the three classes are quite different. While for acceptable and fair-bounded programs it is not needed to know the level mapping, that information is necessary in the case of bounded programs. This is not a real drawback when level mappings are automatically inferred. Moreover, it should be observed that also in the case of acceptability, level mappings must be found out (automatically or manually) to prove a program acceptable. However, for the reasons above, we will maintain the distinction between the three classes in the rest of the Chapter.

The decision procedures for $C$ and $S$-semantics reduce decidability to the problem of finding out the set of refutations of ground queries. For the classes of programs under consideration, those sets are computable.

**Lemma 6.1.26** Let $P$ be an acceptable program, or a fair-bounded program or a program bounded by a given computable level mapping, and $Q, Q'$ two queries. Then it is decidable whether $Q'$ is a correct instance of $P$ and $Q$.

**Proof.** $Q'$ is a correct instance of $P$ and $Q$ iff $Q'$ is a logical consequence of $P$ and $Q$ is an instance of $Q$. Since it is decidable whether a query is an instance of another one, we have only to show that it is decidable whether $Q'$ is a logical consequence of $P$. By the well-known Theorem on Constants (see e.g. Shoenfield [146]) $Q'$ is a logical consequence of $P$ iff $Q'\theta$ is a logical consequence of $P$, where $\theta$ is a substitution mapping all variables of $Q'$ into distinct fresh constants not appearing in $P$, $Q$ or $Q'$. Since $Q'\theta$ is ground, it is a logical consequence of $P$ iff every atom in it is in $M(P)$. By Theorem 6.1.23, we conclude that it is decidable whether $Q'\theta$, and a fortiori $Q'$, is a logical consequence of $P$.

Decidability in the case of computed instances is stated in the next lemma.

**Lemma 6.1.27** Let $P$ be an acceptable program, or a fair-bounded program, or a program bounded by a given computable level mapping, and $Q, Q'$ two queries. Then it is decidable whether $Q'$ is a computed instance of $P$ and $Q$.

**Proof.** We provide a decision procedure for establishing whether $Q'$ is a computed instance of $P$ and $Q$ in the case $P$ is bounded (resp., acceptable - fair-bounded). Let $\theta$ be the given and computable level mapping.

Let $\theta$ be a substitution mapping all variables of $Q'$ into distinct fresh constants. The query $Q'\theta$ is ground and therefore by Theorem 6.1.20 (resp., Theorem 6.1.10–6.1.15) there is a finite set of SLD-refutations for $Q'\theta$ via the leftmost selection rule (resp., via the leftmost selection rule - via any fair selection rule).

Moreover, $Q'\theta$ is bounded by $\theta$ and any Herbrand interpretation, by fixing in Definition 2.5.6 $k$ to $1 +$ the maximum level of atoms in $Q'\theta$. Therefore, we can compute $\xi_1, \ldots, \xi_m$ since the length of each $\xi_i$ is bounded by $f(k)$ (resp., since the
 Chapter 6. From Termination to Validation

LD-tree of $Q\theta$ is finite – since the SLD-tree of $Q\theta$ via any fair selection rule is finite) for a given and computable function $f$, as shown in the proof of Theorem 2.5.14 and Lemma 2.5.13.

For each $\xi_i$, consider now the prefix $\xi_i'$ of the LD-derivation (resp., LD-derivation – SLD-derivation via the round-robin selection rule) for $Q$ using the same sequence of clauses of $\xi_i$ until possible, i.e. until success or failure is reached. Let $Q'_1, \ldots, Q'_n$ be the computed instances of the successful $\xi_i'$s (in general $n \in [0, m]$). Let us show the following fact:

\[ Q' \text{ is a computed instance of } P \text{ and } Q \iff Q' \text{ is a variant of } Q'_i \text{ for some } i \in [1, n]. \tag{6.1} \]

The if part is trivial. To prove the only-if part, we notice that if $Q'$ is a computed instance of $P$ and $Q$ then by Strong Completeness of SLD-resolution, $Q'$ is a computed instance of $P$ and $Q'$ with a SLD-refutation $\xi$ via any selection rule. Then $Q'$ is a computed instance of $P$ and $Q'$ with a LD-refutation (resp., LD-refutation – SLD-refutation via the round-robin selection rule) $\xi'$, which uses the same sequence of clauses of $\xi$. The conclusion then follows from the observation that $\xi'\theta$ (the refutation obtained by replacing in $\xi'$ every variable $x$ of $Q'$ by $x\theta$) is a refutation for $Q'\theta$ which uses the same sequence of clauses of $\xi'$, and then of $\xi$.

We point out that the $Q'_i$'s are computable since we can compute $\xi_1, \ldots, \xi_m$. Finally, since it is decidable whether two queries are variants, it follows from (6.1) that it is decidable whether $Q'$ is a computed instance of $P$ and $Q$.

An intuitive way of interpreting the decision procedures for $C$ and $S$-semantics consists of interpreting them as visits of the LD-tree of $P$ and $Q'$ limited to branches that do no instantiate variables in $Q'$. The visit is finite, because it can be reducted to a visit of a ground instance of $Q'$, obtained by replacing variables with fresh distinct constants.

As an immediate consequence, the $C$ and $S$-semantics and observables are decidable sets.

**Theorem 6.1.28** For a program $P$ acceptable, fair-bounded or bounded by a given computable level mapping, $O_C(P)$, $O_S(P)$, and $\mathcal{C}(P)$, $\mathcal{S}(P)$ are decidable sets.

**Proof.** The conclusion for observables is immediate from Lemmas 6.1.26, 6.1.27 and Definition 6.1.3. The results for semantics follow by Theorems 6.1.7 and 6.1.5. \hfill \Box

Moreover, since we have shown that semantics decidability and the testing problem are equivalent, we conclude that the testing problem is decidable.

**Corollary 6.1.29** The testing problem is decidable for the class of acceptable and fair-bounded programs, and for programs bounded by a given computable level mapping w.r.t. the $C$ and $S$-semantics and w.r.t. the $O_C$ and $O_S$-observables. \hfill \Box
Example 6.1.30 (SAT Ctd) In Examples 6.1.11, we pointed out that the plain Prolog execution is not an effective decision procedure w.r.t. \( C \) and \( S \)-semantics and observables for the SAT program.

On the contrary, Lemma 6.1.26 provides us with a procedure to test whether the atom \( \text{not}(X \land \text{false}) \) is a correct instance of SAT, i.e. whether it is in \( C(\text{SAT}) \).

Analogously, Lemma 6.1.27 provides us with a procedure to test whether the atom \( \text{not}(X \land \text{false}) \) is in \( C(\text{SAT}) \). □

Example 6.1.31 (SAT-ERROR Ctd) Lemma 6.1.26 allows us to decide that \( \text{not}(\text{false} \land X) \) is not a computed instance of the SAT-ERROR program of Example 6.1.12 and of \( \text{not}(Y \land X) \). □

6.1.4 Prolog Implementation of the Decision Procedures

The decision procedures of Lemmas 6.1.26 and 6.1.27 employ mechanisms from the logic programming paradigm itself, such as substitutions and LD-derivations. It is then natural to implement those procedures in Prolog. In the following, we propose Prolog implementations of the decision procedures for the \( O_C \) and \( O_S \) observables.

Acceptable Programs

First, we consider correct instances. Let us recall the main steps of the decision procedure. Given two queries \( Q \) and \( Q' \), first of all the variables of \( Q' \) are consistently replaced by fresh distinct constants. Then a Prolog interpreter for \( P \) and the resulting query is called, building up a finite LD-tree.

We model replacement of variables with fresh constants by assuming that the Prolog built-in predicate \( \text{freeze}(\text{Term}, \text{Frozen}) \) described in Sterling and Shapiro [150, Section 10.3] is available.

\( \text{freeze} \) makes a copy of the first argument by replacing variables with fresh distinct constants. For instance \( \text{freeze}(p(X), Y) \) succeeds by instantiating \( Y \) to \( p(a_i) \), where \( a_i \) is a fresh constant representing the frozen variable \( X \). Even though \( \text{freeze} \) is not present in existing Prolog implementations, it can be simulated\(^1\). We abstract away from the details of simulating \( \text{freeze} \) (see [137] for them), while concentrating on the central idea.

For a logic program \( P \), we denote by \( \text{DECC-A}(P) \) the following program.

\(^1\)A way to simulate \( \text{freeze} \) is to assume a set of facts \( \text{new-const}(i, a_i) \), where the \( a_i \)'s are fresh distinct constants, for \( 1 \leq i \leq M \), where \( M \) is an upper bound for the number of variables in \( Q' \). Starting from \( \text{new-const} \), a predicate \( \text{constants}(N1, L, N2) \) is easily definable that replaces the \( N2 - N1 \) variables appearing in \( L \) by \( a_{N1}, \ldots, a_{N2-1} \).
test_c(Q1, Q) ←
    freeze(Q1, Qs),
    Qs = Q,
    call(Qs).

augmented by P.

Program DECC-A(P).

Without loss of generality, we assume that the predicate symbol test_c does not appear in P.

Theorem 6.1.32 Let P be an acceptable program, and Q', Q two variable disjoint queries. Then every LD-derivation of DECC-A(P) and the query test_c([Q'], [Q]) is finite.

Moreover, there exists a LD-refutation for them iff Q' is a correct instance of P and Q.

The decision procedure for computed instances can be derived with the same arguments starting from Lemma 6.1.27. In this case, we need a slight modification of the Prolog interpreter, such that when a refutation is found, the computed instance Q'' (if it exists) of P and Q is computed using the same sequence of clauses of that refutation. Finally, if Q'' is a variant of Q' then we can state that Q' is a computed instance of Q. If no variant of Q' is found this way, then Q' is not a computed instance of Q.

We translate this reasoning in a decision procedure in the form of a Prolog meta-program. We design a variant of the Vanilla meta-interpreter, which behaves as expected when a LD-refutation is found. In order to trace back the sequence of clauses used in a derivation, we assume that a distinct identifier k is associated with each clause C_k in P. Also, we assume that the Prolog predicate variants(Term1, Term2) is available, which succeeds iff the terms Term1 and Term2 are variants.

For a logic program P, we denote by DECS-A(P) the following program.

test_s(Q1, Q) ←
    freeze(Q1, Qs),
    demo(Qs, Q),
    variants(Q1, Q).

demo([], []).

demo([A|As], [B|Bs]) ←
    clause(A, Ls, Id),
    demo(Ls, Lis),
demo(As, Bs),
clause(B, L1s, Id).

clause(A, [B1, ..., Bn], k).
for every Ck = A ← B1, ..., Bn ∈ P
augmented by the definition of variants [150, Program 11.7].

Program DECS-A(P).

The next Theorem states termination and correctness of the meta-program DECS-A(P).

**Theorem 6.1.33** Let P be an acceptable program, and Q', Q two variable disjoint queries. Then every LD-derivation for DECS-A(P) and test-s([Q'], [Q]) is finite.

Moreover, there exists a LD-refutation for them iff Q' is a computed instance of P and Q. □

By inspection of the proof of Lemma 6.1.27, we observe that the second part of the theorem holds for every program, i.e. the procedure is correct for every logic program. However, we are in the position to state termination only for acceptable programs.

**Example 6.1.34 (Sat Ctd)** Consider again the SAT program of Example 6.1.11, and suppose to have to test whether satisfiable(¬(X ∧ false)) is in S(SAT).

By the definition of S-semantics, this means to test whether satisfiable(¬(X ∧ false)) is a computed instance of any pure atom, say satisfiable(Z), variable disjoint with it. DECS-A(SAT) and the query
test-s([satisfiable(¬(X ∧ false))], [satisfiable(Z)])

have a finite LD-tree containing a LD-refutation.

By Theorem 6.1.33, satisfiable(¬(X ∧ false)) is a computed instance of SAT and satisfiable(Z), i.e. satisfiable(¬(X ∧ false)) is in S(SAT).

Analogously, DECC-A(SAT) and the query
test-c([satisfiable(¬(X ∧ false))], [satisfiable(Z)])

have a finite LD-tree containing a LD-refutation.

This implies that satisfiable(¬(X ∧ false)) is in C(SAT). □

**Example 6.1.35 (Sat-ERROR)** Consider again the SAT-ERROR program of Example 6.1.12. DECS-A(SAT-ERROR) and the query
test-s([satisfiable(¬(false ∧ X))], [satisfiable(¬(U ∧ V))])
have a finitely failed LD-tree. By Theorem 6.1.33, \( \text{satisfiable}(\text{not}(\text{false} \land X)) \) is not a computed instance of \( \text{satisfiable}(\text{not}(U \land V)) \). Since computed instances are closed under variable renaming (see [10, Note 3.28]), we conclude that \( \text{satisfiable}(\text{not}(\text{false} \land X)) \) is not a computed instance of \( \text{satisfiable}(\text{not}(Y \land X)) \). \( \square \)

**Fair-bounded Programs**

While the decision procedure w.r.t. \( C \)-semantics for acceptable programs uses a Prolog interpreter, in the case of fair-bounded programs a system adopting a fair selection rule is needed. Alternatively, the following variant of the Vanilla meta-interpreter implements in Prolog a round-robin fair selection rule.

```
demo([]).

demo([A|As]) ←
    clause(A, Ls),
    append(As, Ls, Bs),
    demo(Bs).

clause(A, [B_1, \ldots, B_n]).
    for every A ← B_1, \ldots, B_n ∈ P.

augmented by the APPEND program.

Program DEC-M-F(P).
```

Termination and correctness of the meta-interpreter is shown below.

**Theorem 6.1.36** Let \( P \) be a fair-bounded program and \( Q \) a ground query.

Then every LD-derivation of \( \text{DEC-M-F}(P) \) and the query \( \text{demo}([Q]) \) is finite.

Moreover, there exists a LD-refutation for them iff \( Q ∈ \mathcal{O}_M(P) \).

*Proof.* By Lemma 6.1.14, we can assume that \( P \) is fair-bounded w.r.t. an ambivalent language \( L \). We claim that:

\[
\vdash_t \{ \text{Pre} \} \text{DEC-M-F}(P) \{ \text{Post} \}
\]

holds, with \( \text{Post} \) well-supported, by using a level mapping \( | \cdot | \), where:

\[
\text{Pre} = \{ \text{demo}(qs) \mid qs ∈ GList \} \cup \text{clause}(A, B)|_L \\
∪ \text{Pre}_{\text{APPEND}}
\]

\[
\text{Post} = \{ \text{demo}([qs]) \mid qs = t_1, \ldots, t_n \land M^L_P \models t_1, \ldots, t_n \} \\
∪ \{ \text{clause}(a, [b_1, \ldots, b_n]) \mid a ← b_1, \ldots, b_n ∈ \text{ground}_L(P) \} \\
∪ \text{Post}_{\text{APPEND}}
\]
\[ |\text{demo}(qs)| = \begin{cases} 
\text{length}(t_1, \ldots, t_n) \cdot n_P + n & \text{if } qs = [t_1, \ldots, t_n] \\
0 & \text{otherwise}
\end{cases} \]

\[
|\text{clause}(a, ls)| = 0
\]

\[
|\text{append}(xs, ys, zs)| = |zs|.
\]

We recall that \(n_P\) is the maximum number of atoms in the body of a clause from \(P\). Since \(P\) is fair-bounded, by Theorem 6.1.15 every SLD-derivation of \(P\) and any ground query (in any language) via the round-robin selection rule \(rr\) is finite. Therefore the function \(\text{length}(P)\) (see Definition 2.3.17) maps ground queries into natural numbers. This shows that \(\|\) is a level mapping.

By showing that (6.2) holds and that \(Post\) is a well-supported interpretation, we have that every LD-derivation of \(\text{DECM-F}(P)\) and the query \(\text{demo}([Q])\) is finite, by Theorem 4.1.39. Moreover, by Theorem 4.1.40, there is a LD-refutation iff \(\text{demo}([Q])\) is in \(M^P_{\text{DECM-F}(P)}\) iff \(M^P_{\text{DECM-F}(P)} \models Q\) iff \(Q \in \mathcal{O}_{\text{sat}}(P)\).

The only non-trivial proof obligation in showing that (6.2) holds and that \(Post\) is well-supported is the decreasing of the level mapping from the head to the recursive call of the second clause. Consider a ground instance:

\[
\text{demo}([a|as]) \leftarrow \\
\text{clause}(a, ls), \\
\text{append}(as, ls, bs), \\
\text{demo}(bs).
\]

and assume that

\[ Pre \models \text{demo}([a|as]) \land Post \models \text{clause}(a, ls), \text{append}(as, ls, bs). \]

Then \(as = [as']\), \(ls = [ls']\), for some queries \(as'\) and \(ls'\). Moreover, \(a \leftarrow ls' \in \text{ground}_L(P)\) and \(bs = as + ls\). Called \(Q'\) the SLD-resolvent of \(a\) and a clause with instance \(a \leftarrow ls'\), we calculate:

\[
|\text{demo}([a \leftarrow as])| = \text{length}(a, as') \cdot n_P + |as| + 1
\]

\[
\geq \quad \{ \text{Definition of } rr \} \\
\text{length}(a, Q') \cdot n_P + n_P + |as| + 1
\]

\[ \geq \quad \{ \text{Lemma 2.3.18 (ii)} \} \\
\text{length}(a, ls') \cdot n_P + n_P + |as| + 1 \]

\[ > \quad \{ a \leftarrow ls' \in \text{ground}_L(P) \Rightarrow |ls| \leq n_P \} \\
\text{length}(bs') \cdot n_P + |ls| + |as| \]

\[ = \quad \{ bs = as + ls = [bs'] \} \\
|\text{demo}(bs)| \]

Let us now turn the attention on the decision procedure for correct instances.
test_c(Q1, Q) ←
  freeze(Q1, Qs),
  Qs = Q,
  demo(Qs).

augmented by program $\text{DECM-F}(P)$.

Program $\text{DECC-F}(P)$.

The analogous of Theorem 6.1.32 readily holds for fair-bounded programs.

Example 6.1.37 (ProdCons Ctd) As observed in Example 6.1.17, systems adopting fair selection rules are not effective procedures for deciding whether $\text{system(s(X))}$ belongs to $C(\text{PRODCONS})$. On the contrary, $\text{DECC-F(PRODCONS)}$ and the query:

$$test_c([\text{system(s(X)}], [\text{system(s(Y)}])$$

have a finite failed LD-tree. This implies that $\text{system(s(X))}$ is not a correct instance of $\text{system(s(Y))}$, which in turn implies that $\text{system(s(X))} \notin C(\text{PRODCONS})$. □

Finally, consider computed instances. The definition of the meta-interpreter must be consistent with the Prolog implementation of the round-robin fair selection rule provided by program $\text{DECM-F}(P)$ given the object program $P$. We denote by $\text{DECS-F}(P)$ the following program.

$$test_s(Q1, Q) ←$$
  $$freeze(Q1, Qs),$$
  $$demo(Qs, Q),$$
  $$variants(Q1, Q).$$

demo([], []).

demo([A|As], [B|Bs]) ←
  clause(A, Ls, Id),
  append(As, Ls, A1s),
  demo(A1s, B1s),
  clause(B, L1s, Id),
  append(B1s, L1s, B1s).

clause(A, [B1, ..., Bn], k).

for every $C_k = A ← B_1, ..., B_n \in P$

augmented by the APPEND program

and the definition of variants [150, Program 11.7].

Program $\text{DECS-F}(P)$.

The analogous of Theorem 6.1.33, stating termination and correctness of $\text{DECS-F}(P)$, can be readily established for fair-bounded programs $P$. 
Bounded Programs

Analogously, we present a decision procedure for programs bounded by computable level mappings. In this case we need an interpreter that stops computing after having reached a given depth in the attempt to construct a proof tree. The transformation of Definition 2.5.20 provides us with such an interpretation of logic programs. Alternatively, the following variant of Vanilla implements in Prolog the decision procedure w.r.t. $\mathcal{M}$-semantics.

\[
\text{demo([], \_).}
\]
\[
\text{demo([A|As], N) } \leftarrow \text{ N > 0, }
\]
\[
\text{clause(A, Ls), N1 is N - 1, }
\]
\[
\text{demo(Ls, N1), demo(As, N).}
\]
\[
\text{\textbf{clause}(A, [B_1, \ldots, B_n]).}
\]
\[
\text{for every } A \leftarrow B_1, \ldots, B_n \in P.
\]

Program $\text{DECM-B}(P)$. 

Termination and correctness of the meta-interpreter, and run-time absence of arithmetic errors are shown in the next result.

**Theorem 6.1.38** Let $Q$ be a ground query and $k$ a natural number such that $k > |A|$ for every atom $A$ in $Q$.

Then every LD-derivation of $\text{DECM-B}(P)$ and the query $\text{demo}([Q], k)$ is finite.

In addition, no LD-derivation ends in an error.

Moreover, if $P$ is bounded then there exists a LD-refutation for them iff $Q \in \mathcal{O}_\mathcal{M}(P)$.

Let us turn the attention to the decision procedure for correct instances.

\[
\text{test_c(Q1, Q, N) } \leftarrow \text{ freeze(Q1, Qs), Qs = Q, demo(Qs, N).}
\]

augmented by program $\text{DECM-B}(P)$.

Program $\text{DECC-B}(P)$. 

**Theorem 6.1.39** Let $P$ be a program bounded by a given and computable level mapping $|.|$ and $Q',Q$ two variable disjoint queries. Consider any ground instance $Q'' = A_1, \ldots, A_n$ of $Q$ and let $k = \max_{i \in [1,n]} |A_i| + 1$.

Then every LD-derivation of $\text{DECS-B}(P)$ and the query $\text{test}_L([Q'],[Q],k)$ is finite. Moreover, there exists a LD-refutation for them iff $Q'$ is a correct instance of $P$ and $Q$.

**Proof.** Let $P$ be bounded by $|.| : B_L \rightarrow N$ and $\theta = \{X_i/a_i \mid i \in [1,m]\}$ be a mapping of the variables of $Q'$ into distinct fresh constants. By Lemma 6.1.14, $P$ is bounded by some $|.|' : B_{L'} \rightarrow N$ and $L'$, where $L'$ extends $L$ and includes the fresh constants.

Let $\gamma = \{X_i/t_i \mid i \in [1,m]\}$ be such that $Q'\gamma = Q'$.

By an inspection of the proof of Lemma 6.1.14, we can define $|.|'$ such that for every atom $A$ in $Q'$ it results that $|A\theta'| = |A\gamma|$, i.e. we define $|.|'$ by assuming the replacement of $a_i$ by $t_i$. By Definition of $k$, we have $k > |A\gamma| = |A\theta'|$. Therefore, termination and correctness of the call to $\text{demo}$ in the definition of $\text{test}_L$ are implied by Theorem 6.1.38. The conclusion follows from Lemma 6.1.26. \hfill \Box

Finally, consider computed instance. We define $\text{DECS-B}(P)$ as follows:

```
\text{test}_s(Q_1, Q, N) \leftarrow
\text{freeze}(Q_1, Q_s),
\text{demo}(Q_s, Q, N),
\text{variants}(Q_1, Q).

\text{demo}([], [], ).

\text{demo}([A|As], [B|Bs], N) \leftarrow
\begin{align*}
N > 0, \\
\text{clause}(A, L_s, \text{Id}), \\
N_1 \text{ is } N - 1, \\
\text{demo}(L_s, L_1s, N_1), \\
\text{demo}(As, Bs, N), \\
\text{clause}(B, L_1s, \text{Id}).
\end{align*}
```

augmented by the definition of $\text{variants}$ [150, Program 11.7].

**Program $\text{DECS-B}(P)$**.

**Theorem 6.1.40** Let $P$ be a program bounded by a given and computable level mapping $|.|$ and $Q',Q$ two variable disjoint queries. Consider any ground instance $Q'' = A_1, \ldots, A_n$ of $Q$ and let $k = \max_{i \in [1,n]} |A_i| + 1$.

Then every LD-derivation for $\text{DECS-B}(P)$ and $\text{test}_L([Q'],[Q],k)$ is finite. In addition, no LD-derivation ends in an error.

Moreover, there exists a LD-refutation for them iff $Q'$ is a computed instance of $P$ and $Q$. \hfill \Box
6.1. Semantics Decidability and Testing

Example 6.1.41 (Circular Queues Ctd) Consider the \texttt{RQUEUE} program of Example 6.1.22. We observed that the transformational approach of Section 2.5.6 is not applicable in testing whether \texttt{remove(X, [Y,X], [Y])} is in \(S(\texttt{RQUEUE})\).

On the contrary, observe that by instantiating all variables of the atom above with \(a\), we get the ground atom \texttt{remove(a, [a,a], [a])} whose level is 3. Moreover, \texttt{DECC-B(QUEUE)} and the query:

\[
\text{test}(\texttt{remove([X], [Y,X], [Y])), [\texttt{remove(A, B, C)}], 2)}
\]

turn out to have a finitely failed LD-tree. By Theorem 6.1.40, we conclude that

\[
\text{remove([X], [Y,X], [Y])} \notin S(\texttt{RQUEUE}).
\]

\[\square\]

6.1.5 Extensions of the Approach

From now on, we omit the parameter \(P\) when referring to \texttt{DECS-A(P)} if \(P\) is clear from the context. Moreover, when not otherwise specified we reason on acceptable programs and \(S\)-semantics, with the observation that analogous reasonings apply to (fair-)bounded programs and \(C\)-semantics.

In this section, we consider extensions of the decidability procedures to include other semantics, arithmetic built-in's, meta-predicates and negation. The metaprogramming approach will be crucial to successfully model those extensions.

Semantics

Several other semantics have been proposed in the literature in addition to \(C\) and \(S\)-semantics. We recall the definition of some of them.

Definition 6.1.42 We write \(B < A\) for two atoms \(A, B\) if \(A\) is an instance of \(B\), and they are not variants.

\[
\begin{align*}
\mathcal{MC}(P) &= \{ A \in \text{Atom}_L \mid A \in \mathcal{C}(P) \text{ and for every } B < A, B \notin \mathcal{C}(P) \} \\
\mathcal{CW}(P) &= \{ A \in B_L \mid \text{there exists no SLD-refutation for } P \text{ and } A \} \\
\mathcal{FF}(P) &= \{ A \in B_L \mid \text{there exists a fin. failed SLD-tree for } P \text{ and } A \} \\
\mathcal{L}(P) &= \{ (A, n) \mid A \text{ is a computed instance of a pure atom by} \\
&\quad \text{a derivation of length } n \}. 
\end{align*}
\]

The set \(\mathcal{MC}(P)\) of more general correct instances coincides with that of computed instances \(S(P)\) when considering subsumption free programs (see [11]), namely those with no computed instances \(A, B\) such that \(A < B\). \(\mathcal{MC}(P)\) is a decidable set.
since \( C(P) \) is decidable and observing that there are finitely many atoms (modulo renaming) that are more general of a given one.

\[ CWA(P) \] is the closed world assumption set, which coincides with \( B_L \setminus M(P) \). It is a decidable set for acceptable and fair-bounded programs, and programs bounded by a computable level mapping, since its complement is decidable.

\( \mathcal{FF}(P) \) is the finite failure set of \( P \). If \( P \) is acceptable or fair-bounded then \( \mathcal{FF}(P) \) is a decidable set by Theorem 6.1.10 and Theorem 6.1.15 respectively. However, with the results of this Chapter we cannot conclude that \( \mathcal{FF}(P) \) is a decidable set in the case of programs bounded by a computable level mapping.

Finally, the computed answers with depth semantics\(^2\) \cite{49} takes into account the length of refutations. By adding a derivation length counter to \texttt{DECS-A}, we get a decision procedure w.r.t. \( L \)-semantics.

\[
\text{test} \left( Q_1, Q, N \right) \leftarrow \\
\text{freeze} \left( Q_1, Q_s \right), \\
\text{demo} \left( Q_s, Q, N \right), \\
\text{variants} \left( Q_1, Q \right). \\
\text{demo} \left( [], [], 0 \right). \\
\text{demo} \left( [A|As], [B|Bs], N \right) \leftarrow \\
\text{clause} \left( A, Ls, Id \right), \\
\text{demo} \left( Ls, L_1s, N_1 \right), \\
\text{demo} \left( As, Bs, N_2 \right), \\
N \text{ is } N_1 + N_2 + 1, \\
\text{clause} \left( B, L_1s, Id \right).
\]

augmented by the definition of \texttt{variants} \cite[Program 11.7]{150}.

**Example 6.1.43** The query `test \left( \left[ Q' \right], \left[ Q \right], X \right); X > 100` is intended to test whether \( Q' \) is a computed instance of \( Q \) by means of a derivation of length greater than 100.

**Arithmetic Built-in's**

We consider an extension of \( S \)-semantics which includes the treatment of arithmetic built-in's. In particular, we will consider the \( > \) built-in, by pointing out that the same reasonings apply to \( \langle, \leq, \langle=, =\langle, \rangle=, \rangle\rangle \).

\(^2\text{Strictly speaking, we are not in the hypothesis of Definition 6.1.2, since } L(P) \text{ is not a set of atoms. Formally, we should consider the atom } p(A, n) \text{ instead of the pair } (A, n), \text{ where } p \text{ is a fresh predicate symbol.} \)
Definition 6.1.44 The $S_{ar}$-semantics of a program with arithmetic $P$ is the set:

$$S_{ar}(P) = \bigcup_{i \geq 0} S(P \cup M_i^*)$$

where $\{M_i^*\}_{i \geq 0}$ is an increasing chain of finite sets such that:

$$\bigcup_{i \geq 0} M_i^* = \{ n > m \mid n, m \text{ gae's } \land \text{value}(n) > \text{value}(m) \}.$$

We point out that $S_{ar}(P)$ is well-defined and it does not depend on the particular chain chosen. However, the main Theorem 3.1.2 cannot be immediately extended.

Example 6.1.45 For instance, for the program $P$

$$p \leftarrow 1 > x,$$

it is clear that $p \in S_{ar}(P)$, even though the only LD-derivation for $P$ and $p$ ends in an error. $\Box$

Excluding the case of “wrong” computations (see Theorem 4.2.4 for a sufficient method), we are in the position to extend Theorem 3.1.2 as follows.

Theorem 6.1.46 Assume that no LD-derivation for a program with arithmetic $P$ and a query $Q$ ends in an error. Then the set of LD-computed instances of $P$ and $Q$ coincides with $mg_i(Q, S_{ar}(P))$.

Proof. By Definition 6.1.44 and the definition of $mg_i$, if $Q' \in mg_i(Q, S_{ar}(P))$ then there exists $i \geq 0$ such that $Q' \in mg_i(Q, S(P \cup M_i^*))$. Noting that $P \cup M_i^*$ is a logic program, by Theorem 3.1.2 and Strong completeness of SLD-resolution there exists a LD-derivation $\xi$ for $P \cup M_i^*$ and $Q$ with computed instance $Q'$. Since no LD-derivation ends in an error, we have that $\xi$ is a LD-derivation for the program with arithmetic $P$. Conversely, a LD-derivation for $Q$ and the program with arithmetic $P$ with computed instance $Q'$ is also a LD-derivation for $Q$ and the logic program $P \cup M_i^*$, for some $i$. Therefore, by Theorem 3.1.2, $Q' \in mg_i(Q, S(P \cup M_i^*))$ and then $Q' \in mg_i(Q, S_{ar}(P))$. $\Box$

This theorem gives us a method to extend Theorem 6.1.33 to programs with arithmetic, under the additional hypothesis that:

$(i)$ no LD-derivation for $P$ and $Q$ ends in an error,
$(ii)$ and $Q'$ is an instance of $Q$.

Let us suppose that the atom $n > m$ is selected in a LD-derivation for $P$ and $Q'$. The proof of Theorem 6.1.46 shows that LD-resolution for programs with arithmetic behaves like LD-resolution, i.e. either the atom succeeds or fails. Therefore, we simply add to DECS-A the following meta-level interpretation of $>$. 
Chapter 6. From Termination to Validation

\[
demo([X > Y \mid As], [X > Y \mid Bs]) \leftarrow
\begin{align*}
X > Y, \\
demo(As, Bs).
\end{align*}
\]

Hypothesis \((i)\) is imposed by Theorem 6.1.46. \((ii)\) is a sufficient condition to prevent that \textsc{decs-A} ends in an error. In fact, \((i)\) and \((ii)\) imply that no \text{LD}-derivation for \(P\) and \(Q'\theta\) ends in an error, where \(\theta\) maps variables into constants. This property lifts from \(P\) up to the meta-interpreter \textsc{decs-A}, as can be readily shown using the approach of Chapter 5.

The \texttt{call} Predicate

In addition to programs with arithmetic, we can reason on the built-in \texttt{call} (see also Section 5.3.2) by adding to \textsc{decs-A} the following clause:

\[
demo([\texttt{call}(A) \mid As], [\texttt{call}(B) \mid Bs]) \leftarrow
demo([A], [B]),
demo(As, Bs).
\]

Consider a program \(P\) containing meta-calls to \texttt{call}, and suppose that it is acceptable by a level mapping \(|\mid\) such that \(|\texttt{call}(A)| > |A|\) for every \(A \in B_L\). Then termination and correctness of the resulting reflective procedure are formally justified by a variant of Theorem 6.1.33 and the observation that \texttt{call} is defined by the (non-Horn) clause \(\texttt{call}(A) \leftarrow A\). As in the case of arithmetic built-ins, ill-typed \texttt{call}-atoms generate run-time errors. In particular, \(\texttt{call}(X)\) is ill-typed if \(X\) is a variable. Therefore, Theorem 6.1.33 can be extended under the additional hypothesis that:

\((i)\) no \text{LD}-derivation for \(P\) and \(Q\) ends in an error,

\((ii)\) and \(Q'\) is an instance of \(Q\).

Negation

General logic programs allow for the use of negation in the body of clauses. The underlying operational semantics interprets negation by means of the \textit{negation as failure rule}. Here, we discuss how to extend the decidability results to general program. First, we observe that we are not able to state an equivalent of Lemma 6.1.26 or 6.1.27 for a conservative generalization of bounded programs. Otherwise, we would have a decision procedure for \(\mathcal{E}\)-semantics of bounded logic programs. As mentioned in Section 6.1.5, we cannot conclude that with the results of this paper. Also, the extension of fair-boundedness to general programs is an open problem. Therefore, we concentrate on acceptable programs. Apt and Pedreschi \cite{APS} extended acceptability to general programs. In the following, we put \(\models A = |A|\). We recall that \texttt{comp}(\(P\)) denotes Clark's completion of \(P\).
Definition 6.1.47 A general program $P$ is acceptable by $| | : B_L \rightarrow N$ and a Herbrand interpretation $I$ iff $I$ is a model of $\text{comp}(P)$, and for every $A \in L_1, \ldots, L_n$ in $\text{ground}_L(P)$:

$$
\text{for } i \in [1,n], I \models L_1, \ldots, L_{i-1} \text{ implies } |A| > |L_i|.
$$

$P$ is acceptable if it is acceptable by some $| |$ and $I$. \hfill \Box

We observe that a general program $P$ is acceptable iff $\vdash_I \{ B_l \} P \{ I \}$ holds for some $I$, in the sense of Definition 4.3.4. Unfortunately, the basic Theorem 6.1.10 does not lift to general programs in full generality. In particular, independence from the language is lost.

Example 6.1.48 Consider the following program $P$:

$$
p(X) \leftarrow q(X), p(X).
q(a).
$$

$P$ is acceptable w.r.t. $L_P$ by $| |$ and $I$, where $I = \{ q(a) \}$, and

$$
|p(a)| = 1, \quad |q(a)| = 0.
$$

By Theorem 4.3.10, every LDNF-derivation for $P$ and any ground query in $L_P$ is finite. However, if we add a constant $b$ to the language, we have that $p(b)$ has an infinite LDNF-derivation. \hfill \Box

The desired property holds if we restrict to consider a subclass of acceptable programs. In particular, the class of acyclic programs is invariant w.r.t. the language signature, as noted at page 147. Alternatively, we can assume that $P$ is acceptable w.r.t. a sufficiently rich language.

Theorem 6.1.49 Let $P$ be a general program acceptable w.r.t. a language $L$ such that $\Sigma_L$ contains infinitely many constants, $Q'$ and $Q$ two general queries. Moreover, assume that no LDNF-derivation for $P$ and $Q$ flounders. Then it is decidable whether $Q'$ is a LDNF-computed instance of $P$ and $Q$.

Proof. First of all, we can assume that $Q'$ is an instance of $Q$. Otherwise $Q'$ cannot be a computed instance of $Q$. Let $Q''$ be the query obtained by substituting every variable of $Q'$ with distinct constants that do not appear in $P$, $Q$ or $Q''$. By the assumptions on $L$, such constants exist. The proof now continues as in Lemma 6.1.27, by noting that:

- termination of a ground query in $L$ follows by Theorem 4.3.10;
- no LDNF-derivation for $P$ and $Q''$ flounders. Otherwise, since $Q''$ is an instance of $Q$, there exists a LDNF-derivation for $P$ and $Q$ that flounders.
Implementing the treatment of negation is then immediate. We add to the DECS-A meta-interpreter the clause:

\[
\text{demo}([\not (A) \mid As], [\not (A) \mid Bs]) \leftarrow \\
\quad \text{demo}([A], [A]), \\
\quad \text{demo}(As, Bs).
\]

Note that we have assumed to represent the object-level negative literal \( \neg A \) with the term \( \not (A) \).

**Theorem 6.1.50** Let \( P \) be a general program acceptable w.r.t. a language \( L \) such that \( \Sigma_L \) contains infinitely many constants, \( Q' \) and \( Q \) two general queries such that \( Q' \) is an instance of \( Q \).

Also, assume that no LDNF-derivation for \( P \) and \( Q \) flounders.

Then every LDNF-derivation for DECS-A(\( P \)) and \( \text{test}_s([Q'], [Q]) \) is finite.

Moreover, \( Q' \) is a LDNF-computed instance of \( P \) and \( Q \) iff there exists a LDNF-computed instance of DECS-A(\( P \)) and \( \text{test}_s([Q'], [Q]) \).

The hypotheses that \( \Sigma_L \) contains infinitely many constants, and that no LDNF-derivation for \( P \) and \( Q \) flounders, are imposed by Theorems 6.1.49 and 6.1.50. The hypothesis that \( Q' \) is an instance of \( Q \), is a sufficient condition to prevent DECS-A from floundering.

### 6.1.6 Experimental Results and a Compilation-Oriented Approach

We are confident that our framework can be successfully integrated with other ones for which (semi-)automatic tools already exist, including test case generation methodologies, abstract interpreters and constraint solvers for inferring acceptability and boundedness, program structural complexity analysis, and declarative debuggers. Our confidence is also motivated by an undergoing work that is trying to integrate the tool PROTest [20] with the approach presented here and with the debugging approach that will be presented in Section 6.2. Some considerations on efficiency of the Prolog implementations can be already made at this stage as the outcome of preliminary experimental results. We compare the search space of the proposed decision procedure with the search space of the plain Prolog execution. In general, the latter may be infinite while the former is not for acceptable, fair-bounded programs, and programs bounded by computable level mappings. This fact is particularly relevant in the case of \( S \)-semantics, where a plain Prolog execution actually means running a pure atom query \( p(X_1, \ldots, X_n) \) and then checking whether there is computed instance that is a variant of a given atom. Unfortunately, even for trivial programs, the query above has infinitely many computed instances, and then the strategy is not feasible.
Let us consider now the case when the plain Prolog execution terminates. Test data sets include pairs of queries \((Q, Q')\) such that \(Q\) is supposed to have a finite (S)LD-tree and \(Q'\) is supposed to be a computed or correct instance of \(Q\). Let us recall how the decision procedures work, restricting the attention to acceptable programs. Similar considerations hold for bounded programs, while experimental results are not available for fair-bounded programs. The decision procedures search the LD-tree of \(Q\theta\), where \(\theta\) replaces variables with fresh constants. Since \(Q\theta\) is an instance of \(Q\) (if we test that \(Q'\) is an instance of \(Q\) at the beginning of the procedure), we have that the LD-tree of \(Q\theta\) is smaller or equal than that of \(Q\). In the case of \(\mathcal{C}\)-semantics, the search space is the LD-tree of \(Q\theta\), and then at worst the procedure takes the same time as the plain Prolog execution. In the case of \(\mathcal{S}\)-semantics, instead, once a refutation \(\xi\) for \(Q\theta\) is found, the decision procedure checks whether the computed instance of \(Q\) obtained by using the same clauses of \(\xi\) is a variant of \(Q'\). Therefore, in the worst case we visit a search space that is twice the LD-tree of \(Q\). However, the meta-programming implementation adds an overhead that can be significantly high. On one extreme, \textsc{DECS-A} can be several times less efficient than a plain Prolog execution strategy. This happens for:

- deterministic programs, such as \texttt{QuickSort} [150, Program 3.22], i.e. programs such that for every ground atom there is (about) only one LD-derivation;
- programs that construct the computed answer substitution in the last step of a refutation. Among them we found programs that use the technique of accumulators, such as \texttt{Queens} [150, Program 14.3].

On the other extreme, \textsc{DECS-A} is much more efficient than plain Prolog for non-deterministic programs, including \texttt{generate \\& test} and fully declarative specifications such as \texttt{NaiveQueens} [150, Program 14.2] and \texttt{ColorMap} [150, Program 14.4].

The overhead due to meta-programming can be reduced by adopting a compilation-oriented approach. The idea is to transform a program into an equivalent one that keeps track of refutations. We augment every predicate symbol by a further argument, and transform every clause:

\[
p(t) \leftarrow p_1(t_1), \ldots, p_n(t_n).
\]

with distinct identifier \(c\), into the clause:

\[
p(t, c(x_1, \ldots, x_n)) \leftarrow p_1(t_1, x_1), \ldots, p_n(t_n, x_n).
\]

where \(x_1, \ldots, x_n\) are distinct fresh variables.

**Example 6.1.51 (Sat Ctd)** For example, the transformation applied to the SAT program produces:

\[
satisfiable(true, c0).
satisfiable(X \land Y, c1(A, B)) \leftarrow
\]
Chapter 6. From Termination to Validation

Table 6.1: Experimental results (in mmsecs and ratios)

<table>
<thead>
<tr>
<th>Program</th>
<th>Plain Prolog</th>
<th>Meta</th>
<th>Compil.</th>
<th>Plain/Compil.</th>
<th>Meta/Compil.</th>
</tr>
</thead>
<tbody>
<tr>
<td>QuickSort</td>
<td>35</td>
<td>971</td>
<td>59</td>
<td>0.6</td>
<td>16.5</td>
</tr>
<tr>
<td>Queens</td>
<td>1549</td>
<td>7223</td>
<td>1902</td>
<td>0.8</td>
<td>3.8</td>
</tr>
<tr>
<td>ColorMap</td>
<td>141</td>
<td>13</td>
<td>3</td>
<td>47</td>
<td>4.3</td>
</tr>
<tr>
<td>NaiveQueens</td>
<td>25218</td>
<td>24</td>
<td>15</td>
<td>1681</td>
<td>1.6</td>
</tr>
</tbody>
</table>

Table 6.1 reports some experimental average timings (in mmsecs on SWI Prolog) and ratios, that confirm the theoretical considerations made. The meta-programming implementation is more efficient than plain Prolog in the case of non-deterministic
programs, while the overhead due to meta-programming is considerably high for deterministic programs. The compilation-oriented approach removes that overhead, and, in the worst case, its running times are at worst twice that of the plain Prolog execution. However, it should be observed that while the compilation-oriented approach is more efficient, the meta-programming one is more flexible and allows for addressing extensions of the approach in an intuitive and simple way.

\section{Declarative Debugging of Missing Answers}

A test case producing a result which is considered incorrect motivates program debugging. An incorrect result is called a \textit{symptom}, and is caused by an error in programs. Debugging is concerned with locating and solving errors. Declarative debugging is an approach to debugging which relies only on a declarative knowledge of the intended semantics of the program, while not considering its computational behavior.

In logic programming systems, two types of symptoms are usually distinguished. A query such that every atom in it is in the actual semantics of a program but not in its intended semantics is called a \textit{wrong answer}. On the other hand, a query which is valid in the intended meaning of a program but that is not in its actual semantics is a symptom called \textit{missing answer}.

\textbf{Wrong answers}

Wrong answers (sometimes called \textit{false solutions}) are caused by the presence of program clauses that are false in the intended interpretation of the program. False clauses can be detected starting from proof trees of wrong answers, such as the proof trees produced by the $PTB(P)$ meta-interpreter of Section 5.3.3. Below, we report the top-down diagnoser from Sterling and Shapiro \cite{sterling_shapiro93}, Program 17.15 w.r.t. $\mathcal{C}$-semantics.

\begin{verbatim}
wrong_answer( TA & TB, Clause ) ←
    wrong_answer( TA, Clause ).
wrong_answer( TA & TB, Clause ) ←
    wrong_answer( TA, Clause ).

wrong_answer( A if T, Clause ) ←
    false(A),
    extract_body(T).

extract_body(empty, true).
extract_body(A if T, A).
extract_body((A if T) & TB, A & Bs) ←
    extract_body(TB, Bs).
\end{verbatim}
Intuitively, this diagnoser searches in a given proof tree for an atom which is false in the intended meaning of the program. The predicate \texttt{false} is an \textit{oracle} specifying those atoms which are false in the intended semantics of the program. An oracle is some entity external to the diagnosis algorithm that can answer queries concerning the intended meaning of the program. It may be implemented either by queries to the programmer or to the user (validation debugging) or by means of an executable specification (verification debugging) of the program or by a previous version of the program (regression debugging).

Consider a wrong answer $Q = A_1, \ldots, A_n$ w.r.t. \texttt{C}-semantics, namely a query which is a logical consequence of $P$ but which is not true in $\mathcal{C}(P)$. The $\text{PTB}(P)$ meta-interpreter and the query $\text{demo}(A_1 \& \ldots \& A_n, \top)$ produce a proof tree $T$. Then, the diagnoser above and the query $\text{wrong-answer}(T, \text{Clause})$ detect an instance of a program clause in $P$ which is false in the intended interpretation of $P$.

A similar wrong answer debugger w.r.t. \texttt{S}-semantics has been proposed by Comini et al. [50].

\section*{Missing answers}

A missing answer originates from a “failure” in the construction of a proof tree for a valid query. The reason of such a failure is the presence of \textit{uncovered atoms}, i.e. of atoms $A$ in the intended interpretation of the program, for which there is no clause instance whose head is $A$ and whose body is true in the intended interpretation. In other words, there is no immediate justification in the program in order to deduce $A$. Once such an atom is found, a suitable rule should be added to the program, or an existing rule should be modified in order to cover the atom.

Debuggers in the literature find uncovered atoms starting from missing answers that have a \textit{finitely failed SLD-tree}. As we will point out, this assumption is restrictive in some cases, and it is due to a well-known limitation of the \textit{negation as failure} rule.

In this section, we propose two declarative debuggers of missing answers for \texttt{C}- and \texttt{S}-semantics that are correct for any program (in the sense that they return only uncovered atoms), and complete (in the sense that given a missing answer they eventually found an uncovered atom) and terminating for a large class of logic programs, namely acceptable programs. The implementations of the debuggers rely on decidability procedures for \texttt{C}- and \texttt{S}-semantics which are adapted from Section 6.1.1.4.

\subsection*{6.2.1 Program Semantics and Missing Answers}

The theories of \texttt{C}- and \texttt{S}-semantics associate a continuous \textit{immediate consequence operator} to programs. For a program $P$, the least fixpoint of $T^\mathcal{C}_P$, $\mathcal{C}(P)$, and the upward ordinal closure $T^\mathcal{C}_P \uparrow \omega$ coincide [77]. Similarly, the least fixpoint of $T^S_P$, $\mathcal{S}(P)$, and the upward ordinal closure $T^S_P \uparrow \omega$ coincide [76].
Definition 6.2.1 For a logic program \( P \) we define the following functions from sets of atoms into set of atoms:

\[
T^C_P(I) = \{ A \in \text{Atom}_L \mid \exists A \leftarrow B_1, \ldots, B_n \in P, \\
\{B_1, \ldots, B_n\} \subseteq I \}.
\]

\[
T^S_P(I) = \{ A \in \text{Atom}_L \mid \exists A \leftarrow B_1, \ldots, B_n \in P, \\
B'_1, \ldots, B'_n \text{ variants of atoms in } I \text{ and renamed apart} \\
\text{ and } \exists \theta = \text{mgu}((B_1, \ldots, B_n),(B'_1, \ldots, B'_n)) \}
\]

The intended interpretation of a program w.r.t. a semantics is a set of atoms which, in the intentions of the programmer, is supposed to be the actual semantics of the program. Starting points of the debugging analysis are missing answers.

Definition 6.2.2 We say that a query is in a set of atoms if every atom of the query is in the set.

Let \( \mathcal{F} \) be the \( C \)- or \( S \)-semantics, and \( \mathcal{I} \) be the intended interpretation of a program \( P \) w.r.t. \( \mathcal{F} \). A missing answer w.r.t. \( \mathcal{F} \) is any query which is in \( \mathcal{I} \) but that is not in \( \mathcal{F}(P) \).

Missing answer are caused by uncovered atoms, i.e. atoms valid in the intended meaning of a program that have no immediate justification in the program.

Definition 6.2.3 Let \( \mathcal{F} \) be the \( C \)- or \( S \)-semantics, and \( \mathcal{I} \) be the intended interpretation of a program \( P \) w.r.t. \( \mathcal{F} \).

An atom \( A \) is uncovered if \( A \in \mathcal{I} \) and \( A \notin T^C_P(\mathcal{I}) \).

6.2.2 Shapiro’s Debugger

Consider the semantics of correct instances of logic programs, i.e. \( C \)-semantics. A query \( Q \) which is supposed to be a logical consequence of the program, “fails” if it is not. However, by saying that a query \( Q \) “fails” it is often meant \( Q \) finitely fails. This stronger assumptions is due a well-known limitation of the negation as failure rule, and it affects several declarative debuggers in the literature. Let us consider the Shapiro’s debugger [144] [121, Debugger S.1] as an example. Let \( P \) be the program under analysis.

\[
\text{miss([A | B], Goal) } \leftarrow \\
\quad \neg \text{call(A)}, \\
\quad \text{miss(A, Goal)}. \\
\text{miss([A | B], Goal) } \leftarrow \\
\quad \text{call(A)}, \\
\quad \text{miss(B, Goal)}. 
\]
Chapter 6. From Termination to Validation

\[
\text{miss}(A, \text{Goal}) \leftarrow \\
\quad \text{user}\_\text{pred}(A), \\
\quad \text{clause}(A, B), \\
\quad \text{valid}(B), \\
\quad \text{miss}(B, \text{Goal}). \\
\text{miss}(A, A) \leftarrow \\
\quad \text{user}\_\text{pred}(A), \\
\quad \neg \text{covered}(A). \\
\text{covered}(A) \leftarrow \\
\quad \text{clause}(A, B), \\
\quad \text{valid}(B). \\
\text{clause}(A, [B_1, \ldots, B_n]). \quad \text{for every } A \leftarrow B_1, \ldots, B_n \in P
\]

\text{augmented by } P.

\text{user}\_\text{pred} \text{ characterizes user-defined predicates. Its definition consists of a set of facts } \text{user}\_\text{pred}(p(X_1, \ldots, X_n)) \text{ for every predicate symbol } p \text{ of arity } n. \text{ valid is an oracle defining the intended meaning of } P.

\text{Example 6.2.4} \text{ Consider now the program:}

\begin{align*}
\text{s.} \\
p(X) & \leftarrow q(Y), r(Y, X). \\
q(a). \\
\% q(b). \% \text{ missing} \\
r(a, c). \\
r(b, x).
\end{align*}

Shapiro's debugger correctly works when the missing answer in input has a finitely failed tree. On the other hand, the query \(p(X), s\) is a missing answer, since \(p(X)\) is not a logical consequence of the program, but there is no finitely failed tree, since there is a refutation that instantiates \(X\) to \(c\). A call \(\text{miss}([p(X), s], A)\) to the Shapiro's debugger fails to return that \(q(b)\) is uncovered.

\(\square\)

The need for the hypothesis of finite failure lies in the use of negation in clauses such as:

\[
\text{miss}([A \mid B], \text{Goal }) \leftarrow \text{call}(A), \text{miss}(A, \text{Goal}).
\]

where the debugger tries to prove \(\neg \text{call}(A)\). Due to well-known limitations of the negation as failure rule, \(\neg \text{call}(A)\) succeeds if \(P \models \neg \exists A\), i.e. if there is a finitely-failed SLD-tree. Instead, the intended use of \(\neg \text{call}(A)\) is to prove \(P \models \neg \forall A\), i.e. that \(A\) is not a logical consequence of \(P\). A form of completeness has been shown for Ferrand's debugger [78], [21, Debugger F.1], which is obtained from Shapiro's debugger by removing the literals \(\neg \text{call}(A)\) and \(\text{call}(A)\). As an example, the
uncovered atom \( q(b) \) of the program of Example 6.2.4 is detected by Ferrand's debugger. However, Ferrand's approach differs from ours in the fact that it considers \textit{impossible} atoms instead of uncovered atoms. \( A \) is impossible if no instance of \( A \) is uncovered.

**Example 6.2.5** Let us consider the following incorrect \texttt{E-APPEND} version of the \texttt{APPEND} program:

\begin{verbatim}
append([], Xs, Xs).
append([X|Xs], Ys, [X|Zs]) ←
append(Ys, Ys, Zs).  % should be append(Xs, Ys, Zs).
\end{verbatim}

The query \( Q \equiv \text{append}([X], Ys, [X|Ys]) \) is a missing answer, since it is in the intended interpretation, and it is not a logical consequence of \texttt{E-APPEND}. However, Shapiro's debugger is not able to find out that \( Q \) is an uncovered atom, since \( Q \) has not a finitely failed tree. On the other hand, Ferrand's debugger does not show that \( Q \) is uncovered, due to the fact that there exists an instance of \( Q \) which is covered, namely \texttt{append}([X], [1], [X]), and then \( Q \) is not an impossible atom. Finally, it is worth noting that both debuggers ask the oracle for a valid instance of \texttt{append(Ys, Ys, Ys)}. Consequently, the call \texttt{miss(append([], [], []), Goal)} is made, where \texttt{append([], [], [])} is not a missing answer. In general, unnecessary questions are addressed to the oracle, in the sense that the search space includes queries that are not missing answers, and then cannot lead to an uncovered atom.

---

### 6.2.3 Semantics Decidability

In this section, we specialize the decision procedures of Section 6.1.4 for acceptable programs to the case of semantics. We delay the treatment of fair-bounded and bounded programs until Section 6.2.5. The decision procedure for \( C \) - and \( S \) -semantics will be the crucial in the debugging approach of this Chapter. Here, we assume that \texttt{freeze} and its dual \texttt{melt} are available. Given a term \( B \), \texttt{melt}(\( B \), \( A \)) replaces every constants of \( B \) introduced by freezing some term by the original variable to obtain \( A \). For instance \texttt{freeze(p(X), Y)} succeeds by instantiating \( Y \) to \( p(a_X) \), where \( a_X \) is a fresh constant representing the frozen variable \( X \). On the other hand, \texttt{melt(p(a_X, A)} succeeds binding \( A \) to \( p(X) \). The following is the decision procedure for \( S \) -semantics.

\begin{verbatim}
in_s(A) ←
freeze(A, A1),
pure(A, B),
demo([A1],[B]),
variants(A, B).
demo([], []).  
\end{verbatim}
demo([A|As], [B|Bs]) ←
    clause(A, Ls, Id),
    demo(Ls, Lis),
    demo(As, Bs),
    clause(B, L1s, Id).

pure(p(X1, ..., Xn), p(Y1, ..., Yn)).
    for every predicate symbol p of arity n

clause(A, [B1, ..., Bn], k).
    for every Ck = A ← B1, ..., Bn ∈ P

augmented by the definition of variants [150, Program 11.7].

Program IN-S(P).

pure(A, B) computes a pure atom for the predicate symbol of a given atom. Finally, as a corollary of Theorem 6.1.33, given a program P, we have that for an atom A, \( \text{in}_S(A) \) succeeds iff \( A ∈ S(P) \). Moreover, if \( P \) is acceptable then every LD-derivation of \( \text{in}_S(A) \) is finite.

Next we present the procedure for \( \mathcal{C} \)-semantics. Given a program \( P \) and an atom \( A \), \( \text{in}_C(A) \) succeeds iff \( A ∈ C(P) \). Moreover, if \( P \) is acceptable then every LD-derivation of \( \text{in}_C(A) \) is finite.

\[
\begin{align*}
\text{in}_C(A) & \leftarrow \\
    \text{freeze}(A, A1), \\
    \text{call}(A1).
\end{align*}
\]

augmented by \( P \).

Program IN-C(P).

Without loss of generality, we assume that the predicate symbol \( \text{in}_C \) does not appear in \( P \).

Example 6.2.6 (Pre-order Tree Traversal) Consider the following program E-PREORDER, which is a variant of the PREORDER program of Example 4.1.2 where the variable \( \text{x} \) in clause \( (p2) \) has been erroneously typed in lower case.

\[
\begin{align*}
(p1) \quad & \text{preorder(nil, [] ).} \\
(p2) \quad & \text{preorder(leaf(x), [x] ).} \\
(p3) \quad & \text{preorder(tree(X, Left, Right), Ls) ← } \\
& \text{preorder(Left, As),} \\
& \text{preorder(Right, Bs),} \\
& \text{append([X|As], Bs, Ls).}
\end{align*}
\]
append([], Xs, Xs).
append([X|Xs], Ys, [X|Ys]) ←
append(Xs, Ys, Zs).

\textbf{E-PREORDER} is acceptable by \( \| \) and \( I \), where

\[
\begin{align*}
|\text{preorder}(t, \text{ls})| &= \|t\| + 1 \\
|\text{append}(xs, ys, zs)| &= |xs| \\
I &= \{ \text{append}(xs, ys, zs) \mid |zs| = |xs| + |ys| \} \cup \\
&\quad \cup \{ \text{preorder}(t, \text{ls}) \mid |ls| = \|t\| \}
\end{align*}
\]

where \( \|t\| \) is the number of nodes of a tree \( t \) (see definition at page 100). \textbf{E-PREORDER} is acceptable by \( \| \) and \( I \), hence it is acceptable. Note that \texttt{preorder(tree(E, leaf(X), leaf(Y)), [E, X, Y])} is a missing answer w.r.t. \( \mathcal{C} \) and \( \mathcal{S} \)-semantics. However, Shapiro's debugger is not able to find an uncovered atom starting from it, due to the fact that there is no finitely failed SLD-tree for it. On the contrary, observe that the LD-tree of \texttt{IN-S(E-PREORDER)} and the query:

\texttt{in}_s(\text{preorder(tree(E, leaf(X), leaf(Y)), [E, X, Y]))}

is finitely failed. \hfill \Box

\subsection{Declarative Debuggers}

\textit{C-semantics}

We revise the Shapiro's debugger, by integrating the decision procedure \texttt{in}_c within it.

\begin{enumerate}
\item[(o)] \texttt{missing_answers_c(Q, Goal) ←}
\item[(i)] \texttt{miss([A | B], Goal) ←}
\item[(ii)] \texttt{miss([A | B], Goal) ←}
\item[(iii)] \texttt{miss(A, Goal) ←}
\end{enumerate}

\item user\_pred(A),
\item freeze(A, A1),
\item clause(A1, B),
\item valid\_c(B),
\[
\text{miss}(E, \text{Goal}).
\]

\((iv)\) \hspace{1em} \text{miss}(A, A1) \leftarrow \text{user}\_\text{pred}(A), \\
\hspace{2em} \neg \text{covered}(A).

\((v)\) \hspace{1em} \text{covered}(A) \leftarrow \\
\hspace{2em} \text{freeze}(A, A1), \\
\hspace{3em} \text{clause}(A1, B), \\
\hspace{4em} \text{valid}\_c(B).

\text{clause}(A, [B_1, \ldots, B_n]). \hspace{1em} \text{for every } A \leftarrow B_1, \ldots, B_n \in P

augmented by program \text{IN}\_c(P).

\textbf{Program MISS\_c(P)}.

Without loss of generality, we assume that the predicates symbols of the object-program \(P\) are disjoint from the symbols defined by the debugger.

\text{valid}\_c is an oracle describing the queries in the intended interpretation \(I\). Formally, called \(V\) the definition of \text{valid}\_c, an atom \text{valid}\_c([B]) is in \(C(V)\) iff \(B\) is in \(I\).

\textbf{Example 6.2.7} Consider now the program of Example 6.2.4, which is readily checked to be acceptable. We have the following definitions of \text{user}\_\text{pred} and \text{valid}\_c:

\begin{align*}
\text{user}\_\text{pred}(s). \\
\text{user}\_\text{pred}(p(X)). \\
\text{user}\_\text{pred}(q(X)). \\
\text{user}\_\text{pred}(r(X, Y)).
\end{align*}

\begin{align*}
\text{valid}\_c(s). \\
\text{valid}\_c(p(X)). \\
\text{valid}\_c(q(a)). \\
\text{valid}\_c(q(b)). \\
\text{valid}\_c(r(a,c)). \\
\text{valid}\_c(r(b,X)). \\
\text{valid}\_c([A|B]) \leftarrow \\
\hspace{1em} \text{valid}\_c(A), \text{valid}\_c(B). \\
\text{valid}\_c([]).
\end{align*}

A call \text{missing\_answers\_c(\ [p(X), s], \ A)} \text{ has a finite LD-tree, and computes the uncovered atom } A = q(b). The computation progresses as follows. First \(p(X)\) is found to be not a logical consequence of the program. Then the clause \(p(X) \leftarrow q(Y), r(Y,X)\) is instantiated by \(Y = b\) in order to find a valid body. Note that the instance \(Y = a, X = c\) cannot be considered as \(X\) is frozen. Finally, \(q(b)\) is found to be an uncovered atom. \(\square\)
**Example 6.2.8** Consider \texttt{E-PREORDER} and the missing answer

\[ Q = \text{preorder(tree(E, leaf(X), leaf(Y)), [E, X, Y])}. \]

A call \texttt{missing_answers.c([Q], A)} returns \( A = \text{preorder(leaf(X), [X])} \), which is indeed the correctly typed clause \((p^2)\).

Consider now the program of Example 6.2.5, i.e. the readily checked acceptable program \texttt{E-APPEND}, and the missing answer \( Q = \text{append([X], Ys, [X|Ys])} \). The proposed debugger shows that \( Q \) is an uncovered atom. The only query to the oracle during the computation is \texttt{valid.c(append(Ys, Ys, Ys))} with \( Ys \) frozen. In other words, the oracle is asked whether \texttt{append(Ys, Ys, Ys)} is valid in the intended meaning of \texttt{E-APPEND}, which is obviously false. \( \Box \)

The debugger is correct for every logic program.

**Theorem 6.2.9 (C-Correctness)** Let \( P \) be a program, and \( Q \) a missing answer w.r.t. \( C \)-semantics. If MISS-\( C(P) \) and \texttt{missing_answers.c([Q], Goal)} have a LD-computed instance \texttt{missing_answers.c([Q], Goal)} then \( \text{Goal} \) is an uncovered atom.

**Proof.** First of all, we consider a language \( L' \) obtained by adding to \( L \) sufficiently many new constants, which are employed by the predicate \texttt{freeze}. Let us show that we are in the hypotheses of the Theorem considering \( L' \) instead of \( L \), and

\[ \mathcal{I}' = \{ A \theta \in \text{Atom}_L' \mid A \in \text{Atom}_L, A \in \mathcal{I} \} \]

instead of \( \mathcal{I} \). Since \( Q \) is a missing answer w.r.t. \( \mathcal{I} \), then \( Q \) is in \( \mathcal{I}' \) and \( P \not\models Q \), i.e. \( Q \) is a missing answer w.r.t. \( \mathcal{I}' \). Moreover the definition \( V \) of \texttt{valid.c} is an oracle w.r.t. \( \mathcal{I}' \). In fact, consider any query \( Q \) in \( L' \). \( Q \) can be written as \( Q' \theta \), where \( Q' \) is in \( L \) and \( \theta \) replaces some variables of \( Q' \) with distinct constants not in \( L \). Then, we have that \( Q \) is in \( \mathcal{I}' \) iff \( Q' \) is in \( \mathcal{I} \), and then, since \( V \) is an oracle, iff \( V \models \text{valid.c}([Q']) \). By the Theorem on Constants (see e.g. [146]), \( V \models \text{valid.c}([Q']) \) iff \( V \models \text{valid.c}([Q']) \theta \), when \( \theta \) is of the considered form. This implies, that \( Q \) is in \( \mathcal{I}' \) iff \( V \models \text{valid.c}([Q]) \), i.e. that \( V \) is an oracle w.r.t. \( \mathcal{I}' \). We now show that any computed instance of \texttt{miss([Q], Goal)} returns an uncovered atom on \( L' \).

The proof proceeds by induction on the number \( n \) of calls to \texttt{miss} in a refutation.

\( (n = 1) \). \texttt{Goal} can be only instantiated by applying rule (iv). There is no clause instance \( A1 \leftarrow B \) whose body is in \( \mathcal{I}' \), i.e. \( A1 \not\in T^c_{\mathcal{I}'}(\mathcal{I}) \), and \( A1 \) is obtained by freezing \( A \), hence \( A1 \) in \( \mathcal{I}' \). Then \texttt{Goal} is instantiated by an uncovered atom.

\( (n > 1) \). We show that the hypothesis of the theorem holds for calls to \texttt{miss} in clauses \( (i, ii, iii) \).

\( (i) \). Since \( \neg \texttt{in.c}(A) \) succeeds, \( A \) is not in \( C(P) \) albeit by hypothesis it is in \( \mathcal{I}' \). Therefore, \( A \) is a missing answer.
(ii) Since $\text{in}_c(A)$ succeeds, $A$ is in $\mathcal{C}(P)$. Therefore, $B$ must be a missing answer.

(iii) $A_1 \leftarrow B$ is a clause instance such that $A_1$ is obtained by freezing $A$ and $B$ is in $T'$. By the Theorem on Constants, $A$ is not in $\mathcal{C}(P)$ implies $A_1$ not in $\mathcal{C}(P)$. As a consequence $B$ is not in $\mathcal{C}(P)$. Otherwise, by Definition 6.2.1, $A_1$ would be in $T_0^c(\mathcal{C}(P)) = \mathcal{C}(P)$. Therefore, the call $\text{miss}(B, \text{Goal})$ satisfies the inductive hypothesis, i.e. $B$ is a missing answer.

In conclusion, the call $\text{miss}(Q, \text{Goal}_1)$ in (o) instantiates $\text{Goal}_1$ with an uncovered atom $\text{Goal}$ on $I'$. By melting the frozen variables of $\text{Goal}_1$, we obtain an atom $\text{Goal}$ such that $\text{Goal}$ is in $\mathcal{I}$ (since $\text{Goal}_1$ is in $T'$) but not in $T_0^c(\mathcal{I})$ (otherwise $\text{Goal}_1$ would be in $T_0^c(\mathcal{I'})$, i.e. $\text{Goal}$ is uncovered.

Restricting the attention to acceptable programs, we are in the position to show completeness of the debugger. However, we need the further hypothesis that there are finitely many oracle’s answers for $Q$.

**Definition 6.2.10** We say that there are finitely many oracle’s answers for $Q$ if there are finitely many LD-derivations for every call to valid$_c$ during a LD-derivation for missing$_c\text{answers}_c([Q], \text{Goal})$. □

**Theorem 6.2.11 (C-Completeness)** Let $P$ be an acceptable program, and $Q$ a missing answer w.r.t. C-semantics such that there are finitely many oracle’s answers for $Q$.

Then there exists a LD-computed instance of miss$_c\text{C}(P)$ and missing$_c\text{answers}_c([Q], \text{Goal})$.

**Proof.** Reasoning as in the proof of Theorem 6.2.9, we can assume a language with infinitely many constants.

We observe that every prefix $\xi$ of a LD-derivation for $Q$ is finite if the variables of $Q$ are never instantiated along $\xi$. In fact, let $\theta$ be a substitution of the variables of $Q$ with new distinct constants. If there is an infinite prefix $\xi$ of a LD-derivation for $Q$ such that the variables of $Q$ are never instantiated along $\xi$, then $\xi\theta$ would be an infinite prefix of a LD-derivation for $Q\theta$. This is impossible by Theorem 6.1.10, since $Q\theta$ is ground. We denote by $d_Q$ the maximum length of a prefix of a LD-derivation for $Q$ that does not instantiate any variable of $Q$.

The proof proceeds by induction on $d_Q$.

($d_Q = 0$). Let $A$ be the leftmost atom in $Q$. We claim that $A \not\in \mathcal{C}(P)$. Otherwise, by strong completeness of SLD-resolution, there exists a LD-refutation for $A$ that does not instantiate any variable of $A$. As a consequence, $d_Q > 0$.

Therefore, $A$ is a missing answer. By applying clause (i) the query $\text{miss}(A, \text{Goal})$ is resolved. We now distinguish two cases: either $A$ is or not a variant of the head of a clause instance $A_1 \leftarrow B$ such that $B$ is in the intended interpretation $\mathcal{I}$. In the latter case, by resolving $\text{miss}(A, \text{Goal})$ with clause (iv) we get a refutation, since there are finitely many oracle answers.
In the former case, \(A\) unifies with a clause head without instantiating its variables, and then \(d_A > 0\) and \(d_Q > 0\). In conclusion, the former case is impossible.

\((d_Q > 0)\). Let \(A\) be the leftmost atom in \(Q = D, A, E\) such that \(A \not \in C(P)\). By repeatedly applying clauses \((i, ii)\) the query \(\text{miss}(A, \text{goal})\) is eventually resolved, since for acceptable programs the calls to \(\text{in}\) terminate. Moreover, since \(D\) is in \(C(P)\) then by strong completeness of SLD-resolution, there exists a LD-refutation for \(D\) that does not instantiate any variable of \(D\), hence \(d_Q \geq d_A\).

Again, we distinguish two cases: either \(A\) is or not a variant of the head a clause instance of \(P\) such that the body is in the intended interpretation \(I\). In the latter case, by resolving \(\text{miss}(A, \text{goal})\) with clause \((iv)\) we get a refutation, since there are finitely many oracle answers.

In the former case, clause \((iii)\) is applicable and a query \(\text{miss}(B, \text{goal})\) is eventually resolved where \(A_1 \leftarrow B\) is an instance of a clause \(c\) from \(P\) such that \(A_1\) is obtained by freezing \(A\) (i.e., \(A_1 = A_{\mu}\) for \(\mu\) substituting variables with fresh constants) and \(B\) is in \(I\). As shown in the proof of Theorem 6.2.9, \(B\) cannot be in \(C(P)\), otherwise \(A\) would be. Therefore, \(B\) is a missing answer. It is readily checked that \(d_A = d_{A_1}\). We claim that \(d_{A_1} > d_B\). In fact, let \(\xi\) be the prefix of a LD-derivation for \(P\) and \(B\) that does not instantiate any variable of \(B\). We observe that the LD-resolvent of \(A_1\) and \(c\) is more general than \(B\). Therefore, there exists \(\xi'\) prefix of a LD-derivation for \(A_1\) longer than \(\xi\). Summarizing, \(d_{A_1} > d_B\). As a consequence, \(B\) is a missing answer and \(d_Q \geq d_A = d_{A_1} > d_B\). Therefore we can apply the inductive hypothesis on \(B\) to obtain the conclusion of the Theorem.

Finally, we have termination of the debugger.

**Theorem 6.2.12 (C-Termination)** Let \(P\) be an acceptable program, and \(Q\) a query such that there are finitely many oracle’s answers for \(Q\). Then \(\text{MISS-C}(P)\) and \(\text{missing answers}_C([Q], \text{Goal})\) universally left terminate.

**Proof.** Suppose there is an infinite LD-derivation. Since calls to \(\text{in}\) and \(\text{valid}\) terminate, then it necessarily happens that clause \((iii)\) is called infinitely many times: \(\text{miss}(A_1, \text{goal}), \ldots, \text{miss}(A_n, \text{goal}), \ldots\). By reasoning as in the proof of Theorem 6.2.11, we have that \(d_{A_1}, \ldots, d_{A_n}, \ldots\) is an infinite decreasing chain of naturals. This is impossible since naturals are well-founded.

**S-semantics**

We observe that clauses \((iii-v)\) of program \(\text{MISS-C}(P)\) followed directly from the definition of uncovered atoms (Definition 6.2.3) and the definition of \(T_P^S\) (Definition 6.2.1). We derive the debugger for S-semantics similarly, but considering now \(T_P^S\).

\((o)\) \[\text{missing answers}_S(Q, \text{Goal}) \leftarrow \text{miss}(Q, \text{Goal}).\]

\((i)\) \[\text{miss}([A \mid B], \text{Goal}) \leftarrow\]
\[- \text{in}_{\mathcal{S}}(A),
\quad \text{miss}(A, \text{Goal}).\]

(ii) \[
\text{miss}([A \mid B], \text{Goal}) \leftarrow
\quad \text{in}_{\mathcal{S}}(A),
\quad \text{miss}(B, \text{Goal}).
\]

(iii) \[
\text{miss}(A, \text{Goal}) \leftarrow
\quad \text{user}_{\text{pred}}(A),
\quad \text{pure}(A, A_1),
\quad \text{clause}(A_1, B),
\quad \text{valid}_{\mathcal{S}}(B, C),
\quad \text{variants}(A, A_1),
\quad \text{miss}(C, \text{Goal}).
\]

(iv) \[
\text{miss}(A, A) \leftarrow
\quad \text{user}_{\text{pred}}(A),
\quad \neg \text{covered}(A).
\]

(v) \[
\text{covered}(A) \leftarrow
\quad \text{pure}(A, A_1),
\quad \text{clause}(A_1, B),
\quad \text{valid}_{\mathcal{S}}(B, C),
\quad \text{variants}(A, A_1).
\]

\[
\text{clause}(A, [B_1, \ldots, B_n]). \quad \text{for every } A \leftarrow B_1, \ldots, B_n \in P
\]

augmented by program \texttt{IN-S}(P).

\textbf{Program \texttt{MISS-S}(P).}

\texttt{valid}_{\mathcal{S}} \text{ is an oracle describing the queries in the intended interpretation } \mathcal{I}. \text{ Formally, called } V \text{ the definition of } \texttt{valid}_{\mathcal{S}}, \text{ an atom } \texttt{valid}_{\mathcal{S}}([B], [C]) \text{ is in } \mathcal{S}(V) \text{ iff } B \text{ and } C \text{ are queries whose atoms are in } \mathcal{I} \text{ and variable disjoint, and } C \text{ is a variant of } B. \text{ By } [76, \text{ Theorems 7.1 and 7.7}], \text{ a call } \texttt{valid}_{\mathcal{S}}(B, C) \text{ has a computed instance } \texttt{valid}_{\mathcal{S}}(B, C)\theta \text{ iff for some renamed apart atom } B \text{ in } \mathcal{S}(V), \mu = \text{mgu}(\texttt{valid}_{\mathcal{S}}(B, C), B) \text{ and } \texttt{valid}_{\mathcal{S}}(B, C)\theta = \texttt{valid}_{\mathcal{S}}(B, C)\mu. \text{ Therefore, for an atom } A \text{ the query}

\[
\texttt{pure}(A, A_1), \texttt{clause}(A_1, B), \texttt{valid}_{\mathcal{S}}(B, C), \texttt{variants}(A, A_1)
\]

has a LD-refutation iff there exists a renamed apart clause \(A_1 \leftarrow B\) such that \(\theta = \text{mgu}(B, C)\) for some \(C\) in \(\mathcal{I}\) and \(A_1\theta\) is a variant of \(A\), i.e. iff \(A \in T^S_p(\mathcal{I})\).

\textbf{Example 6.2.13} Consider now, as an example, the following variant of the program of Example 6.2.4.
We have the following definition of valid:\$s:\$
\begin{align*}
\text{valid}_s(s, s). \\
\text{valid}_s(p(X), p(Y)). \\
\text{valid}_s(p(c), p(c)). \\
\text{valid}_s(q(a), q(a)). \\
\text{valid}_s(q(b), q(b)). \\
\text{valid}_s(r(a, c), r(a, c)). \\
\text{valid}_s(r(b, X), r(b, Y)). \\
\text{valid}_s([A|As], [B|Bs]) \leftarrow \\
\text{valid}_s(A, B), \\
\text{valid}_s(As, Bs). \\
\text{valid}_s([], []). 
\end{align*}

A call missing\_answers\_s([p(c), s], A) has a finite LD-tree, and returns the uncovered atom q(a). \[\square\]

The debugger is correct for every logic program.

**Theorem 6.2.14 (S-Correctness)** Let \( P \) be a program, and \( Q \) a missing answer w.r.t. S-semantics. If \( \text{MISS}\_S(P) \) and missing\_answers\_s([Q], Goal) have a LD-computed instance missing\_answers\_s([Q], Goal) then \( \text{Goal} \) is an uncovered atom.

**Proof.** The proof is by induction on the number \( n \) of calls to \text{miss} in a refutation.

\( (n = 1) \). \( \text{Goal} \) can be only instantiated by applying rule (\textit{iv}), i.e. if \( Q \) is an atom and there is no clause \( A1 \leftarrow B \) whose body unifies with a query in the intended interpretation \( \mathcal{I} \) with \text{mgu} \( \theta \) and \( A1\theta \) is a variant of \( Q \), namely if \( Q \not\in T_\mathcal{I}(\mathcal{I}) \).

\( (n > 1) \). We show that the hypothesis of the theorem holds for calls to \text{miss} in clauses \((i, ii, iii)\).

\( (i) \) Since \( \text{in}_S(A) \) succeeds, \( A \) is not in \( S(P) \) albeit by hypothesis it is in \( \mathcal{I} \). Therefore, \( A \) is a missing answer.

\( (ii) \) Since \( \text{in}_S(A) \) succeeds, \( A \) is in \( S(P) \). Therefore, \( B \) must be a missing answer.

\( (iii) \) Assume that
\begin{align*}
\text{pure}(A, A1), \text{clause}(A1, B), \text{valid}_s(B, C), \text{variants}(A, A1)
\end{align*}
succeeds. Then there exists a renamed apart clause \( A_1 \leftarrow B \) such that \( \theta = \text{mgu}(B, C) \) for some \( C \) in \( I \) and \( A_1 \theta \) is a variant of \( A \). Since \( A \) is not in \( \mathcal{S}(P) \), then \( C \) is not in in \( \mathcal{S}(P) \). Otherwise, by Definition 6.2.1, \( A \) would be in \( T(S(P)) = \mathcal{S}(P) \). Summarizing, by definition of \( \text{valids} C \) is in \( I \), and we showed that \( C \) is not in \( \mathcal{S}(P) \). Therefore, the call \( \text{miss}(C, \text{Goal}) \) satisfies the inductive hypothesis, i.e. \( C \) is a missing answer. □

Restricting the attention to acceptable programs, we are in the position to show completeness of the debugger. Also, we assume that there are finitely many oracle’s answers.

**Definition 6.2.15** We say that there are finitely many oracle’s answers for \( Q \) iff there are finitely many LD-derivations for every call to \( \text{valids} \) during a LD-derivation for \( \text{missing answers}([Q], \text{Goal}). \)

**Theorem 6.2.16 (S-Completeness)** Let \( P \) be an acceptable program, and \( Q \) a missing answer w.r.t. \( S \)-semantics such that there are finitely many oracle’s answers. Then there exists a LD-computed instance of \( \text{MISS-S}(P) \) and \( \text{missing answers}([Q], \text{Goal}). \)

**Proof.** As shown in the proof of Theorem 6.2.11, every prefix \( \xi \) of a LD-derivation for \( Q \) is finite if the variables of \( Q \) are never instantiated along \( \xi \). We denote by \( d_Q \) the maximum length of a prefix of a LD-derivation for \( Q \) that does not instantiate any variable of \( Q \). The proof proceeds by induction on \( d_Q \).

\( (d_Q = 0) \). Let \( A \) be the leftmost atom in \( Q \). We claim that \( A \not\in \mathcal{S}(P) \). Otherwise \( A \in \mathcal{C}(P) \). Then by strong completeness of SLD-resolution, there exists a LD-refutation for \( A \) that does not instantiate any variable of \( A \). As a consequence, \( d_Q > 0 \).

Therefore, \( A \) is a missing answer. By applying clause \( (i) \) the query \( \text{miss}(A, \text{Goal}) \) is resolved. We now distinguish two cases: either \( A \) is or not a variant of \( A_1 \mu \) where \( A_1 \leftarrow B \) is a clause of \( P \) and \( \mu = \text{mgu}(B, C) \) for some \( C \) in \( I \). In the latter case, we observe that by resolving \( \text{miss}(A, \text{Goal}) \) with clause \( (iv) \) we get a refutation, since there are finitely many oracle answers. In the former case, \( A \) unifies with a clause head without instantiating its variables, and then \( d_A > 0 \) and \( d_Q > 0 \). In conclusion, the latter case is impossible.

\( (d_Q > 0) \). Let \( A \) be the leftmost atom in \( Q = D, A, E \) such that \( A \not\in \mathcal{S}(P) \). By repeatedly applying clauses \( (i, ii) \) the query \( \text{miss}(A, \text{Goal}) \) is eventually resolved, since for acceptable programs the calls to \( \text{ins} \) terminate. Moreover, since \( \mathcal{S}(P) \subseteq \mathcal{C}(P) \), then by strong completeness of SLD-resolution, there exists a LD-refutation for \( D \) that does not instantiate any variable of \( D \), hence \( d_Q \geq d_A \).

Again, we distinguish two cases: either \( A \) is or not a variant of \( A_1 \mu \) where \( c: A_1 \leftarrow B \) is a clause of \( P \) and \( \mu = \text{mgu}(B, C) \) for some \( C \) in \( I \). In the latter case, we observe that by resolving \( \text{miss}(A, \text{Goal}) \) with clause \( (iv) \) we get a refutation, since there are finitely many oracle answers.
In the former case, clause (iii) is applicable and miss(C, Goal) is eventually resolved. In fact, by definition of valid, the query pure(A, A1), clause(A1, B), valid(B, C), variants(A, A1) succeeds under the stated hypothesis. As shown in the proof of Theorem 6.2.9, C cannot be in S(P), otherwise A would be in S(P), and then C is a missing answer. We claim that d_A > d_C.

Let ξ be the prefix of a LD-derivation for P and C that does not instantiate any variable of C. Since A and A1 are variants, then there exists a renaming substitution σ such that A = A1μσ. Moreover, A ← Cμσ is an instance of c. Let θ be a substitution mapping all variables of A ← Cμσ into distinct fresh constants. Since no variable of C is instantiated in ξ, then there exists a prefix ξ′ of a LD-derivation for Cμσθ of the same length of ξ.

We observe that the LD-resolvent of Aθ and c is more general than Cμσθ, since Aθ ← Cμσθ is an instance of c. Therefore, there exists a prefix ξ″ of a LD-derivation for Aθ longer than ξ′. By substituting in ξ″, every fresh constant introduced by θ with the variable it replaced, we get a prefix of a LD-derivation for P and A that does not instantiate any variable of A and whose length is greater than that of ξ. Summarizing, C is a missing answer and d_Q ≥ d_A > d_C. Therefore we can apply the inductive hypothesis on C to obtain the conclusion of the Theorem.

Finally, we have termination of the debugger.

Theorem 6.2.17 (S-Termination) Let P be an acceptable program, and Q a query such that there are finitely many oracle’s answers for Q. Then MISS-S(P) and missing_answers_S([Q], Goal) universally left terminate.

Observe that the hypothesis that there are finitely many oracle’s answers is rather restrictive in the case of S-semantics. Looking at program MISS-S, we quickly realize that clauses (iii, v) call valid_S(B, C) with B instantiated by the body of some program clause. In the case of a simple program such as APPEND, the resulting query valid_S([append(Xs, Ys, Zs)], C) has infinitely many LD-derivations. In general, we have that “finitely many oracle’s answers” actually requires that I is a finite set. However, by noting that variants(A, A1) must succeed, a weaker assumption can be made by defining an oracle valid_S(B, C, A, A1) that stops a LD-derivation if A1 becomes more instantiated than A (or some sufficient condition that implies that, e.g. by checking that the size of A1 remains lower or equal than the size of A). By such an enhanced oracle, we have that the assumption of “finitely many oracle’s answers” is less restrictive, and allows for reasoning on intended interpretations that are infinite sets. As an example, starting from the missing answer append([X], Ys, [X|Ys]) for the program of Example 6.2.5, the debugger finds out that it is an uncovered atom, when valid_S is as described above.
6.2.5 A Completeness Result

Acceptable programs

The following result is an immediate consequence of Theorems 6.2.11 and 6.2.16.

**Theorem 6.2.18** Let $P$ be an acceptable program. If there is a missing answer $Q$ w.r.t. $C$-semantics (resp., $S$-semantics) and there are finitely many oracle’s answers for $Q$ then there exists an uncovered atom.

A non-constructive proof has been established by Comini et al. [50] also in the case that there are not finitely many oracle’s answers. Indeed, the proofs of Theorems 6.2.11 and 6.2.16 (non-constructively) show that result if we remove the hypothesis that there are finitely many oracle’s answers.

Fair-bounded programs

The existence of an uncovered atom extends to fair-bounded programs as well. Next, we show the case of $C$-semantics.

**Theorem 6.2.19** Let $P$ be a fair-bounded program. If there is a missing answer $Q$ w.r.t. $C$-semantics then there exists an uncovered atom.

**Proof.** Let $\mathcal{I}$ be the intended interpretation of $P$. Consider now a maximal sequence of queries $\xi = Q_1, \ldots, Q_k, \ldots$ such that:

- $Q_1 = Q$, and
- for $i \geq 1$, let $Q_i = A, Q'$ and $A \leftarrow B_1, \ldots, B_n$ an instance of a clause of $P$ such that $B_1, \ldots, B_n \in \mathcal{I}$. Then $Q_{i+1} = Q', B_1, \ldots, B_n$.

We observe that $\xi$ is an instance of a (prefix of a) SLD-derivation of $P$ and $Q$. Since $Q$ is a missing answer and $\xi$ does not instantiate variables of $Q$, then $\xi$ cannot end with an empty query. Otherwise, $Q$ would be a correct instance of $P$ and $Q$, which implies $Q$ in $\mathcal{C}(P)$.

Moreover, called $\theta$ the substitution that maps every variable in $Q$ into fresh distinct constants, $\xi\theta$ has the same length as a prefix of a SLD-derivation for $P$ and $Q\theta$ via the round-robin selection rule. Since $P$ is fair-bounded, $\xi\theta$ is finite, and a fortiori $\xi$ is finite.

Let $Q_k$ be the last (non-empty) query in $\xi$, and $A$ the leftmost atom in $Q_k$. By the definition of $\xi$, $A \in \mathcal{I}$. Moreover, for every instance $A \leftarrow B_1, \ldots, B_n$ of a clause from $P$ we have that some $B_i$ is not in $\mathcal{I}$ — otherwise $Q_k$ would not be the last query.

Therefore, $A$ is uncovered.

We conclude by observing that the proof can be coded into a meta-program which is a variant of program MISS-$C$. 


Bounded programs

The approach presented in this section cannot be extended to bounded programs. In fact, Theorem 6.2.18 does not hold for bounded program, in general.

Example 6.2.20 The query p is a missing answer for the following (bounded) program, but there is no uncovered atom.

\[
\begin{align*}
p & \leftarrow p \\
\% p & \% missing
\end{align*}
\]

The absence of uncovered atoms (e.g., w.r.t. C-semantics) can be concisely written as \( T \subseteq T^c(I) \). However, this only implies \( T \subseteq gfp(T^c(I)) \), while nothing can be said about the relation between \( T \) and \( T^c \uparrow \omega = \mathcal{C}(P) \). When \( P \) is acceptable or fair-bounded, the \( T^c \) operator can be shown (see Related Work) to have a unique fixpoint, and then \( gfp(T^c) = T^c \uparrow \omega \). In this case, absence of uncovered atoms implies \( T \subseteq T^c \uparrow \omega \), i.e. absence of missing answers.

A possible way that could be followed in the case of programs bounded but not acceptable or fair-bounded is to reason on the transformed program \( Ter(P) \) (see Definition 2.5.20) rather than on \( P \).

6.3 Related Work

Semantics decidability

Logic programs are computationally complete, in the sense that they have the same computational power as recursive functions. Andréka and Németi [1] showed that for every recursive function \( f \) there is a program such that its least Herbrand model restricted to atoms with a fixed relation symbol \( p \) defines the graph of \( f \).

Börger [27] and Plümer [135] showed a universal program consisting only of binary and unit clauses without local variables and containing a single predicate.

The class of programs consisting of binary and unit clauses seems then to be the boundary between decidability and undecidability of the least Herbrand model semantics. Consider, in fact, a program with a single binary clause and a single fact:

\[
\begin{align*}
p(S_1, \ldots, S_n) & \leftarrow p(S'_1, \ldots, S'_n) \\
p(T_1, \ldots, T_n)
\end{align*}
\]

and a query \( p(V_1, \ldots, V_n) \). Devienne [70] showed that when the query and the fact are linear, i.e. variables occurs at most once, it is decidable whether the query has a correct answer. This implies that when the fact is linear, then the least Herbrand model of the program is decidable.

Finally, we observe that, while it is undecidable whether a program is acceptable, fair-bounded or bounded, in Section 2.8 we have discussed how sufficient automatic
methods can be adapted to infer acceptability and (fair-)boundedness. The results of this Chapter apply to every program for which those methods are able to infer acceptability or (fair-)boundedness.

Expressive power

Bezem [23] showed that recurrent programs compute every total recursive function. By Theorem 2.7.1, the same property is shared by acceptable, fair bounded and bounded programs.

Testing

Ducassé and Noyé [74, Section 6] reviewed automated debugging and testing approaches. As they point out, logic programming has been mainly used as a basis for generating test cases for other languages. The only works on testing of logic programming languages we are aware of are due to Belli and Jack [20, 92] and Luo et al. [108]. More broadly, the topic of extending testing methodologies from conventional programming to rule-based (expert) systems has been addressed in [100, 143].

Belli and Jack compared testing of imperative and logic programs, by giving formal basis for test case generation and testing methodologies of logic programs. They adopt a program instrumentation technology based on typing and modeling of programs, and a clause coverage-based test case generation. As a result, a testing tool PROTest has been developed. Compared with the main theme of this Chapter, they lack of a terminating test driver, in the sense that PROTest simply calls a Prolog interpreter on test cases, and then it is not guaranteed to terminate. As already mentioned, we have integrated [119] the DECS-A decision procedure within the PROTest system, thus obtaining a tool that is guaranteed to terminate on acceptable programs.

Luo et al. defined a control flow graph of Prolog programs that captures the structural complexity of programs, and develop a fault model for guidance on test case generation in terms of graph coverage.

Summarizing, both the approaches reason about the selection of the finite sets \( I \) and \( S \) such that a program has to be tested with respect to (see Definitions 6.1.2 and 6.1.4). This phase is called test case generation, and is crucial in conventional programming testing methodologies (see [57, 120] for introductory texts). Once \( I \) and \( S \) are generated, the decision procedure is called, and then results are reported. Test report documents may be very poor, reporting only yes or not, or highly informative with statistics, graphics, reliability analysis and estimations, and test coverages.

Without going into further detail, we observe that a meta-programming approach facilitates adding more and more features to make a tool practical. In addition, the meta-programming approach allows us to achieve a smooth integration of tools, including test case generation, test drivers and debuggers.
Declarative debugging

Declarative debugging was introduced in logic programming by Shapiro [144]. A recent survey is by Ferrand and Tessier [81].

The notions of symptoms can be defined in an abstract scheme [49, 79, 126] that reconciles partial correctness verification, testing and debugging.

Let $T : 2^H \rightarrow 2^H$ be a monotonic operator such that $lfp(T)$ defines the semantics of a program $P$, and let $\mathcal{I} \subseteq H$ be the intended semantics of $P$. The relation one expects is $lfp(T) = \mathcal{I}$.

When $\mathcal{I}$ is a finite representation of the intended semantics, program verification is a means to prove that the the relation above is satisfied.

When $\mathcal{I}$ is extensionally or partially specified, program testing is a means to discover incorrectness or incompleteness symptoms, i.e. elements in $lfp(T) \cap H \setminus \mathcal{I}$, or in $H \setminus lfp(T) \cap \mathcal{I}$.

Incorrectness symptoms cannot appear when $T(\mathcal{I}) \subseteq \mathcal{I}$, since this implies $lfp(T) \subseteq \mathcal{I}$. This leads to the notion of wrong clauses.

Dually, if $T$ has a unique fixpoint then incompleteness symptoms cannot appear when $\mathcal{I} \subseteq T(\mathcal{I})$, since this implies $\mathcal{I} \subseteq gfp(T) = lfp(T)$. Comini et al. [50] showed that acceptable programs have unique fixpoint w.r.t. $T^c$ and $T^s$. In the case that $\mathcal{I}$ is a finite set of elements, they propose a bottom-up declarative diagnosis consisting of constructing $T(\mathcal{I})$ and then checking it against $\mathcal{I}$. Also, they propose top-down abstract diagnosers (coded in Prolog) whose observables are wrong clauses and uncovered atoms.

The results presented in this Chapter improve on Shapiro's and Ferrand's proposals for completeness and termination. Moreover, efficiency is improved in the following sense. As shown in Theorem 6.2.9, only calls $\text{miss}([Q], \text{goal})$ where $Q$ is a missing answer are made. On the contrary, from Example 6.2.5, we realize that Shapiro's and Ferrand's debuggers search space include queries that are not missing answers, and then cannot lead to uncovered atoms.

As already mentioned, Comini et al. [50] present a method for finding all uncovered atoms starting from the intended interpretation $\mathcal{I}$ of an acceptable program. However, their approach is effective iff $\mathcal{I}$ is a finite set, and actually they refine the approach to sets $\mathcal{I}$ which are partially specified. On the contrary, we require a weaker condition than finiteness, namely that there are finitely many oracle's answers.

6.4 Conclusion

Starting from the declarative characterizations of terminating programs provided in Chapter 2, we have systematically derived methods for the validation of logic programs.

In the first section of the Chapter, we have recognized that semantics decidability and testing are equivalent problems. We have presented decision procedures for
acceptable, fair-bounded and bounded programs w.r.t. the \( C \)- and \( S \)-semantics, and have provided an implementation for them in the form of Prolog meta-programs. The decision procedures are then recognized to be automatic tools for testing logic programs. The relevance of the results presented lies mainly:

- in the dimensions of the classes of programs under consideration. It is worth mentioning that most of the programs reported in any basic book of programming, such as Sterling and Shapiro [150], belong to those classes. While acceptable programs are closely related to left-to-right selection rules, fair-bounded and bounded programs abstract away from the underlying operational model;
- in the meta-programming approach, which revealed to be successful in modeling extensions of pure logic programming including programs with arithmetic, meta-programs, general logic programs and some other declarative semantics;
- in having recognized that the procedures for observable decidability are practical tools for testing logic programs.

Also, we have discussed some preliminary experimental results and outlined an efficient compilation-oriented approach which overcomes meta-programming overheads.

In the second section of the Chapter, some specializations of the decision procedures have been employed as the basic components for a declarative debugging approach of \textit{missing answers}. We have presented two declarative debuggers of missing answers with respect to \( C \)- and \( S \)-semantics. They are correct for any program, and complete and terminating for acceptable programs.

Finally, we point out that while we reasoned on acceptable programs rather than in the context of the \( \triangleright \) proof relation of Chapter 4, all the results can be rephrased for the \( \triangleright \) relation. For instance, the extension of Lemmas 6.1.26 and 6.1.27 to programs such that \( \triangleright \text{ Pre} \{ P \text{ Post} \) holds is stated in Theorem 4.2.12 at page 135. Also, the decision procedures turn out to be the same meta-programs presented in this Chapter.
Chapter 7

From LP to CLP

The approach followed in this thesis started with the study of declarative characterizations of terminating programs and queries. Verification and validation proof theories were then systematically developed starting from those characterizations.

When trying to apply this approach to one of the many extensions of pure logic programming, the first step consists then of extending the declarative characterizations of Chapter 2. At least in principle, verification and validation proof theories should systematically be derived from those extensions by generalizing the approach followed in the thesis.

In this Chapter, we investigate the extension of the classes of terminating programs of Chapter 2 to a widespread extension of logic programming, namely the family of constraint logic programming (CLP) languages, which merges logic programming with constraint solving. We will show that for a class of languages, namely ideal CLP languages, the definitions of acceptability and boundedness extend in a natural and intuitive way. In particular, Prolog implementations that omit the occur check in the unification algorithm can be seen as a particular ideal CLP language.

Unfortunately, implementations of CLP languages are not complete in general, in the sense that the constraint solver is not complete with respect to the declarative semantics of the constraint domain. Since the characterizations of acceptable, fair-bounded and bounded programs and queries are tightly related to declarative semantics, the natural extensions of those characterizations to non-ideal CLP languages may not result in sound termination proof methods, in general.

We study this issues in the specific case of the CLP(\$R \$) system, which merges logic programming executed via the leftmost selection rule with constraint solving over real numbers.
7.1 Constraint Logic Programs

The Constraint Logic Programming (CLP) Scheme was introduced by Jaffar and Lassez [94]. It merges logic programming with constraint solving, by generalizing the term equations of logic programming to constraints from any pre-defined computation domain. The CLP Scheme defines a family of languages, CLP(\mathcal{C}), which are parametric in the constraint domain \mathcal{C}, and which share rich operational, denotational and logical semantic theories.

The survey of Jaffar and Maher [96] provides a comprehensive introduction to motivations, foundations, and applications of CLP languages. A formal presentation of the semantics can be found in a recent work of Jaffar et al. [95]. We substantially adhere to this paper, unless otherwise specified.

Constraint Domains

A constraint domain \mathcal{C} is a tuple (\Sigma_{\mathcal{C}}, \mathcal{L}_{\mathcal{C}}, \mathcal{D}_{\mathcal{C}}, \mathcal{T}_{\mathcal{C}}, \text{solv}_{\mathcal{C}}).

\Sigma_{\mathcal{C}} is the constraint domain signature. The class of constraints, \mathcal{L}_{\mathcal{C}}, is a set of first-order \Sigma_{\mathcal{C}} formulas. We denote constraints by \textit{c}, \textit{d}, etc. The domain of computation, \mathcal{D}_{\mathcal{C}}, is a \Sigma_{\mathcal{C}}-structure that is the intended interpretation of the constraints. \mathcal{D}_{\mathcal{C}} is the domain (or support) of \mathcal{D}_{\mathcal{C}}. The constraint theory, \mathcal{T}_{\mathcal{C}}, is a \Sigma_{\mathcal{C}} theory describing the logical semantics of the constraints. The constraint solver, \text{solv}_{\mathcal{C}}, is a (computable) function which maps each formula in \mathcal{L}_{\mathcal{C}} to one of true, false, unknown, indicating if the formula is satisfiable, unsatisfiable or it cannot tell.

We assume that the predicate symbol "=" is in \Sigma_{\mathcal{C}} and that it is interpreted as identity in \mathcal{D}_{\mathcal{C}}. Also, we assume that \mathcal{L}_{\mathcal{C}} contains all atoms constructed from "=": the always satisfiable constraint true and the unsatisfiable constraint false, and that \mathcal{L}_{\mathcal{C}} is closed under variable renaming, existential quantification and conjunction. A primitive constraint is an atom of the form \textit{p}(T_{1}, \ldots, T_{n}) where \textit{p} is a predicate in \Sigma_{\mathcal{C}}. The smallest set of constraints which satisfies the above assumptions and contains all primitive constraints is called the set of constraints generated by primitive constraints.

We assume that the solver does not take variable names into account. Also, the domain, the theory and the solver agree in the sense that \mathcal{D}_{\mathcal{C}} is a model of \mathcal{T}_{\mathcal{C}} and for every \textit{c} \in \mathcal{L}_{\mathcal{C}}:

- \text{solv}_{\mathcal{C}}(\text{true}) implies \mathcal{T}_{\mathcal{C}} \models \exists c, and
- \text{solv}_{\mathcal{C}}(\text{false}) implies \mathcal{T}_{\mathcal{C}} \models \neg \exists c.

With these assumptions a solver is allowed to be incomplete, i.e. to return unknown. On the contrary, a solver is called complete if it only ever returns true or false. A constraint domain whose solver is complete is called ideal.

Example 7.1.1 (Real) The constraint domain Real has \textless, \leq, \textasciitilde, \geq, \textgreater as relation symbols; *, -, \ast, / as function symbols; and sequences of digits (possibly with a deci-
7.1. Constraint Logic Programs

The domain of computation is the structure with reals as domain, and where the predicate symbols $<$, $\leq$, $=$, $\geq$, $>$ and the function symbols $\ast$, $\div$, $+$, $-$, $/$ are interpreted as the usual relations and functions over reals. Finally, the theory $T_{\text{Real}}$ is the theory of real closed fields.

A possible constraint solver is provided by the CLP($\mathcal{R}$) system [97], which relies on Gauss-Jordan elimination to handle linear constraints. Non-linear constraints are not taken into account by the solver (i.e., their evaluation is delayed) until they become linear. Thus, the domain of the CLP($\mathcal{R}$) system is not ideal.

A complete solver is proposed in the RISC-CLP(Real) system [91]. Non-linear constraints are handled by means of two algebraic methods: 
Partial Cylindrical Algebraic Decomposition and Gröbner basis. In addition, support for exact arithmetic (vs floating point) is provided.

Example 7.1.2 (Logic Programming) The constraint domain $\text{Term}$ has $=$ as relation symbols and strings of alphanumeric characters as function or constant symbols. The domain of computation of $\text{Term}$ is the set of finite trees (or, equivalently, of finite terms), $\text{Tree}$, while the theory $T_{\text{Term}}$ is Clark’s equality theory.

The interpretation of a constant is a tree with a single node labeled with the constant. The interpretation of a $n$-ary function symbol $f$ is the function $f_{\text{Tree}} : \text{Tree}^{n} \rightarrow \text{Tree}$ mapping the trees $T_{1}, \ldots, T_{n}$ to a new tree with root labeled with $f$ and with $T_{1}, \ldots, T_{n}$ as children.

A complete constraint solver is provided by the unification algorithm. CLP($\text{Term}$) coincides then with logic programming.

Example 7.1.3 (Logic Programming Without Occur-Check) Consider now a constraint domain $\mathcal{R}\text{Term}$ constructed as $\text{Term}$ but with rational trees (see [52, 44]) (or, equivalently, with rational terms) as the domain of computation. Rational trees are (possibly infinite) trees which have a finite set of subtrees.

It has been pointed out by Colmerauer [44, 45] that unification which omits the occur check solves equations over the rational trees. Therefore, CLP($\mathcal{R}\text{Term}$) coincides with logic programming without occur check.

Also, Jaffar and Maher [96] point out that the domain $\mathcal{R}\text{Term}$ is essentially the same as the domain of infinite trees, in the sense that the two structures are elementarily equivalent.

CLP Programs and Operational Semantics

We recall below the operational semantics of CLP languages, which is parametric to constraint domains. Therefore a CLP language is determined by its constraint domain. For a particular constraint domain $\mathcal{C}$, we denote by CLP($\mathcal{C}$) the CLP language based on $\mathcal{C}$. We say that CLP($\mathcal{C}$) is ideal if $\mathcal{C}$ is ideal.
As in the case of logic programs, our results are parametric to a language \( L \) in which all programs and queries under consideration are included. \( \Sigma_L \) coincides with the set of function symbols of \( \mathcal{L}_c \), while \( \Pi_L \) includes the predicate symbols of \( \mathcal{L}_c \).

A constraint logic program, or program, is a finite set of rules of the form:

\[
A \leftarrow B_1, \ldots, B_n.
\]

where \( A \) is an atom, called the head, and \( B_1, \ldots, B_n \) \((n \geq 0)\) are literals. A literal is either an atom (whose predicate symbol is in \( L \) but not in \( \Sigma_c \)) or a primitive constraint. If every literal in \( B_1, \ldots, B_n \) is a constraint, then the clause is called a fact. A query is a sequence of literals. The empty sequence of literals is denoted by \( \Box \).

The operational semantics is given in terms of derivations from states to states, in the logic programming style. A state is a pair \( \langle A_1, \ldots, A_n \mid c \rangle \) where \( A_1, \ldots, A_n \) is a query and \( c \) is a constraint, called the constraint store.

A state \( \langle A_1, \ldots, A_n \mid c \rangle \) is reduced to another state, called resolvent, as follows. Select a literal \( A_i \):

1. If \( A_i \) is a primitive constraint and \( \text{solv}_c(c \land A_i) \neq \text{false} \), the state is reduced to \( \langle A_1, \ldots, A_{i-1}, A_i, A_{i+1}, \ldots, A_n \mid c \land A \rangle \).
2. If \( A_i \) is a primitive constraint and \( \text{solv}_c(c \land A_i) = \text{false} \), the state is reduced to \( \langle \Box \mid \text{false} \rangle \).
3. If \( A_i \) is an atom of the form \( p(T_1, \ldots, T_h) \) and

\[
p(S_1, \ldots, S_h) \leftarrow B_1, \ldots, B_k.
\]

is a renamed apart clause from \( P \), and \( \text{solv}_c(c \land \bigwedge_{i=1}^{h} T_i = S_i) \neq \text{false} \), then the state is reduced\(^1\) to

\[
\langle A_1, \ldots, A_{i-1}, B_1, \ldots, B_k, A_{i+1}, \ldots, A_n \mid c \land \bigwedge_{i=1}^{h} T_i = S_i \rangle.
\]

4. If \( A_i \) is an atom of the form \( p(T_1, \ldots, T_h) \) and there is no clause of the form

\[
p(S_1, \ldots, S_h) \leftarrow B_1, \ldots, B_k \text{ in } P
\]

then the state is reduced to \( \langle \Box \mid \text{false} \rangle \).

We point out that rule 4 is irrelevant from the point of view of termination.

A derivation from a state \( S \) is a (finite or infinite) maximal sequence of states \( S_0 = S, S_1, \ldots, S_n, \ldots \) such that there is a reduction from \( S_i \) to \( S_{i+1} \), for \( i \geq 0 \). A derivation for a query \( Q \) is a derivation from the state \( \langle Q \mid \text{true} \rangle \).

\(^1\)This reduction rule is from [96]: In [95], instead, \( A_i \) and the clause above are reduced to

\[
\langle A_1, \ldots, A_{i-1}, T_1 = S_1, \ldots, T_h = S_h, B_1, \ldots, B_k, A_{i+1}, \ldots, A_n \mid c \rangle.
\]

While the two forms are equivalent from the point of view of the semantics of success, we observe that the one from [95] does not conservatively extend pure logic programming. In fact, the one-clause program \( p(a) \leftarrow p(b) \) and the query \( p(a) \) have only a failed SLD-derivation. Considered as a CLP(Term) program and query, and assuming the rule of [95], they have an infinite SLD-derivation via the rightmost selection rule.
The last state of a derivation is of the form \( \langle \square \| c \rangle \). If \( c = \text{false} \) the derivation is failed. Otherwise, it is successful, or a refutation. For a query \( Q \), and a refutation from \( \langle Q \| \text{true} \rangle \) to \( \langle \square \| c \rangle \), we call answer constraint the existential closure of \( c \) w.r.t. the variables of \( c \) which are not in \( Q \). Selection rules, trees, finitely failed trees, etc. are natural extensions of logic programming concepts. For a large class of solvers, called well-behaved solvers, Jaffar et al. showed that the property of independence from the selection rule. In particular, complete solvers are well-behaved.

**Example 7.1.4 (Prolog)** In many implementations of Prolog the occur check is omitted by default. Therefore, to properly reason on those Prolog programs, one should see them as \( \text{CLP}(R \text{Term}) \) programs executed via the leftmost selection rule rather than logic programs.

The only difference between Prolog and \( \text{CLP}(R \text{Terms}) \) consists of the fact that Prolog's computed answers are displayed as substitutions rather than as equations. This may cause Prolog to run into an infinite loop when displaying rational trees explicitly. Some successors of Prolog, such as Prolog II and Prolog III \([46]\), adopt the implicit representation of computed answers as set of equations.

### Interpretaions and Fixpoints

A \( \mathcal{C} \)-interpretation for a \( \text{CLP}(\mathcal{C}) \) program is an interpretation which agrees with \( \mathcal{D}_\mathcal{C} \) on the interpretations of the symbols in \( \mathcal{L}_\mathcal{C} \). Since the meaning of primitive constraints is fixed by \( \mathcal{C} \), we represent a \( \mathcal{C} \)-interpretation \( I \) by a subset of the \( \mathcal{C} \)-base of \( P \), written \( \mathcal{C} \text{-base}_{\mathcal{L}} \), which is the set:

\[
\{ p(d_1, \ldots, d_n) \mid p \text{ predicate in } L \setminus \Sigma_\mathcal{C}, d_1, \ldots, d_n \in \mathcal{D}_\mathcal{C} \}.
\]

The definitions of \( \mathcal{C} \)-models, and least \( \mathcal{C} \)-model \( \text{lm}(P, \mathcal{C}) \) are natural extensions of the logic programming concepts.

A valuation \( \sigma \) is a function that maps variables into \( \mathcal{D}_\mathcal{C} \). For an interpretation \( I \) and a formula \( \phi \), we write \( I \models_\sigma \phi \) iff \( \phi \) is valid in \( I \) w.r.t. the valuation \( \sigma \). We write \( I \models \phi \) iff \( \phi \) is valid in \( I \) (w.r.t. every valuation). We write \( P, T_\mathcal{C} \models \phi \) iff for every interpretation \( I \) that is a model of \( P \cup T_\mathcal{C} \), we have \( I \models \phi \).

A \( \mathcal{C} \)-ground instance \( A' \) of a literal \( A \) is obtained by applying a valuation \( \sigma \) to the literal, thus producing a construct of the form \( p(a_1, \ldots, a_n) \) with \( a_1, \ldots, a_n \) elements from \( \mathcal{D}_\mathcal{C} \). Similarly \( \mathcal{C} \)-ground instances are defined for queries and clauses.

For an atom \( A, [A]_{\mathcal{L}}^\mathcal{C} \) is the set of \( \mathcal{C} \)-ground instances of \( A \). With a small abuse of notation, we write \( I \models p(a_1, \ldots, a_n) \) when \( I \models_\sigma A \). We denote by \( \text{ground}_\mathcal{C}(P) \) (resp., \( \text{ground}_\mathcal{C}(Q) \)) the set of \( \mathcal{C} \)-ground instances of clauses from \( P \) (resp., the set of \( \mathcal{C} \)-ground instances of \( Q \)).

The following theorems summarize the relation between the declarative and operational semantics.
Theorem 7.1.5 (Correctness) Let $P$ and $Q$ be a CLP($C$) program and query. For every answer constraint $c$ for them, we have:

- $P, T_C \models c \rightarrow Q$,
- $lm(P, C) \models c \rightarrow Q$

\[ \square \]

Theorem 7.1.6 (Completeness) Let $P$ and $Q$ be a CLP($C$) program and query. If $lm(P, C) \models_\sigma Q$ for a valuation $\sigma$, then $Q$ has an answer constraint $c$ such that $T_C \models_\sigma c$.

Moreover, if $P, T_C \models c \rightarrow Q$ for a constraint $c$ then there exist answer constraints $c_1, \ldots, c_n$ such that $T_C \models c \rightarrow (c_1 \lor \ldots \lor c_n)$.

A constraint $c$ such that $P, T_C \models c \rightarrow Q$ is called a correct constraint of $P$ and $Q$. Correct constraints are the CLP version of correct answer substitutions of logic programming.

Miscellaneous

A flat program is a program in which every clause head and body atom has the form $p(X_1, \ldots, X_n)$, where $X_1, \ldots, X_n$ are (not necessarily distinct) variables.

A norm is a function from the domain of $D_C$ to $N^\infty$. Norms are usually defined on the structure of elements in the domain. Since for some domains, e.g. $RTerm$, the structure of elements can be infinite, we now consider $N^\infty$ as codomain of norms instead of $N$.

Example 7.1.7 The list-length norm on rational trees $| | : D_{RTerm} \rightarrow N^\infty$ is defined as follows:

\[
|\mathbf{f}(t_1, \ldots, t_n)| = 0 \quad \text{if } \mathbf{f} \neq [\ldots],
\]

\[
|t| = \infty \quad \text{if } t = [t_1, \ldots, t_n \mid t]
\]

\[
|[x \mid t]| = |t| + 1 \quad \text{otherwise}
\]

where for convenience $\infty + 1$ is set to $\infty$. Obviously, the list-length of a tree $[t_1, \ldots, t_n]$, i.e. a list, is $n$.

Similarly, we extend the size norm, by defining:

\[
size(t) = \infty
\]

if $t$ is not a finite tree

\[
size(f(t_1, \ldots, t_n)) = 1 + size(t_1) + \ldots, size(t_n)
\]

if $f(t_1, \ldots, t_n)$ is a finite tree

\[
size(a) = 0
\]

if $a$ is a leaf.

\[ \square \]
7.2 From LP to Ideal CLP

The generalization of the definitions of universal termination, left termination, \( \exists \)-universal termination to CLP(\( \mathcal{C} \)) programs and queries are immediate. We report below the one of universal termination.

**Definition 7.2.1** A CLP(\( \mathcal{C} \)) program \( P \) and query \( Q \) universally terminate w.r.t. a set of selection rules \( S \) if every derivation of \( P \) and \( Q \) via any selection rule from \( S \) is finite.

In this Section, we generalize acceptability and boundedness to CLP programs. We show that for the class of ideal CLP languages, these (immediate) extensions turn out to be sound and complete proof methods w.r.t left termination and bounded non-determinism respectively. Due to space limitations, we will not discuss the extension of fair-boundedness.

7.2.1 Acceptable CLP programs

Intuitively, a generalization of acceptability to CLP(\( \mathcal{C} \)) has to consider \( \mathcal{C} \)-ground objects, in order to involve the domain to the proof level.

**Example 7.2.2** (Member) The program **MEMBER**:

\[
\begin{align*}
\text{member}(X, [X| Xs]). \\
\text{member}(X, [Y| Xs]) & \leftarrow \\
\text{member}(X, Xs).
\end{align*}
\]

and the query \( Xs = [a| Xs], \text{member}(b, Xs) \) can be seen as a program and a query both in the CLP(\( \text{Term} \)) language, alias logic programming, and in the CLP(\( \text{RTerm} \)) language, alias Prolog without occur check. However, in the former language they left terminate, while in the latter they do not.

The definitions of extended level mappings and acceptability readily lift to CLP(\( \mathcal{C} \)) programs by replacing ground objects (i.e., atoms, clauses) with \( \mathcal{C} \)-ground objects.

**Definition 7.2.3** An extended level mapping for a constraint domain \( \mathcal{C} \) is a function \( \| \): \( \mathcal{C} - \text{base} \rightarrow N^\infty \) of \( \mathcal{C} \)-ground atoms to \( N^\infty \).

For a \( \mathcal{C} \)-ground atom \( A \), \( \|A\| \) is called the level of \( A \).

**Definition 7.2.4** Let \( \| \| \) be an extended level mapping for \( \mathcal{C} \), and \( I \) a \( \mathcal{C} \)-interpretation.

- A CLP(\( \mathcal{C} \)) program \( P \) is acceptable by \( \| \| \) and \( I \) iff \( I \) is a \( \mathcal{C} \)-model of \( P \), and for every \( A \leftarrow B_1, \ldots, B_n \) in \( \text{ground}_\mathcal{C}(P) \):
for $i \in [1, n]$, if $B_i$ is an atom then

$$I \models B_1, \ldots, B_{i-1} \implies |A| \supset |B_i|.$$ 

- A state $(Q|c)$ is acceptable by $|$ $|$ and $I$ iff there exists $k \in N$ such that for every $A_1, \ldots, A_n \in \text{ground}_c(c, Q)$:
  for $i \in [1, n]$, if $A_i$ is an atom then
  $$I \models A_1, \ldots, A_{i-1} \implies k \supset |A_i|.$$ 

- A query $Q$ is acceptable by $|$ $|$ and $I$ iff there exists $k \in N$ such that for every $A_1, \ldots, A_n \in \text{ground}_c(Q)$:
  for $i \in [1, n]$, if $A_i$ is an atom then
  $$I \models A_1, \ldots, A_{i-1} \implies k \supset |A_i|. \quad \square$$

In logic programming a state in a SLD-derivation can be represented as a query, thus the definitions of acceptability for states and queries coincide. In CLP, instead, a query $Q$ can be viewed as an abbreviation for the state $(Q||true)$.

**Example 7.2.5 (CURRY)** Let us consider the CLP($RTerm$) program **CURRY** from Reddy [1.36], which implements the rules of a simple Curry’s type system.

```prolog
\begin{align*}
\text{type}(E, \text{var}(X), T) & \leftarrow \\
\text{in}(E, X, T).
\end{align*}
\begin{align*}
\text{type}(E, \text{apply}(M, N), T) & \leftarrow \\
\text{type}(E, M, \text{arrow}(S, T)), \\
\text{type}(E, N, S).
\end{align*}
\begin{align*}
\text{type}(E, \text{lambda}(X, M), \text{arrow}(S, T)) & \leftarrow \\
\text{type}([[X, S]|E], M, T).
\end{align*}
\begin{align*}
\text{in}([[X, T]|E], X, T).
\end{align*}
\begin{align*}
\text{in}([[Y, T1]|E], X, T) & \leftarrow \\
X \neq Y, \\
\text{in}(E, X, T).
\end{align*}
\begin{align*}
a \neq b. \quad \text{for distinct} \ a, b \in \text{Var}.
\end{align*}
```

Intuitively, lambda terms are represented by the grammar:

\[
M ::= \text{var}(\text{Var}) \mid \text{apply}(M, M) \mid \text{lambda}(\text{Var}, M)
\]

where $\text{Var}$ is a set of constants denoting lambda variables. $E$ is a list representing an environment, i.e. pairs $[x, T]$ where $T$ is a type. A type is defined by the grammar:

\[
T ::= B T \mid X \mid \text{arrow}(T, T)
\]
where $BT$ is a set of basic types, denoted by constants, and $X$ is the set of logic variables.

As an example, $\text{arrow}(\text{int}, \text{int})$ is the type of a function from a set called int into itself. On the other hand, $\text{arrow}(\text{Alpha}, \text{Alpha})$ is the polymorphic type of a function from any set into itself.

Since the elements of the domain are rational trees, recursive polymorphic types are allowed. Recursive types can be represented by sequences of term equations (i.e., answer constraints) such as $\text{Alpha} = \text{arrow}(\text{Alpha}, \text{Beta})$. In general:

$$RT ::= X = T | RT, RT$$

A query $\text{type}(E, M, T)$ is intended to return an answer constraint binding $T$ to the type of the term $M$ in the environment $E$. For instance, the query:

$$\text{type}([], \lambda \text{x} . \text{var}(\text{x})), T),$$

compute the type of the lambda term $\lambda x.x$, represented by the term:

$$\lambda \text{x} . \text{var}(\text{x})$$

The answer constraint is $T = \text{arrow}(\text{Alpha}, \text{Alpha})$. As another example, the query:

$$\text{type}([], \lambda \text{x} . \text{apply}(\text{var}(\text{x}), \text{var}(\text{x}))), T).$$

computes the type of the self-application term $\lambda x.x \ x$, which turns out to be

$$T = \text{arrow}(\text{Alpha}, \text{Beta}), \text{Alpha} = \text{arrow}(\text{Alpha}, \text{Beta}).$$

$\text{CURRY}$ and the queries above are acceptable by $||$ and $RTerm - base_L$, where

$$|\text{type}(e, m, t)| = |e| + 2 \cdot \text{size}(m)$$

$$|\text{in}(e, x, t)| = |e|$$

$$|x \neq y| = 0$$

As will be shown, acceptability implies left termination of the queries above. On the contrary, the query $\text{type}([], \lambda \text{x} \text{m}, T)$ is not acceptable by the same $||$ and $RTerm - base_L$, since there are rational trees of the form $\lambda \text{x} \text{m}$ which are infinite. Therefore, their size is $\infty$. Notice that in this case $\text{CURRY}$ and the query above do not left terminate.

The extension of acceptability to CLP languages maintains the property of persistency, also for non-ideal languages.

**Lemma 7.2.6 (Persistency)** Let $P$ be a CLP($\mathcal{C}$) program and $S$ a state both acceptable by $||$ and $I$.

Every resolvent $S'$ of $P$ and $S$ is acceptable by $||$ and $I$.
Left termination of acceptable programs and queries can be shown only for ideal CLP languages.

**Theorem 7.2.7 (Termination Soundness)** Let CLP(\mathcal{C}) an ideal CLP language. Let \( P \) be a CLP(\mathcal{C}) program and \( Q \) a query both acceptable by \( \mid \mid \) and \( I \). Then every derivation for \( P \) and \( Q \) via the leftmost selection rule is finite.

**Example 7.2.8 (Curry Ctd)** Consider the program CURRY and the query:

\[
\text{type([],lambda(x,\text{var}(x))),T)}
\]

We have seen that they are both acceptable by the same level mapping and RTerm-interpretation. Therefore, they left terminate, i.e. Prolog without occur check terminates.

Focusing on termination completeness, we observe that it holds also for non-ideal languages. Intuitively, complete constraint solvers cut more derivations than incomplete ones, since they find failure as soon as the constraint store is unsatisfiable. Therefore, termination using incomplete solvers implies termination using complete ones, from which acceptability can be shown by reasoning as in Chapter 2.

**Theorem 7.2.9 (Termination Completeness)** Let \( P \) be a CLP(\mathcal{C}) program and \( S \) a state such that every derivation for \( P \) and \( S \) via the leftmost selection rule is finite. Then there exist \( \mid \mid \) and \( I \) such that \( P \) and \( S \) are both acceptable by \( \mid \mid \) and \( I \).

**Example 7.2.10 (Reasoning on Prolog without occur check programs)** Since the notions and the results on acceptability extend to the CLP(RTerm) language adopting the leftmost selection rule, we can readily apply the approach of Chapter 3 in order to derive a verification proof method for Prolog without occur check.

Actually, the resulting method turns out to be defined by the \( \vdash_t \) proof relation studied in Chapter 4, where pre- and post-conditions are RTerm-interpretations, rather than Herbrand interpretations. All the declarative properties of \( \vdash_t \) can be readily stated in the CLP(RTerm) framework.

**Example 7.2.11 (Validation does not lift)** The results on the characterization of computed instances (see Section 4.1.6), testing and debugging, instead, do not lift to ideal CLP languages, in general.

Maher [109] observes that the characterization of computed answers as minimal correct constrains (see Theorem 4.1.31) extends to CLP languages for which the Completeness Theorem 7.1.6 holds for \( n = 1 \) (e.g., CLP(Term), CLP(RTerm)).

On the contrary, the characterization does not extend to ideal languages in general. Consider, in fact, the RISC-CLP(Real) program:
and the query \( p(X) \). The least \( \text{Real} \) model of the program is \([p(X)]^\text{Real}_L\). Moreover, the program clauses are non-overlapping [109], in the sense that they return distinct solutions. However, the minimal correct constraints of \( p(X) \) in the least \( \text{Real} \) model are:

\[
\text{Min} \{ c \mid [p(X)]^\text{Real}_L \models c \Rightarrow p(X) \} = \{ \text{true} \},
\]

where \( \text{Min} \mathcal{A} \) is the set of constraints (modulo renaming) in \( \mathcal{A} \) which are not strictly implied by other constraints in \( \mathcal{A} \). On the other side, the answer constraints of the query above are \( X > 1, X = 1 \) and \( X > 1 \).

Also, Mesnard [118] points out that our approach to testing logic programs is not always applicable to CLP languages, since the set of constants in CLP languages is fixed once for all. In the case of \( \text{CLP(Term)} \) or \( \text{CLP(RTerm)} \) the solver is parametric to the set of function symbols. However, this is not the case for \( \text{CLP(Real)} \) for instance. Therefore, “adding more constants” means switching to another CLP language.

\[\square\]

### 7.2.2 Bounded CLP programs

Bounded programs were introduced in Section 2.5 as a declarative characterization of bounded nondeterminism. Here, we extend their definitions to CLP languages.

**Definition 7.2.12** Let \( || \) be an extended level mapping for \( C \), and \( I \) a \( C \)-interpretation.

- A \( \text{CLP}(C) \) program \( P \) is **bounded by \( || \)** and \( I \) iff \( I \) is a \( C \)-model of \( P \), and for every \( A \leftarrow B_1, \ldots, B_n \) in \( \text{ground}_C(P) \):
  \( I \models B_1, \ldots, B_n \) implies
  \quad for \( i \in [1, n] \), if \( B_i \) is an atom then \( |A| \triangleright |B_i| \).

- A state \( \langle Q||c \rangle \) is **bounded by \( || \)** and \( I \) iff there exists \( k \in N \) such that for every \( A_1, \ldots, A_n \in \text{ground}_C(c,Q) \):
  \( I \models A_1, \ldots, A_n \) implies
  \quad for \( i \in [1, n] \), if \( A_i \) is an atom then \( k \triangleright |A_i| \).

- A query \( Q \) is **bounded by \( || \)** and \( I \) iff there exists \( k \in N \) such that for every \( A_1, \ldots, A_n \in \text{ground}_C(Q) \):
  \( I \models A_1, \ldots, A_n \) implies
  \quad for \( i \in [1, n] \), if \( A_i \) is an atom then \( k \triangleright |A_i| \).
It turns out that the definition of boundedness is persistent along derivations.

**Lemma 7.2.13 (Persistency)** Let \( P \) be a CLP(\( \mathcal{C} \)) program and \( S \) a state both bounded by \( || \) and \( I \).

Every resolvent \( S' \) of \( P \) and \( S \) is bounded by \( || \) and \( I \). □

Moreover, boundedness is a sound and complete characterization of bounded non-determinism restricted to consider satisfiable answer constraints.

**Theorem 7.2.14 (Soundness)** Let \( P \) be a CLP(\( \mathcal{C} \)) program and \( Q \) a query both bounded by \( || \) and \( I \). Then for every selection rule \( s \) there are finitely many refutations for \( P \) and \( Q \) via \( s \) with satisfiable answer constraints. □

**Example 7.2.15** Consider the CLP(\( \mathcal{R} \)) program:

\[
\begin{align*}
p(x), \\
p(x) \leftarrow x \ast x = -1, \ p(x).
\end{align*}
\]

It is readily checked that it is bounded by \( || \) and any \( \text{Real} \)-model of it, where \(|p(x)| = 0 \) for every real \( x \). Therefore, there are finitely many refutations for it and the query \( p(x) \) such that the answer constraints are satisfiable. □

**Theorem 7.2.16 (Termination Completeness)** Let \( P \) be a CLP(\( \mathcal{C} \)) program, \( s \) a selection rule and \( S \) a state such that there are finitely many refutations for \( P \) and \( S \) via \( s \) with satisfiable answer constraints. Then there exist \( || \) and \( I \) such that \( P \) and \( S \) are both bounded by \( || \) and \( I \). □

Note that these results hold for any CLP language, including non-ideal ones. The reason lies in the fact that incomplete solvers never cut derivations when the constraint store is satisfiable. Incompleteness, in fact, causes to continue a derivation from a state with unsolvable constraint store. Therefore, the number of refutations with satisfiable answer constraints via any selection rule is the same for any solver, either complete or incomplete.

Note, however, that the number of refutations (possibly with unsatisfiable answer constraints) depends on completeness of the solver.

**Example 7.2.17** Consider again the program:

\[
\begin{align*}
p(x), \\
p(x) \leftarrow x \ast x = -1, \ p(x).
\end{align*}
\]

In the CLP(\( \mathcal{R} \)) system there are infinitely many refutations of it and the query \( p(x) \). On the contrary, in the RISC-CLP(\( \text{Real} \)) system there are finitely many. □

Also, the number of refutations may depend on the selection rule. This is shown by Jaffar et al [95, Examples 3.4 and 3.5]. They observe that most real world solvers do not exhibit this pathological behavior, and define the class of well-behaved solvers, for which independence from the selection rule holds.
7.3 From Ideal CLP to CLP(\(\mathbb{R}\))

7.3.1 The incomplete solver of CLP(\(\mathbb{R}\))

Unfortunately, many implementations of CLP languages are not ideal, i.e. their solvers are incomplete. The choice of developing incomplete solvers is motivated by a trade off between declarativeness and efficiency.

Example 7.3.1 (Linear and Non-linear Constraints)
Consider the CLP(\(\text{Real}\)) language. A primitive constraint \(c\) is called linear if it can be written as a linear combination of its variables, i.e. it is of the form
\[
a_0 + a_1 x_1 + \ldots + a_n x_n > 0
\]
and analogously for the predicates \(=\), \(\geq\), \(\leq\), \(<\), where the \(a_i\)'s are constants.

An example of non-linear constraint is the primitive constraint \(X * Y > 0\).

Satisfiability of linear disequations and equations can be checked by using the simplex algorithm and Gauss-Jordan elimination. This is the approach of the CLP(\(\mathbb{R}\)) system [97]. Its constraint solver checks only satisfiability of the linear constraints in the constraint store. Non-linear constraints \(c\) are delayed until they become linear, i.e. the linear constraints imply that some variables in the constraint store assume a unique value in such a way that \(c\) is linear.

Example 7.3.2 Let us see the behavior of the solver of the CLP(\(\mathbb{R}\)) system on different cases.

For the the constraint:
\[
X \geq 2, \ Y < X, \ Y = X*X
\]
satisfiability is checked only for \(X \geq 2, \ Y < X\), which turns out to be satisfiable. The solver returns \(\text{unknown}\), since satisfiability depends on the unseen part of the constraint store.

For the the constraint store:
\[
X > Y, \ Y < X, \ Y = X*X
\]
satisfiability is checked only for \(X > Y, \ Y < X\), which turns out to be unsatisfiable. The solver returns \(\text{false}\), since the constraint is unsatisfiable independently from the rest of the constraint store.

For the the constraint store:
\[
X = 2, \ Y > X, \ Y = X*X
\]
satisfiability is checked for \(X = 2, \ Y > X, \ Y = 4\), which turns out to be unsatisfiable. The solver returns \(\text{true}\).  \(\square\)
On the contrary, solving non-linear equations or disequation is computationally expensive. Thus, CLP(\(\mathbb{R}\)) sacrifices completeness of the solver for efficiency reasons. A different choice is done by the RISC-CLP(\(\mathbb{R}\)) system [91], which checks satisfiability of non-linear constraints and provide support for exact arithmetic.

**Example 7.3.3 (Fact)** Let us consider now the following CLP(\(\mathbb{R}\)) program FACT for computing factorial numbers:

```plaintext
fact(0, 1).
fact(1, 1).
fact(N, N * F) ←
    F >= 1,
    N >= 2,
    fact(N-1, F).
```

The query `fact(4, F)` is intended to compute in \(F\) the 4th factorial number, i.e. 24. Moreover, the same program can be used to check whether a number is factorial, by means of a query such as \(Q = \text{fact}(N, 24)\). We point out that FACT and \(Q\) are both acceptable by \(||\) and \(\text{Real} - \text{base}\), where:

\[
|\text{fact}(n, f)| = \max\{\text{int}(f), 0\}
\]

where \(\text{int}(f)\) is the integer part of a real \(f\). From Definition 7.2.4, the only proof obligation is to show that:

\[
\text{int}(n \cdot f) > \text{int}(f)
\]

when \(f \geq 1, n \geq 2\). This clearly holds.

Every derivation of the program and the query \(Q\) via the leftmost selection rule is finite, if the constraint solver is complete. Termination, in fact, is stated in Theorem 7.2.7.

On the contrary, the computation of the (non-ideal) CLP(\(\mathbb{R}\)) system returns the answer constraint \(N = 4\) and then diverges. In fact, the system runs into an infinite loop by applying the third clause again and again. The constraints along the infinite derivation are:

\[
\begin{align*}
24 &= N \times F \\
24 &= N \times F, F >= 1, N >= 2 \\
24 &= N \times F, F >= 1, N >= 2, F = N1 \times F1, N1 = N - 1 \\
24 &= N \times F, F >= 1, N >= 2, F = N1 \times F1, N1 = N - 1, F1 >= 1, N1 >= 2 \\
\ldots
\end{align*}
\]

We observe that the first argument of a call to \(\text{fact}\) decreases by 1 at each call. This and \(24 = N \times F, F >= 1\) imply that the first argument must be necessarily bounded by 24. After 23 calls, the first argument is constrained to be at least 2. This implies that the argument of the top level call must be at least 25, which is not possible.
Unfortunately, proving unsatisfiability in this way means reasoning on non-linear constraints, such as \(24 = N \times F\). On the contrary, the CLP(\(\mathcal{R}\)) systems has visibility only of the linear constraints:

\[
\begin{align*}
\text{true} \\
F &\geq 1, \ N \geq 2 \\
F &\geq 1, \ N \geq 2, \ M_1 = N - 1 \\
F &\geq 1, \ N \geq 2, \ M_1 = N - 1, \ F_1 \geq 1, \ M_1 \geq 2 \\
\ldots
\end{align*}
\]

which are all satisfiable. \(\square\)

### 7.3.2 Proving Termination of CLP(\(\mathcal{R}\)) programs

**Recurrency**

In general, a sound termination method for non-ideal languages consists of preventing the use of any declarative reading of programs in the definition of acceptability, i.e. by fixing the \(C\)-interpretation \(I\) to \(C = \text{base}_L\).

**Definition 7.3.4** A CLP(\(C\)) program \(P\) is *recurrent* by an extended level mapping \(\mid\mid\) iff for every \(A \leftarrow B_1, \ldots, B_n\) in \(\text{ground}_C(P)\):

\[
\text{for } i \in [1, n], \text{ if } B_i \text{ is an atom then } |A| \triangleright |B_i|.
\]

\(\square\)

The definition of recurrent queries and states is derived accordingly. It can be easily shown that any derivation is finite with respect to any selection rule, when considering programs and queries both recurrent by a same level mapping.

**Example 7.3.5** *(Map and Mapflat)* Consider the CLP(\(\mathcal{R}\)) program \(\text{MAP}\).

\[
\begin{align*}
\text{map}([], \ [\]). \\
\text{map}([X|Xs], \ [Y|Ys]) &\leftarrow \\
&\quad Y = X \times X, \\
&\quad \text{map}(Xs, \ Ys).
\end{align*}
\]

Strictly speaking, \(\text{MAP}\) is not a CLP(\(\mathcal{R}\)) program. In fact, CLP(\(\mathcal{R}\)) and almost any other CLP system are languages over many-sorted domains (usually over \(\text{Term}\) and the specific domain of the system). Definitions and results lift to many sorted languages.

It is easy to see that \(\text{MAP}\) is recurrent by defining: \(|\text{map}(ls, \ rs)| = |ls|\). However, if we write \(\text{MAP}\) in a flat form, namely the following \(\text{MAPflat}\)
map(A, B) ←
  A = [], B = [].
map(A, B) ←
  A = [X|Xs], B = [Y|Ys], Y = X * X,
  map(Xs, Ys).

we obtain a program that is not recurrent by any level mapping. □

In the rest of this section, we give some termination conditions (more powerful than recurrence) specially targeted to CLP(ℛ) programs.

Excluding non-linear constraints

Our first proposal is to exclude non-linear constraints from the termination analysis. The next theorem states that when removing some non-linear constraints from a program and a query we get termination, then the original program and query terminate as well. Intuitively, the conclusion follows since adding constraints to a clause can only produce shorter derivations.

**Theorem 7.3.6** Consider the CLP(ℛ) system (which adopts the leftmost selection rule). Let P be a program and Q a query, and consider P' and Q' obtained by deleting some non-linear constraints from P and Q.

If every derivation for P' and Q' is finite then every derivation for P and Q is finite. □

As a corollary, acceptability of P' and Q' obtained by removing all non-linear constraints is a sufficient condition for termination.

**Corollary 7.3.7** Consider the CLP(ℛ) system.

Every derivation of a program P and a query Q is finite if P' and Q' are both acceptable by || and I, where P' (resp., Q') is obtained by deleting all non-linear constraints from P (resp., Q). □

**Example 7.3.8** (MAPFLAT) Consider again the MAPFLAT program. It is immediate to observe that the non-linear constraint Y = X * X does not play a relevant role in termination of a query such as map([X, 3, 5], Z). In fact, termination is implied by the decreasing of the length of the list in the first argument of map. By deleting Y = X * X we get the program MAPFLAT':

map(A, B) ← A = [], B = [].
map(A, B) ← A = [X|Xs], B = [Y|Ys], map(Xs, Ys).
which is acceptable by $\parallel$ and \( Real - base_L \), by fixing:

$$\text{map}(ls, rs) = \lceil ls \rceil.$$

By Corollary 7.3.7, MAPFLAT and \( \text{map}([X,3,5], Z) \) terminate. \hfill \square

However, this approach is not sufficient to prove termination when it depends on non-linear constraints.

**Example 7.3.9 (SQRT)** Consider the program SQRT for computing square roots of naturals.

\[
\text{sqrt}(X, R) \leftarrow \\
A = 0, \\
\text{sqrt2}(X, A, R).
\]

\[
\text{sqrt2}(X, A, A) \leftarrow \\
(A+1)*(A+1) > X.
\]

\[
\text{sqrt2}(X, A, B) \leftarrow \\
(A+1)*(A+1) \leq X, \\
A1 = A + 1, \\
\text{sqrt2}(X, A1, B).
\]

By removing the non-linear constraints, we get a program that has an infinite derivation for any query by applying the third rule again and again. \hfill \square

**Moding CLP(\( \mathbb{R} \)) programs**

To reason on programs containing non-linear constraints that become linear at runtime, we extend the notion of moding (see Definition 4.2.8) to CLP(\( \mathbb{R} \)). Without any loss of generality, we restrict to consider flat programs.

**Definition 7.3.10**

- Consider an \( n \)-ary predicate symbol \( p \) in \( L \setminus \Sigma_c \). A mode for \( p \) is a function \( d_p \) from \( \{1, \ldots, n\} \) in \( \{+, -, \rho\} \). If \( d_p(i) = + \) we call \( i \) an input position. If \( d_p(i) = - \) then \( i \) is called an output position. If \( d_p(i) = \rho \) then \( i \) is called a blank position (with respect to \( d_p \)). We write \( d_p \) in the form \( p(d_p(1), \ldots, d_p(n)) \).

- A mode for a primitive constraint \( c(X_1, \ldots, X_n) \) whose variables are \( X_1, \ldots, X_n \) is a function \( d_p \) from \( \{X_1, \ldots, X_n\} \) in \( \{+, -, \rho\} \). We write \( d_p \) in the form \( c(X_1 d_p(1), \ldots, X_n d_p(n)) \).

- A mode for a program \( P \) and a query \( Q \) is a collection of modes, one for each predicate and constraint in \( P \) or in \( Q \).
Chapter 7. From LP to CLP

- For an atom or a constraint \( A \), we write \( A(X, Y, Z) \) to denote that \( X \) are the variables occurring in input positions, \( Y \) are those occurring in output positions, and \( Z \) are those occurring in blank positions.

- We say that a flat program \( P \) is well-moded iff for every clause

\[
A_0(Y_0, X_{n+1}, Z_0) \leftarrow A_1(X_1, Y_1, Z_1), \ldots, A_n(X_n, Y_n, Z_n)
\]

of \( P \), for \( i \in [1, n+1] \quad X_i \subseteq \cup_{k \leq i} Y_k. \)

- We say that a flat query \( A_1(X_1, Y_1, Z_1), \ldots, A_n(X_n, Y_n, Z_n) \) is well-moded iff for \( i \in [1, n] \quad X_i \subseteq \cup_{k \leq i} Y_k. \)

The intuition underlying this definition is to force the input variables in an atom or a constraint selected along a derivation via the leftmost selection rule to be "grounded" (i.e., to be constrained in such a way that they can only assume a unique value) by the linear constraints. Variables not involved in the input-output relation are marked as blank.

Suppose now that the modes of constraints are consistent with the operational semantics, i.e. if a constraint \( c(X, Y, Z) \) is selected and the linear constraints imply \( X = a \) for some tuple \( a \) of elements of the domain, then the linear constraints of the resolvent (if exists) imply \( Y = b \) for some tuple \( b \). Under this assumption, when a non-linear constraint is selected then the input variables are grounded by the linear one. We can exploit this fact to impose that non-linear constraints become linear at run-time.

**Definition 7.3.11** A mode for a program \( P \) and a query \( Q \) is consistent w.r.t. \( \text{CLP}(\mathcal{R}) \) if for every primitive constraint \( c(X, Y, Z) \) in \( P \) or in \( Q \) either

(i) \( Y \) is an empty tuple and \( c(X, Y, Z) \) is linear in \( Z \), or

(ii) \( Y \) is a tuple of only one variable, \( Z \) is an empty tuple and \( c(X, Y, Z) \) is an equation linear in \( Y \).

It is worth noting that both well-modedness and consistency w.r.t. \( \text{CLP}(\mathcal{R}) \) are syntactic notions.

**Example 7.3.12** (Sqrt Ctd) Consider again the program \texttt{SQRT}. It is immediate to see that it is well-moded with the mode

\[
\begin{align*}
\texttt{sqrt}(j, j), & \quad A- = 0 \\
\texttt{sqrt2}(j, +, j) & \\
(A++1)*(A++1) > X & \\
(A++1)*(A++1) \leq X & \\
\end{align*}
\]
Moreover, the modes for the constraints are consistent w.r.t. CLP(R). Consider, for instance, \((A + 4) \times (A + 1) \leq X / Y\). Here, there are no output variables, thus Definition 7.3.11 (i) requires that the constraint is linear in \(X\), which is the case. 

The next theorem relates modes, acceptability and termination by providing a sufficient condition for termination of well-modal acceptable CLP(R) programs.

**Theorem 7.3.13** Consider the CLP(R) system. Let \(P\) be a well-modal flat program and \(Q\) a well-modal flat query, and assume that the mode of \(P\) and \(Q\) is consistent w.r.t. CLP(R).

If \(P\) and \(Q\) are both acceptable by \(I\) and \(|\|\) then every derivation of \(P\) and \(Q\) is finite.

**Example 7.3.14** The program \(\text{SQRT}\) and the query \(\text{sqrt}(n, R)\) for \(n \in N\) are well-modal and acceptable by \(|\|\) and Real - base_{\ell}, where:

\[
|\text{sqrt2}(x, a, b)| = \begin{cases} 
  \max\{x - a, 0\} & \text{if } x, a \in N \\
  \infty & \text{otherwise}
\end{cases}
\]

\[
|\text{sqrt}(x, r)| = \begin{cases} 
  x + 1 & \text{if } x \in N \\
  \infty & \text{otherwise}
\end{cases}
\]

Therefore, Theorem 7.3.13 allows us to state termination of \(\text{SQRT}\) and \(\text{sqrt}(n, R)\) when \(n \in N\).

Theorem 7.3.13 can be used together Theorem 7.3.6 in order to prove termination of programs \(P\) and queries \(Q\), by means of the following strategy:

(i) delete some (non-linear) constraints from \(P\) and \(Q\), and

(ii) show that the resulting program and query are well-modal and acceptable by the same model and level mapping.

**A transformational approach**

In general, however, the combined strategy outlined above is only a sufficient condition for proving termination.

**Example 7.3.15 (FACT Ctd)** Consider the flat version of the FACT program of Example 7.3.3.

```haskell
fact(0, 1).
fact(1, 1).
fact(N, F) ←
    F = N * F1,
    F1 >= 1,
    N >= 2,
    fact(N-1, F1).
```
We have already observed that for a query such as \texttt{fact(N, 24)} the presence of the non-linear constraint \( F = N \ast F1 \) is crucial. Moreover, we note that at run-time this constraint remains non-linear. Thus, the program and the query cannot be well-moded. Actually, they do not terminate in the CLP(\( \mathcal{R} \)) system, as observed in Example 7.3.3.

In the following, we propose the CLP(\( \mathcal{R} \)) extension of the transformational strategy of Section 2.5.4.

**Definition 7.3.16** Consider the CLP(\( \mathcal{R} \)) system. Let \( P \) be a program, \( Q \) a query and \( k \in \mathbb{N} \).

- For an atom \( A = p(T_1, \ldots, T_n) \), we define \( A(D) \) the atom \( p(T_1, \ldots, T_n, D) \).
  
  For a primitive constraint \( A \), instead, we define \( A(D) = A \).

- We define \( \text{Ter}(P) \) as the program such that:
  
  for every clause in \( P \)
  
  \[ A \leftarrow B_1, \ldots, B_n. \]
  
  which is not a fact, the clause
  
  \[ A(D + 1) \leftarrow D \geq 0, B_1(D), \ldots, B_n(D). \]
  
  is in \( \text{Ter}(P) \), where \( D \) is a fresh variable, and

  for every fact in \( P \)
  
  \[ A \leftarrow B_1, \ldots, B_n. \]
  
  the fact
  
  \[ A(D) \leftarrow B_1, \ldots, B_n. \]
  
  is in \( \text{Ter}(P) \), where \( D \) is a fresh variable.

- Let \( Q \) be \( A_1, \ldots, A_n \). We define \( \text{Ter}(Q, k) \) as the query
  
  \[ A_1(k), \ldots, A_n(k). \]

On the one hand, since Theorem 7.2.14 holds for any CLP language, given a bounded program and query the transformation produces a CLP(\( \mathcal{R} \)) program and query which terminate.

**Theorem 7.3.17** Consider the CLP(\( \mathcal{R} \)) system. Let \( P \) be a program and \( Q \) a query both bounded by \( \| \) and \( I \), and let \( k \) be a given natural number satisfying Definition 7.2.12.

Then, for every \( n \in \mathbb{N} \), every derivation of \( \text{Ter}(P) \) and \( \text{Ter}(Q, n) \) is finite.

Moreover, there is a bijection between refutations of \( P \) and \( Q \) with satisfiable answer constraints and refutations of \( \text{Ter}(P) \) and \( \text{Ter}(Q, k - 1) \) with satisfiable answer constraints.

\[ \square \]
On the other hand, consider a program and a query such that every derivation in CLP(\(\mathcal{R}\)) is finite. By Theorem 7.2.9, \(P\) and \(Q\) are acceptable by some \(||\) and \(I\). Since proof obligations of acceptability are stronger than those of boundedness, this implies that \(P\) and \(Q\) are bounded by the same \(||\) and \(I\). Therefore, the transformational approach is complete.

**Example 7.3.18 (FACT Ctd.)** Consider again the FACT program and the query \(\text{fact}(N, 24)\) of Example 7.3.3. We have shown that they are both acceptable (hence bounded) by \(||\) and \(\text{Real} - \text{base}_L\) where

\[
|| \text{fact}(n, f) || = \max\{\text{int}(f), 0\}. 
\]

By Theorem 2.5.21, \(\text{Ter}(\text{FACT})\) and the query \(\text{fact}(N, 24, 24)\) terminate, where \(\text{Ter}(\text{FACT})\) is:

\[
\begin{align*}
\text{fact}(0, 1, D). \\
\text{fact}(1, 1, D). \\
\text{fact}(N, N * F, D + 1) \leftarrow \\
\quad D >= 0, \\
\quad F >= 1, \\
\quad N >= 2, \\
\quad \text{fact}(N-1, F, D). 
\end{align*}
\]

The only answer constraint produced is \(N = 4\). □

Summarizing, the transformational strategy is a sound and complete method for proving termination of CLP(\(\mathcal{R}\)) programs. Obviously, to be applied, it requires that the natural \(k\) in Definition 7.2.12 is given. This gives rise to the problem of inferring extended level mappings and C-models, which is currently a challenging research topic (see Related work).

### 7.4 Related work

**Termination of CLP programs**

There is still little work on the extension of termination method for logic programs up to CLP programs. The only works we are aware of are from Colussi et al. [48], Mesnard [117], Mesnard et al. [118, 89].

Colussi et al. [48] proposed a necessary and sufficient condition for termination inspired by the works of Floyd on termination of flowchart programs. The method is based on assigning a dataflow graph to a program, whose nodes are the program points and whose arcs are abstractions of the rules describing operational semantics of CLP languages. The termination method consists of proof obligations that forbid
a state to enter an infinite loop in the graph. The approach is presented in the case of ideal systems, and an extension to non-ideal ones is sketched in the conclusions. Unfortunately, the extension involves the constraint solver $solve_C$ at the proof level, thus losing in declarativeness.

Mesnard [117] provided sufficient termination conditions based on approximation techniques and boolean mu-calculus, with the aim of inferring a class of queries such that that the program and each query in the class universally left terminate. The approach has been refined and implemented by Hoarau and Mesnard [89] to infer classes for programs obtained by a static reordering of atoms in clause bodies of a given program.

Mesnard et al. [118] showed a method to check where a conjunction of constraints is a logical consequence of a CLP program. This approach seems to be suitable to check (and, possibly, to infer) $C$-models of CLP programs, which can be used in checking (and, possibly, inferring) acceptability or boundedness.

We also mention that Hanus [85] characterized a class of programs with no delayed constraints at the end of successful computations. The method is able to discover possible sources of non-termination due to the delaying of non-linear constraints.

**Groundness analysis**

Methods for groundness analysis, such as our notion of mode, have been lifted from of logic programming (see e.g., Decorte and De Schreye [60], Cousot and Cousot [53]) up to CLP (see e.g., [83, 85]).

However, groundness analysis is much more complicated in CLP languages than in logic programming. For instance, the constraint $X = Y + Z$ implies the groundness of $Y$ if $X$ and $Z$ are ground, but not the groundness of $Y$ and $Z$ if $X$ is ground. Our notion of well-modalness overcomes the problem by associating modes with variables and not with argument positions. However, it is still not able to prove groundness of $X$ and $Y$ after the query $X + Y = 5, X - Y = 1$.

Groundness analysis can be seen as a special case of functional dependencies among variables. Maher [109] extended the notion of functional dependencies to constraint dependencies, which express that a variables is functionally dependent on another variable on a subset of the least $C$-model of a program for which a given constraint hold. When the dependent variable can assume only a finite number of values, then the dependency is called a constraint finiteness dependency. Maher proposed an algorithm for inference of constraint dependencies and studied some applications of constraint dependencies.
7.5 Conclusion

We have presented natural extensions to the CLP Scheme of acceptability and boundedness. For a class of CLP languages, namely ideal languages, acceptability is a sound proof method w.r.t. left termination. An example of ideal languages is the CLP$(RTerm)$, alias Prolog without occur check. Boundedness, instead, is a sound and complete characterization of bounded nondeterminism w.r.t. any CLP language.

In the second part of the Chapter, we have investigated termination for a specific (non-ideal) language, namely the CLP$(R)$ system, by proposing three approaches.

As the first step, we tried to exclude constraints that cause incompleteness of the solver from the termination analysis.

Then, acceptability was combined with an extension to CLP$(R)$ of the notion of mode.

Finally, boundedness and the transformational approach of Section 2.5.4 have been extended to CLP$(R)$. This leads to a sound and complete termination proof method when the extended level mapping is given and computable.

We have pointed out that starting from the definition of acceptability for CLP programs, a proof relation for weak total correctness can be derived in the case of ideal CLP languages by following the approach of Chapter 3. Concerning non-ideal CLP languages, a characterization of terminating programs and queries must be found first, e.g. by proceeding as in the case of CLP$(R)$.

Most of the properties of the $\vdash_t$ proof relation directly extend to ideal languages, including (weak) partial and total correctness, persistency, call and success patterns, modularity, weakest preconditions, correct instances characterization. Characterization of computed instances, testing and debugging, instead, do not lift to all ideal CLP languages (see Example 7.2.11).
Chapter 8

Conclusions

**declarative** (Computing) designating high-level programming languages which can be used to solve problems without requiring the programmer to specify an exact procedure to be followed.

The Concise Oxford Dictionary, ninth edition

### 8.1 Summary

Due to its mathematical roots, the family of logic programming languages is advocated as an ideal support to declarative programming.

Practical experience, however, shows that programs and specifications often do not correspond, in the sense that the intended interpretation of a program is not a model of the program. In addition, programs may fail to terminate, may end in run-time errors, may deliver unexpected output, may behave differently in different implementations of logic programming systems. These arguments motivate the need for the verification of logic programs.

In addition, even when a formal proof of correctness is provided, programs must be validated against the user requirements, and, in case a wrong or missing answer is found, the source of the error must be located. These arguments motivate the need for the validation of logic programs.

The main original contribution of this thesis consists of having provided a single unifying framework able to reason about several verification and validation properties at the same time. Our approach started from the declarative characterization of classes of universal terminating programs and queries.
On the one hand, we have systematically lifted those characterizations to verification proof methods for weak total correctness. In particular, we have investigated the method derived from acceptability, showing how the proof relation $\vdash_t$ allows us to reason on:

(i) left termination,
(ii) (weak) partial correctness,
(iii) (weak) total correctness,
(iv) weakest (liberal) preconditions and strongest postconditions,
(v) characterization of call and success patterns,
(vi) absence of run-time errors,
(vii) characterization of correct and computed instances,
(viii) safe omission of the occur-check,
(ix) modular program development,
(x) meta-programs.

The unifying framework for verification of logic programs has been obtained by a stepwise definition of increasingly higher levels of verification, from termination up to full-fledged total correctness, including the properties (i-x). On the other hand, we have systematically derived testing and debugging tools for the classes of terminating programs introduced. In particular, we concentrated on formal approaches to:

(ii) testing,
(iii) and debugging.

While the two ways were presented separately, we have observed that they can be integrated, in the sense that the testing and debugging approaches can be reformulated in the context of the $\vdash_t$ proof relation.

From a theoretical perspective, a single proof method for reasoning on several program properties may be less powerful than using many separated ones — each specialized for one of the properties above. However, this is not always the case. For instance, the $\vdash_t$ proof relation has been shown to be complete w.r.t. weak partial correctness plus left termination.

From a practical perspective, our approach has aimed at producing a framework which is a trade-off between full expressive power and easy of use in paper & pencil proofs, i.e. a framework where fewer proof obligations are needed with respect to the use of separate methods, and redundancy (of pieces) of proofs is avoided. The framework should be sufficient for reasoning on a large subset of programs and for a large spectrum of properties. Of course, for “unusual” programs, specialized methods for reasoning on call patterns, absence of run-time errors, and other operational properties may be needed.
Other original contributions of this thesis to specific research fields are summarized in the following list (also pointing out the papers where the results have been described for the first time):

**Termination**
- The approach of Apt and Pedreschi has been revisited, providing a sound and complete characterization of left terminating programs and queries.
- The notions of \( \exists \)-universal termination \([140]\) and bounded nondeterminism have been introduced, and a declarative characterization of them has been provided by means of fair-bounded and bounded programs and queries.

**Verification**
- We have presented a formal derivation of weak total correctness proof relations from declarative characterization of universal terminating programs.
- The \( \vdash_t \) relation has been investigated in depth, and enriched with additional proof obligations in order to prove total correctness \([133]\).
- A characterization of the weakest (liberal) preconditions as ordinal closures of a monotonic operator has been provided \([132]\).
- Relation \( \vdash_t \) has been extended to LDNF-resolution, and, as a byproduct, a completeness theorem has been obtained \([133]\).

**Meta-programming**
- We have shown that, under certain natural assumptions, the verification properties \((i-iii, v-vii)\) lift from the object program up to the *Vanilla* meta-interpreter \([128, 131]\);
- A general criterion \([GC]\) for proving declarative correctness of meta-programs has been introduced \([131]\).

**Validation**
- The notion of testing of logic programs has been introduced and related to semantics decidability \([137]\);
- Decision procedures w.r.t. \( C \) and \( S \)-semantics have been proposed and implemented as Prolog meta-programs for large classes of programs \([137]\);
- A declarative debugger of missing answers has been derived from the decision procedures, improving on completeness of existing debuggers \([139]\).

**Constraint Logic Programming**
- Acceptability \([138]\) and boundedness have been extended to ideal CLP languages;
- Some (necessary and) sufficient conditions for universal termination of \( \text{CLP}(\mathcal{R}) \) programs and queries have been proposed, where \( \text{CLP}(\mathcal{R}) \) is a widely used *non-ideal* language.
8.2 Future Research

We foresee two main lines for future research, which are opened from the thesis work: tools and extensions.

Tools

In the last years, the logic programming community has realized that a major difficulty in the dissemination and the use of logic programming languages lies in the lack of tools, that help users in designing, producing, verifying and validating software.

Since the basic results of the integrated theoretical framework proposed in the thesis are fixed, we believe that the step ahead is the development of an integrated tool.

This does not necessarily mean starting from scratch. Since parts of the framework can be seen as the combination of existing approaches for which automated tools already exist, we argue that the development of an integrated tool actually should consist of some glue software. In particular, we believe in the possibility to integrate (adaptation of) existing tools reasoning on:

- termination (e.g., [60]);
- typing (e.g., [62] and [67]);
- testing (e.g., [20] and [108]);
- and debugging (e.g., [67]).

In Chapter 6, we have already presented some preliminary results on the integration of the decision procedures with the PROTest tool.

Extensions

We foresee three orthogonal research lines: languages, methods, and applications.

Languages. A natural question is how far we can go from pure logic programming without betraying the principles of our approach. Possible extensions include typed languages, languages supporting dynamic selection rules, higher order languages, concurrent constraint languages, deductive databases.

Methods. A second issue is how programming methodologies not treated here can be integrated within and/or validated by the framework. Possible methodologies include program patterns, program refinement, program transformation schemas and program composition.

Applications. Another issue is how some attracting theoretical results of the thesis can be turned into practical applications. Among the others, we mention the theory of the weakest (liberal) preconditions.

For those three issues, we propose below three specific lines which we believe worth investigating.
Languages: Weak and strong fairness

Logic programs have a natural interpretation in terms of nondeterministic concurrent systems, where atoms model processes, shared variables model multiparty communication channels, clauses model process activations, and queries model dynamic networks of parallel processes. Therefore, fair-boundedness models termination w.r.t. unconditional fairness (see e.g., Francez [82]).

One extension of logic programming that tries to provide a declarative view of control is the family of languages that adopt dynamic selection rules. In Gödel [88], for instance, program annotations called delay declarations allow the user to specify restrictions on the admissible selection rules. In the concurrent view, delay declarations models synchronizations among processes. Therefore, the notions of weak and strong fairness [82] naturally arise in this context.

It would be interesting then to characterize termination w.r.t. weak and strong fair selection rules.

A preliminary study on the analysis of concurrent logic programs is reported in [130], where a class of programs with only successful derivations has been characterized using an approach similar to the one of this thesis.

Methods: Program patterns

In this thesis we have reasoned about verification of modular programs. However, other good programming practices have been proposed that allow for program reuse, systematic development, and maintenance. As an example, Sterling and Yalcinalp [151] explicitly identify program patterns that have emerged within logic programming. They propose a systematic construction of logic programs that standardize the use of skeletons and programming techniques, and call this stepwise enhancement, an adaptation of stepwise refinement.

A challenging issue is to validate the correctness of stepwise refinement methodologies by means of the verification principles of the thesis.

Applications: Weakest (liberal) preconditions

Another issue that deserves more investigation is the study of the weakest (liberal) preconditions. We have observed that even for small programs, the direct construction of the weakest (liberal) preconditions may result in a complex task. The origin of this complication lies in the fact that logic programming in an untyped language, so the weakest (liberal) preconditions may contain unintended atoms. We believe, however, that in the context of a typed logic programming language, the theory of weakest preconditions may be fruitful for practical calculations (in the style of Hoare’s logic), or for inferring larger preconditions starting from a given one.
Bibliography


<table>
<thead>
<tr>
<th>Symbol</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>□</td>
<td>240</td>
</tr>
<tr>
<td>*</td>
<td>10</td>
</tr>
<tr>
<td>≤_{tx}</td>
<td>115</td>
</tr>
<tr>
<td>&lt;</td>
<td>116</td>
</tr>
<tr>
<td>⊮</td>
<td>144</td>
</tr>
<tr>
<td>⊳</td>
<td>19</td>
</tr>
<tr>
<td>⪰_{m}</td>
<td>10</td>
</tr>
<tr>
<td>⪰_{m}</td>
<td>10</td>
</tr>
<tr>
<td>□_{m}</td>
<td>48</td>
</tr>
<tr>
<td>□_{m}</td>
<td>48</td>
</tr>
<tr>
<td>⊥</td>
<td>131</td>
</tr>
<tr>
<td>⊥</td>
<td>9</td>
</tr>
<tr>
<td>≠</td>
<td>253</td>
</tr>
<tr>
<td>≃</td>
<td>196,244</td>
</tr>
<tr>
<td></td>
<td>, list-length</td>
</tr>
<tr>
<td>a</td>
<td>Q</td>
</tr>
<tr>
<td>a</td>
<td>Q</td>
</tr>
<tr>
<td>a</td>
<td>Q</td>
</tr>
<tr>
<td>a</td>
<td>Q</td>
</tr>
<tr>
<td>a</td>
<td>Q</td>
</tr>
<tr>
<td>a</td>
<td>Q</td>
</tr>
<tr>
<td>c</td>
<td>9</td>
</tr>
<tr>
<td>[.]_L</td>
<td>8</td>
</tr>
<tr>
<td>(Pre, Post)</td>
<td>71</td>
</tr>
<tr>
<td>{Pre} P {Post}</td>
<td>97</td>
</tr>
<tr>
<td>C</td>
<td>241</td>
</tr>
<tr>
<td>C-ground instances</td>
<td>241</td>
</tr>
<tr>
<td>C-interpretation</td>
<td>241</td>
</tr>
<tr>
<td>D_C</td>
<td>238</td>
</tr>
<tr>
<td>L_C</td>
<td>238</td>
</tr>
<tr>
<td>T_C</td>
<td>238</td>
</tr>
<tr>
<td>⟨Σ_L, Π_L⟩</td>
<td>7</td>
</tr>
<tr>
<td>n_p</td>
<td>8</td>
</tr>
<tr>
<td>Π_P</td>
<td>8</td>
</tr>
<tr>
<td>Σ_P</td>
<td>8</td>
</tr>
<tr>
<td>Σ_c</td>
<td>238</td>
</tr>
<tr>
<td>μ()</td>
<td>181</td>
</tr>
<tr>
<td>ḥ_{P,P_{out}}()</td>
<td>123</td>
</tr>
<tr>
<td>comp()</td>
<td>10</td>
</tr>
<tr>
<td>gfp()</td>
<td>9</td>
</tr>
<tr>
<td>ground_C()</td>
<td>241</td>
</tr>
<tr>
<td>ground_L()</td>
<td>8</td>
</tr>
<tr>
<td>length_P()</td>
<td>26</td>
</tr>
<tr>
<td>lfp()</td>
<td>9</td>
</tr>
<tr>
<td>lmax()</td>
<td>32</td>
</tr>
<tr>
<td>mgi()</td>
<td>70</td>
</tr>
<tr>
<td>rel()</td>
<td>131</td>
</tr>
<tr>
<td>length_P()</td>
<td>50</td>
</tr>
<tr>
<td>size()</td>
<td>13,242</td>
</tr>
<tr>
<td>solVC()</td>
<td>238</td>
</tr>
<tr>
<td>sp(P, Pre)</td>
<td>105</td>
</tr>
<tr>
<td>unknown</td>
<td>238</td>
</tr>
<tr>
<td>value()</td>
<td>55</td>
</tr>
<tr>
<td>wp(P, Post)</td>
<td>105</td>
</tr>
<tr>
<td>Atomp_L</td>
<td>7</td>
</tr>
<tr>
<td>B_L</td>
<td>7</td>
</tr>
<tr>
<td>B_P</td>
<td>8</td>
</tr>
<tr>
<td>C_k</td>
<td>202</td>
</tr>
<tr>
<td>Card()</td>
<td>22</td>
</tr>
<tr>
<td>FF_p</td>
<td>8,137</td>
</tr>
<tr>
<td>GList</td>
<td>10</td>
</tr>
<tr>
<td>HIS</td>
<td>9</td>
</tr>
</tbody>
</table>

278
Index

I → J ........................................... 9
L ........................................... 7, 8
L_P ........................................... 8
M_P ........................................... 9, 137
Mim() ........................................... 116
N ........................................... 10
N^∞ ........................................... 19
P_> ........................................... 55
P_{is} ........................................... 55
Pre-termination ........................................... 119
Query_y_L ........................................... 8
T_P() ........................................... 9
T_{P}^b() ........................................... 219
T_{P}^g() ........................................... 219
Ter(P) ........................................... 52,256
Ter(Q,k) ........................................... 52,256
Tree(α, β) ........................................... 100
U_L ........................................... 7
Var() ........................................... 118
Post_> ........................................... 129
Post_{is} ........................................... 129
Post_{V m(P)} ........................................... 160
Pre_> ........................................... 129
Pre_{is} ........................................... 129
Pre_{V m(P)} ........................................... 160
[R-GC] - R-General Criterion ............................ 176
[GC], general criterion ........................................... 167
[sc], soundness criterion ........................................... 169
[wGC], weak soundness criterion .............................. 169
Σ^b ........................................... 86
Σ^f ........................................... 81
- ........................................... 98
call patterns ........................................... 109,140
computed instances ........................................... 116
correct instances ........................................... 116
partial correctness ........................................... 112
persistency ........................................... 108,140
semantics decidability ........................................... 134
success patterns ........................................... 109,140
weak partial correctness ........................................... 110
\( \mathcal{L}_t \) ........................................... 78,98
general programs ........................................... 137

- persistency ........................................... 118
semi-termination completeness ........................................... 131
total correctness ........................................... 121,144
weak total correctness ........................................... 119,143
acceptable programs ........................................... 19,192
CLP programs ........................................... 243
general programs ........................................... 213

acceptability

Apt and Pedreschi ........................................... 65
completeness ........................................... 27,246
persistency ........................................... 22,245
soundness ........................................... 25,246
acyclic programs ........................................... 147,213
amalgamation ........................................... 158,169
safe ........................................... 170
simple ........................................... 169
ambivalence ........................................... 159
annotation method ........................................... 93
arithmetic built-in’s ........................................... 55,129,210
in meta-interpreters ........................................... 163
arity ........................................... 7
bag ........................................... 10
bounded nondeterminism ........................................... 45
bounded programs ........................................... 46,196
CLP programs ........................................... 247
boundedness

completeness ........................................... 52,248
inference ........................................... 59
persistency ........................................... 48,248
soundness ........................................... 49,248
call patterns ........................................... 72
claus ........................................... 8
complete control ........................................... 11
computed instance ........................................... 8
constraint domain ........................................... 238
constraint logic programs ........................................... 238
constraint solver ........................................... 238
constraints ........................................... 238
correct ........................................... 242
answer ........................................... 241
linear ........................................... 249
<table>
<thead>
<tr>
<th>Term</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>non-linear</td>
<td>249, 253</td>
</tr>
<tr>
<td>primitive</td>
<td>238</td>
</tr>
<tr>
<td>correct instance</td>
<td>9</td>
</tr>
<tr>
<td>characterization</td>
<td>72</td>
</tr>
<tr>
<td>correctness</td>
<td>90</td>
</tr>
<tr>
<td>general programs</td>
<td>137</td>
</tr>
<tr>
<td>partial</td>
<td>71</td>
</tr>
<tr>
<td>success</td>
<td>91</td>
</tr>
<tr>
<td>total</td>
<td>71</td>
</tr>
<tr>
<td>weak partial</td>
<td>71</td>
</tr>
<tr>
<td>weak total</td>
<td>71</td>
</tr>
<tr>
<td>de Boer</td>
<td>94</td>
</tr>
<tr>
<td>debugger</td>
<td></td>
</tr>
<tr>
<td>C-semantics</td>
<td>223</td>
</tr>
<tr>
<td>S-semantics</td>
<td>227</td>
</tr>
<tr>
<td>Ferrand's</td>
<td>220</td>
</tr>
<tr>
<td>Shapiro's</td>
<td>219</td>
</tr>
<tr>
<td>declarative correctness</td>
<td>156</td>
</tr>
<tr>
<td>( \mathcal{R} )-declarative correctness</td>
<td>176</td>
</tr>
<tr>
<td>declarative debugging</td>
<td>217</td>
</tr>
<tr>
<td>diagnosis problem</td>
<td>189</td>
</tr>
<tr>
<td>fact</td>
<td>240</td>
</tr>
<tr>
<td>fair selection rule</td>
<td>9</td>
</tr>
<tr>
<td>fair-bounded programs</td>
<td>31, 193</td>
</tr>
<tr>
<td>fair-boundedness</td>
<td></td>
</tr>
<tr>
<td>completeness</td>
<td>42</td>
</tr>
<tr>
<td>persistency</td>
<td>36</td>
</tr>
<tr>
<td>soundness</td>
<td>39</td>
</tr>
<tr>
<td>fairness</td>
<td>265</td>
</tr>
<tr>
<td>first order language</td>
<td>7</td>
</tr>
<tr>
<td>flat programs</td>
<td>242</td>
</tr>
<tr>
<td>floundering</td>
<td>137</td>
</tr>
<tr>
<td>function symbol</td>
<td>7</td>
</tr>
<tr>
<td>general programs</td>
<td>135</td>
</tr>
<tr>
<td>ground term</td>
<td>7</td>
</tr>
<tr>
<td>groundness analysis</td>
<td>258</td>
</tr>
<tr>
<td>inductive method</td>
<td>75, 93</td>
</tr>
<tr>
<td>intended interpretation</td>
<td>1, 70, 219</td>
</tr>
<tr>
<td>interargument relations</td>
<td>66</td>
</tr>
<tr>
<td>level mapping</td>
<td>13</td>
</tr>
<tr>
<td>computable</td>
<td>196</td>
</tr>
<tr>
<td>extended</td>
<td>19, 243</td>
</tr>
</tbody>
</table>
Index

locality .................................. 43
local ................................... 65
semilocal .................................. 66

semantics
\mathcal{C} .................................. 70, 91
\mathcal{CP} .................................. 92
\mathcal{CWA} .................................. 209
\mathcal{F} .................................. 90, 189
\mathcal{FF} .................................. 209
\mathcal{IO} .................................. 91
\mathcal{L} .................................. 209
\mathcal{M} .................................. 70
\mathcal{MC} .................................. 209
\mathcal{S} .................................. 70, 91
\mathcal{S}_\text{er} .................................. 211

semantics decidability .................. 134, 188, 221
signature ..................................... 7
specification ................................. 71, 90, 91
strongest postconditions ................. 105

symptom .................................. 189, 217, 235

termination
exists-universal .................................. 29
CLP programs .................................. 243
existential ................................... 12
inference ................................... 59, 66
left ........................................ 17
reuse ........................................ 58
universal ..................................... 12
testing problem .................................. 189
uncovered atoms .................................. 218
unit clause ................................... 8

van Gelder .................................. 65
weakest liberal precondition ............... 105, 123
weakest precondition ....................... 90, 107, 123
well-m-asserted programs .................... 148
well-supported interpretation ............. 112, 121

wrong answers .................................. 217
Andréka ..................................... 233
Apt .................................. 2, 3, 7, 65, 70, 90, 93, 134, 148
Börger ..................................... 233
Barklund .................................... 183
Baudinet .................................... 66

Bellì ........................................... 234
Bézia .................................. 13, 65, 234
Bol ........................................ 66
Bossi .................................. 65, 93, 148
Bowen ........................................ 183
Boyce ..................................... 91
Brogg ........................................ 183
Bronsard .................................. 67, 94, 152
CLP ........................................ 6, 237
CLP(\mathcal{C}) .................................. 238

CLP(\mathcal{R}) .................................. 239
CLP(RTerm) .................................. 239
CLP(Term) .................................. 239
Cavedon .................................... 65, 147
Clark .................................. 10, 69, 75
Coccor .................................. 94, 148
Codish ..................................... 67
Colmerauer .................................. 239
Colussi .................................... 94, 257

Comini .................................. 232, 235, 258
Crnogorac .................................. 93
De Schreye .................................. 64, 93, 159, 183, 258
Decorte .................................. 64, 93, 258
Dembinski .................................. 94
Deransart .................................. 67, 69, 91, 93, 152
Dershowitz .................................. 10
Devienne .................................... 66, 233
Devil ........................................ 93, 153
Dershowitz .................................. 10

d'Etalle ..................................... 67
Falaschi ..................................... 69
Ferrand .................................. 94, 152, 235
Gödel .................................. 12, 65, 159
Gallagher .................................... 183
Hanus .................................... 258
Hill ......................................... 183
Index

Hoarau .................. 258
Hoare's logic ............. 2,90,123
Hogger .................. 93
Jack ..................... 234
Jacquet .................. 93
Jaffar ................... 238
Jang ..................... 159
Kalsbeek .................. 159, 184
King ..................... 66
Kowalski .................. 1, 11, 67, 183
Krishna Rao ................ 66
LD-resolution .............. 9
LDNF resolution .......... 135
LDNF-resolution .......... 137
completeness ............. 147
Levi ..................... 184
Lindenstrauss ............... 67
Lloyd .................... 1, 7, 183
Lučišes ................... 94
Luo ...................... 234
Maher .................... 90, 238, 246, 258
Malfon .................... 94, 150
Marchiori ................ 65, 93, 148
Martens .................. 159, 183
Martin .................... 66
Maluszynski ................. 67, 91, 94, 152
Mellish ................... 67
Mercury .................. 12, 65
Mesnard .................. 94, 247, 257
NSTO ..................... 133
Naish .................... 65, 94
Noyé ..................... 93, 234
Németi ................... 233
Olderog .................. 2, 90
PROTest .................. 214, 234
Pedreschi ................ 3, 65, 94
Pellegrini ................ 134
Plümer .................. 65, 233
Potter ................... 64
Prolog III ................. 241
Prolog .................. 12, 241, 246
RISC-CLP (Real) ........... 239

Ramundo .................. 184
Ruggieri .................. 94
SEM1 ..................... 117
SEM2 ..................... 117
SLD-resolution .......... 8
SYN1 ..................... 118
SYN2 ..................... 118
Sagiv ..................... 67
Shapiro .................. 183, 235
Speirs ................... 67
Stärl ........ ............ 153
Sterling .................. 183, 265
Taboch ................... 67
Tessier ................... 235
Teusink ................... 65
Turini .................... 184
Ullman ................... 65

Vanilla ................... 157

call patterns ........... 162
computed instances ...... 179
correct instances ...... 179
declarative correctness .... 166
declarative soundness .... 164
lifting of \( \vdash \) .......... 160
lifting of \( \vdash_{l} \) .......... 180
success patterns .......... 163
termination ............ 182
typed .................. 184
weak declarative soundness ... 164

Vasak .................... 64
Yalcinalp ................. 265

> ................................ 55
\leq ................................ 55
\lt ................................ 55
\neq ................................ 55
\geq ................................ 55
\emptyset .......................... 55
\B_1 \& \ldots \& \B_n .......... 157
& ................................ 158
call ......................... 158, 212
clause/2 .................. 157, 204
clause/3 ................. 203
Index

freeze ........................................... 201
is .................................................. 55
melt ............................................... 221
pure ............................................... 222
true ............................................... 158
user.pred ........................................ 220
valid ............................................... 220
valid.s ........................................... 224
APPEND ..........................................15,70,72,107
Call(P) .......................................... 174
CURRY ............................................ 244
DECC-A(P) ........................................ 202
DECC-B(P) ........................................ 207
DECC-F(P) ........................................ 206
DECM-B(P) ........................................ 207
DECM-F(P) ........................................ 204
DECS-A(P) ........................................ 203
DECS-B(P) ........................................ 208
DECS-F(P) ........................................ 206
E-APPEND ........................................ 221,225
E-PREORDER .................................... 222,225
EVEN ............................................. 16,21,117
FACT ............................................. 250,255,257
FLAT .............................................. 151
IN-C(P) .......................................... 222
IN-S(P) .......................................... 222
LEXORD ........................................... 115
MAPFLAT ......................................... 251
MAP ............................................... 251
MEMBER .......................................... 243
MISS-C(P) ........................................ 224
MISS-S(P) ........................................ 228
NAIVE REVERSE .................................. 17,20,25
ODDEVEN ......................................... 42,44,47
PART ............................................. 56,129
PERMUTATION .................................... 28,34,44,47,54,64,87
PREORDER ....................................... 99,103,110,119,122,133
PRODCONS ...................................... 30,39,84,195,206
PTB(P) ........................................... 177,218
RQUEUE .......................................... 196,209
SAT-ERROR ...................................... 193,203

SAT .................................. 14,192,201,203,215
SQRT .................................. 253
STRANGE ................................... 169,173
SUBLIST .................................. 130
TRANS .................................. 144
TRANSF .................................. 18,21
VU(P) .................................. 171
Van(P) .................................. 157