KATHOLIEKE UNIVERSITEIT LEUVEN
FACULTEIT WETENSCHAPPEN
FACULTEIT TOEGEPASTE WETENSCHAPPEN
DEPARTEMENT COMPUTERWETENSCHAPPEN
Celestijnenlaan 200A — B-3001 Leuven (Heverlee)

ON THE SEMANTICS OF META-PROGRAMMING AND THE CONTROL OF PARTIAL DEDUCTION IN LOGIC PROGRAMMING

Examencommissie :
Voorzitter Prof. Dr. ir. Y.D. Willems
Prof. Dr. D. De Schreye, promotor
Prof. ir. M. Gobin, promotor
Prof. Dr. ir. M. Bruynooghe
Prof. Dr. J. Gallagher (University of Bristol)
Prof. Dr. F. van Harmelen (Univ. Amsterdam)

Proefschrift voorgedragen tot het behalen van het doctoraat in de Informatica
door
Bernhard MARTENS

Februari 1994

On the Semantics of Meta-Programming and the Control of Partial Deduction in Logic Programming

Bernhard Martens

Department of Computer Science, K.U.Leuven

ABSTRACT

In logic programming, *meta-programming* has been advocated as a major route towards increased knowledge representation and reasoning capabilities. And writing programs that treat other programs as data is not difficult to accomplish within its framework. However, in many cases, the practice seemed to lack a clear *semantical foundation*.

In the *first part* of this thesis, we therefore study a *semantics for untyped, vanilla meta-programs, using the non-ground representation* for object level variables. We do not only address the basic vanilla meta-interpreter, but also some interesting extensions, including programs which allow some forms of amalgamation. We show that for stratified object programs, associated meta-programs are weakly stratified. For a large class of object programs, we establish a natural correspondence between the object level perfect and the meta level weakly perfect Herbrand models, thus providing a sensible meta-program semantics. Finally, for definite object programs, we reconsider and generalise these results in the context of an extended Herbrand semantics, designed to closely mirror the operational behaviour of logic programs.

Another problem faced by meta-programming applications is a considerable loss of *execution efficiency* (compared with reasoning directly at the object level). Specialising meta-interpreters with respect to object programs helps. *Partial deduction* constitutes one technique used to pursue this effect. However, its relevance is not limited to this particular setting.

This leads us, in the *second part* of this thesis, to a study of *partial deduction for (pure) definite logic programs*. Within that context, we focus on the (online) *control of unfolding*, devising methods to ensure its *termination* in a way that reflects structural properties of the query and program to be unfolded. We propose a general framework for finite unfolding, based on well-founded orderings. We extensively investigate several concrete instances and present fully automatic algorithms. Using such unfolding to construct finite SLD-trees, we formulate a sound and complete, always terminating, completely automatic method for partial deduction. Finally, some experiments, comparing various approaches, are briefly discussed. Throughout our presentation, we particularly emphasise detailed formalisations, allowing formal proofs for interesting properties of the various algorithms.

Acknowledgements

My first words of thanks are meant for you, reader of this thesis. After all, what greater joy can there be for one who writes than the existence of one who reads ? Or yes, perhaps the very act of writing itself. Indeed, working on this thesis has often (though not always) given me tremendous pleasure. I realise it is probably asking too much, but I can not rid myself of the secret hope that some of this enjoyment might occasionally leak through the subsequent pages.

Now, there is no escape possible, I have to face a task which has proven itself beyond my limited capabilities: finding the right words to thank my supervisor, Professor Danny De Schreye. After many hours of pondering, I decided to simply give up and admit defeat. Let me, taking advantage of the Dutch word for "supervisor", just cite Chuck Berry: "He is the most wonderful promoter I have ever had !"

Next, I would like to express my gratitude towards Professor Marc Gobin for accepting to be my second supervisor. I thank him, as well as the remaining members of the thesis committee, Professors Maurice Bruynooghe, John Gallagher, Frank van Harmelen and Yves Willems, for the time spent on reading my thesis and for very valuable feedback.

Science, like life, is often a very lonely activity. However, neither can be performed in complete solitude. Luckily, there were colleagues, at the Leuven Department of Computer Science, at ECRC, in Compulog, and elsewhere, as well as some (other) friends who decided, to a greater or lesser extent, that it was fun to keep me company for a while. One, named Ingrid, even choose to share an unusually large part of my life. Thanks to all of them.

At a more technical level, the content of this thesis greatly benefited from discussions with and contributions made by many people throughout the past seven years. I mention just three who were particularly important: Danny De Schreye (of course), Maurice Bruynooghe and Tamás Horváth. If you find you like the work below, please realise that at least part of the credit is theirs. However, if you do not, the blame is entirely mine.

Bern Martens
Leuven, January 1994

Financial support for my work has been kindly provided by: the Katholieke Universiteit Leuven, the Belgian government project RFO-AI-03, the Belgian National Fund for Scientific Research, ESPRIT Basic Research Action COMPULOG II, and the GOA "Non-Standard Applications of Abstract Interpretation" (Belgium).

This thesis is dedicated
to my father
and my mother.

And also to Jaak Moors,
whose wonderful teaching
first introduced me
to the delights of mathematics.

Contents

List of Figures

Chapter 1

Introduction

Logic programming developed out of the work on automated theorem proving between 20 and 30 years ago. An important discovery, enabling this breakthrough, was the invention of the resolution principle by Alan Robinson, whose main paper on the subject ([145]) appeared in 1965. The next major step was taken by Robert Kowalski, who realised that reasoning with a particularly simple form of logic formulas, called Horn clauses, only requires a simple and efficient instance of general resolution (LUSH, later renamed to SLD, see [101]). This led to his milestone paper on predicate calculus as a programming language ([96]). Finally, the semantic foundations of the new programming paradigm were firmly established in [170]. Meanwhile, a prototype implementation of a practical programming language was developed by Alain Colmerauer and Philippe Roussel, whose wife coined the name Prolog ("Programmation en Logique") ([149]). Two graciously written personal accounts on the roots and early development of logic programming and Prolog, can be found in [98] and [146].

Later developments include the transfer of the logic programming approach to the field of (relational) databases, leading to the birth of the nowadays flourishing deductive database concept (see a.o. [167], [168], [31], [74]), and the adoption of (parallel) logic programming as the basic computing paradigm underlying the Japanese Fifth Generation Computer Systems Project ([61]). One of the major technical extensions of the logic programming framework itself meanwhile has been the inclusion of the so-called "negation as failure" rule ([35]). This enhancement greatly increased the expressivity of the framework, but also caused quite a few semantical and procedural problems. We refer to chapter 2 for some further details.

Against this background, Kowalski recently sketched the ambitions of the logic programming community as no less than (we quote from the abstract of [99]):

> "The ultimate goal ... is to develop the use of logic for all aspects of computation. This includes not only the development of a single logic for representing programs, program specifications, databases, and knowledge representations in artificial intelligence, but also the development of logic-based management tools."

continuing as follows:

> "I shall argue that, for these purposes, two major extensions of logic programming are needed — abduction and metalevel reasoning."

It is the latter extension which is part of this thesis' subject matter.

Meta-level reasoning can be characterised as "reasoning about reasoning". It is, in fact, common practice in a human's everyday life. (A delightfully phrased and carefully analysed example, of a particular kind, namely reasoning about one's own reasoning, also termed "reflection", can be found in [78].) Many authors have therefore argued that a full-fledged knowledge representation and reasoning tool should include such a facility. In a logic setting, the idea boils down to the construction of theories about theories; in a programming context, it translates into programs that take as input, manipulate, and/or produce as output other programs. Since logic programming does not really distinguish between programs and data, it has no conceptual difficulties with such a practice. And indeed, applications of logic "meta"-programming have been manifold (see section 3.1 for some references).

However, many such applications seemed to lack a well-established semantics (amalgamation, the merging of object and meta theory, a practice closely related to the concept of reflection, posing especially challenging problems). Since a firm semantical foundation in first order logic is one of the main features boasted by logic programming, this situation can hardly be deemed satisfactory. In the first part of this thesis, we therefore consider in detail the semantics of one particularly influential kind of meta-programming. We remain as close as possible to the basic Herbrand semantics framework for logic programs (see chapter 2), and establish correspondence results between meta- and object-level semantics.

As mentioned above, logic programming does not only provide a knowledge representation tool with a clear semantic foundation. It is also a programming paradigm, providing the opportunity to write and execute programs. This dual capacity can in fact be considered its main outstanding feature. Writing programs, however, implies being concerned about their execution efficiency. Now, a substantial volume of research in logic programming has aimed at relieving the programmer as much as possible from this task, and shifting the burden of finding efficient execution strategies to the underlying system. The work on recursive query answering in deductive databases ([31]) can be mentioned as one example, the work on transformation and specialisation of logic programs as another (see e.g. [106]). Taken to its limits, such an approach leads to the complete reduction of programming to knowledge representation. This luring prospect, however, still seems to lie well beyond the realm of the currently feasible.

However, it is not only its dual nature, combining knowledge representation and programming, which makes logic programming a suitable vehicle for such undertakings. An equally important aspect is its well-established semantics. Indeed, this feature enables comparisons of various programs, including for example the demand that they should have the same "meaning", in spite of perhaps huge operational differences. Clearly, in a program transformation context, such an asset is a precious one. It makes possible the formulation of important results as the basic soundness and completeness theorem for partial deduction by Lloyd and Shepherdson (see section 5.4).

Which brings us to the particular transformation technique studied in part II of this thesis: *partial evaluation*, nowadays usually renamed to *partial deduction* in a logic programming context. Originally borrowed from other fields of computing, its basic idea is the (automatic) specialisation of a given program with respect to partially known input. The resulting more specific program should then be capable of dealing with concrete values for the rest of the input in a more efficient way than the original general program. (For a more extensive introduction to partial evaluation/deduction, we refer to chapter 5.)

A particularly important application of partial deduction is the specialisation of a meta-interpreter with respect to various object programs. Typically, the resulting program runs an order of magnitude faster on (remaining) concrete object level input. Even two orders of magnitude speedups have been reported in a context of specialising Gödel ([80]) ground representation meta-interpreters ([75]). In fact, practical meta-programming in logic programming seems hardly feasible without sophisticated program specialisation techniques of this kind.

The latter consideration provides the main conceptual link between the first and the second part of this thesis. Indeed, in part II below, we shift our attention from the semantical aspects of meta-programming, addressed in part I, and turn to more operational issues. We study partial deduction of logic programs, largely concentrating on one subproblem: the control of unfolding. We construct a framework to ensure its termination without resorting to ad hoc techniques and develop several fully automatic algorithms. We also include a method for overall partial deduction, which we show to always terminate and satisfy important correctness properties. Finally, we discuss some experiments comparing various ways to control partial deduction and unfolding.

Apart from chapters 3 and 4 in part I, chapters 5 to 8, constituting part II, and the present introduction, this thesis comprises two more chapters. Chapter 9 concludes the thesis. It briefly returns to the topical concluding discussions ending most chapters, reconsiders the overall picture, and indicates some additional directions for possible future research. Finally, in chapter 2, the one which immediately follows this introduction, we have compiled the main technical background on logic programming semantics, with which we suppose familiarity throughout the rest of this thesis.

Chapter 2

Technical Background

2.1 Introduction

In this chapter, we summarise the main technical background on the semantics (declarative as well as procedural) of logic programming, with which we suppose familiarity throughout the rest of this thesis. The presentation below is based on [110] and [5], mainly following the terminology adopted in the former work. In the present context, we can only include the briefest possible introduction, attempting to provide that material which seems indispensable for a good understanding of what follows. We refer to [110] and/or [5] for a more general and thorough treatment with further details, including extensive references to relevant papers.

Next, since (definite) logic programs are a (particularly simple) kind of first order logic theories, notions and results of plain (mathematical) logic are also relevant. We refer to any of the numerous textbooks available ([50] and [58] can be mentioned as examples). Finally, our presentation below will be in a setting of first order *predicate* calculus. All relevant notions can of course be straightforwardly applied to a *propositional* calculus program.

In sections 2.3 and 2.4, we sum up concepts and results on, respectively, definite and normal logic program semantics. But first, we present a bird's-eye view of some underlying basic notions.

2.2 Basics

Suppose that some first-order language, containing variables, constants, function symbols (or functors), predicate (or relation) symbols (and propositional constants) is given. Function and predicate symbols have an associated *arity*, a natural number indicating how many arguments the symbol takes in the definitions which now follow. *Terms* are defined as follows:

- a variable is a term
- a constant is a term
- a function symbol applied to terms is a term

and *atoms* as follows:

a predicate symbol applied to terms is an atom.

A *literal* is an atom possibly preceded by a negation sign. Literals of the latter kind will be called *negative*; *Positive* literals are simple atoms. A *program clause* is a formula of the following form:

$A \leftarrow B_1, \ldots, B_n,\ n \geq 0$

This formula is to be understood as a logical implication, with the condition part being a conjunction of literals and the consequent an atom. All variables occurring in its literals are supposed to be *universally* quantified, with the scope of the quantifiers extending over the entire clause. A is called the *head* of the clause and $B_1, \ldots, B_n$ its body. The body may be empty. In that case, the resulting clause is a *fact*. A *query* or *goal* is a clause with an empty head and a non-empty body:

$\leftarrow B_1, \ldots, B_n,\ n \geq 1$

It can be read as a logical implication the consequent part of which is *false*. A term, atom, clause, goal, ... without any variables is called *ground*. We will use the word "expression" to refer to any object which is a term, an atom, a clause, a goal.

Throughout the rest of this thesis, we will adopt the following *syntactical conventions for the notation of clauses and goals*:

- Variables are denoted by strings starting with a capital from the latter half of the alphabet.
- Constants, function symbols and predicate symbols are denoted by strings starting with a lower case character. Often $a, b, c, \ldots$ will be used for constants, $f, g, h, \ldots$ for function, and $p, q, r, \ldots$ for predicate symbols, occasionally enhanced with numerical indices, accents, etc..
- Lists, constructed through the use of the *list-forming functor* . will be represented in the conventional [|] Prolog notation, where [] denotes the empty list constant, nil. $[a, b|X] = .(a, .(b, X))$, $[a, b] = .(a, .(b, []))$ and $[X] = .(X, [])$ are some examples.

- Occasionally, we will need to name terms, atoms, clauses, etc.. Strings starting with a lower case character (usually $s, t, \ldots$) will normally refer to the former; Strings headed by a capital from the former half of the alphabet to atoms and clauses.

Next, a *substitution* θ is a (finite) mapping from (distinct) variables to terms, denoted as:

$$\theta = \{X_1/t_1, \ldots, X_n/t_n\}$$

where each $X_i \neq t_i$. A *ground* substitution is one that maps to ground terms. Substitutions can be applied to expressions. If E is an expression and θ a substitution, than $E\theta$ is the expression obtained by replacing domain variables of θ, occurring in E, by their corresponding term. We call $E\theta$ an *instance* of E. If E and F are expressions such that E is an instance of F and F is an instance of E, we call the two expressions *variants*. A substitution θ such that $E\theta$ is a variant of E is called a *renaming (substitution) for* E. Finally, if E is an expression and F an instance of E, than E is called *more general than* (or a generalisation of, or an anti-instance of) F. Substitutions can be composed:

Definition 2.2.1 Let $\theta = \{X_1/s_1, \ldots, X_m/s_m\}$ and $\sigma = \{Y_1/t_1, \ldots, Y_n/t_n\}$ be substitutions. Then the *composition* $\theta\sigma$ of θ and σ is the substitution obtained from the set

$$\{X_1/s_1\sigma, \ldots, X_m/s_m\sigma, Y_1/t_1, \ldots, Y_n/t_n\}$$

by deleting any binding $X_i/s_i\sigma$ for which $X_i = s_i\sigma$ and deleting any binding Y_j/t_j for which $Y_j \in \{X_1, \ldots, X_m\}$.

A particularly important operation on expressions is called *unification*. We only need unification of terms or atoms. Let us call expressions of the latter kind *simple*. And let us extend the applicability of a substitution to sets of expressions in the obvious way. Then:

Definition 2.2.2 Let S be a finite set of simple expressions. A substitution θ is called a *unifier* for S if $S\theta$ is a singleton. A unifier θ for S is called a *most general unifier* for S if, for each unifier σ of S, there exists a substitution γ such that $\sigma = \theta\gamma$.

If a set of expressions is *unifiable*, then its most general unifier (mgu) is unique (modulo renaming of its target terms). Unification *algorithms*, checking whether a given set of expressions is unifiable and, if so, computing its mgu, can be found in [110] and [5].

Next, an outstanding feature of logic is its *model theory*. It allows *reasoning about* truth in a theory, without actually performing any *reasoning within* the theory. (A delightful account of the historical background and the fundamental importance of model theory in mathematics can be found in [90].) Since a logic program can be understood as a logic theory, logic programming inherits this asset. Moreover, in a logic programming context, for most purposes it suffices to consider interpretations of a particularly simple kind, the domain of which coincides with the set of ground terms in the given language. They are called *Herbrand* interpretations. In such interpretations, a constant is interpreted as itself, a function symbol as a function of tuples of ground terms to ground terms, where the image of a given tuple is the term obtained by using the terms in the tuple as arguments for the given functor, and a predicate symbol as a relation on tuples of ground terms. It can be noted that a Herbrand interpretation (in a given language) is completely characterised by the set of its true ground atoms. The following result is basic for the semantics of logic programs:

Proposition 2.2.3 Let S be a set of program clauses and queries. Then S has a model iff S has a Herbrand model.

For a given collection of program clauses, and a query, we will be interested in finding instances of the query, inconsistent with the collection of clauses (i.e. its body literals are satisfied in any model of the clauses, i.e. logically implied by these clauses). The above proposition ensures that this question can be considered on the basis of Herbrand interpretations only.

We introduce some terminology. Let P be a set of program clauses in some language $\mathcal{L}_P$ (usually assumed to contain those constants, functors and predicate symbols which actually appear in P; see section 3.2), then:

- The *Herbrand universe*, U_P is the set of all ground terms in $\mathcal{L}_P$.
- The *Herbrand base*, B_P is the set of all ground atoms in $\mathcal{L}_P$.

Observe that any Herbrand interpretation can be identified with a subset of the Herbrand base, and vice versa, the atoms in the subset being true, and the rest false. We will make this identification throughout what follows. Then the following definition makes sense:

Definition 2.2.4 Let P be a set of program clauses and $H \subseteq B_P$, a Herbrand model of P. Then H is a *minimal* Herbrand model of P if there is no $H' \subset H$ which is also a model of P.

Clearly, a ground literal is a logical consequence of a set of program clauses P, iff it is true in every minimal Herbrand model of P.

2.3 Definite Logic Programs

A *definite program clause (or goal)* is a program clause (resp. goal) that does not contain any negative literals. A *definite logic program* is a finite set of definite program clauses. For such programs, there is a firmly established, and very appealing, semantical theory.

Proposition 2.3.1 Let P be a definite program and S_H a non-empty set of Herbrand models of P. Then the intersection of the elements in S_H is also a Herbrand model of P.

The set of *all* Herbrand models of a given definite program P is non-empty since B_P is always a model. The intersection of all P's Herbrand models is therefore again a model, denoted H_P. Obviously, it is the smallest, in the sense that it is a subset of all other Herbrand models and none of its (strict) subsets is a model. It is called the *least Herbrand model* of P. It is P's unique minimal model and decides on its own which ground literals are logical consequences of the program.

Theorem 2.3.2 Let P be a definite program. Then:

$H_P = \{A \in B_P | A \text{ is a logical consequence of } P\}$

The least Herbrand model of a definite program can therefore be considered as a declarative description of its meaning, independent from any operational issues. However, the above characterisations are not very constructive. Luckily, there is a much more convenient way to obtain a program's least Herbrand model than computing the intersection of all its Herbrand models. To this end, one associates a so-called *fixpoint operator* T_P with P. It maps Herbrand interpretations of P into Herbrand interpretations of P. If we denote by $\mathcal{P}(B_P)$ the set of subsets of P's Herbrand base, i.e. the set of P's Herbrand interpretations, a formal definition looks as follows:

Definition 2.3.3 Let P be a definite program. The mapping

$T_P : \mathcal{P}(B_P) \rightarrow \mathcal{P}(B_P)$

is defined as follows. Let $I \in \mathcal{P}(B_P)$, then:

$T_P(I) = \{A \in B_P | A \leftarrow B_1, \ldots, B_n \text{ is a ground instance of a clause in } P \text{ and } \{B_1, \ldots, B_n\} \subseteq I\}$

The next property provides one reason why T_P is interesting:

Proposition 2.3.4 Let P be a definite program and I a Herbrand interpretation of P. Then I is a model of P iff $T_P(I) \subseteq I$.

Clearly, T_P is monotonic with respect to the $\subseteq$-order on $\mathcal{P}(B_P)$. One is usually interested in the series of interpretations obtained through repeated application of T_P, starting from $\emptyset$. The interpretations thus computed are denoted by $T_P\!\uparrow\! m$, where m is some natural number, taking $T_P\!\uparrow\! 0 = \emptyset$, $T_P\!\uparrow\! 1 = T_P(\emptyset)$, $T_P\!\uparrow\! 2 = T_P(T_P\!\uparrow\! 1)$, etc.. One can prove that, for any definite program, the union of all these sets is the least fixpoint of T_P. In other words, it is the smallest set such that T_P maps it to itself. But this is exactly P's least Herbrand model. Formally:

Theorem 2.3.5 Let P be a definite program. Then $H_P = lfp(T_P) = \bigcup_{i<\omega} T_P \uparrow i$.

This union is also denoted as $T_P \uparrow \omega$ and it might or might not be equal to $T_P \uparrow n$ for some finite n (and any m larger than such n).

Next, we introduce a notion that will serve as a bridge to the procedural semantics of logic programming: correct answer substitutions.

Definition 2.3.6 Let P be a definite program and G a definite goal. An *answer* for $P \cup \{G\}$ is a substitution for variables of G.

Definition 2.3.7 Let P be a definite program, G a definite goal $\leftarrow A_1, \ldots, A_k$ and θ an answer for $P \cup \{G\}$. θ is called a *correct answer* for $P \cup \{G\}$ if $\forall((A_1 \& \ldots \& A_k)\theta)$ is a logical consequence of P.

Theorem 2.3.8 Let P be a definite program and G a definite goal $\leftarrow A_1, \ldots, A_k$. Suppose θ is an answer for $P \cup \{G\}$ such that $(A_1 \& \ldots \& A_k)\theta$ is ground. Then the following are equivalent:

- θ is correct.
- $(A_1 \& \ldots \& A_k)\theta$ is true in every Herbrand model of P.
- $(A_1 \& \ldots \& A_k)\theta$ is true in the least Herbrand model of P.

Let us now turn to the commonly used execution mechanism for definite logic programs. The material to be presented now is often termed the *procedural semantics* of (definite) logic programming. We first introduce the special form of resolution (called SLD-resolution) used to reason with definite logic programs.

Definition 2.3.9 Let G be $\leftarrow A_1, \ldots, A_m, \ldots, A_k$ and C be $A \leftarrow B_1, \ldots, B_q$. Then G' is *derived* from G and C using mgu θ if the following conditions hold:

1. A_m is an atom, called the *selected* atom in G
2. θ is an mgu of A_m and A
3. G' is the goal $\leftarrow (A_1, \ldots, A_{m-1}, B_1, \ldots, B_q, A_{m+1}, \ldots, A_k)\theta$.

In fact, in part II of this thesis, keeping a possible generalisation to normal programs in mind, we will usually speak about selected *literals*, rather than atoms.

Definition 2.3.10 Let P be a definite program and G a definite goal. An SLD-derivation of $P \cup \{G\}$ consists of a (finite or infinite) sequence $G_0 = G, G_1, \ldots$ of goals, a sequence $C_1, C_2, \ldots$ of variants of program clauses of P and a sequence $\theta_1, \theta_2, \ldots$ of mgu's such that each G_{i+1} is derived from G_i and C_{i+1} using θ_{i+1}.

SLD-derivations may be *finite* or *infinite*. A finite SLD-derivation may be *successful* or *failed*. A successful SLD-derivation (also called SLD-*refutation*) is one that ends in a goal without any literals. A failed SLD-derivation is one that ends in a non-empty goal with the property that the selected atom in this goal does not unify with the head of any program clause.

Now, we can define the following important concept:

Definition 2.3.11 Let P be a definite program and G a definite goal. A *computed answer* θ for $P \cup \{G\}$ is the substitution obtained by restricting the composition $\theta_1 \ldots \theta_n$ to the variables of G, where $\theta_1, \ldots, \theta_n$ is the sequence of mgu's used in an SLD-refutation of $P \cup \{G\}$.

As the term suggests, computed answer substitutions are what is actually "computed" when constructing SLD-derivations for a given program and query. Two theorems, fundamental in the theory of logic programming, assure that computed answers correspond nicely to correct answers.

Theorem 2.3.12 (soundness of SLD-resolution) Let P be a definite program and G a definite goal. Then every computed answer for $P \cup \{G\}$ is a correct answer for $P \cup \{G\}$.

Theorem 2.3.13 (completeness of SLD-resolution) Let P be a definite program and G a definite goal. For every correct answer θ for $P \cup \{G\}$, there exists a computed answer σ for $P \cup \{G\}$ and a substitution γ such that $\theta = \sigma\gamma$.

Against this background, the importance of results such as theorem 5.4.7 below, can be appreciated.

Let us finally introduce two representations of the overall reasoning process involved in computing answers for a given program and query, to be used frequently in part II of the thesis.

Definition 2.3.14 Let P be a definite program and G a definite goal. An *SLD-tree* for $P \cup \{G\}$ is a tree satisfying the following:

1. Each node of the tree is a (possibly empty) definite goal

2. The root node is G

3. Let $\leftarrow A_1, \ldots, A_m, \ldots, A_k$ $(k \geq 1)$ be a node in the tree and suppose that A_m is the selected atom. Then, for each (variant of a) clause $A \leftarrow B_1, \ldots, B_q$ in P such that A_m and A are unifiable with mgu θ, the node has a child:

$$\leftarrow (A_1, \ldots, A_{m-1}, B_1, \ldots, B_q, A_{m+1}, \ldots, A_k)\theta.$$

Obviously, the branches of an SLD-tree are SLD-derivations. When one or more branches are infinite derivations, the tree is called *infinite*, if there is none, *finite*. Observe that for a finite program P, a finite SLD-tree is also a finite tree, in the sense that its set of nodes is finite. This is clear, since the tree is finitely branching and therefore Königs lemma (see e.g. [58]) can be applied. Next, an SLD-tree all branches of which are failed derivations, is called *finitely failed*. It indicates that no answers can be found for the query which is its root. Finally, leaves of an SLD-tree are called *success* or *failure* nodes, depending on whether they terminate a successful or a failed derivation.

Definition 2.3.15 A *computation rule* is a function from a set of definite goals[1] to a set of atoms such that the value of the function for a goal is an atom, called the *selected* atom, in that goal.

Note that the shape of an SLD-tree (modulo permutations of its branches) is completely determined by a program, a goal and a computation rule.

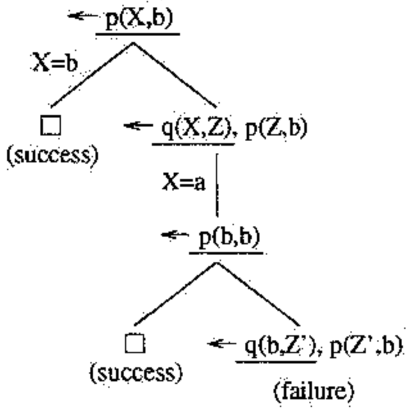


Figure 2.1: An SLD-tree.

Consider the following example, borrowed from [110]:

Example 2.3.16 Take the following program:

$p(X, X) \leftarrow$
$p(X, Y) \leftarrow q(X, Z), p(Z, Y)$
$q(a, b) \leftarrow$

and query:

$\leftarrow p(X, b)$

[1]Possibly augmented with a history.

Assuming a "choose the leftmost atom" computation rule, the resulting (finite, not finitely failed) SLD-tree is depicted in figure 2.1. Selected atoms are underlined, and branches are annotated with the necessary equalities to allow the reconstruction of computed answer substitutions.

We will also need the related notion of a *proof tree*. For a given program, query and computation rule, it can be constructed in parallel with the SLD-tree, and registers in finer detail how an atom, considered for a derivation step, derives from atoms considered earlier. The latter feature is the reason for its (implicit) use in this thesis. There are two different kinds of nodes, called and- and or-nodes respectively. These names refer to whether every or just one descending branch should be followed in order to find an answer for the given query. We will not include formal definitions, but simply reconsider example 2.3.16.

Example 2.3.17 The proof tree corresponding to the SLD-tree in figure 2.1 can be found in figure 2.2. And-nodes are indicated by dots and the links originating from them are joined by an arc to stress the fact that they should be considered together. Or-nodes can be enhanced by a label, indicating relevant variable bindings produced in other parts of the tree, before this particular node is considered. (In figure 2.2, there is actually one such or-node.) Observe how in figure 2.2, the two (unrelated) q atoms do not appear in the same branch of the tree, while they do in figure 2.1.

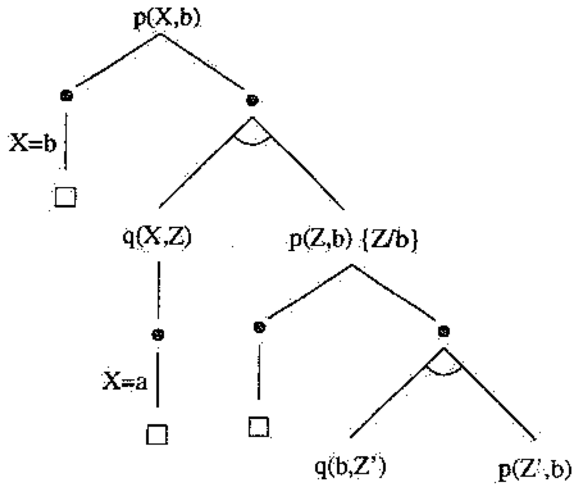


Figure 2.2: A proof tree.

2.4 Normal Logic Programs

If the restriction to positive literals in program clause bodies and goals is lifted, one speaks about *normal* program clauses, goals and programs. The possibility to add negations is not strictly necessary from a computational viewpoint (definite logic programming is Turing complete), but it adds greatly to the knowledge representation capabilities. However, there is a considerable price to pay.

Indeed, normal logic programs usually no longer have a unique (least) minimal Herbrand model, nor is their associated T_P-operator monotonic. (Iterative fixpoint computations are no longer possible.) However, it turns out that for a large class of programs, using negation in a restricted way, there is an obvious choice of *one particular model among its minimal Herbrand models*, that somehow reflects best the intended meaning of the program. These programs are called *stratified* and the preferred model their *perfect* Herbrand model. Moreover, perfect models *can* be characterised through an iterative construction. Thus, the perfect Herbrand model concept generalises the least Herbrand model one and inherits its agreeable properties.

We now include an extremely brief formal account of the stratification approach to the declarative semantics of normal logic programs, presenting required background material for part I of this thesis.

Let us first identify stratified programs.

Definition 2.4.1 The *dependency graph* $Dep(P)$ of a program P is defined as follows:

- Its vertices are the predicate symbols of P
- There is a positive (resp. negative) directed edge from q to p if there is a clause in P with p in its head and q in a positive (resp. negative) body literal.

One says that a predicate symbol p *depends positively* (resp. *negatively*) on a predicate symbol q if there is a path from q to p in $Dep(P)$ containing only positive edges (resp. at least one negative edge).

Definition 2.4.2 Let P be a normal program. We call a partition $P^1, \ldots, P^n$ of P's set of predicate symbols a *stratification* of P, if the following two conditions hold for $i = 1, \ldots, n$:

1. All predicate symbols on which the elements of P^i depend positively are contained in $\bigcup_{j \leq i} P^j$
2. All predicate symbols on which the elements of P^i depend negatively are contained in $\bigcup_{j < i} P^j$.

Observe that such a stratification exists iff $Dep(P)$ contains no cycles with a negative edge. In other words, no predicate should be defined (directly or indirectly) in terms of its own negation. If this is the case, P is called ***stratified*** and $P^1, \ldots, P^n$ its ***strata***. Usually, there is more than one stratification possible for a stratified program. Obviously, definite programs are stratified.

Now, suppose that P is stratified, then we can compute a model of P, proceeding stratumwise, starting from the first (bottom) stratum, and interpreting negative literals with respect to the already computed part of the model. It is exactly condition 2 in definition 2.4.2 which makes the latter possible.

Formally, one can first associate with a normal program P an (in general non-monotonic) *operator* T_P, through a straightforward generalisation of definition 2.3.3. Suppose now that P is a normal program and $I \subseteq B_P$. Then the following notation is introduced:

- $T_P{\Uparrow}0(I) = I$
- $T_P{\Uparrow}(n+1)(I) = T_P(T_P{\Uparrow}n(I)) \cup T_P{\Uparrow}n(I)$
- $T_P{\Uparrow}\omega(I) = \bigcup_{n<\omega} T_P{\Uparrow}n(I)$

Next, suppose that $P^1, \ldots, P^n$ is a stratification of the program P. Then one can consider normal programs $P_1, \ldots, P_n$, where each P_i contains precisely all clauses of P with head predicate symbol in P^i. Obviously, these are (possibly empty) normal programs, and one can define the following sequence of Herbrand interpretations:

$$H_1 = T_{P_1}{\Uparrow}\omega(\emptyset), \quad H_2 = T_{P_2}{\Uparrow}\omega(H_1), \quad \ldots, \quad H_n = T_{P_n}{\Uparrow}\omega(H_{n-1})$$

If one takes $H_P = H_n$, the following theorem can be proved:

Theorem 2.4.3 Let P be a stratified normal program and $P^1, \ldots, P^n$ a stratification of P. Let H_P be defined as above. Then the following hold:

1. H_P is a minimal Herbrand model of P.
2. H_P does not depend on the particular stratification of P, used in the above construction.

So, among the minimal Herbrand models of a stratified program P, there is one which seems particularly natural; It tends to minimise positive information on predicates in lower strata. We include one elementary example.

Example 2.4.4 Consider as P the following program, containing a single clause:

$p(a) \leftarrow not\ q(a)$

P has two minimal Herbrand models:

$\{p(a)\}$ and $\{q(a)\}$

H_P equals the first of these two.

One calls H_P the *perfect* Herbrand model of P. Note that perfect Herbrand models are a proper generalisation of least Herbrand models. Indeed, result 1 in theorem 2.4.3 implies that for a definite program, the notions of least and perfect Herbrand model coincide.

Perfect models have been defined for a wider class of programs. To this end, one does not stratify the set of predicate symbols in a program P, as in definition 2.4.2 above, but instead its Herbrand base, B_P. (Using a ground dependency graph, constructed on the basis of all possible ground instances of clauses in P. See definitions 3.3.1 and 3.3.2 in the next chapter.) Programs admitting such an operation are called *locally stratified*. Again a unique perfect model can be defined. Stratified programs are locally stratified and the two notions of perfect model coincide on this class of programs.

It is possible to take yet a further step, and construct the mentioned stratification of B_P *dynamically*, using information obtained in the already built part of the Herbrand model to delete irrelevant remaining clause instances. In this way, the class of *weakly* stratified programs is introduced, generalising locally stratified programs. Section 3.3 below deals in detail with weak stratification.

The *procedural* semantics of (locally) stratified normal logic programs has also been considered. A proof procedure, called *SLS-resolution* was proposed, and shown to be sound and complete with respect to the perfect model. However, this procedure can not be implemented (since it would incorporate a solution to the halting problem). A workable, sound approximation is *SLDNF-resolution*: an extension of SLD-resolution that allows dealing with *negation as failure*. Further details on these topics are not needed in the context of this thesis.

Much more can be said on the semantics of negation in logic programming. One approach to attach such semantics to (almost) *all* normal programs, in effect predating the stratification method, is called *completion* semantics. More recent research has, in various ways, aimed at *extending the preferred model idea to all normal logic programs*. And *unifying frameworks* for the resulting plethora of semantics have been proposed. Some of this work will be briefly referred to below, but since it is not immediately relevant to the bulk of this thesis, we leave it undiscussed here. Let us only mention that all semantics collapse into least Herbrand model semantics on the class of definite programs and that *all preferred model approaches coincide on weakly stratified programs*.

Part I

Why Untyped Non-Ground Meta-Programming is Not (Much of) a Problem

Chapter 3

Two Preliminary Concepts

3.1 Introduction

Since the appearance of [19] and [71] in [36], meta-programming has become increasingly important in logic programming and deductive databases. Applications in knowledge representation and reasoning, program transformation, synthesis and analysis, debugging and expert systems, the modeling of evaluation strategies, the specification and implementation of sophisticated optimisation techniques, the description of integrity constraint checking, etc. are constituting a significantly large part of the recent work in the field (see e.g. [18], [99], [161], [64], [159], [77], [158], [28], [164], [29]). A biennial, specialised workshop is dedicated to the subject, and its proceedings ([1], [24] and [130]) provide excellent reading material on foundations, implementational issues and various applications.

[82] and [160] were the first to seriously investigate theoretical foundations for meta-programming in logic programming. Particularly the ideas and results in [82] formed the starting point for the development of the novel logic programming language Gödel ([80]). Originally mounted as an attempt to take meta-programming seriously ([113]), the latter language now constitutes a full-fledged declarative successor to Prolog, providing extensive support for the sound development of further meta-programming applications.

As pointed out in chapter 1 however, it was not the sound semantics for meta-programming, nor the existence of Gödel, that attracted so much interest into meta-programming in logic programming to start with. (Although they have clearly accelerated the activity in the area). Indeed, we already mentioned that one main motivation certainly is the desire to extend the expressiveness of Horn clause logic augmented with negation as failure. Meta-programming adds extra knowledge representation and reasoning facilities ([99]). A second attraction is related to practicality. In applicative languages (both pure functional and pure logical), data and programs are syntactically indistinguishable. This is an open invitation to writing programs that take other programs as input. We believe that the practical success of, in particular, untyped vanilla-type meta-programming has resulted from this. However, in spite of this success, little or no effort was made to provide it with a sensible semantics in the usual framework of untyped Herbrand interpretations. Doing just this, is the main motivation for the work presented in this part of the thesis.

In [82], the possibility of providing such a declarative semantics is rejected immediately, on the basis that the intended interpretations of vanilla meta-programs can never be models in such a framework. Now, this statement seems somewhat inaccurate. The intended *meaning* of a vanilla-type meta-theory (in which different variables range over different domains) can simply not be captured within the formal notion of an interpretation, as it is defined for untyped, first order logic. So, a more precise statement would be that *the intended meaning can not be formalised as an untyped interpretation.* However, this problem is not typical for untyped vanilla programs; it generally appears in the semantics of most untyped logic programs. Indeed, any such program in which a functor is used to represent a partial function suffers from the same semantical problem and, in practice, total functions seldom occur in real applications. (See [47] for a thorough discussion of this issue.)

Whether this and other arguments ([112]) in favour of typed logic programs ([131]) should convince us to abandon the notational simplicity of untyped logic programs all together, is an issue we will not address.

From here on, we will assume that the semantics of an (untyped) program is captured by the alternative notion of its (least/perfect/well-founded/...) Herbrand model, avoiding the problems with intended interpretations. Even in this more restricted context, problems with the semantics of untyped vanilla meta-programs are present. Consider the (definite) object program P:

$p(X) \leftarrow$
$q(a) \leftarrow$

Let M denote the standard (definite) *solve* interpreter:

$solve(empty) \leftarrow$
$solve(X\&Y) \leftarrow solve(X), solve(Y)$
$solve(X) \leftarrow clause(X,Y), solve(Y)$

In addition, let M_P denote the program M augmented with the following facts:

$clause(p(X), empty) \leftarrow$
$clause(q(a), empty) \leftarrow$

Although the least Herbrand model of our object program is $\{p(a), q(a)\}$, the least Herbrand model of the meta-program M_P contains completely unrelated atoms, such as *solve*(*p*(*empty*)), *solve*(*p*(*q*(*a*))), etc..

This is certainly undesirable, since we, in general, would like at least that the atoms of the form *solve*($p(t)$) in the least Herbrand model of M_P correspond in a one-to-one way with the atoms of the form $p(t)$ in P's least Herbrand model.

Our main result shows that for a certain, important class of object programs, this property is indeed satisfied. That class of programs is characterised by the *language independence* condition, introduced below. However, we have judged it appropriate not to restrict our treatment to definite programs, but to widen our perspective so as to also consider stratified object programs and their associated meta-programs. Such meta-programs turn out not to be stratified themselves, but they do satisfy the required conditions for the broader notion of *weak stratification*. So, we can consider their weakly perfect Herbrand model and compare it with the perfect Herbrand model of the underlying object program. Again, language independence is the key to good results.

In chapter 4, we prove that the perfect Herbrand model of a stratified, language independent object program corresponds in a one-to-one way with a natural subset of its meta-theory's weakly perfect Herbrand model. In addition, we show how our approach can be extended to provide a semantics for various related meta-programs, including a limited form of amalgamation. Moreover, we demonstrate that the language independence condition can often be skipped in a semantics that reflects more closely the program's operational behaviour. But first, in the next two sections, constituting the main body of this preliminary chapter, we introduce and discuss the notions of language independence and weak stratification respectively.

Finally, a condensed version of both this and the next chapter can be found in [121].

3.2 Language Independence

The intuition behind *language independence* is simple. In the logic programming community, there seems to be a broad agreement that the declarative semantics of most or all logic programs can be described by a particular Herbrand model of the program[1]. Two well-known classes of programs whose semantics is the subject of little or no controversy[2] are *definite* programs with their *least* Herbrand model (see e.g. [170]) and stratified programs with their *perfect* Herbrand model (see e.g. [7], [139]).

[1] In the context of this work, we do not consider what has been termed "the universal query problem". See e.g. [141] and [147].

[2] See however section 4.6.

In general however, these models depend on the language in which we are considering the Herbrand models. A simple example suffices to show this. Consider the program consisting of the single clause $p(X) \leftarrow$. Its least Herbrand model (in fact, its only Herbrand model) in a language with one constant symbol a and no function symbols is $\{p(a)\}$. If we however add the constant symbol b, we obtain $\{p(a), p(b)\}$. Basically, we will call a program *language independent* when its characteristic Herbrand model does not depend on the particular choice of language.

A formal introduction and characterisation of the notion for stratified programs follows below. Some comments and results on language independence for other classes of programs are included at the end of the section.

3.2.1 Language independent stratified programs

Suppose P is a logic program. Let $\mathcal{R}_P$, $\mathcal{F}_P$ and $\mathcal{C}_P$ denote the sets of predicate, function and constant symbols occurring in the program. We can then consider a first order language $\mathcal{L}_P$, which has exactly $\mathcal{R}_P$ and $\mathcal{F}_P$ as its sets of predicate and function symbols, respectively. As its set of constant symbols, we take $\mathcal{C}_P$ if it is *not empty* and $\{*\}$, a set with a single arbitrary element $*$, if it is. $\mathcal{L}_P$ is called the *language underlying the program* P. Although this is not imposed as a limitation in e.g. [110], Herbrand interpretations of the program are usually constructed with this underlying language in mind. For our purposes in this work, however, we need more flexibility. We therefore introduce the following two definitions.

Definition 3.2.1 Let P be a normal program with underlying language $\mathcal{L}_P$. We call a language $\mathcal{L}'$, determined by $\mathcal{R}'$, $\mathcal{F}'$ and $\mathcal{C}'$ an *extension* of $\mathcal{L}_P$ iff $\mathcal{R}_P \subseteq \mathcal{R}'$, $\mathcal{F}_P \subseteq \mathcal{F}'$ and $\mathcal{C}_P \subseteq \mathcal{C}' \neq \emptyset$.

Notice that if $\mathcal{C}_P$ is empty, $\mathcal{C}'$ may be any non-empty set of constant symbols. In particular, it does not have to contain $*$. The following definition makes explicit the language in which Herbrand interpretations are constructed.

Definition 3.2.2 Let P be a normal program with underlying language $\mathcal{L}_P$. A Herbrand interpretation of P in a language $\mathcal{L}'$, extension of $\mathcal{L}_P$, is called an *$\mathcal{L}'$-Herbrand interpretation* of P.

In the sequel, we will often refer to *Herbrand* interpretations and models of a program P with underlying language $\mathcal{L}_P$, when in fact, we mean *$\mathcal{L}_P$-Herbrand* interpretations or models.

We are now in a position to introduce the notion of *language independence*.

Definition 3.2.3 A stratified program P with underlying language $\mathcal{L}_P$ is called *language independent* iff for any extension $\mathcal{L}'$ of $\mathcal{L}_P$, its perfect $\mathcal{L}'$-Herbrand model is equal to its perfect $\mathcal{L}_P$-Herbrand model.

Notice that this definition entails that no atom in any perfect Herbrand model can contain any predicate, function and/or constant symbols not occurring in P. In particular, when C_P is empty, any perfect Herbrand model can only contain propositions. (This observation will be used implicitly in the sequel. It ensures that the infamous $*$ constant does not cause too much trouble. See also the remark following proposition 4.2.9.)

We illustrate definition 3.2.3 with some simple examples.

Example 3.2.4 Of the following programs, P_1 and P_2 are *not* language independent, while P_3 and P_4 are.

$P_1 :\ p(X) \leftarrow$
$P_2 :\ p(X) \leftarrow not\ q(X)$
$\quad\ q(a) \leftarrow$
$P_3 :\ p(X) \leftarrow r(X), not\ q(X)$
$\quad\ r(a) \leftarrow$
$P_4 :\ p(X, Y) \leftarrow r(X), not\ q(X)$
$\quad\ q(X) \leftarrow h(X)$
$\quad\ h(a) \leftarrow$
$\quad\ r(a) \leftarrow$

It is clear that language independence, introduced here as a concept tuned towards the semantics of logic programs, is strongly related to *domain independence*. The latter concept was defined for any formula in the context of full first order logic (see e.g. [45] and further references given there). The following example shows that the two notions do *not* coincide.

Example 3.2.5

$p(X) \leftarrow q(X), not\ r(X, Y)$
$q(a) \leftarrow$
$r(a, a) \leftarrow$
$s(b) \leftarrow$

Indeed, this program is language independent. But there are (non-Herbrand) models, having only one element in their domain of interpretation on which both a and b are mapped, in which $p(a)$ is *not* true. It therefore is *not* domain independent.

However, it seems that, when restricted to Herbrand interpretations (with free equality), both notions *do* coincide. And indeed, the concept of *domain independent databases* (*function-free*, with equality) as introduced in [163], captures the same basic property as our definition of *language independent programs* (see lemma 3 on page 229 in [163]). We feel, however, that the above introduced terminology better reflects the underlying intuition. We should point out that [163] contains an extensive discussion on the relation between *domain independence* and *allowedness*, a notion similar to the notion of *range restriction*, introduced below. Moreover, it is argued extensively that non-domain independent databases are unreasonable and should be avoided. Finally, it is shown that for *function free*, stratified, normal programs, domain independence (or, rather, language independence) is *decidable*.

In general, however, it is clear that, like domain independence for full first order logic (see e.g. theorem 2 on page 224 in [163] and further references given there), language independence is an undecidable property. To see this, observe that checking language independence boils down to checking refutability of goals[3]. It is therefore important to investigate the existence of syntactically recognisable, and thus decidable, subclasses of the class of language independent programs. It turns out that the well-known concept of *range restriction* determines such a class.

Let us first repeat its definition, specialised to the context of normal logic programs.

Definition 3.2.6 A clause in a program P is called *range restricted* iff every variable in the clause appears in a positive body-literal.
A program P is called *range restricted* iff all its clauses are range restricted.

It is obvious that *range restriction* is a syntactic property. It has been defined for more general formulas and/or programs and was used in other contexts. See e.g. [128] and [30] for its use in the context of integrity checking in relational and deductive databases. Two related notions are *safety*, used by Ullman in [167] and *allowedness*, defined in [110] and important for avoiding floundering of negative goals in SLDNF. The limitation to range restricted programs is natural in many contexts. Moreover, it can be noted that [119] includes a general method to transform a non range restricted program P into a range restricted program P' through the addition of $dom(X)$ calls to the bodies of clauses for every variable X which is not "restricted". dom itself is defined to hold for every ground term in the Herbrand universe of the given language via a range restricted definite program. Obviously, this transformation preserves stratification. Moreover, there is a one-to-one correspondence between Herbrand models of P and P' such that they coincide for all predicates in P.

The following proposition shows that this important class of logic programs is a subclass of the language independent ones.

Proposition 3.2.7 Let P be a stratified program. If P is range restricted then P is language independent.

[3]This observation was first made by an anonymous referee of [122]. More comments on the impossibility of finding an effective and always terminating query answering procedure for perfect Herbrand model semantics can e.g. be found in [141].

Proof Let $\mathcal{L}_P$ be the underlying language of P and let $\mathcal{L}'$ be an extension of $\mathcal{L}_P$. We prove that for any predicate p, the perfect $\mathcal{L}_P$-Herbrand model of P and its perfect $\mathcal{L}'$-Herbrand model coincide. The proof proceeds through induction on the strata of P. Suppose first that p is defined in the bottom stratum P_b. P_b is a definite program. Let T_{P_b} be the immediate consequence operator applied in the context of $\mathcal{L}_P$ and T'_{P_b} the corresponding operator applied in the context of $\mathcal{L}'$. We prove that for each $n \geq 0 : T_{P_b}\uparrow n = T'_{P_b}\uparrow n$.
The proof is through induction on n. The base case is trivial. Furthermore, if the (positive) body literals in a ground instance of a range restricted clause are instantiated with terms in $\mathcal{L}_P$, then so is the head. The induction step now follows immediately.
Next, suppose p is defined in the ith stratum $(i > 1)$ of P. Then an argument similar to the one above, and considering the fact that also all negative body literals will be instantiated with terms in $\mathcal{L}_P$, shows that application of the fixpoint operator gives identical results in the context of $\mathcal{L}_P$ and $\mathcal{L}'$. The result follows. □

Example 3.2.8 It can be noted that of the programs in example 3.2.4, only P_3 is range restricted. In particular, P_4 presents a simple case of a program that is language independent, but *not* range restricted. The Y in the head of its first clause does not constitute a problem, because the body of that clause can not be satisfied and therefore the perfect Herbrand model of P_4 remains the same in any language.

From this example, one is tempted to infer that the difference between language independence and range restriction lies largely in the fact that the first notion allows the presence of some non range restricted, but in the present program also "non applicable" clauses like the first one of program P_4. However, the situation is more complex, as the following example programs show.

Example 3.2.9 Of the programs below, none is range restricted, but all except P_8 are language independent.

$$\begin{aligned}
P_5 :\ & p(X,Y) \leftarrow q(X) \\
P_6 :\ & p(X,Y) \leftarrow q(X), \mathit{not}\ r(X) \\
& r(X) \leftarrow q(X) \\
& q(a) \leftarrow \\
P_7 :\ & p(X) \leftarrow q(X), \mathit{not}\ r(X,Y) \\
& q(a) \leftarrow \\
P_8 :\ & p(X) \leftarrow q(X), \mathit{not}\ r(X,Y) \\
& q(a) \leftarrow \\
& r(a,a) \leftarrow
\end{aligned}$$

It can be noted that P_7 *is* language independent, exactly because all conditions in the only ground instance of its non range restricted rule are ***satisfied*** in the given program's perfect Herbrand model. For P_8, the opposite holds.

For a more precise result on the relation between language independence and range restriction in the context of definite programs, we refer to proposition 3.2.10.

Readers who judge the difference between the two notions insignificant are free to substitute "range restricted" for "language independent" throughout most what follows. They may however observe that the notion of language independence renders more elegant most proofs in the next chapter.

3.2.2 Language independence for other classes of programs

Above, we have introduced the notion of language independence for ***stratified*** programs. It is obvious that this generalises the concept of language independence for ***definite*** programs, based on the invariance of the *least* Herbrand model, as it was defined in [41]. It is also obvious that on the class of definite programs, both definitions coincide.

However, for definite programs, the difference between language independence and range restriction is more easily characterisable, as the following proposition shows[4].

Proposition 3.2.10 Let P be a definite program. Then P is language independent iff all ground ($\mathcal{L}_P$)-instances of every non range restricted clause in P contain at least one body atom, not true in its least Herbrand model.

Proof First suppose there is a non range restricted clause that does not satisfy the stated condition. It immediately follows that P is not language independent. This proves the ***only-if***-part.
The proof of the ***if***-part is completely analogous to the reasoning on the bottom layer in the proof of proposition 3.2.7. □

Notice that this result still leaves language independence as an undecidable property.

We conclude our reflections on definite, language independent programs with the following proposition.

Proposition 3.2.11 Let P be a definite program. Then P is language independent iff for any definite goal G, all (SLD-)computed answers for $P \cup \{G\}$ are ground.

[4]This observation is due to an anonymous referee of [41].

Proof We first prove the *only-if*-part. Suppose there is a goal $\leftarrow A_1, \ldots, A_k$ such that θ is a non-ground computed answer for $P \cup \{\leftarrow A_1, \ldots, A_k\}$. Then there must be at least one $A_i, 1 \leq i \leq k$ such that $\forall \overline{x} A_i \theta$ is a logical consequence of P (here $\overline{x}$ represents the *non-empty* set of variables, free in $A_i\theta$). In particular, $\forall \overline{x} A_i \theta$ must be satisfied in any least Herbrand model. This implies that P is *not* language independent.
For the *if*-part, first observe that for any particular goal, computed answers are not affected by language extensions. Suppose now that all computed answers to all goals are ground. This means that in particular all computed answers to any goal G where each argument is an uninstantiated variable, are ground. From the completeness of SLD-resolution (theorem 8.6 in [110]), it now follows that the set of computed answers for $P \cup \{G\}$ is exactly equal to the set of correct answers for $P \cup \{G\}$. We can conclude that P is language independent. □

Proposition 3.2.11 provides yet another characterisation of the class of definite, language independent programs. We will find occasion to apply it in section 4.6.

The notion of language independence can also be considered for classes, broader than the one of stratified programs. A generalisation to *locally* stratified programs is straightforward. And also *weakly* stratified programs with their *weakly* perfect Herbrand model (introduced in the next section) allow an obvious adaptation of definition 3.2.3 and proposition 3.2.7. A complete analysis of its usefulness in the context of different classes of logic programs and their semantics is outside the scope of the present work. Notice however, that in those semantics in which the "preferred" Herbrand model is three-valued (e.g. well-founded semantics, [172]), one will demand only the positive information in the model to be invariant under language extensions. Indeed, it is obvious that the negative information will be affected for almost all reasonable programs and semantics. We return briefly to this issue in section 4.9.

3.3 Weak Stratification

In section 3.1, we pointed out that, in general, there is a problem with the least Herbrand model of vanilla meta-programs. In the next chapter, we show that for language independent programs, this problem disappears. However, as was mentioned above, we do not wish to restrict our development to definite programs. In particular, we would like to consider stratified object and meta-programs and compare their perfect Herbrand models.

But in this context, another difficulty has to be addressed first. We illustrate it through the following example. Consider the stratified, language independent object program P:

$p(X) \leftarrow r(X), not\ q(X)$
$r(a) \leftarrow$

Let M denote the standard (normal) *solve* interpreter:

$solve(empty) \leftarrow$
$solve(X\&Y) \leftarrow solve(X), solve(Y)$
$solve(\neg X) \leftarrow not\ solve(X) \qquad (i)$
$solve(X) \leftarrow clause(X,Y), solve(Y) \qquad (ii)$

In addition, let M_P denote the program M augmented with the following facts:

$clause(p(X), r(X)\&\neg q(X)) \leftarrow$
$clause(r(a), empty) \leftarrow$

Obviously, M_P is *not* stratified. Moreover, it is not even *locally* stratified. To see this, consider clause (*ii*) of M_P. For any two ground atoms, $solve(t_1)$ and $solve(t_2)$ in the Herbrand base for the language underlying M_P, we have that both

$solve(t_1) \leftarrow clause(t_1, t_2), solve(t_2)$

and

$solve(t_2) \leftarrow clause(t_2, t_1), solve(t_1)$

are ground instances of (*ii*). Therefore, in any local stratification of M_P, all the ground atoms of the form $solve(t)$ must be in the same stratum. On the other hand, by clause (*i*), any ground atom $solve(\neg t)$ must be placed in a higher stratum than the corresponding atom $solve(t)$. So no local stratification can be possible.

However, there is a simple way to overcome this problem. Consider the new theory, M'_P, obtained from M_P by performing one unfolding step of the atom $clause(X,Y)$ in clause (*ii*), using every available *clause*-fact of M_P. Clause (*ii*) is replaced by the resultants:

$solve(p(X)) \leftarrow solve(r(X)\&\neg q(X))$
$solve(r(a)) \leftarrow solve(empty)$

One can easily verify that M'_P *is* locally stratified. It can be shown that for any stratified object program P, the program obtained from its associated vanilla meta-program through a similar unfolding transformation, is locally stratified. In [136] and [138], Przymusinska and Przymusinski introduced *weakly* stratified logic programs and their unique *weakly* perfect Herbrand model. Now, from their definitions, it follows that programs which can be unfolded into a locally stratified one, are weakly stratified. It can therefore be shown that a stratified object program gives rise to a weakly stratified vanilla meta-program. This allows us to consider the weakly perfect Herbrand model of the latter as the description of its semantics.

So, before we actually embark on a study of meta-program semantics, we devote the next subsection to a formal introduction of *weakly stratified programs* and *weakly perfect models*. Moreover, it turns out that we do not need the fully general concepts in our restricted context. We will therefore derive a more easily verifiable *sufficient* condition for weak stratification, to be used throughout chapter 4.

3.3.1 Weakly stratified programs and weakly perfect models

Weakly stratified programs and weakly perfect models were first introduced in [136]. However, that paper is restricted to the case of *function-free* (so-called datalog) programs. Obviously, our needs surpass that limitation: the example meta-programs above clearly contain functors. Since the necessary generalisations are not completely straightforward, we turn to the fully general treatment in [138]. And since both weak stratification and weakly perfect models are not (yet?) fully standard concepts in logic programming, we include the relevant parts of their definition. Finally, it can be pointed out that the overview paper [137] also has a section on weak stratification and weakly perfect models. The presentation there is essentially the same, in spite of the fact that some of its basic definitions are chosen differently.

We first introduce both concepts in an informal way, and then give the (somewhat more general) formal development.

To decide whether a normal logic program P is *weakly stratified* and if so, to determine its *weakly perfect* Herbrand model, one basically proceeds as follows. Consider all ground instances of clauses in P. Consider the set A of ground atoms that do not depend negatively on other atoms (either directly or indirectly). If A is empty, then the program is *not* weakly stratified and the construction fails. Otherwise, the clause instances whose head is in A constitute a definite logic program only containing atoms in the chosen set. Compute the least Herbrand model H of this (definite) partial program. Consider now the clause instances left. Eliminate clause instances with a body literal such that its atom belongs to A, when the literal is not satisfied according to H. Simplify the remaining clause instances by deleting condition literals with atoms in A, satisfied according to H. Repeat the above construction on this new set of ground clauses. If this construction can be carried out (possibly transfinitely) until no clause instances are left, take the union of the computed least Herbrand models. This is the *weakly perfect* Herbrand model of P.

The following formal development is abridged (and slightly adapted) from [138]. For further details and examples, we refer to that paper.

Definition 3.3.1 Let P be a normal program. We denote by $Ground(P)$ the (possibly infinite) set of ground instances of clauses in P and we call it the *ground program associated with* P.

In the sequel, we will often apply notions defined for (finite) programs in the context of infinite ground programs. The generalisation of the "classical" definitions is straightforward. As usual, we use the notation B_P to denote the *Herbrand base* for the language underlying a program P.

Definition 3.3.2 The *ground dependency graph* $Depg(P)$ of a program P is defined as follows:

- Its vertices are the atoms in B_P.
- There is a positive directed edge from A to B if $Ground(P)$ contains a clause $B \leftarrow \ldots, A, \ldots$.
- There is a negative directed edge from A to B if $Ground(P)$ contains a clause $B \leftarrow \ldots, not A, \ldots$.

Then we define the following relations between atoms in B_P:

Definition 3.3.3

- $A \leq B$ iff there is a directed path from A to B in $Depg(P)$.
- $A < B$ iff there is a directed path from A to B in $Depg(P)$ passing through a negative edge.
- $A \sim B$ iff $(A = B) \vee (A < B \wedge B < A)$.

$\sim$ is an equivalence relation on B_P, the equivalence classes of which we will call *components* of B_P. We can define a partial order between these components.

Definition 3.3.4 Suppose C_1 and C_2 are two components of B_P. We define:

$C_1 \prec C_2$ iff $C_1 \neq C_2 \wedge \exists A_1 \in C_1, \exists A_2 \in C_2 (A_1 < A_2)$

A component C_1 is called *minimal*, iff there is no component C_2 such that $C_2 \prec C_1$.

Definition 3.3.5 By the *bottom stratum* $S(P)$ of a ground program P, we mean the union of all minimal components of P, i.e.

$S(P) = \bigcup\{C | C$ is a minimal component of $B_P\}$.

By the *bottom layer* $L(P)$ of a ground program P, we mean the corresponding subprogram of P, i.e.

$L(P) =$ the set of all clauses from P, whose heads belong to $S(P)$.

In the context of the ensuing construction, (possibly partial, three-valued) interpretations and models that explicitly register negative information will be used. They contain not only positive, but also negative ground literals. $Pos(I)$ will denote the positive subset of such an interpretation. It is of course itself a standard (two-valued) interpretation. In the next definition, as usual, we assume $not(not L) = L$ for any ground literal.

Definition 3.3.6 Let P be a ground program of which I is a partial interpretation. By a *reduction of P modulo I*, we mean a new (ground) program $\frac{P}{I}$ obtained from P by performing the following two reductions:

- removing from P all clauses which contain a premise L such that $notL \in I$ or whose head $\in I$ (i.e. clauses true in I)
- removing from the remaining clauses all premises $L \in I$.

Finally, remove all non-unit clauses whose heads appear as unit clauses.

We can now describe the (transfinite) iteration process that leads to the construction of a *weakly* perfect model. (The union of three-valued interpretations, used in definition 3.3.7, is defined as straightforward set union. Since the S_α sets are disjunct, the corresponding H_α models can not contain contradictory information and their union is well-defined.)

Definition 3.3.7 Suppose that P is a logic program and let $P_0 = Ground(P)$, $H_0 = \emptyset$. Suppose that $\alpha > 0$ is a countable ordinal such that programs P_δ and partial interpretations H_δ have been already defined for all $\delta < \alpha$. Let

$$N_\alpha = \bigcup_{\delta<\alpha} H_\delta, \quad P_\alpha = \frac{P_0}{N_\alpha}, \quad S_\alpha = S(P_\alpha), \quad L_\alpha = L(P_\alpha)$$

- If the program P_α is empty, then the construction stops and $H_P = Pos(N_\alpha)$ is the *weakly perfect Herbrand model* of P.
- Otherwise, if $S_\alpha = \emptyset$ or if L_α has no least Herbrand model, then the construction also stops. (P has no weakly perfect Herbrand model.)
- Otherwise, H_α is defined as the least Herbrand model of L_α and the construction continues.

Finally, we define *weakly stratified* programs.

Definition 3.3.8 We say that a program P is *weakly stratified* if it has a weakly perfect model and all L_α are definite.

The following result is immediate.

Proposition 3.3.9 Every (locally) stratified program is weakly stratified and its perfect and weakly perfect Herbrand model coincide.

Proof Immediate from theorem 4.1 and corollary 4.5 in [138]. □

Thus we see that weak stratification is a conservative extension of (local) stratification. It can even be argued that weak stratification is in many ways a more suitable extension of stratification than local stratification is. Indeed, the latter property is not invariant under some elementary program transformations, as was shown above.

In chapter 4, we show that meta-programs associated with stratified object programs are indeed weakly stratified. However, a number of details in the above general construction are rather inconvenient in that restricted context.

- We want to avoid the use of three-valued interpretations, even during construction of the (two-valued) weakly perfect model.
- We will not need transfinite induction.
- We are only interested in weakly perfect models for programs that are actually weakly stratified.
- Definition 3.3.7 takes S_α always equal to the entire union of minimal components of B_{P_α}. In other words, the weak stratification built is as "tight" as possible. Following this practice would considerably damage the elegance of the proofs in the next chapter. We need the ability to only consider "safe" subsets of that union.
- Finally, we will not remove non-unit clauses whose heads appear as unit clauses, as stipulated in definition 3.3.6.

For all these reasons, we present a modified construction procedure in proposition 3.3.11. Successful application of this procedure guarantees weak stratification and constructs the weakly perfect model. In other words, we show that proposition 3.3.11 provides a *sufficient* condition for weak stratification. A condition which will then actually be used throughout the rest of our work. Some simple programs, not satisfying it, but nevertheless weakly stratified according to definitions 3.3.7 and 3.3.8 are included among the examples below. Before we can state proposition 3.3.11, we need one more auxiliary concept.

Definition 3.3.10 If P is a ground program, then we denote by $CH(P)$ the set of all heads of clauses in P.

Proposition 3.3.11 Let P be a normal program. Then we define the following:

- $P_1 = Ground(P)$

If $P_i \neq \emptyset$ is defined and B_{P_i} has one or more minimal components, we define:

- $S_i = S(P_i)$
- V_i: a non-empty subset of S_i such that if $B \in V_i$ and $A \leq B$ (and hence $A \in S_i$) then $A \in V_i$

- L_i: the set of clauses in P_i whose head is in V_i
- H_i: the least Herbrand model of L_i if L_i is definite

If H_i is defined, we define the following:

- P'_{i+1}: the set of all clauses in $P_i \setminus L_i$ for which the following holds:
  - for every positive body literal B, $B \in V_i \Rightarrow B \in H_i$
  - for every negative body literal $notB$, $B \in V_i \Rightarrow B \notin H_i$
- P_{i+1}: P'_{i+1} with the V_i body literals deleted from the clauses

If there is an i such that $P_i = \emptyset$, then take $H_P = \bigcup_{j<i} H_j$.
Else if H_j is defined for all $j < \omega$ and $\bigcap_{j<\omega} CH(P_j) = \emptyset$,
then take $H_P = \bigcup_{j<\omega} H_j$.
In both cases, P is weakly stratified and H_P is its weakly perfect Herbrand model.

Proof First, it can be verified that our use of two-valued interpretations is a correct recasting of definition 3.3.6 and its use in definition 3.3.7. Not removing non-unit clauses in the presence of corresponding unit clauses, might lead to failure (see also example 3.3.20 and the comment that follows it), but does not influence H_P upon success. A detailed formal proof of this point can proceed in a similar way to the proof for lemma 3.3.12 below. Furthermore, the definition of the H_i now incorporates the condition that each L_i be definite, and P therefore weakly stratified. Next, note that $\bigcap_{j<\omega} CH(P_j) = \emptyset$ means that all clauses in *Ground*(P) are consumed during the iteration process, and therefore halting successfully is correct. It remains to be shown that constructing a non-tight stratification is safe. In other words: that any choice of the V_i, leading to a successful halt, returns the same H_P. (The fact that some choices might lead to success, while others might not, is not a problem, since we are only claiming a sufficient condition.) This follows from lemma 3.3.12 below. □

Lemma 3.3.12 Let P be a normal program such that a series $V_1, \ldots$ leads to successful termination of the construction in proposition 3.3.11 with resulting model ${H_P}^V$. Then the (maximal choice) series $S_1, \ldots$ also terminates successfully with ${H_P}^S = {H_P}^V$.

Proof The proof of this lemma can be found in appendix A. □

Let us now briefly turn to some examples. The first one is taken from [138], where it was borrowed from [72].

Example 3.3.13
Let P be the following program:

$p(1,2) \leftarrow$
$q(X) \leftarrow p(X,Y), not\ q(Y)$

P is neither stratified, nor locally stratified. It is however weakly stratified:

$P_1 : p(1,2) \leftarrow$
$q(1) \leftarrow p(1,1), not\ q(1)$
$q(1) \leftarrow p(1,2), not\ q(2)$
$q(2) \leftarrow p(2,1), not\ q(1)$
$q(2) \leftarrow p(2,2), not\ q(2)$
$S_1 = \{p(1,1), p(1,2), p(2,1), p(2,2)\}$
$V_1 = S_1$
$L_1 = \{p(1,2) \leftarrow\}$
$H_1 = \{p(1,2)\}$
$P'_2 = \{q(1) \leftarrow p(1,2), not\ q(2)\}$
$P_2 = \{q(1) \leftarrow not\ q(2)\}$
$S_2 = \{q(2)\}$
$V_2 = \{q(2)\}$
$L_2 = \emptyset$
$H_2 = \emptyset$
$P'_3 = \{q(1) \leftarrow not\ q(2)\}$
$P_3 = \{q(1) \leftarrow\}$
$S_3 = \{q(1)\}$
$V_3 = \{q(1)\}$
$L_3 = \{q(1) \leftarrow\}$
$H_3 = \{q(1)\}$
$P'_4 = \emptyset = P_4$
P's weakly perfect Herbrand model $H_P = \bigcup_{j<4} H_j = \{p(1,2), q(1)\}$

The following well-known example is often presented as a motivation for extending the concept of stratification to *local* stratification (see e.g. [139]). We demonstrate that it is also *weakly* stratified.

Example 3.3.14
Let P name the following program:

$even(0) \leftarrow$
$even(s(X)) \leftarrow not\ even(X)$

$P_1 = \{even(0) \leftarrow\} \cup \{even(s^{n+1}(0)) \leftarrow not\ even(s^n(0)) | n \geq 0\}$
$S_1 = \{even(0)\}$
$V_1 = S_1$
$L_1 = \{even(0) \leftarrow\}$

$H_1 = \{even(0)\}$
$P_2' = \{even(s^{n+1}(0)) \leftarrow not\ even(s^n(0)) | n \geq 1\}$
$P_2 = P_2'$
$S_2 = \{even(s(0))\}$
$V_2 = S_2$
$L_2 = \emptyset$
$H_2 = \emptyset$
$P_3' = P_2$
$P_3 = \{even(s(s(0))) \leftarrow\} \cup \{even(s^{n+1}(0)) \leftarrow not\ even(s^n(0)) | n \geq 2\}$
$S_3 = \{even(s(s(0)))\}$
$V_3 = S_3$
$L_3 = \{even(s(s(0))) \leftarrow\}$
$H_3 = \{even(s(s(0)))\}$
$\vdots$
$\bigcap_{j<\omega} CH(P_j) = \emptyset$

In this way, by constructing the successive V_i sets, we dynamically build a stratification of B_P, containing an infinite amount of strata, each with one single atom. Notice that this stratification is identical to the (most tight) *local* stratification of B_P. Put otherwise, *Ground*(P) can be split into an infinite amount of "layers", each containing one single ground clause instance, the head of which corresponds to the sole element of the respective V_i.

Example 3.3.15 The M_P example program on page 28 is also weakly stratified. *Ground*(M_P) contains a (countably) infinite amount of layers, each of which is composed of an infinite amount of ground clause instances. For more details, we refer to theorem 4.2.5 and its proof.

The program in the next example has no weakly perfect Herbrand model.

Example 3.3.16

$p \leftarrow not\ q$
$q \leftarrow not\ p$

Indeed, the complete program is contained in its single layer, which has two distinct minimal Herbrand models, and therefore no least.

And also the following program P has no weakly perfect Herbrand model.

Example 3.3.17

$even(0) \leftarrow$
$even(X) \leftarrow not\ even(s(X))$

The bottom stratum of *Ground*(P) is empty.

A small change to example 3.3.16 gives us a program that does have a weakly perfect Herbrand model, even though it is still not weakly stratified.

Example 3.3.18

$p \leftarrow q$
$q \leftarrow not\ p$

The modified program still consists of a single (non-definite) layer, which now however has a least Herbrand model: $\{p\}$.

Finally, we include two weakly stratified programs that do *not* satisfy the condition in proposition 3.3.11.

Example 3.3.19

P: $p(a) \leftarrow$
$p(f(X)) \leftarrow not\ p(X)$
$q(X) \leftarrow not\ p(X)$
$r(a) \leftarrow p(X), q(X)$
$r(f(X)) \leftarrow not\ r(X)$

Clearly, $\bigcap_{j<\omega} CH(P_j) \neq \emptyset$ and yet P is weakly stratified. However, establishing the latter fact requires transfinite induction.

Example 3.3.20

$p \leftarrow not\ q$
$p \leftarrow r$
$q \leftarrow not\ p$
$r \leftarrow$

Example 3.3.20 shows a case where the removal of clauses (with a non-empty body) having a head for which also a fact is present (see the bottom line in definition 3.3.6), makes a crucial difference. As a motivation for proceeding thus, the authors of [138] mention the desire to keep separated so-called *intensional* and *extensional* predicates. In the context of logic programming, this difference is however far less significant than in the context of deductive databases. And example 3.3.20 illustrates that the influence of this measure goes beyond the concern stated above. In general, it seems that one of its effects is incorporating a kind of "tie breaking" strategy, where definite information originating from another source is used to resolve conflicts resulting from mutual recursion through negation. It is up to the reader to decide whether programs like the one above should indeed be called "weakly stratified" or not. For our applications, this practice is not needed. We therefore did not incorporate it in the construction presented in proposition 3.3.11.

Finally, we point out that in [148] the closely related notion of ***modular stratification*** has been defined for *datalog* programs. Essentially, it builds a componentwise dynamic local stratification. It is worth noting that the above mentioned "tie breaking" strategy is *not* built into its definition. Since every modularly stratified (datalog) program is weakly stratified (theorem 3.1 in [148]), the latter is the more general property.

Chapter 4

Herbrand Semantics for Meta-Programs

4.1 Introduction

In chapter 3, we indicated some apparent problems with the Herbrand semantics of untyped vanilla-like meta-programs. We introduced the concepts of language independence and weak stratification and announced that both would be used as basic tools in our development. Having worked our way through the necessary preparations, we can in this chapter address our proper subject: meta-program semantics.

Our presentation is structured as follows. Section 4.2 contains our basic results for stratified object programs and their straightforward untyped vanilla meta-version. Some related (more "useful") meta-programs are considered in section 4.3. We justify overloading logical symbols in section 4.4 and address various (limited) forms of amalgamation in section 4.5. Section 4.6 contains some interesting results for definite programs in the context of S-semantics. The latter is a variant of standard Herbrand semantics, allowing also non-ground atoms to appear in interpretations and models ([55],[56]). It was designed to more closely mirror the operational behaviour of (definite) logic programs. Section 4.6 includes a short presentation of its main characteristics, sufficient for a good understanding of its use in the present context. Next, section 4.7 deals with applications that involve explicit references to theories and explores the applicability boundaries of our approach. We discuss and compare related work in section 4.8. Finally, some concluding remarks and ideas for further research, listed in section 4.9, round off this part of the thesis.

4.2 Vanilla Meta-Programs

In this section, we present the two key results that lie at the heart of our work.

4.2.1 Definitions

We set out with the following definitions, formally introducing the concept of a *vanilla meta-program.*

Definition 4.2.1 Suppose $\mathcal{L}$ is a first order language and $\mathcal{R}$ its finite (or countable) set of predicate symbols. Then we define $\mathcal{F}_{\mathcal{R}}$ to be a *functorisation* of $\mathcal{R}$ iff $\mathcal{F}_{\mathcal{R}}$ is a set of function symbols such that there is a one-to-one correspondence between elements of $\mathcal{R}$ and $\mathcal{F}_{\mathcal{R}}$ and the arity of corresponding elements is equal.

We introduce the following notation: Whenever A is an atom in a first order language $\mathcal{L}$ with predicate symbol set $\mathcal{R}$ and a functorisation $\mathcal{F}_{\mathcal{R}}$ of $\mathcal{R}$ is given, A' denotes the term produced by replacing in A the predicate symbol by its corresponding element in $\mathcal{F}_{\mathcal{R}}$.

Definition 4.2.2 The following normal program M will be called *vanilla meta-interpreter*:

$solve(empty) \leftarrow$
$solve(X\&Y) \leftarrow solve(X), solve(Y)$
$solve(\neg X) \leftarrow not\ solve(X)$
$solve(X) \leftarrow clause(X, Y), solve(Y)$

Notice M is neither language independent, nor stratified (nor locally stratified). And these properties carry over to M_P-programs, defined as follows:

Definition 4.2.3 Let P be a normal program. Then M_P, the *vanilla meta-program associated with* P, is the normal program consisting of M together with a fact of the form

$clause(A', \ldots \& B' \& \ldots \& \neg C' \& \ldots) \leftarrow$

for every clause $A \leftarrow \ldots, B, \ldots, not C, \ldots$ in P and a fact of the form

$clause(A', empty) \leftarrow$

for every fact $A \leftarrow$ in P.

A number of remarks are in order:

- If $\mathcal{L}_P$, the language underlying P, is determined by $\mathcal{R}_P$, $\mathcal{F}_P$ and $\mathcal{C}_P$, then $\mathcal{L}_{M_P}$, the language underlying M_P, is determined by:
  - $\mathcal{R}_{M_P} = \{solve, clause\}$
  - $\mathcal{F}_{M_P} = \mathcal{F}_P \cup \mathcal{F}_{\mathcal{R}_P} \cup \{\&, \neg\}$
    where $\mathcal{F}_{\mathcal{R}_P}$ is a functorisation of $\mathcal{R}_P$, pre-supposed in definition 4.2.3.

- $\mathcal{C}_{M_P} = \mathcal{C}_P \cup \{empty\}$

- For clarity, except when explicitly stated otherwise, we will demand:
  - $\mathcal{F}_P \cap \{solve, clause\} = \emptyset$
  - $\mathcal{R}_P \cap \{solve, clause\} = \emptyset$
  - $\mathcal{F}_P \cap \mathcal{F}_{\mathcal{R}_P} = \emptyset$
  - $empty \notin \mathcal{C}_P$

  for all object programs P throughout this chapter.

- In the sequel, when we refer to Herbrand interpretations and/or models (of a program P or M_P) and related concepts, this will be in the context of the languages $\mathcal{L}_P$ and $\mathcal{L}_{M_P}$, as defined above, unless when stated explicitly otherwise.

- Finally, we introduce the following notation:
  - U_P: the Herbrand universe of a program P
  - $U_P{}^n = U_P \times \ldots \times U_P$ (n copies)
  - p/r: a predicate symbol with arity r in $\mathcal{R}_P$
    p'/r: its associated function symbol in $\mathcal{F}_{M_P}$

The following proposition, which will implicitly be used in the sequel, is immediate.

Proposition 4.2.4 Let P be a normal program with $\mathcal{C}_P \neq \emptyset$ and M_P its vanilla meta-program. Then $U_P \subset U_{M_P}$.

Proof Obvious from the definitions. □

Notice, however, that the property does *not* hold when $\mathcal{C}_P = \emptyset$. Indeed, in that case U_P contains terms with $*$, while U_{M_P} does not.

4.2.2 Weak stratification of M_P

We are now ready to formulate and prove the first of our two main results. It shows that the concept of *weak stratification has a very natural application in the realm of meta-programming.*

Theorem 4.2.5 Let P be a stratified normal program. Then M_P, the vanilla meta-program associated with P, is weakly stratified.

Proof We can choose:

$V_1 = \{clause(t_1,t_2) | clause(t_1,t_2)$ is a ground instance of a $clause(t,s)$-fact in $M_P\}$

It immediately follows that L_1 is definite and $H_1 = V_1$.
It also follows that

$$\begin{aligned} P_2 = &\{solve(empty)\} \\ &\cup\{C | C \text{ is a ground instance of } solve(X\&Y) \leftarrow solve(X), solve(Y)\} \\ &\cup\{C | C \text{ is a ground instance of } solve(\neg X) \leftarrow not\ solve(X)\} \\ &\cup\{solve(t_1) \leftarrow solve(t_2) | clause(t_1,t_2) \in V_1\} \end{aligned}$$

Now suppose that $P^1, \ldots, P^k$ is a stratification of P. Then we can choose:

$$\begin{aligned} V_2 = &\{solve(t) | t \in \tau^1\} \text{ with} \\ \tau^1 = &\{p'(\bar{t}) | p/n \in P^1, \bar{t} \in U_{M_P}{}^n\} \\ &\cup\{t_1\&t_2 | t_1, t_2 \in \tau^1\} \\ &\cup\{f(\bar{t}) | f/n \in \mathcal{F}_P, \bar{t} \in U_{M_P}{}^n\} \\ &\cup \mathcal{C}_{M_P} \end{aligned}$$

It follows immediately that L_2 is a definite program.
Moreover, for $3 \leq i \leq k+1$, we can choose:

$$\begin{aligned} V_i = &\{solve(t) | t \in \tau^{i-1}\} \text{ with} \\ \tau^{i-1} = &\{p'(\bar{t}) | p/n \in P^{i-1}, \bar{t} \in U_{M_P}{}^n\} \\ &\cup\{t_1\&t_2 | t_1, t_2 \in \textstyle\bigcup_{j \leq i-1} \tau^j \text{ and } t_1 \text{ or } t_2 \in \tau^{i-1}\} \\ &\cup\{\neg t | t \in \tau^{i-2}\} \end{aligned}$$

and for $i > k+1$:

$$\begin{aligned} V_i = &\{solve(t) | t \in \tau^{i-1}\} \text{ with} \\ \tau^{i-1} = &\{t_1\&t_2 | t_1, t_2 \in \textstyle\bigcup_{j \leq i-1} \tau^j \text{ and } t_1 \text{ or } t_2 \in \tau^{i-1}\} \\ &\cup\{\neg t | t \in \tau^{i-2}\} \end{aligned}$$

It follows that for all $3 \leq i$

$$\{solve(\neg t) \leftarrow not\ solve(t) | t \in U_{M_P} \setminus \tau^{i-2}\}$$

is the set of clauses with a negative literal in P_i. From this, we can conclude that every $L_i (i \geq 3)$ is definite. Moreover, $\bigcap_{j<\omega} CH(P_j) = \emptyset$. □

This result shows that the meta-program on page 28 is indeed weakly stratified, since its object program is stratified. In fact, we can infer from the proof above that our observation on page 28 about the local stratification of the program M_P' can also be generalised. Indeed, once P_2 is obtained, the rest of the stratification process is essentially static: P_2 is locally stratified. It can therefore be argued that the concept of *weak stratification* is perhaps a bit too strong for our purposes. However, we conjecture that all results in this chapter formulated for *stratified* object programs, also hold for *weakly* stratified object programs. We will not explicitly address this topic in the rest of this work, but it is clear that in the latter context, meta-programs will in general no longer be "almost" locally stratified.

4.2.3 A sensible semantics for M_P

Now follows our basic result:

Theorem 4.2.6 Let P be a stratified, language independent normal program and M_P its vanilla meta-program. Let H_P denote the perfect Herbrand model of P and H_{M_P} the weakly perfect Herbrand model of M_P. Then the following holds for every $p/r \in \mathcal{R}_P$:

$$\forall \bar{t} \in {U_{M_P}}^r : \; solve(p'(\bar{t})) \in H_{M_P} \Longleftrightarrow \bar{t} \in {U_P}^r \;\&\; p(\bar{t}) \in H_P$$

Proof Suppose $P^1, \ldots, P^k$ is a stratification of P. The proof is through *induction on the P-stratum* to which p belongs. Suppose first that $p \in P^1$. Let P_1 name the collection of clauses in P corresponding to P^1. (P_1 is a definite, language independent program.) Let T_{P_1} name its immediate consequence operator. We learn from the proof of theorem 4.2.5 that atoms of the form $solve(p'(\bar{t}))$ are in the V_2 layer of B_{M_P}. But this means:

$$solve(p'(\bar{t})) \in H_{M_P} \Longleftrightarrow solve(p'(\bar{t})) \in H_2 \qquad (*)$$

Let T_{L_2} name the immediate consequence operator corresponding to the (infinite) definite ground program L_2. (Throughout the rest of this proof, names such as V_2, H_2 and L_2 refer to the construction showing the weak stratification of M_P.)
We first prove the following:

$$\forall \bar{t} \in {U_P}^r, \forall n \in \mathbb{N} : p(\bar{t}) \in T_{P_1}{\uparrow}n \Longrightarrow \exists m \in \mathbb{N} : solve(p'(\bar{t})) \in T_{L_2}{\uparrow}m \quad (1)$$

The proof proceeds through induction on n. The base case ($n = 0$; $T_{P_1}{\uparrow}0 = \emptyset$) is trivially satisfied. Now suppose that $p(\bar{t}) \in T_{P_1}{\uparrow}n, n > 0$. Then there must be at least one clause C in P_1 such that $p(\bar{t}) \leftarrow C_1, \ldots, C_k$ ($k \geq 0$) is a ground instance of C and $C_1, \ldots, C_k \in T_{P_1}{\uparrow}(n-1)$. Consider first the case that we have one with $k = 0$. In other words, $p(\bar{t}) \leftarrow$ is a ground instance of a fact in P. In that case, L_2 contains the clause $solve(p'(\bar{t})) \leftarrow solve(empty)$. It follows that $solve(p'(\bar{t})) \in T_{L_2}{\uparrow}2$.
Suppose now $k \geq 1$. Then L_2 contains the clause

$$solve(p'(\bar{t})) \leftarrow solve(C'_1 \& \ldots \& C'_k),$$

as well as

$$solve(C'_l \& \ldots \& C'_k) \leftarrow solve(C'_l), solve(C'_{l+1} \& \ldots \& C'_k) \qquad \forall\, 1 \leq l < k.$$

The induction hypothesis guarantees for every C_i the existence of an $m_i \in \mathbb{N}$ such that $solve(C'_i) \in T_{L_2}{\uparrow}m_i$. Let mm denote the maximum of these m_i. It takes only a straightforward proof by induction on l to show the following:

$$\forall\, 1 \leq l \leq k : solve(C'_l \& \ldots \& C'_k) \in T_{L_2}{\uparrow}(mm + k - l).$$

From this, it follows that in particular

$$solve(C'_1 \& \ldots \& C'_k) \in T_{L_2}{\uparrow}(mm + k - 1)$$

and therefore

$$solve(p'(\bar{t})) \in T_{L_2}{\uparrow}(mm + k).$$

This completes the proof of (1). (Notice that, in this part of the proof the language independence of P is only implicitly used, to deal with the case that $C_P = \emptyset$. See also the remark following proposition 4.2.9.)

Next, we prove:

$$\forall \bar{t} \in U_{M_P}{}^r, \forall n \in I\!N : solve(p'(\bar{t})) \in T_{L_2}\!\uparrow\! n \\ \Longrightarrow \bar{t} \in U_P{}^r \ \& \ \exists m \in I\!N : p(\bar{t}) \in T_{P_1}\!\uparrow\! m \quad (2)$$

We first define $\mathcal{L}'$ to be the language determined by $\mathcal{R}_{P_1}$, $\mathcal{F}_{M_P}$ and $\mathcal{C}_{M_P}$. $\mathcal{L}'$ is an extension of $\mathcal{L}_{P_1}$.

The proof again proceeds through an induction on n. The base case where $n = 0$ and $T_{L_2}\!\uparrow\! 0 = \emptyset$ is trivially satisfied. Suppose that $solve(p'(\bar{t})) \in T_{L_2}\!\uparrow\! n$ where $n > 0$. Then either there is a *clause*-fact in M_P of which $clause(p'(\bar{t}), empty) \leftarrow$ is a ground instance or this is not the case. Suppose first there is. Then P_1 must contain a fact of which $p(\bar{t}) \leftarrow$ is a ground instance in $\mathcal{L}'$. This means that $p(\bar{t}) \in T_{P_1}\!\uparrow\! 1$ in $\mathcal{L}'$. But, since P_1 is language independent, this implies that $\bar{t} \in U_{P_1}{}^r$ (and thus $\bar{t} \in U_P{}^r$) and $p(\bar{t}) \in T_{P_1}\!\uparrow\! 1$.

If there is no such *clause*-fact in M_P, then there must be one with a ground instance $clause(p'(\bar{t}), C_1'\& \ldots \& C_k')$ where $k \geq 1$, such that $solve(C_1'\& \ldots \& C_k') \in T_{L_2}\!\uparrow\!(n-1)$. A simple induction argument on k shows that we obtain the following:

$$\forall\ 1 \leq i \leq k : \exists\ n_i < n \in I\!N : solve(C_i') \in T_{L_2}\!\uparrow\! n_i.$$

Through the induction hypothesis, we get:

$$\forall\ 1 \leq i \leq k : \exists m_i \in I\!N : C_i \in T_{P_1}\!\uparrow\! m_i \ \& \ \bar{t_i} \in U_P{}^{r_i}$$

(where $\bar{t_i}$ is the tuple of arguments appearing in the atom C_i and r_i the arity of its predicate). From the above and the fact that P_1 is language independent, (2) follows.

The desired result for the bottom stratum is an immediate consequence of (*), (1) and (2).

Now, let $p \in P^i$ ($i > 1$) and assume that the result is obtained for all $q \in P^j, j < i$. We first prove the *if*-part. From $p(\bar{t}) \in H_P$ and $p \in P^i$, we know there must be a clause in P such that

$$p(\bar{t}) \leftarrow C_1, \ldots, C_n\ (n \geq 0)$$

is a ground instance of it for which holds that all positive $C_j \in H_P$ and for no negative $C_j = not\,B_j, B_j \in H_P$. Now:

- If there is no C_j containing a predicate symbol $\in P^i$, then we can prove (through induction on i):

  $solve(p'(\bar{t})) \leftarrow\ \in L_{i+1}$.

  From this, the desired result follows immediately.

- In the other case, we can, without loss of generality, suppose that $C_1, \ldots, C_l$ are the literals containing predicate symbols $\in P^i$. We then have in L_{i+1}:

$solve(p(\bar{t})) \leftarrow solve(C_1' \& \ldots \& C_n')$
$solve(C_1' \& \ldots \& C_n') \leftarrow solve(C_1'), solve(C_2' \& \ldots \& C_n')$
$solve(C_2' \& \ldots \& C_n') \leftarrow solve(C_2'), solve(C_3' \& \ldots \& C_n')$
$\vdots$
$solve(C_l' \& \ldots \& C_n') \leftarrow solve(C_l')$

(Here, if C_n is a negative literal $notB$, C_n' of course denotes $\neg B'$.)
$solve(p'(\bar{t})) \in H_{M_P}$ now follows through an induction argument analogous to the one used in the proof of (1) above.

Finally, we turn to the *only-if*-part. We have one of the following two cases:

- If $solve(p'(\bar{t})) \leftarrow \; \in L_{i+1}$, then there is a clause in P such that
  $p(\bar{t}) \leftarrow C_1, \ldots, C_n \; (n \geq 0)$
  is a ground instance of it, all C_j contain predicate symbols $\in \bigcup_{j<i} P^j$ and are true in H_P. The desired result now follows from the language independence of P.

- Otherwise, L_{i+1} must contain a clause:
  $solve(p'(\bar{t})) \leftarrow solve(C_1' \& \ldots \& C_n')$
  such that $solve(C_1' \& \ldots \& C_n') \in H_{i+1}$. (Here again, C_n' possibly denotes $\neg B'$ for some atom B.) Let us (without loss of generality) suppose that $C_1, \ldots, C_l$ are the only literals containing predicate symbols $\in P^i$, then it follows that L_{i+1} also contains:
  $solve(C_1' \& \ldots \& C_n') \leftarrow solve(C_1'), solve(C_2' \& \ldots \& C_n')$
  $solve(C_2' \& \ldots \& C_n') \leftarrow solve(C_2'), solve(C_3' \& \ldots \& C_n')$
  $\vdots$
  $solve(C_l' \& \ldots \& C_n') \leftarrow solve(C_l')$
  An induction argument similar to the one in the proof of (2) above now brings us the desired result.

□

Strictly speaking $\bar{t} \in U_P{}^r$ is of course implied by $p(\bar{t}) \in H_P$, but since we judge the former fact to be an important result in its own right, we have chosen to include the statement explicitly in the formulation of the theorem.

Theorem 4.2.6 shows that untyped non-ground vanilla meta-programs have a very reasonable Herbrand model semantics for a large class of object programs. Notice, however, that it does not incorporate any results on "negative" information. It is obvious that a straightforward generalisation to atoms of the form $solve(\neg p'(\bar{t}))$ is not possible:

Example 4.2.7 Indeed, consider a very simple stratified, language independent object program P:

$p(a) \leftarrow$

We have $solve(\neg p'(empty)) \in H_{M_P}$ while of course $empty \notin U_P$.

However, we do have the result below:

Corollary 4.2.8 Let P be a stratified, language independent program such that $C_P \neq \emptyset$ and M_P its vanilla meta-program. Let H_P denote the perfect Herbrand model of P and H_{M_P} the weakly perfect Herbrand model of M_P. Then the following holds for every $p/r \in \mathcal{R}_P$:

$\forall \bar{t} \in U_P{}^r : \; solve(\neg p'(\bar{t})) \in H_{M_P} \Longleftrightarrow p(\bar{t}) \notin H_P$

Proof From theorem 4.2.6, we have:

$\forall \bar{t} \in U_P{}^r : \; solve(p'(\bar{t})) \in H_{M_P} \Longleftrightarrow p(\bar{t}) \in H_P$

And therefore:

$\forall \bar{t} \in U_P{}^r : \; not \; solve(p'(\bar{t}))$ is satisfied in $H_{M_P} \Longleftrightarrow p(\bar{t}) \notin H_P$

But then definition 4.2.2 implies:

$\forall \bar{t} \in U_P{}^r : \; solve(\neg p'(\bar{t})) \in H_{M_P} \Longleftrightarrow p(\bar{t}) \notin H_P$ □

Obviously, this result can be extended to $\bar{t} \in U_{M_P}{}^r$, if one so desires. Note that it is essential that $C_P \neq \emptyset$.

To conclude this section, we briefly dwell upon the "strength" of theorem 4.2.6. In other words, can a similar result be proven for classes of programs, significantly larger than the class of language independent programs ? We believe this not to be the case. One argument in favour of this is the fact that the proof of theorem 4.2.6 relies on the language independence of the object program in a very natural way. However, one direction of the $\Longleftrightarrow$ *can* be shown to hold for (almost) *all definite* object programs (in proposition 4.2.9, M_P is supposed to be defined *without* the third clause of definition 4.2.2, see definitions 4.6.10 and 4.6.11):

Proposition 4.2.9 Let P be a definite program with $C_P \neq \emptyset$ and M_P its vanilla meta-program. Let H_P denote the least Herbrand model of P and H_{M_P} the least Herbrand model of M_P. Then the following holds for every $p/r \in \mathcal{R}_P$:

$\forall \bar{t} \in U_P{}^r : p(\bar{t}) \in H_P \Longrightarrow solve(p'(\bar{t})) \in H_{M_P}$

Proof The result immediately follows from:

$\forall \bar{t} \in U_P{}^r, \forall n \in I\!N : p(\bar{t}) \in T_P{\uparrow}n \Longrightarrow \exists m \in I\!N : solve(p'(\bar{t})) \in T_{M_P}{\uparrow}m$

The proof of the latter statement is analogous to the one for (1) in the proof of theorem 4.2.6. □

(In fact, we can just as well use M_P as defined in definitions 4.2.2 and 4.2.3, as the first part of the proof for theorem 4.2.6 shows.) Again, $C_P \neq \emptyset$ is an essential condition. Indeed, the least Herbrand model of a non-language independent program contains non-propositional elements. All terms in these atoms are of course ground and the only constant involved is $*$, which is not even present in the language of the meta-program. It should also be mentioned that when the restriction to terms in the *object* universe is imposed as a precondition, the reverse of proposition 4.2.9 does hold. This is proved in [87].

The following example shows that proposition 4.2.9 can *not* be extended to the class of *stratified* programs and their (weakly) perfect Herbrand models.

Example 4.2.10 The following program P is stratified, but *not* language independent.

$r(X) \leftarrow not\ p(X)$
$p(X) \leftarrow not\ q(Y)$
$q(a) \leftarrow$

We find that $r(a) \in H_P$, and yet $solve(r'(a)) \notin H_{M_P}$.

It seems then that theorem 4.2.6, corollary 4.2.8 and proposition 4.2.9 are about the best we can do in the context of "classical" ground Herbrand semantics. In section 4.6, we show that it *is* often possible to drop the condition of language independence in the framework of an extended Herbrand semantics, designed to mirror more closely the operational behaviour of logic programs. But first, in the next few sections, we present some interesting extensions of the basic results obtained above.

4.3 Extensions

Theorem 4.2.6 is interesting because, for a large class of programs, it provides us with a reasonable semantics for non-ground vanilla meta-programming. However, it also shows that we do not seem to *gain* much by this kind of programming. Indeed, (the relevant part of) the meta-semantics can be *identified* with the object semantics. So, why going through the trouble of writing a meta-program in the first place ? The answer lies of course in useful *extensions* of the vanilla interpreter (see e.g. [159] and further references given there). In this section, we study meta-programs that capture the essential characteristics of many such extensions. We will first consider definite object and meta-programs and turn to the normal case afterwards.

4.3.1 Definite programs and their extended meta-programs

In the definitions below, the prefix "d" serves to make a distinction with normal meta-interpreters. However, when it is clear from the context whether a definite or a normal meta-program is intended, we will often not write down that "d" explicitly.

Definition 4.3.1 A definite program of the following form will be called *extended (d-)meta-interpreter*:

$solve(empty, t_{11}, \ldots, t_{1n}) \leftarrow C_{11}, \ldots, C_{1m_1}$
$solve(X\&Y, t_{21}, \ldots, t_{2n}) \leftarrow solve(X, t_{31}, \ldots, t_{3n}), solve(Y, t_{41}, \ldots, t_{4n}),$
$C_{21}, \ldots, C_{2m_2}$
$solve(X, t_{51}, \ldots, t_{5n}) \leftarrow clause(X, Y), solve(Y, t_{61}, \ldots, t_{6n}), C_{31}, \ldots, C_{3m_3}$

where the t_{ij}-terms are extra arguments of the *solve*-predicate and the C_{kl}-atoms extra conditions in its body, together with defining clauses for any other predicates occurring in the C_{kl} (none of which contain *solve* or *clause*).

Definition 4.3.2 Let P be a definite program and E an extended (d-)meta-interpreter. Then E_P, the *E-extended d-meta-program associated with P*, is the definite program consisting of E together with a fact of the form

$clause(A', B'_1 \& \ldots \& B'_n) \leftarrow$

for every clause $A \leftarrow B_1, \ldots, B_n$ in P and a fact of the form

$clause(A', empty) \leftarrow$

for every fact $A \leftarrow$ in P.

As an example of this kind of meta-programming, we include the following program E, adapted from [159]. It builds proof trees for definite object level programs and queries.

Example 4.3.3

$solve(empty, empty) \leftarrow$
$solve(X\&Y, Px\&Py) \leftarrow solve(X, Px), solve(Y, Py)$
$solve(X, X \ \mathit{if} \ Py) \leftarrow clause(X, Y), solve(Y, Py)$

As is illustrated in [159], the proof trees thus constructed can be used as a basis for explanation facilities in expert systems. Further examples can be found in e.g. [158].

We have the following proposition:

Proposition 4.3.4 Let P be a definite, language independent program and E_P an E-extended d-meta-program associated with P. Let H_P and H_{E_P} denote their respective least Herbrand models. Then the following holds for every $p/r \in \mathcal{R}_P$:

$$\forall \bar{t} \in {U_{E_P}}^r : (\exists \bar{s} \in {U_{E_P}}^n : solve(p'(\bar{t}), \bar{s}) \in H_{E_P}) \Longrightarrow \bar{t} \in {U_P}^r \ \& \ p(\bar{t}) \in H_P$$

Proof It follows immediately from an obvious property of definite logic programs that $solve(p'(\bar{t}), \bar{s}) \in H_{E_P}$ implies $solve(p'(\bar{t})) \in \mathcal{L}_{E_P}$-$H_{M_P}$ (M_P's least $\mathcal{L}_{E_P}$-Herbrand model). Let $\mathcal{L}'$ be the language determined by $\mathcal{R}_P$, $\mathcal{F}_{E_P}$ and $\mathcal{C}_{E_P}$. $\mathcal{L}'$

is an extension of $\mathcal{L}_P$. The result now follows from the language independence of P. □

It can be noted that the right hand side of the implication in proposition 4.3.4 is equivalent with $\bar{t} \in U_{M_P}{}^{\tau}$ & $solve(p'(\bar{t})) \in H_{M_P}$ (where M_P denotes the vanilla d-meta-program associated with P, defined as in definition 4.6.11, and H_{M_P} its least Herbrand model). This follows from the definite program version of theorem 4.2.6 (theorem 11 in [41]), the proof of which is similar to the bottom stratum part of the proof for theorem 4.2.6.

Proposition 4.3.4 essentially ensures us that working with extended meta-programs is "safe" for definite, language independent programs. Indeed, no unwanted solutions of the kind mentioned in section 3.1 are produced. Further research can perhaps determine conditions on E that allow an equivalence in proposition 4.3.4. We know such extended interpreters exist: The proof tree building program in example 4.3.3 above presents an instance. In general, programs where the extra arguments and conditions neither cause failures nor additional bindings on the main arguments are obviously safe. (See subsection 4.6.4 for a related comment.)

4.3.2 Normal extensions

We will now address normal object programs and their normal extended meta-programs. The definitions we set out with, are of course very similar to those in the previous subsection.

Definition 4.3.5 A normal program of the following form will be called *extended meta-interpreter*:

$$
\begin{aligned}
&solve(empty, t_{11}, \ldots, t_{1n}) \leftarrow C_{11}, \ldots, C_{1m_1}\\
&solve(X \& Y, t_{21}, \ldots, t_{2n}) \leftarrow solve(X, t_{31}, \ldots, t_{3n}), solve(Y, t_{41}, \ldots, t_{4n}),\\
&\qquad\qquad C_{21}, \ldots, C_{2m_2}\\
&solve(\neg X, t_{51}, \ldots, t_{5n}) \leftarrow not\ solve(X, t_{61}, \ldots, t_{6n}), C_{31}, \ldots, C_{3m_3}\\
&solve(X, t_{71}, \ldots, t_{7n}) \leftarrow clause(X, Y), solve(Y, t_{81}, \ldots, t_{8n}), C_{41}, \ldots, C_{4m_4}
\end{aligned}
$$

where the t_{ij}-terms are extra arguments of the *solve*-predicate and the C_{kl}-literals extra conditions, defined through a stratified program included in the extended meta-interpreter (but not containing *solve* or *clause*).

Definition 4.3.6 Let P be a normal program and E an extended meta-interpreter. Then E_P, the *E-extended meta-program associated with* P, is the normal program consisting of E together with a fact of the form

$clause(A', \ldots \& B' \& \ldots \& \neg C' \& \ldots) \leftarrow$

for every clause $A \leftarrow \ldots, B, \ldots, notC, \ldots$ in P and a fact of the form

$clause(A', empty) \leftarrow$

for every fact $A \leftarrow$ in P.

The first question now is: Are such programs weakly stratified ? The following proposition shows they indeed are, when the object program is stratified.

Proposition 4.3.7 Let P be a stratified program. Let E be an extended meta-interpreter. Then E_P, the E-extended meta-program associated with P, is weakly stratified.

Proof A construction completely analogous to the one in the proof of theorem 4.2.5 can be used, since:

- The strata of the program that defines the C_{kl}-literals can be considered first.
- We can still divide the *solve*-atoms in strata, based on the structure of their first argument.

□

Having established this result, we would like to generalise proposition 4.3.4. However, the following simple examples demonstrate that this is not possible.

Example 4.3.8

P: $p \leftarrow not\ q$
q

Notice P is language independent and (trivially) range restricted.

E: First 3 clauses as in M (definition 4.2.2)
$solve(X) \leftarrow clause(X,Y), solve(Y), good(Y)$
$good(\neg q')$

We have $p \notin H_P$ and yet $solve(p') \in H_{E_P}$.

Example 4.3.9

P: $p(X,Y) \leftarrow r(X), not\ q(X)$
$q(a) \leftarrow h(a)$
$r(a)$
$h(a)$

Notice P is language independent, but *not* range restricted.

E: First 3 clauses as in M (definition 4.2.2)
$solve(X) \leftarrow clause(X,Y), solve(Y), not\ bad(Y)$
$bad(h'(a))$

We have $solve(p'(a, empty)) \in H_{E_P}$ (and, of course, $(a, empty) \notin {U_P}^2$).

However, we *do* have the following result:

Proposition 4.3.10 Let P be a stratified, *range restricted* program and let E_P be an extended meta-program associated with P. Let H_{E_P} be its weakly perfect Herbrand model. Then the following holds for every $p/r \in \mathcal{R}_P$:

$$\forall \bar{t} \in {U_{E_P}}^r : (\exists \bar{s} \in {U_{E_P}}^n : solve(p'(\bar{t}), \bar{s}) \in H_{E_P}) \Longrightarrow \bar{t} \in {U_P}^r$$

Proof First, observe that all facts in a range restricted program P are ground. Secondly, all variables in the head of a clause appear in positive body literals. Therefore, basically, when applying T_P, all variables are instantiated with terms propagated upwards from the ground facts. Now, the same holds for the $\bar{t}$-arguments of $solve(p'(\bar{t}), \bar{s})$-atoms in the context of E_P (provided P is range restricted). Therefore, such arguments can not be instantiated with terms outside U_P. A fully formal elaboration of this reasoning involves a completely straightforward induction proof. □

A few remarks are in order:

- Example 4.3.8 shows that this proposition cannot be strengthened to also include $p(\bar{t}) \in H_P$ in the right-hand side of the implication. And, in view of what might be required in actual applications, this seems very natural. Consider for example a jury finding a guilty person innocent through lack of sufficient proof. When formalised by means of an extended meta-interpreter, this would result in something like $solve(innocent')$ being true at the meta-level, while *innocent* would be false at the object level.
- However, proposition 4.3.10 does certify that one does not obtain nonsensical $solve(p'(\bar{t}, \bar{s}))$-atoms in the weakly perfect Herbrand model of the extended meta-program, at least for *range restricted* object programs. Indeed, example 4.3.9 shows that even this minimum result can *not* be extended to the class of all stratified, *language independent* programs. In other words, language independence proves to be too weak a concept in the context of extended normal meta-programs.

4.4 A Justification for Overloading

In the next section, we extend some of the results of section 4.2 to provide a semantics for a limited form of amalgamation. The simplest example of the kind of programs we will consider is the (textual) combination $P + M_P$ of the clauses of an object program P with the clauses for its associated vanilla program M_P. A more complex case is obtained by further (textually) extending $P + M_P$ with additional *clause*/2-facts and -statements, covering the clauses in M_P itself. In the most general case, we allow moreover the occurrence of *solve*/1-calls in the bodies of clauses of P. Furthermore, we will impose the use of one particular functorisation $\mathcal{F}_{\mathcal{R}_P}$, namely the one in which all functors in $\mathcal{F}_{\mathcal{R}_P}$ are identical to their associated predicate symbols in $\mathcal{R}_P$. (And, in the more complex cases, we will proceed similarly for the predicates *solve* and *clause*.)

Postponing the discussion on the generalisation of our results, we first address the more basic problem with the semantics of such programs, caused by overloading the symbols in the language. Clearly, the predicate symbols of P occur both as predicate symbol and as functor in $P + M_P$ (and in any further extensions). Now, although this was not made explicit in e.g. [110], an underlying assumption of first order logic is that the class of functors and the class of predicate symbols of a first order language $\mathcal{L}$, are disjoint (see e.g. [50], [142]). So, if we aim to extend our results to amalgamated programs, without introducing any kind of naming to avoid the overloading, we need to verify whether the constructions, definitions and results on the foundations of logic programming are still valid if the functors and predicate symbols of the language overlap.

We have checked the formalisation and proofs in [110], starting from the assumption that the set of functors and the set of predicate symbols may overlap. We found that none of the results become invalid. Of course, under this assumption, there is in general no way to distinguish well-formed formulas from terms. They as well have a non-empty intersection. But this causes no problem in the definition of pre-interpretations, variable- and term-assignments and interpretations (see e.g. [110], p.12). It is clear however, that a same syntactical object can be both term and formula and can therefore be given two different meanings, one under the pre-interpretation and variable-assignment, the other under the corresponding interpretation and variable-assignment. But this causes no confusion on the level of truth-assignment to well-formed formulas under an interpretation and a variable-assignment. This definition performs a complete parsing of the well-formed formulas, making sure that the appropriate assignments are applied for each syntactic substructure. In particular, it should be noted that no paradoxes can be formulated in these languages, since each formula obtains a unique truth-value under every interpretation and variable-assignment.

On the level of declarative logic program semantics, the main results both for definite programs, the existence of a least Herbrand model and its characterisation as the least fixpoint of T_P ([110], Prop.6.1, Th.6.2, Th.6.5), and for (weakly) stratified normal programs, the existence of a (weakly) perfect Herbrand model for P ([110], Cor.14.8; [7], [139], [138]), remain valid in the extended languages. Thus, the amalgamated programs we aim to study can be given a unique semantics. In the next section, we demonstrate that it is also a sensible semantics.

4.5 Amalgamation

From here on, throughout the rest of this chapter, functorisations will always contain ***exactly the same symbols*** as their corresponding sets of predicate symbols.

A justification for this practice was given in the previous section. It leads to an increased flexibility in considering meta-programs with several layers. In fact, as shown in subsection 4.5.2 below, we can now deal with an unlimited amount of meta-layers. However, we first briefly consider a completely straightforward extension: Including the object-program in the resulting meta-program.

4.5.1 Amalgamated vanilla meta-programs

Definition 4.5.1 Let P be a normal program and M_P its associated vanilla meta-program (see definition 4.2.3). Then we call the textual combination $P + M_P$ of P and M_P the *amalgamated vanilla meta-program associated with* P.

Notice that $\mathcal{L}_{P+M_P}$ is determined by

$\mathcal{R}_{P+M_P} = \mathcal{R}_P \cup \{solve, clause\}$
$\mathcal{F}_{P+M_P} = \mathcal{F}_P \cup \mathcal{R}_P \cup \{\&, \neg\}$
$\mathcal{C}_{P+M_P} = \mathcal{C}_P \cup \{empty\}$

We immediately have the following:

$U_{P+M_P} = U_{M_P}$

The semantic properties of $P + M_P$ are of course straightforward variants of those obtained above for M_P. First, we have the following:

Proposition 4.5.2 Let P be a stratified program, then $P + M_P$, its associated amalgamated vanilla meta-program, is weakly stratified.

Proof For the construction in proposition 3.3.11, we can first consider the strata of P and then continue as in the proof of theorem 4.2.5. □

This enables us to formulate the next theorem:

Theorem 4.5.3 Let P be a stratified, language independent program, M_P its vanilla and $P + M_P$ its amalgamated vanilla meta-program. Let H_P, H_{M_P} and H_{P+M_P} denote their (weakly) perfect Herbrand models. Then the following holds for every $p/r \in \mathcal{R}_P$:

$\forall \bar{t} \in U_{P+M_P}{}^r : \; solve(p(\bar{t})) \in H_{P+M_P} \Longleftrightarrow p(\bar{t}) \in H_{P+M_P}$
$\forall \bar{t} \in U_{P+M_P}{}^r : \; solve(p(\bar{t})) \in H_{P+M_P} \Longleftrightarrow \bar{t} \in U_P{}^r \;\&\; p(\bar{t}) \in H_P$
$\forall t \in U_{P+M_P} : \; solve(t) \in H_{P+M_P} \Longleftrightarrow t \in U_{M_P} \;\&\; solve(t) \in H_{M_P}$

Proof Obvious from definition 4.5.1 and theorem 4.2.6. □

Naturally, adapted versions of corollary 4.2.8 and proposition 4.2.9 also hold.

Considering *extended amalgamated meta-programs* is likewise straightforward. We will not do this explicitly and only illustrate by an example the extra programming power one can gain in this context.

Example 4.5.4 In applications based on the proof tree recording program from example 4.3.3, it may be the case that users are not interested in branches for particular predicates. To reflect this, clauses of the form:

$solve(p(x), some\_info) \leftarrow p(x)$

can be added (combined with extra measures to avoid also using the standard clause for these cases).

In other words, on the basis laid out in this subsection, we can build a sensible semantics for a number of meta-programs involving *reflection*. Some brief comments on the issue of reflection can be found below, at the end of this section.

4.5.2 Meta2-programs

In this subsection, we consider meta-programs that include *clause*-information for the M_P-clauses themselves, thus allowing the use of an unlimited amount of meta-layers. Programming of this kind is relevant in e.g. the contexts of reasoning about reasoning (see e.g. [89]) and proof-plan construction and manipulation (see e.g. [73]).

We start with a formal definition.

Definition 4.5.5 Let P be a normal program. Then $M^2{}_P$, the *vanilla meta2-program associated with P*, is the program M (see definition 4.2.2) together with the following clause:

$clause(clause(X, Y), empty) \leftarrow clause(X, Y) \qquad (*)$

and a fact of the form

$clause(A, \ldots \& B \& \ldots \& \neg C \& \ldots) \leftarrow$

for every clause $A \leftarrow \ldots, B, \ldots, notC, \ldots$ in P or M and a fact of the form

$clause(A, empty) \leftarrow$

for every fact $A \leftarrow$ in P or M.

Notice that this definition essentially adds to the vanilla meta-program *clause*-facts for the four *solve*-clauses in M and for every *clause*-fact. An actual textual execution of the latter intention would however demand the addition of an infinite amount of *clause*-facts. Indeed, we do not only want *clause*-facts for the clauses in P and M and the *clause*-facts in M_P, but also for the *clause*-facts about these *clause*-facts, etc.. Rule $(*)$ in definition 4.5.5 covers all the "facts about facts" cases. Definition 4.5.5 implies that $\mathcal{L}_{M^2{}_P}$ is determined by

$\mathcal{R}_{M^2{}_P} = \{solve, clause\}$

$\mathcal{F}_{M^2{}_P} = \mathcal{F}_P \cup \mathcal{R}_P \cup \{solve, clause, \&, \neg\}$

$\mathcal{C}_{M^2{}_P} = \mathcal{C}_P \cup \{empty\}$

It follows that $\mathcal{C}_P \neq \emptyset \Longrightarrow U_P \subset U_{M^2{}_P}$

The proof of the following theorem is a straightforward adaptation of the one for theorem 4.2.5.

Theorem 4.5.6 Let P be a stratified program. Then $M^2{}_P$, the vanilla meta2-program associated with P, is weakly stratified.

Proof Suppose that $P^1, \ldots, P^k$ is a stratification of P. To see that $M^2{}_P$ is indeed weakly stratified, it suffices to take the sets V_i in the construction described in proposition 3.3.11 as follows:

- $V_1 = \{clause(t_1, t_2) | t_1, t_2 \in U_{M^2{}_P}\}$
- $V_2 = \{solve(t) | t \in \tau^1\}$ with
  $\tau^1 = \{p(\bar{t}) | p/n \in P^1, \bar{t} \in U_{M_P}{}^n\}$
  $\cup \{t_1 \& t_2 | t_1, t_2 \in \tau^1\}$
  $\cup \{f(\bar{t}) | f/n \in \mathcal{F}_P, \bar{t} \in U_{M_P}{}^n\}$
  $\cup\, C_{M_P}$
  $\cup \{clause(t_1, t_2) | t_1, t_2 \in U_{M^2{}_P}\}$
  $\cup \{solve(t) | t \in \tau^1\}$
- For $3 \leq i \leq k+1$:
  $V_i = \{solve(t) | t \in \tau^{i-1}\}$ with
  $\tau^{i-1} = \{p(\bar{t}) | p/n \in P^{i-1}, \bar{t} \in U_{M_P}{}^n\}$
  $\cup \{t_1 \& t_2 | t_1, t_2 \in \bigcup_{j \leq i-1} \tau^j$ and either t_1 or $t_2 \in \tau^{i-1}\}$
  $\cup \{\neg t | t \in \tau^{i-2}\}$
  $\cup \{solve(t) | t \in \tau^{i-1}\}$
- For $k+1 < i$:
  $V_i = \{solve(t) | t \in \tau^{i-1}\}$ with
  $\tau^{i-1} = \{t_1 \& t_2 | t_1, t_2 \in \bigcup_{j \leq i-1} \tau^j$ and either t_1 or $t_2 \in \tau^{i-1}\}$
  $\cup \{\neg t | t \in \tau^{i-2}\}$
  $\cup \{solve(t) | t \in \tau^{i-1}\}$

□

At least part of the following theorem will by now no longer come as a surprise.

Theorem 4.5.7 Let P be a stratified, language independent program and $M^2{}_P$ its vanilla meta2-program. Let H_P denote the perfect Herbrand model of P and $H_{M^2{}_P}$ the weakly perfect Herbrand model of $M^2{}_P$. Then the following holds:

$\forall t \in U_{M^2{}_P} : \ solve(solve(t)) \in H_{M^2{}_P} \iff solve(t) \in H_{M^2{}_P}$

Moreover, the following holds for every $p/r \in \mathcal{R}_P$:

$\forall \bar{t} \in U_{M^2{}_P}{}^r : \ solve(p(\bar{t})) \in H_{M^2{}_P} \iff \bar{t} \in U_P{}^r \ \& \ p(\bar{t}) \in H_P$

Proof The proof of the second equivalence is analogous to the proof of theorem 4.2.6. To prove the first, we discern between different possibilities for the structure of t.

- $t = empty$

  From definition 4.5.5, it is clear that both $solve(solve(empty))$ and $solve(empty)$ are in $H_{M^2_P}$.

- $t = t_1 \& t_2$ and suppose the equivalence holds for t_1 and t_2. Then we have:

  $solve(solve(t)) \in H_{M^2_P}$
  $\Longleftrightarrow \exists y\ clause(solve(t_1 \& t_2), y)$ and $solve(y) \in H_{M^2_P}$
  $\Longleftrightarrow solve(solve(t_1) \& solve(t_2)) \in H_{M^2_P}$
  $\Longleftrightarrow solve(solve(t_1))$ and $solve(solve(t_2)) \in H_{M^2_P}$
  $\Longleftrightarrow solve(t_1)$ and $solve(t_2) \in H_{M^2_P}$
  $\Longleftrightarrow solve(t_1 \& t_2) \in H_{M^2_P}$

- $t = \neg t'$ and suppose the equivalence holds for t'. Then we have:

  $solve(solve(t)) \in H_{M^2_P}$
  $\Longleftrightarrow \exists y\ clause(solve(\neg t'), y)$ and $solve(y) \in H_{M^2_P}$
  $\Longleftrightarrow solve(\neg solve(t')) \in H_{M^2_P}$
  $\Longleftrightarrow solve(solve(t')) \notin H_{M^2_P}$
  $\Longleftrightarrow solve(t') \notin H_{M^2_P}$
  $\Longleftrightarrow solve(\neg t') \in H_{M^2_P}$

- Finally, we deal with the remaining cases.

  $solve(solve(t)) \in H_{M^2_P}$
  $\Longleftrightarrow \exists y\ clause(solve(t), y)$ and $solve(y) \in H_{M^2_P}$
  $\Longleftrightarrow \exists y'\ solve(clause(t, y') \& solve(y')) \in H_{M^2_P}$
  $\Longleftrightarrow \exists y'\ solve(clause(t, y'))$ and $solve(solve(y')) \in H_{M^2_P}$
  $\Longleftrightarrow \exists y', y''\ clause(clause(t, y'), y''), solve(y'')$ and $solve(solve(y')) \in H_{M^2_P}$
  $\Longleftrightarrow \exists y'\ clause(t, y')$ and $solve(solve(y')) \in H_{M^2_P}$

  Now suppose $t \in \tau^i$. Then we can use induction on the level at which the $T_{L_{i+1}}$-operator derives $solve(solve(t))$. Indeed, $solve(solve(y'))$ will be derived at a lower level, and therefore we have:

  $\Longleftrightarrow \exists y'\ clause(t, y')$ and $solve(y') \in H_{M^2_P}$
  $\Longleftrightarrow solve(t) \in H_{M^2_P}$

□

Theorem 4.5.7 shows that vanilla meta2-programs have a sensible Herbrand semantics, just like plain vanilla meta-programs. Notice that the language independence of P is not used in the proof of the first equivalence.

It is obvious that corollary 4.2.8 can be rephrased for meta2-programs. And we also have the following:

Corollary 4.5.8 Let $M^2{}_P$ be the vanilla meta2-program associated with a stratified object program P. Let $H_{M^2{}_P}$ denote its weakly perfect Herbrand model. Then the following holds:

$\forall t \in U_{M^2{}_P} : \ solve(\neg solve(t)) \in H_{M^2{}_P} \Longleftrightarrow solve(t) \notin H_{M^2{}_P}$

Proof The result follows immediately from definition 4.5.5 and the first equivalence in theorem 4.5.7. □

Various amalgamated and/or extended meta2-programs can be treated. We will just point out one interesting further step it is possible to take. Indeed, we can consider meta2-programs in which the "object" clauses contain meta-calls. It is clear that we can in such cases no longer discern between an object- and a meta-level. Results similar to what we obtained before make no sense. But we *can* state the following proposition:

Proposition 4.5.9 Let P be a stratified program and let $P + M^2{}_P$ be its *amalgamated vanilla meta2-program.* Let PM be a program textually identical to $P + M^2{}_P$, except that an arbitrary number of atoms A in the bodies of clauses in the P-part of it have been replaced by $solve(A)$. Then the following holds:

$H_{P+M^2{}_P} = H_{PM}$

Proof (Sketch)
First, both $P+M^2{}_P$ and PM can be shown to be weakly stratified. For PM, $p(t)$- and $solve(p(t))$-atoms of course have to be taken together in the same stratum. Moreover, the same can be done in the dynamic stratification of $P + M^2{}_P$. A proof through induction can then be produced to show that every layer in one program has the same least Herbrand model as its corresponding layer in the other program. □

Note that language independence is not immediately relevant here, since $P+M^2{}_P$ and PM have identical underlying languages.

Example 4.5.10 In this framework, we can address interesting examples from [19]. Consider e.g. the following clause, telling us that a person is innocent when he is not found guilty:

$innocent(X) \leftarrow person(X), not\ solve(guilty(X))$

Of course, such possibilities only become really interesting when using extended meta-interpreters, involving e.g. an extra *solve*-argument limiting the resources available for proving a person's guilt. Results similar to those in section 4.3, but now pertaining to the relationship between PM-like programs and their extended versions, can be stated and proved.

Further extensions are possible, but we believe that the above sufficiently illustrates the flexibility, elegance and power of our approach.

We conclude this section with a brief comment on *reflection*. Basically, the term refers to the transfer of queries/answers from one level of reasoning to another. In this way, in a meta-programming system, we obtain additional inference rules: When something is shown to be true at one level, conclude the truth of the corresponding statement at the other level. Translation of a meta-level goal to the object level is generally termed *downward* reflection, while proceeding in the other direction is known as *upward* reflection. *Reflection rules* were first introduced to artificial intelligence in [177]. Bowen and Kowalski mentioned them in the context of logic programming in [19]. An example of a meta-programming approach in logic programming that relies heavily on reflection, can be found in [37]. A number of papers addressing the issue of reflection, together with further references, can be found in [117]. In terms of our framework, we can point out that the amalgamated meta-programs of subsection 4.5.1 provide a basis for considering programs involving *downward* reflection (see example 4.5.4), while meta2-programs are a starting point for the incorporation of *upward* reflection (see proposition 4.5.9). Of course, in this context, these terms do not really refer to inference rules, but rather to the embodiment of the underlying reasoning in actual program clauses.

4.6 S-Semantics for Meta-Interpreters

4.6.1 Introduction

We believe that in many applications, the restrictions we imposed on the object programs in the previous sections, are very naturally satisfied. Nevertheless, the condition of language independence can be regarded as a somewhat annoying limitation. Indeed, the actual *practice* of meta-programming reaches beyond this boundary, without experiencing much trouble. The underlying reason for this phenomenon is the fact that standard least/(weakly) perfect Herbrand semantics does not really accurately reflect the operational behaviour of many logic programs. Indeed, Herbrand models contain only ground atoms, while logic program execution often produces *non-ground answer substitutions*.

In [55] and [56], non-ground Herbrand models are introduced to bridge the gap between declarative semantics and operational behaviour. In this section, we will show that many of our results can be generalised beyond language independence in the context of the so-called *S-semantics*. (Readers judging an approach to declarative semantics which involves non-ground atoms in models to be unacceptable, can perhaps regard the sequel as a formal treatise on the operational behaviour of untyped non-ground meta-programs.)

The rest of this section then, contains first a short introduction to S-semantics. We continue with a look at vanilla meta-interpreters, point out some interesting complications for extended meta-programs and finish our exposé with a brief conclusion. A last remark must be made: Since currently S-semantics is only fully developed for definite programs, this whole section is restricted to *definite* object and meta-programs.

4.6.2 S-Semantics

We first recapitulate some relevant basic notions and results concerning the S-semantics for definite logic programs, as it was introduced in [55] and [56].

For atoms A and A', we define $A \leq A'$ (A is *more general* than A') iff there exists a substitution θ such that $A\theta = A'$. The relation $\leq$ is a preorder. Let $\approx$ be the associated equivalence relation (*renaming*). (Similarly for terms.) Then we can define the following.

Definition 4.6.1 Let P be a definite program with underlying language $\mathcal{L}_P$. Then its *S-Herbrand universe* ${U^S}_P$ is the quotient set of all terms in $\mathcal{L}_P$ with respect to $\approx$.

So, ${U^S}_P$ basically contains *all* possible terms, not only ground ones. Notice however that terms which are renamings of each other are considered to be one and the same element of ${U^S}_P$. The following definition similarly extends the concept of Herbrand *base*.

Definition 4.6.2 Let P be a definite program with underlying language $\mathcal{L}_P$. Then its *S-Herbrand base* ${B^S}_P$ is the quotient set of all atoms in $\mathcal{L}_P$ with respect to $\approx$.

We can now extend the notions of *interpretation*, *truth* and *model*.

Definition 4.6.3 Let P be a definite program. Any subset ${I^S}_P$ of ${B^S}_P$ is called an *S-Herbrand interpretation* of P.

Definition 4.6.4 Let P be a definite program and ${I^S}_P$ an S-Herbrand interpretation of P.

- A (possibly non-ground) *atom* A in $\mathcal{L}_P$ is *S-true* in ${I^S}_P$ iff there exists an atom A', such that (the equivalence class of) A' belongs to ${I^S}_P$ and $A' \leq A$.
- A definite *clause* $A \leftarrow B_1, \ldots, B_n$ in $\mathcal{L}_P$ is *S-true* in ${I^S}_P$ iff for all atoms ${B_1}', \ldots, {B_n}'$ belonging to ${I^S}_P$, if there exists a substitution
  $\theta = mgu(({B_1}', \ldots, {B_n}'), (B_1, \ldots, B_n))$
  then $A\theta$ belongs to ${I^S}_P$.

Definition 4.6.5 Let ${I^S}_P$ be an S-Herbrand interpretation of a definite program P. ${I^S}_P$ is an *S-Herbrand model* of P iff every clause of P is S-true in ${I^S}_P$.

It is clear that S-Herbrand interpretations and models contain non-ground atoms. Notice that the notion of S-truth is defined differently for atoms and for facts (i.e. clauses with no literals in the body). The reason for demanding $A\theta \in {I^S}_P$, instead of $A\theta$ S-true in ${I^S}_P$, is the wish to attach a different semantics to programs such as P_2 and P_3 in example 4.6.9 below. Definitions 4.6.3 to 4.6.5 are taken from [55]. In [56], a more elegant, but also slightly more elaborate approach leads to the same results.

On the set of S-Herbrand interpretations of a given program, we impose an ordering through set inclusion, just as in the case of "ordinary" ground Herbrand interpretations. We can then include the following result from [55].

Theorem 4.6.6 For every definite logic program P, there is a unique least S-Herbrand model ${H^S}_P$.

We will consider this least S-Herbrand model of a definite program P as the description of its S-semantics.

A fixpoint characterisation of the least S-Herbrand model is possible.

Definition 4.6.7 Let P be a definite program. The mapping ${T^S}_P$ on the set of S-Herbrand interpretations, associated with P, is defined as follows:

$$\begin{aligned}{T^S}_P({I^S}_P) = \{A' \in {B^S}_P \mid \exists\, A \leftarrow B_1,\ldots,B_n \text{ in } P,\ \exists\, {B_1}',\ldots,{B_n}' \in {I^S}_P,\\ \exists\theta = mgu(({B_1}',\ldots,{B_n}'),(B_1,\ldots,B_n)), \text{ and } A' = A\theta\}.\end{aligned}$$

The following theorem can now be included from [55]. It provides the desired least fixpoint characterisation of ${H^S}_P$.

Theorem 4.6.8 For every definite program P:

$${H^S}_P = lfp({T^S}_P) = \bigcup_{n<\omega} {{T^S}_P}^n(\emptyset)(= {T^S}_P \uparrow \omega)$$

Finally, it can be pointed out that the least S-Herbrand model of a program exactly characterises computed answer substitutions for completely uninstantiated queries with respect to this program. We refer to [55] and/or [56] for a full formal development with soundness and completeness results.

We conclude this brief introduction to S-semantics with a few simple examples to illustrate the concept of a least S-Herbrand model.

Example 4.6.9

$P_1 : \quad p(a) \leftarrow$

${H^S}_{P_1} = \{p(a)\}$

$P_2 : \quad p(x) \leftarrow$

${H^S}_{P_2} = \{p(x)\}$

$P_3:\quad p(x) \leftarrow$
$\qquad p(a) \leftarrow$
$H^S{}_{P_3} = \{p(x), p(a)\}$
$P_4:\quad p(x) \leftarrow q(x)$
$\qquad q(a) \leftarrow$
$H^S{}_{P_4} = \{p(a), q(a)\}$
$P_5:\quad p(x,y) \leftarrow q(x)$
$\qquad q(a) \leftarrow$
$H^S{}_{P_5} = \{p(a,x), q(a)\}$

Notice that $*$ (see section 3.2) does *not* show up in $H^S{}_{P_2}$. Indeed, definition 4.6.7 and theorem 4.6.8 show that atoms in the least S-Herbrand model of a program without constants do not contain any constants either. In particular, the special constant $*$, added to the underlying language, plays no role in least S-Herbrand semantics although it does of course occur in the S-Herbrand universe of such programs.

4.6.3 Vanilla meta-interpreters

As pointed out above, this section is about definite programs. For clarity and completeness, we include the definition of a definite program's vanilla meta-program below. We leave out the "d", used in similar circumstances in subsection 4.3.1, since in the context of the present section, no confusion with normal meta-programs is possible. Finally, we remind the reader of the fact that we use functorisations which are identical to their associated sets of predicate symbols.

Definition 4.6.10 The following definite program M is called *vanilla meta-interpreter*:

$solve(empty) \leftarrow$
$solve(X \& Y) \leftarrow solve(X), solve(Y)$
$solve(X) \leftarrow clause(X, Y), solve(Y)$

Definition 4.6.11 Let P be a definite program. Then M_P, the *vanilla meta-program associated with* P, is the definite program consisting of M together with a fact of the form

$clause(A, B_1 \& \ldots \& B_n) \leftarrow$

for every clause $A \leftarrow B_1, \ldots, B_n$ in P and a fact of the form

$clause(A, empty) \leftarrow$

for every fact $A \leftarrow$ in P.

We have:

Proposition 4.6.12 Let P be a definite program with $C_P \neq \emptyset$ and M_P its vanilla meta-program. Then $U^S{}_P \subset U^S{}_{M_P}$.

Proof Obvious from the definitions. □

Just like before, $*$ precludes a generalisation of proposition 4.6.12 to all definite object programs. However, our observation above guarantees the absence of problems with $*$ when considering the least S-Herbrand model of programs where $C_P = \emptyset$ and their meta-programs.

We are now in a position to start proving the main result of this section. We set out with the following two lemmas.

Lemma 4.6.13 Let P be a definite program and M_P its vanilla meta-program. Then the following holds for every $p/r \in \mathcal{R}_P$:

$\forall \bar{t} \in U^S{}_P{}^r, \forall n \in I\!N : p(\bar{t}) \in T^S{}_P\!\uparrow\! n \Longrightarrow \exists m \in I\!N : solve(p(\bar{t})) \in T^S{}_{M_P}\!\uparrow\! m$

Proof The proof is through induction on n. The base case ($n = 0$; $T^S{}_P\!\uparrow\!0 = \emptyset$) is trivially satisfied. Now suppose that $p(\bar{t}) \in T^S{}_P\!\uparrow\! n, n > 0$. Then there must be at least one clause $A \leftarrow B_1, \ldots, B_k (k \geq 0)$ in P such that $\exists\, C_1, \ldots, C_k \in T^S{}_P\!\uparrow\!(n-1)$, $\exists \theta = mgu((C_1, \ldots, C_k), (B_1, \ldots, B_k))$ and $p(\bar{t}) = A\theta$. Consider first the case that we have one with $k = 0$. In other words, $p(\bar{t}) \leftarrow$ is a fact in P. In that case, definition 4.6.11 immediately implies that $solve(p(\bar{t})) \in T^S{}_{M_P}\!\uparrow\!2$. Suppose now $k \geq 1$. The induction hypothesis guarantees for every C_i the existence of an $m_i \in I\!N$ such that $solve(C_i) \in T^S{}_{M_P}\!\uparrow\! m_i$. Let mm denote the maximum of these m_i. This means that $\forall\, 1 \leq i \leq k : solve(C_i) \in T^S{}_{M_P}\!\uparrow\! mm$.
In particular, $solve(C_k) \in T^S{}_{M_P}\!\uparrow\! mm$.
Moreover, for any $1 \leq l < k$,

$solve(C_{l+1} \& \ldots \& C_k) \in T^S{}_{M_P}\!\uparrow\!(mm + k - l - 1)$

implies

$solve(C_l \& C_{l+1} \& \ldots \& C_k) \in T^S{}_{M_P}\!\uparrow\!(mm + k - l)$.

It follows (induction on l) that

$\forall 1 \leq l \leq k : solve(C_l \& \ldots \& C_k) \in T^S{}_{M_P}\!\uparrow\!(mm + k - l)$.

In particular,

$solve(C_1 \& \ldots \& C_k) \in T^S{}_{M_P}\!\uparrow\!(mm + k - 1)$.

Since we also know that

- $clause(A, B_1 \& \ldots \& B_k) \in T^S{}_{M_P}\!\uparrow\!1$,
- $\theta = mgu((C_1, \ldots, C_k), (B_1, \ldots, B_k))$,
- $p(\bar{t}) = A\theta$,

it follows that $solve(p(\bar{t})) \in T^S{}_{M_P}\!\uparrow\!(mm + k)$. □

Lemma 4.6.14 Let P be a definite program and M_P its vanilla meta-program. Then the following holds for every $p/r \in \mathcal{R}_P$:

$$\forall \bar{t} \in {U^S}_{M_P}{}^r, \forall n \in I\!N : solve(p(\bar{t})) \in {T^S}_{M_P}\uparrow n \Longrightarrow \bar{t} \in {U^S}_P{}^r \,\&\, \exists m \in I\!N : p(\bar{t}) \in {T^S}_P\uparrow m$$

Proof The proof uses induction on n. The base case ($n = 0$; ${T^S}_{M_P}\uparrow 0 = \emptyset$) is trivially satisfied. Suppose that $solve(p(\bar{t})) \in {T^S}_{M_P}\uparrow n, n > 0$. Then either there is a fact $clause(p(\bar{t}), empty) \leftarrow$ in M_P or this is not the case. Suppose first there is. Then there must be a fact $p(\bar{t}) \leftarrow$ in P and the result follows.
If there is no such *clause*-fact in M_P, then the following must be true:

- $\exists\, clause(A, B_1 \& \ldots \& B_k)(k > 0) \in {T^S}_{M_P}\uparrow(n-1)$
- $\exists\, solve(C_1 \& \ldots \& C_k) \in {T^S}_{M_P}\uparrow(n-1)$
- $\exists\theta = mgu((clause(A, B_1 \& \ldots \& B_k), solve(C_1 \& \ldots \& C_k)), (clause(X, Y), solve(Y)))$ and $solve(p(\bar{t})) = solve(X)\theta$

From the last point, it follows that,
if $\sigma = mgu((B_1 \& \ldots \& B_k), (C_1 \& \ldots \& C_k))$, then $p(\bar{t}) = A\sigma$.
Furthermore, the first point implies the presence of a clause $A \leftarrow B_1, \ldots, B_k$ in P and the second can only be true if
$\forall 1 \le i \le k, \ \exists n_i < n \in I\!N : solve(C_i) \in {T^S}_{M_P}\uparrow n_i$.
Through the induction hypothesis, we obtain
$\forall 1 \le i \le k, \ \exists m_i \in I\!N : C_i \in {T^S}_P\uparrow m_i \ \& \ \overline{t_i} \in {U^S}_P{}^{r_i}$,
where $\overline{t_i}$ is the tuple of arguments appearing in the atom C_i and r_i the arity of its predicate.
The desired result now follows. □

Our main result on the S-semantics of vanilla meta-interpreters is expressed by the following theorem.

Theorem 4.6.15 Let P be a definite program and M_P its vanilla meta-program. Let ${H^S}_P$ and ${H^S}_{M_P}$ denote the least S-Herbrand model of P and M_P respectively. Then the following holds for every $p/r \in \mathcal{R}_P$:

$$\forall \bar{t} \in {U^S}_{M_P}{}^r : solve(p(\bar{t})) \in {H^S}_{M_P} \Longleftrightarrow \bar{t} \in {U^S}_P{}^r \,\&\, p(\bar{t}) \in {H^S}_P$$

Proof The theorem follows immediately from lemmas 4.6.13 and 4.6.14. □

When we compare this theorem with theorem 4.2.6, we first of all notice that it is restricted to *definite* object and meta-programs. However, we conjecture that this limitation follows from the current state of S-semantics, and is not inherent to meta-programming. We briefly return to this issue at the end of this section. More important is the *absence of the language independence condition.* Indeed, theorem 4.6.15 shows that there is a sensible correspondence between the S-semantics of *any* definite object program and its vanilla meta-program. Therefore, this theorem can be regarded as a formal confirmation that this kind of programming gives no *practical* problems, not even for programs that are *not* language independent. Notice, by the way, that theorem 4.6.15 also generalises proposition 4.2.9. Indeed, as indicated above, S-semantics allows a more elegant treatment of the $C_P = \emptyset$ case than classical ground Herbrand semantics.

4.6.4 Extended meta-interpreters

In the previous subsection, we have shown how our earlier results on vanilla meta-semantics can be generalised beyond language independence in the context of S-semantics. Having established theorem 4.6.15, the next question that comes to mind is: Can we accomplish a similar feat for proposition 4.3.4? Contrary to our initial expectations, this question has to be answered negatively.

The definitions of definite extended meta-interpreters and -programs were already given in subsection 4.3.1. We will not repeat them here and simply refer to definitions 4.3.1 and 4.3.2. Now, consider the following example. It involves a very simple, definite, non-language independent object program and an equally simple associated extended meta-program.

Example 4.6.16

$P:\quad p(x) \leftarrow$
$H^S{}_P = \{p(x)\}$
$E_P:\quad solve(empty) \leftarrow$
$\quad\quad solve(x\&y) \leftarrow solve(x), solve(y)$
$\quad\quad solve(x) \leftarrow clause(x, y), solve(y), inst(x)$
$\quad\quad clause(p(x), empty) \leftarrow$
$\quad\quad inst(p(a)) \leftarrow$

It is easy to see that $solve(p(a)) \in H^S{}_{E_P}$.

The source of the problem is clearly the fact that answers might become further instantiated than is the case in the object program. Since the least S-Herbrand model represents the most general answer substitutions, this means that a straightforward generalisation of proposition 4.3.4 is impossible.

However, we *can* prove a more modest variant of proposition 4.3.4. Indeed, proposition 3.2.11 ensures us that computed answers for language independent definite programs are ground. It is clear that such answers *can not be further instantiated.* Therefore, language independent programs again prove to be "safe".

We first prove the following:

Proposition 4.6.17 Let P be a definite program. Let H_P denote its least Herbrand and $H^S{}_P$ its least S-Herbrand model respectively. Then P is language independent iff $H_P = H^S{}_P$.

Proof The proposition follows from definition 4.6.7, theorem 4.6.8 and proposition 3.2.10 via a straightforward induction proof. □

Notice that $H_P = H^S{}_P$ implies that $H^S{}_P$ contains only ground atoms. Seen in the light of our earlier comments and results, proposition 4.6.17 can hardly be called surprising. Indeed, it is just an S-semantics reformulation of proposition 3.2.11. It follows that:

Lemma 4.6.18 Let P be a definite, language independent program and M_P its associated vanilla meta-program. Let $H^S{}_{M_P}$ denote the least S-Herbrand model of M_P. Then the following holds for every $p/r \in \mathcal{R}_P$:

$\forall \bar{t} \in {U^S{}_{M_P}}^r : solve(p(\bar{t})) \in H^S{}_{M_P} \Longrightarrow \bar{t}$ is ground

Proof The lemma follows from theorem 4.6.15 and proposition 4.6.17 □

We need a second lemma:

Lemma 4.6.19 Let P be a definite program, M_P the vanilla and E_P an extended meta-program associated with P. Then the following holds:

$$\forall t \in U^S{}_{E_P} : (\exists \bar{s} \in {U^S{}_{E_P}}^n : solve(t, \bar{s}) \in H^S{}_{E_P}) \Longrightarrow \exists \theta,\ \exists solve(t') \in H^S{}_{M_P} : t = t'\theta$$

Proof Obvious from the definitions. □

This allows us to prove the following variant of proposition 4.3.4:

Proposition 4.6.20 Let P be a definite, language independent program and E_P an E-extended meta-program associated with P. Let $H^S{}_P$ and $H^S{}_{E_P}$ denote the least S-Herbrand model of P and E_P respectively. Then the following holds for every $p/r \in \mathcal{R}_P$:

$$\forall \bar{t} \in {U^S{}_{E_P}}^r : (\exists \bar{s} \in {U^S{}_{E_P}}^n : solve(p(\bar{t}), \bar{s}) \in H^S{}_{E_P}) \Longrightarrow \bar{t} \in {U^S{}_P}^r \ \&\ p(\bar{t}) \in H^S{}_P$$

Proof Lemmas 4.6.18 and 4.6.19 ensure that $solve(p(\bar{t}), \bar{s}) \in H^S{}_{E_P}$ implies $solve(p(\bar{t})) \in H^S{}_{M_P}$ (and therefore $\bar{t} \in {U^S{}_{M_P}}^r$). The result now follows from theorem 4.6.15. □

Example 4.6.16 shows that the S-semantics results for vanilla meta-programs can not immediately be carried over to extended meta-programs. And, indeed, in practice, logic programming with extended meta-programs *can* generate unwanted answer substitutions. Proposition 4.6.20 shows that this is not the case for language independent object programs. It is of course possible to investigate conditions on the meta-program which would ensure that proposition 4.6.20 holds for any definite object program. It is not even very difficult to conjecture some such conditions. (See the comment concluding subsection 4.3.1.) However, we will not pursue this topic in the present work.

4.6.5 Concluding remarks

A treatment of amalgamated meta-programs and meta2-programs in the context of S-semantics is straightforward and not particularly enlightening. All the results from section 4.5 can be generalised in the expected way. We will neither state nor prove them explicitly.

Concerning the limitation to definite programs, we can remark that [166] extends [55] and [56], drawing from work on constructive negation, and in this way perhaps provides a setting for addressing normal object and meta-programs.

4.7 Reasoning about Theories and Provability

4.7.1 An explicit theory argument

An interesting variant of the meta-programs we considered above, is obtained when *an extra argument, denoting a particular object theory*, is added to the *solve* and *clause* predicates.

The basic definitions are as follows:

Definition 4.7.1 We call the following normal program M^t *th-vanilla meta-interpreter*:

$solve(T, empty) \leftarrow$ (*)
$solve(T, X\&Y) \leftarrow solve(T, X), solve(T, Y)$
$solve(T, \neg X) \leftarrow not\ solve(T, X)$ (**)
$solve(T, X) \leftarrow clause(T, X, Y), solve(T, Y)$

Definition 4.7.2 Let P be a normal program. Then $M^t{}_P$, the *th-vanilla meta-program associated with P*, is the normal program consisting of M^t together with a fact of the form

$clause(p, A, \ldots \&B\& \ldots \&\neg C\& \ldots) \leftarrow$

for every clause $A \leftarrow \ldots, B, \ldots, notC, \ldots$ in P and a fact of the form

$clause(p, A, empty) \leftarrow$

for every fact $A \leftarrow$ in P.

In the above definition, the constant p in the *clause*-facts indicates that they correspond to clauses in the object program P.

Obviously, all earlier results can straightforwardly be generalised to programs of this form. Notice that clauses (∗) and (∗∗) in definition 4.7.1 do not provide a range for their theory argument. Thus, H_{M^*P} will contain numerous irrelevant ground atoms, where the theory argument is instantiated to some other term than the constant p. This can be avoided by either introducing a *theory* range predicate for theories, or typing (see section 4.8 for some related comments). Of course, an S-semantics approach is likewise free from such inconveniences.

The meta-program in definition 4.7.2 deals with a single object program. Extending it to cope with several such object programs is straightforward: Simply introduce more constants referring to object theories and annotate the *clause* facts correspondingly. More interesting are meta-programs surpassing the vanilla context by defining interactions between the different object theories.

Consider the following example, adapted from [22]. It consists of a meta-interpretative definition of (definite) logic program intersection.

Example 4.7.3

$solve(T, empty) \leftarrow$
$solve(T, X\&Y) \leftarrow solve(T, X), solve(T, Y)$
$solve(T, X) \leftarrow clause(T, X, Y), solve(T, Y)$
$clause(T \cap T', X, Y\&Y') \leftarrow clause(T, X, Y), clause(T', X, Y')$ (1)

And, of course, a number of facts of the form

$clause(p_1, A, B_1\& \ldots \&B_n)$

$\vdots$

$clause(p_k, C, D_1\& \ldots \&D_m)$

to represent the object programs P_1 to P_k, respectively.

The union of (definite) programs can be defined similarly, as shown in [22].

Example 4.7.4

We replace clause (1) of the program in example 4.7.3, by the following two clauses:

$clause(T \cup T', X, Y) \leftarrow clause(T, X, Y)$
$clause(T \cup T', X, Y) \leftarrow clause(T', X, Y)$

Notice that the latter two clauses again do not provide a (full) range for their theory argument.

4.7.2 The *demo* predicate

In many meta-programming applications in logic programming, not the *solve* meta-interpreter, but a related program is used. It involves the so-called *demo* or *demonstrate* predicate. This name refers to a somewhat different origin and/or underlying motivation of most of this work. Indeed, while the *solve* interpreter refers to (possibly modified) query answering in logic programming, the *demo* program is situated in the context of formalising provability. The difference is obviously rather subtle, but, in general, one can perhaps say that *solve* has a slightly more procedural flavour than *demo*.

The *demo* predicate was originally introduced in chapter 12 of [97] and further elaborated upon in a.o. [19] and [54]. In [99], we find the following definition of a *demo* for definite propositional programs.

Definition 4.7.5

$demo(T, true) \leftarrow$
$demo(T, P\&Q) \leftarrow demo(T, P), demo(T, Q)$
$demo(T, P) \leftarrow demo(T, P{\leftarrow}Q), demo(T, Q)$

A similar meta-interpreter was used in [23] as the basis for a reformulation of the work in [22].

The most striking difference with (the definite variant of) the M^t program in definition 4.7.1 above, is *the absence of a clause predicate*. Kowalski prefers the formulation in definition 4.7.5 because of its greater generality and its similarity with a modal logic approach. However, in the context of a meta extension/simulation of an object program, the formulation with *clause* seems more natural to us. Indeed, the distinction between data in the program and results derivable from the program remains more clear.

A related issue is the choice between *a term- and a constant-based representation of the object theory*. In definition 4.7.2 and in examples 4.7.3 and 4.7.4, we introduced an extra argument in the *clause*-facts and -clauses, and named object theories through meta-level constants. The same technique can be used with the *demo* predicate. In that case, definition 4.7.5 needs to be extended with *demo*-facts, representing the object theory, in exactly the same way as the *clause*-facts above. An alternative possibility is representing the object theory as a term in the meta-goal. We adapt the following simple example from [99].

Example 4.7.6 Execution of the object level program and query

$p \leftarrow q, r$
$q \leftarrow s$
$r \leftarrow$
$s \leftarrow$
$\leftarrow p$

can be simulated by executing the meta-query

$\leftarrow demo([p{\leftarrow}q\&r, q{\leftarrow}s, r{\leftarrow}true, s{\leftarrow}true], p)$

on the condition that the program in definition 4.7.5 is extended with the following clause:

$demo(T, P) \leftarrow member(P, T)$

and a definition of the *member* predicate.

This works well for propositional object theories, but can produce inconvenient variable bindings when used for object programs that contain variables. Moreover, since the object theory is no longer represented in the meta-program itself, comparing least Herbrand models does not immediately bring us anything. Even an extended least Herbrand model, somehow incorporating the information contained in the query, does not really refer to the object theory, but to ground instances of that theory. Therefore, in these cases, a ground representation of the object theory (using meta-level constants to refer to object-level variables) might be preferable. We briefly return to this issue in section 4.8.

Further applications and comments, including a number of interesting further references, can be found in [99]. (There exists e.g. quite a lot of work on the relationship between meta-logic and modal logic. However, that intriguing issue is clearly outside the scope of this thesis.)

We conclude this section with a brief comment on one of the most well-known, and also most advanced, applications of the *demo* predicate: the so-called 3 wisemen puzzle. It was proposed as a benchmark for testing the expressive power and naturalness of knowledge representation formalisms and, in a logic programming setting, has been addressed in a.o. [4], [89] and [100]. For more details about the problem and possible solutions, we refer the interested reader to these papers and further references given there. Also [88] contains some relevant "meta"-remarks, particularly pertaining to the solution presented in [100]. Here, it suffices to point out that problems like this one, and their solutions, fall outside the framework built above. Indeed, consider the following simplified subproblem of the 3 wisemen puzzle.

Example 4.7.7 Given is the object knowledge C:

$white1 \vee white2 \leftarrow$

$white2 \leftarrow$

At the meta-level the truth of *white*1 has to be considered, without using a closed world assumption. In particular, we do *not* want to conclude $demo(c, \neg white1)$.

Clearly, issues of disjunctive logic programming and incomplete knowledge are involved. The former can possibly be dealt with by an extension of our results to the semantics devised for disjunctive programs (see e.g. [57] and [116]). The latter means negation as failure is not fit as a general reasoning mechanism. Which implies that (weakly) perfect Herbrand semantics is not satisfactory.

Finally, notice that [100] as well as [99] uses symbol overloading as addressed in section 4.4.

4.8 Discussion, Some Related Work

This section contains some further comments on our approach to meta-semantics and its results, mainly through a discussion of some related work. Since there is a vast literature on meta-logic, its semantics, possible applications, advantages and disadvantages, we do not strive for completeness. Instead, we only consider some more or less closely related papers within logic programming.

First, we would like to mention [154]. It presents a truth predicate for full first order logic, extendable into a meta-interpreter which is executable in a first order programming system. Unlike our framework, the approach incorporates a full naming mechanism for first order terms and formulas. This enables self-reference, which, together with negation, gives rise to the possible expression of paradoxes. So, a three-valued semantics is inevitable. Summarising, we can conclude that the approach to meta-programming proposed in [154] is more complicated, but also more powerful than the vanilla meta-interpreter and its extensions, considered here. In particular, full quantification inside named expressions is possible, while this is as yet lacking from vanilla related meta-programs (see below for further comments on this issue).

Next, a rather theoretically oriented treatment of the *demo* predicate, including a consideration of self-reference and the simulation of non-classical logics can be found in [17]. The paper also contains some further references that might be relevant to the interested reader.

A work more closely related to what we presented here is [82]. Indeed, in the first part of that paper, it is shown that through the use of appropriate typing, vanilla meta-programs can be given a suitable declarative (the well-known Clark's completion semantics is used) and procedural semantics. Moreover, in a second part, a ground representation for object level terms at the meta-level is considered, and it is shown how a number of problematic Prolog built-ins (static and of the type called "first-order", i.e. not referring to clauses or goals, in [8]) can be given a declarative meaning in this setting.

Addressing the latter topic first, it should be noted that such Prolog meta-predicates, of which *var*/1 and *nonvar*/1 are prototypical examples, are not included in our language. This certainly puts some limitation on the obtained expressiveness. Observe, however, that in the typed non-ground representation proposed in [82], no alternative for *var*/1 was introduced either and that the declarative *var*/1 predicate introduced in the ground representation approach provides no direct support for the sort of functionalities (e.g. control and corouting facilities) that the *var*/1 predicate in Prolog is typically used for. Recently, [8] proposed a declarative semantics for such predicates. We conjecture that, if one so desires, our basic methodology can be adapted to the semantics described there, thus enabling the inclusion of such built-ins in our language. Finally, a perhaps superior alternative for providing the latter kind of facilities is the use of *delay* control annotations as in Gödel.

For the *assert*/1 and *retract*/1 Prolog built-in predicates, the solution of [81], to represent dynamic theories as terms in the meta-program, can as well be applied in our approach. However, as was noted in section 4.7 above, this requires special care with variable bindings, and leads to some inconveniences in the context of a ground Herbrand model approach to semantics. A thorough discussion of the problems related to these predicates is given in [81] and [111]. They are also treated in [160], discussed below. As a final remark about Prolog built-ins, we would like to mention that our use of overloading largely eliminates the need for a *call*/1 predicate, as example 4.5.4 illustrates.

Next, observe that the condition of range restriction, which is the practical, verifiable, sufficient condition for language independence our approach was mostly designed for, is strongly related to typing. Indeed, typing can in principle be converted into additional atoms that are added in the bodies of clauses, expressing the range of each variable. See e.g. [50] and [110]. In this context, it is interesting to note the apparent duality between range restriction at the object level and typing at the meta-level. If one "hardwires" types into the code of the object program through range restriction, typing (or range restriction) at the meta-level is no longer required for a sensible declarative semantics. [69] can also be mentioned here. It presents a program transformation technique that enables to minimise run-time type checking in systems which represent types as (unary) predicates, thus largely eliminating one of the main advantages types might offer in comparison with ranges.

Finally, [82] does not address amalgamation. And Gödel (see [80]), the programming language whose extensive meta-programming facilities are largely based upon the foundations laid out in [82] and [81] does not allow it. While dealing with amalgamated programs of the kind addressed in subsection 4.5.1 seems relatively straightforward, a generalisation of the typed approach to meta2-programs is probably not immediate.

With respect to the extension to amalgamated programs, we should point out that our use of overloading is strongly related to the logic proposed in [144]. Indeed, in his analysis of the problems connected with reference and modality, Richards considers logical languages that contain their *well-formed formulas as terms*. He interprets these languages on specially devised models the domains of which are a union of *the constants and the sentences* (i.e. closed formulas) in the language.

Kalsbeek ([87]) recently proposed a variant of Richards' logic as a suitable basis for studying the semantics of meta-programs. She allows *any formula to be a term* in the language and uses (Herbrand) interpretations with *arbitrary closed terms* in the domain. In the logic framework thus obtained, soundness and completeness of definite vanilla meta-programs with respect to their definite object program are proved, restricted to terms in the *object* level language.

Jiang ([84]) proposes an even more ambivalent language, also allowing *terms as formulas*. He shows that a number of interesting properties (Herbrand theorem, completeness), lost in Richards' logic, are recovered. Vanilla meta-programs are considered, leading to similar results as obtained by others. The framework is however more powerful and particularly suitable for addressing quantified object level statements and full amalgamation. To this end, a sophisticated treatment of variables, not syntactically distinguishing between variables and their names is proposed. This allows the consideration meta-theories dealing with full first order object statements. However, at the time of writing, this work is not yet in a finalised state. Particularly the treatment of variables seems to require further study.

Summarising, our technique of overloading function and predicate symbols is probably less powerful than approaches using more fully ambivalent syntax, but it requires no modification of the familiar notion of Herbrand interpretations and models.

A comparison with the work in [34] basically leads to the same conclusion. The semantic techniques proposed there provide a first order semantics for a class of programming constructs that includes the kind of amalgamation we consider, but significantly surpasses it. Clauses like $solve(x) \leftarrow x$ are allowed, as well as higher order functions, generic predicate definitions, etc.. The basic characteristic of HiLog logic is the fact that "terms may represent individuals, functions, predicates, and atomic formulas in different contexts". However, the semantics required to support this, is a less immediate extension of the common first-order logic semantics.

Next, it should be noted that [108] independently extended the result for vanilla meta-programs of language independent programs in [41] to all definite object programs and their vanilla meta-programs in the context of S-semantics. Moreover, a sizeable meta-programming application is sketched and its semantics discussed in the proposed framework.

We would like to conclude this overview of related work with a brief discussion of [160]. In that paper, the foundations of a theory of metalogic programming in (*definite*) logic programming are laid out. The framework incorporates a full-fledged naming device, like the well-known Gödel numbering. Thus, a name can be attached to any formula in the object language. Axiom schemes for a number of meta-predicates, among which a provability predicate, an assertional and a retractional predicate, are introduced. This gives rise to metalogic programs including clauses for these predicates. A Herbrand model semantics for such programs is then developed. Particularly interesting is the fact that universal quantifiers in named object formulas cause problems in the fixpoint semantics, as shown by the following example, taken from [160].

Example 4.8.1

$p \leftarrow demo('(\forall x)q(x)')$
$q(a) \leftarrow$
$q(s(x)) \leftarrow q(x)$

If we consider the immediate consequence operator T that can be associated with this program (for an exact definition, we refer to [160]), we notice that all atoms of the form $q(t)$ are in $T \uparrow \omega$, but p is not. However, $p \in T \uparrow (\omega + 1)$. So, T is *not* continuous.

Subrahmanian then goes on to define a model theoretic forcing technique, which makes it possible to re-establish the basic semantic results, known for "plain", non-metalogic definite logic programs. A number of interesting further issues are discussed, among which are meta-unification and -proof theory, naming and its relationship with different degrees of amalgamation. A detailed discussion leads too far; we just mention that a brief treatment of the three wisemen puzzle is also included. With respect to the relationship to our own work, we can once again state that, for definite programs, the framework in [160] is definitely more powerful than ours. Indeed, it provides quantification of object level formulas in meta-level statements, as well as a fully developed naming mechanism. However, the price to be paid for this is a non-trivial modification of the standard Herbrand semantics for logic programs. How this approach generalises to programs *with negation* is also not immediately clear.

It seems then that our approach is a good compromise between complicating semantics and enhancing programming power. If one considerably wants to extend the latter, e.g. by allowing full quantification in named object level statements, a more complex semantics is probably inevitable.

4.9 Conclusion

In the first part of this thesis, we have considered untyped non-ground meta-programs. We have studied rather extensively the semantic properties of vanilla meta-interpreters of this kind. And we have looked at interesting extensions and variants involving (a limited form of) amalgamation. It turned out that in most of these cases, the least or weakly perfect Herbrand semantics is well-behaved for definite respectively stratified *language independent* object programs. (And if not, then at least for *range restricted* ones. See proposition 4.3.10.) This is an interesting result since these semantics are widely accepted as good declarative semantics for logic programs. Moreover, we believe that our methodology can often also be applied when considering untyped, non-ground meta-programs that do not immediately fall within one of the categories we explicitly considered. So, contrary to what was generally assumed, untyped non-ground meta-programming often does not really present semantic problems.

We have also shown how, for non-extended (definite) meta-programs, the restriction of *language independence* can be lifted in the context of a declarative semantics that more closely reflects the procedural behaviour of logic programs. These results explain why the language independence condition almost never surfaces in logic programming *practice*. Next, we addressed some issues which arise in the context of meta-programs with an explicit theory argument. And we briefly discussed the formalisation of provability in the related *demo* predicate, including an indication of some limitations inherent to our current work. Finally, we compared our approach with some relevant other work.

Along the way, we have moreover shown that vanilla meta-programs, as well as their variants, associated with stratified object programs are *weakly* stratified. We have conjectured that this result can be generalised to *weakly* stratified *object* programs. Our work therefore seems to provide evidence in favour of the view that *weak* stratification is a much more natural and useful generalisation of stratification than *local* stratification appears to be. It is in any case the former concept which finds an important area of application in the realm of meta-programming. Finally, it can be noted that SLS-resolution, introduced in [141] as a query answering procedure which is sound and complete with respect to the perfect model semantics of stratified programs, retains the same properties in the context of weakly stratified programs and weakly perfect model semantics.

It seems to us that almost all realistic logic programs are at least weakly stratified. Nevertheless, it can be a topic of further research to investigate whether our results can be generalised in the context of a semantics that is able to deal with *any* logic program. Well-founded semantics (see e.g. [172]) is one such approach which recently has gained popularity (see e.g. [147], [171], [140], [137]). It attaches to any logic program a (possibly partial, 3-valued) unique *well-founded* Herbrand model. If we want to recast our results in this setting, the first step is of course a reformulation of the *language independence* notion. This provides no fundamental difficulties, but some care has to be taken since for most sensible programs, the negative information implied (now explicitly incorporated in the well-founded model), *does* depend on the language. So, one should only require stability of the positive information. Having established this, a generalisation of our results can be attempted. We feel no fundamental difficulties or big surprises should be met, but proofs can perhaps be technically quite complicated and not very elegant. Finally, well-founded semantics remains just one of various formalisms proposed as a semantics for arbitrary normal logic programs. It might therefore be more worthwhile to study meta-program semantics in the context of a generic unifying semantic framework such as provided in e.g. [46]. In some sense, by restricting ourselves to a class of programs the semantics of which is the subject of little or no controversy, we have, in the present work, taken a dual approach.

Finally, other possible topics for further research include the following:

- A precise characterisation of extended meta-programs that would allow more powerful variants of propositions 4.3.4, 4.3.10 and/or 4.6.20. The classification of enhancements of the vanilla meta-interpreter presented in [158], can perhaps be a starting point here.
- An extension of section 4.6 to normal programs.
- A reformulation of our results in the context of a declarative semantics for Prolog built-ins, as proposed in [8].
- A study of object theories beyond the scope of normal logic programs, and their associated meta-programs.

Part II

Partial Deduction and the Gentle Art of Finite Unfolding

Chapter 5

Prelude

5.1 Introduction

In this second main part of the thesis, we address partial deduction of definite logic programs. Within this context, we particularly focus on methods to perform sensible finite unfolding, obtaining termination in a way that reflects structural properties of the unfolded query and program.

The three subsequent chapters contain the following material. In chapter 6, we construct a general framework in which unfolding strategies can be described. We formulate algorithms and show that they indeed terminate. In chapter 7, we use a particular, fully automatic unfolding algorithm developed within the framework laid out in chapter 6, as a basis for a partial deduction method. We show that the latter satisfies the conditions of the fundamental theorem 5.4.7 below, and always terminates. We also briefly discuss the results of an experimental study, testing the proposed method on some benchmark programs and comparing its results with those obtained through related strategies. Finally, chapter 8 further elaborates on finite unfolding. Several algorithms, incorporating advanced features, are developed, with a particular stress on full automation.

But first, in the present preliminary chapter, we present some background material on partial deduction. To set out, we briefly situate the notion of partial evaluation in computer science as a whole. Next, in section 5.3, we dwell somewhat more extensively on work in the more narrow context of logic programming. Section 5.4 sums up essential notions and results contained in an influential foundational paper on partial deduction of logic programs. In section 5.5, we conclude chapter 5 with a sketch of a first partial deduction method explicitly based on the given formal foundations. We illustrate its operation on some simple examples and show that its unfolding does not always terminate.

Finally, throughout part II, we will explicitly address only definite logic programs. Therefore, unless stated otherwise, from now on, the term "(logic) program" will refer to a *definite* logic program.

5.2 Partial Evaluation

What is nowadays usually called "partial deduction" in a logic programming setting, derives from partial evaluation, research on which originated, grew and still (also) flourishes outside the context of logic programming. In this section, we present an extremely brief introduction to the development of partial evaluation in computer science as a whole and provide a minimal amount of references. Our account is largely based on (chapter 18 of) [85]. We refer to that source for more details, example applications and comments on related topics, as well as a very recent, rather extensive bibliography.

In the introduction to [85], a program performing partial evaluation is characterised as follows:

> A *partial evaluator* is given a *subject program* p together with part of the latter's input data. Its effect is to construct a new program which, when given p's remaining input, will yield the same result that p would have produced given both inputs. In other words, a partial evaluator is a program specialiser.

Of course, the underlying intention is that the resulting specialised program does whatever it is supposed to do with that remaining input, in a way superior to the original program's performance on the complete input.

The theoretical feasibility of partial evaluation is asserted by Kleene's s-m-n theorem ([90]), showing that any given program can be specialised with respect to any part of its input. However, obtaining a performance improvement was not considered an issue. Pursuing just that has been the main objective of subsequent research by computer scientists.

Following some early work in the 60's, the first contribution of major importance was made by Futamura ([62]). He proposed the use of partial evaluation to *specialise an interpreter to an object program*, and in this way obtain a transformed object program (or, since the new program is in the language of the interpreter, which, in general, does not have to be identical to the language of the original object program, a *compiled* object program). It is exactly this so-called *first Futamura projection* which has received much attention in a logic programming context.

Futamura also proposed to *apply a partial evaluator to itself*, including some interpreter as known partial input. The result of such an exercise is a compiler. One can even proceed one further step, choosing the partial evaluator itself as the interpreter program that serves as input, thus obtaining what has been called a compiler generator. These conceptually quite advanced applications, known as the *second and third Futamura projections*, have been intensively pursued by researchers working in a functional programming context (see e.g. [86]). Within logic programming, however, until recently, they have received very little attention. In this thesis too, the issue of self-application is not (explicitly) addressed.

Other influential early work, in the context of imperative languages, was performed by Ershov, who introduced the term *mixed computation* (see e.g. [51]). Closely related also is the research by Turchin on *supercompilation* (see e.g. [165]).

Around 1985, interest in partial evaluation had increased considerably. A first specialised workshop was held in October 1987, and reported on in [53]. The latter reference also contains a historical account by Ershov ([52]) and an annotated bibliography ([155]) of all papers on partial evaluation that were known at that time. Recent developments can be observed in the proceedings of the annual ACM Sigplan symposium/workshop devoted to partial evaluation and semantics based program manipulation (see e.g. [2] and [3]).

This conludes our brief introduction to partial evaluation in general. Before taking a closer look at the work in a more narrow logic programming setting, we adopt a piece of useful terminology. In a functional programming context, it is customary to discern between *online* and *offline* approaches to partial evaluation. The former term refers to work which takes most or all decisions on how (and whether) partial evaluation should proceed, while actually performing it. Offline approaches on the other hand include a more or less sophisticated preprocessing phase, during which supporting information is derived, and a number of control decisions that will steer the actual partial evaluation, are made. An assessment of advantages and disadvantages of either methodology is not within the scope of this thesis. We refer to [85] for further details. But it *is* important to note that, throughout our whole development in the ensuing chapters, we remain within a context of *online* partial deduction and unfolding, only presupposing a rather elementary analysis phase that discovers which predicates in a given program are recursive.

5.3 Partial Deduction in Logic Programming

Komorowski ([92]) was the first to consider partial evaluation in a logic programming context. Later, the same author proposed the use of the term *partial deduction*, since, after all, the basic computation principle in logic programming is deduction rather than evaluation (see e.g. [94]). Other *early work*, applying partial evaluation to optimise (deductive) database queries, is described in [175].

The *specialisation of logic program meta-interpreters* with respect to object programs was first explicitly studied in [64], [161] and [151]. The former concentrated on interpreters specifying control strategies, while [151] aimed at the enhancement of object programs with various extra functionalities (see also chapter 19 of [159] and section 4.3 of this thesis). [158] can be cited as a well-known example of further work along the latter line. Finally, [109] describes partial deduction of meta-programs in a context of full Prolog, including some non-logical system predicates.

Various points connected to the treatment of (impure, not always free of side effects) *Prolog* programs were also discussed in [176]. In fact, the procedural aspects of Prolog programs pose many additional challenges to a partial deduction system. So much so, that it has been proposed to keep the term *partial evaluation* in that context, to emphasise the distinction with work on pure, declarative logic programs. *Since in this thesis, we clearly stay within the latter line of research*, we do not include any further comments on issues specific to the treatment of full Prolog. For a detailed discussion of many such aspects, e.g. [153] can be consulted.

Returning to the specialisation of meta-interpreters, we can mention that some authors have explicitly addressed the (possible) *limitations* of partial deduction in that context; [173] provides one example, [129] another. Particularly in the latter paper, it is argued that partial deduction should not only rely on unfoldings to transform programs (as formally described in the next section), but also allow foldings and the definition of fresh predicates. The latter two notions are central in so-called *unfold/fold* transformations of logic programs (see e.g. [162]). Which leads us to the overall research concerning *transformation of logic programs*, a setting more general than partial deduction, both in its aims and its techniques. An interesting comparative discussion is offered in [135]. Comments pertaining to this issue can also be found in [65] and [93], two recent tutorial papers on partial deduction in logic programming.

We just note that the view taken in [129] is countered in [102], where Lakhotia argues that partial deduction should not unnecessarily be complicated: *Unfolding suffices.* In the context of concrete applications, further program optimisations can be pursued through the use of various other, complementary transformation techniques. It is this latter view, restricting the concepts involved in partial deduction to the bare essentials, which is adhered to in this thesis.

Partial deduction for some *variants and/or extensions* of the basic logic programming paradigm has also been considered. Concurrent Prolog is addressed in section 4 of [68], constraint logic programming in [157] and [79]. Another topic of interest has been the relationship between partial deduction and some reasoning schemes in artificial intelligence, particularly machine learning (see e.g. [174]).

Two noteworthy *recent developments* are the use of sophisticated *analysis* techniques ([68], [66], [70]), based on abstract interpretation ([38]), and an increasing interest in *self-application* ([127], [107] and [75]). [75] specifically addresses the partial deduction of Gödel meta-programs, for which some quite spectacular results are obtained.

Finally, we can mention that the Journal of Logic Programming recently dedicated an issue to partial deduction ([95]). Relevant material can also be found in the proceedings of the annual LOPSTR workshop on logic program transformation and synthesis (see e.g. [106]).

More can be said (and occasionally will be, see e.g. section 7.4) and many more references can be given. However, we presently conclude this short survey section and simply refer to [65] and/or [93] as possible starting points for further study. Instead, in the next section, we look in somewhat more detail at a particularly influential paper on formal foundations for partial deduction in logic programming.

5.4 Foundations

Much of the above mentioned work, though often presenting valuable ideas as well as interesting results, had a somewhat heuristic, empirical flavour, and lacked sound formal foundations. This situation was rectified by Lloyd and Shepherdson in [114]. In that paper, formal correctness criteria for partial deduction of logic programs are derived.

Our work is built on this theoretical basis. Chapter 7 in particular will rely heavily upon it. We therefore judged it appropriate to render briefly [114]'s main definitions and results. Moreover, since some of these also play a background role in those subsequent chapters not immediately concerned with overall partial deduction, but rather focusing on the control of unfolding, we decided to include a section on [114] in this preliminary chapter. Finally, since our work only addresses *definite* programs, we will adapt the treatment in [114], presented for normal programs in the context of completion semantics, to our more limited needs.

First, we extend the notion of SLD-tree by allowing it to be *incomplete*. This means that apart from success and failure nodes (occasionally called *trivial* goal nodes in what follows), also arbitrary goal statements (where no literal has been selected for further unfolding) can be leaves.

Next, we adapt the following *basic definitions on partial deduction*.

Definition 5.4.1 Let P be a definite program, A an atom and $\leftarrow A, G_1, \ldots, G_n$ with $n > 0$, an SLD-derivation for $P \cup \{\leftarrow A\}$. Let $\theta_1, \ldots, \theta_n$ be the corresponding sequence of substitutions and let G_n be $\leftarrow A_1, \ldots, A_m$.
$A\theta_1 \cdots \theta_n \leftarrow A_1, \ldots, A_m$ is called the *resultant* of the derivation $\leftarrow A, G_1, \ldots, G_n$.

Definition 5.4.2 Let P be a definite program, A an atom and τ a finite SLD-tree for $P \cup \{\leftarrow A\}$. Let $\{G_i | i = 1, \ldots, r\}$ be the (non-root) leaves of the non-failing branches of τ and $\{R_i | i = 1, \ldots, r\}$ the resultants corresponding to the

derivations $\{\leftarrow A, \ldots, G_i | i = 1, \ldots, r\}$. The set $\{R_i | i = 1, \ldots, r\}$ is called a *partial deduction for A in P*.
If $\mathbf{A} = \{A_1, \ldots, A_s\}$ is a finite set of atoms, then a *partial deduction for* $\mathbf{A}$ *in* P is the union of the partial deductions for $A_1, \ldots, A_s$ in P.
A *partial deduction of P wrt to* $\mathbf{A}$ is a definite logic program obtained from P by replacing the set of clauses in P whose head contains one of the predicate symbols appearing in $\mathbf{A}$ (called the *partially deduced predicates*) by a partial deduction for $\mathbf{A}$ in P.

We also include the following:

Definition 5.4.3 Let $\mathbf{A}$ be a finite set of atoms. We say $\mathbf{A}$ is *independent* if no pair of atoms in $\mathbf{A}$ have a common instance.

Definition 5.4.4 Let $\mathbf{S}$ be a set of first-order formulas and $\mathbf{A}$ a finite set of atoms. We say $\mathbf{S}$ is $\mathbf{A}$*-closed* if each atom in $\mathbf{S}$ containing a predicate symbol occurring in an atom in $\mathbf{A}$ is an instance of an atom in $\mathbf{A}$.

Definition 5.4.5 Let P be a definite program and G a definite goal. We say G *depends* upon a predicate p in P if there is a path from a predicate in G to p in the dependency graph for P.

Definition 5.4.6 Let P be a definite program, G a definite goal, $\mathbf{A}$ a finite set of atoms, P' a partial deduction of P wrt $\mathbf{A}$, and P^* the subprogram of P' consisting of the definitions of the predicates in P' upon which G depends. We say $P' \cup \{G\}$ is $\mathbf{A}$*-covered* if $P^* \cup \{G\}$ is $\mathbf{A}$-closed.

The following basic *soundness and completeness* theorem can now be formulated.

Theorem 5.4.7 Let P be a definite logic program, G a definite goal, $\mathbf{A}$ a finite, independent set of atoms, and P' a partial deduction of P wrt $\mathbf{A}$ such that $P' \cup \{G\}$ is $\mathbf{A}$-covered. Then the following hold :

- $P' \cup \{G\}$ has an SLD-refutation with computed answer θ iff $P \cup \{G\}$ does.
- $P' \cup \{G\}$ has a finitely failed SLD-tree iff $P \cup \{G\}$ does.

In other words, under the conditions stated in this theorem, computation with a partial deduction of a program is sound and complete with respect to computation with the original program. This is clearly a very desirable characteristic of any procedure for partial deduction. It is therefore important to devise methods for partial deduction that ensure the conditions of theorem 5.4.7 are satisfied. (In fact, the independence condition is not strictly necessary when only definite programs are considered. However, it does avoid the creation of duplicate computations in P'. For this reason, and with a possible later extension of our method to normal programs in mind, we decided to keep it.)

The presentation above reproduces the core ingredients of [114] needed in the context of our work. For further motivation, examples, proofs and results (including some on declarative semantics), we refer to [114]. Finally, it can be noted that Lloyd and Shepherdson present their work in the context of completion semantics. Recent papers have recasted it to suit other, more recently developed semantics for normal logic programs; [10] provides an example. Since we restrict our treatment in this thesis to definite logic programs, these efforts are not immediately relevant to what follows. (We refer to sections 6.6 and 7.4 for some brief further comments on negation.)

5.5 A Partial Deduction Method

A method for partial deduction of (normal) logic programs, based on the framework in [114], was first presented in [14]. Before giving a rough, somewhat simplified account of it, we include the following definition of the concept of *most specific generalisation* (or *msg*) from [14].

Definition 5.5.1 Let S be a set of atoms. Then an atom A is an *msg* of S iff

- For every atom B in S, A is more general than B.
- If C is an atom more general than each atom in S, then C is more general than A.

Notice an msg is uniquely defined up to variable renaming. (The notion of most specific generalisation was introduced in [132] and [143]. See also [105].)

Basically, the algorithm presented in [14] proceeds as follows. For a given goal G and program P, a partial deduction for G in P is computed. This is repeated for any goal occurring in the resulting clauses which is not an instance of one already processed. Assuming the procedure terminates, one gets in this way a set of clauses S and a set A of partially deduced atoms satisfying definition 5.4.4. But one also wants A to be independent. In order to achieve this, the procedure is modified as follows. Whenever a goal occurring in S is not an instance (nor a variant) of one in A, but has a common instance with it, the latter is removed from A and a partial deduction is computed for their msg (which itself is therefore added to A) and added to S. The original partial deduction for the removed goal is itself also removed from S. The process stops if A becomes independent and S A-closed. S can then be used to synthesise a partial deduction of P wrt A which satisfies the conditions of theorem 5.4.7 for any goal G' which is an instance of G.

We illustrate the algorithm with a simple example.

Example 5.5.2
source program:

$append([], Y, Y) \leftarrow$
$append([X|Xs], Y, [X|Zs]) \leftarrow append(Xs, Y, Zs)$

query:

$\leftarrow append([1, 2|Xs], [7], Zs)$

partial deduction for $\leftarrow append([1, 2|Xs], [7], Zs)$:

$append([1, 2], [7], [1, 2, 7]) \leftarrow$
$append([1, 2, X|Xs], [7], [1, 2, X|Zs]) \leftarrow append(Xs, [7], Zs)$

partial deduction for $\leftarrow append(Xs, [7], Zs)$:

$append([], [7], [7]) \leftarrow$
$append([X|Xs], [7], [X|Zs]) \leftarrow append(Xs, [7], Zs)$

resulting partial deduction of the *append* program:

$append([], [7], [7]) \leftarrow$
$append([X|Xs], [7], [X|Zs]) \leftarrow append(Xs, [7], Zs)$

This example shows how the tactic of taking msgs to make A independent causes an unacceptable loss of specialisation in the resulting partial deduction of the *append* program. To remedy this, the authors of [14] introduce a renaming transformation as a preprocessing stage before running their algorithm. It amounts to duplicating and renaming the definitions of those predicates, occurring in the original goal G, which are likely to pose specialisation problems. The details can be found in [14]; we will only reconsider the previous example.

Example 5.5.3
program after preprocessing:

$append([], Y, Y) \leftarrow$
$append([X|Xs], Y, [X|Zs]) \leftarrow append'(Xs, Y, Zs)$
$append'([], Y, Y) \leftarrow$
$append'([X|Xs], Y, [X|Zs]) \leftarrow append'(Xs, Y, Zs)$

partial deduction for $\leftarrow append([1, 2|Xs], [7], Zs)$:

$append([1, 2], [7], [1, 2, 7]) \leftarrow$
$append([1, 2, X|Xs], [7], [1, 2, X|Zs]) \leftarrow append'(Xs, [7], Zs)$

partial deduction for $\leftarrow append'(Xs, [7], Zs)$:

$append'([], [7], [7]) \leftarrow$
$append'([X|Xs], [7], [X|Zs]) \leftarrow append'(Xs, [7], Zs)$

resulting partial deduction of the *append* program:

$append([1, 2], [7], [1, 2, 7]) \leftarrow$

$append([1, 2, X|Xs], [7], [1, 2, X|Zs]) \leftarrow append'(Xs, [7], Zs)$
$append'([], [7], [7]) \leftarrow$
$append'([X|Xs], [7], [X|Zs]) \leftarrow append'(Xs, [7], Zs)$

The latter program ∪ the goal $\leftarrow append([1, 2|Xs], [7], Zs)$ is indeed covered by the independent set $\{append([1, 2|Xs], [7], Zs), append'(Xs, [7], Zs)\}$. Moreover, the result does show the desired specialisation.

One question is left more or less unanswered until now: How to obtain the (incomplete) SLD-trees used as a basis for producing partial deductions ? In other words, which computation rule should be used for building these trees (including the question of deciding when to stop the unfolding) ? [14] mentions 4 criteria and proposes the following one as the best : The computation rule R_v selects the leftmost atom which is not a variant of an atom already selected on the branch down to the current goal. (This rule was actually used in examples 5.5.2 and 5.5.3.) However, this rule fails to guarantee the production of finite SLD-trees in all cases. We present a counter-example. It is the well-known *reverse* program with accumulating parameter.

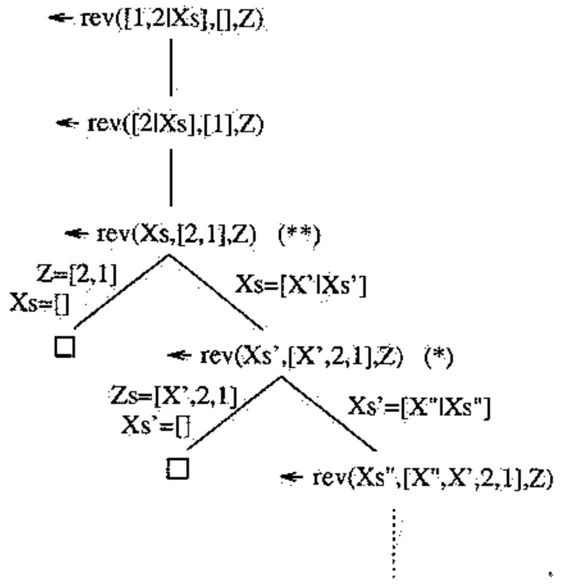


Figure 5.1: An infinite SLD-tree.

Example 5.5.4
source program:

$reverse([], Z, Z) \leftarrow$
$reverse([X|Xs], Y, Z) \leftarrow reverse(Xs, [X|Y], Z)$

query:

$\leftarrow reverse([1, 2|Xs], [], Z)$

The infinite SLD-tree, generated by R_v, is depicted in figure 5.1. Notice *reverse* has been abbreviated to *rev*. Along the rightmost branch of the tree, *rev*'s second argument grows with each unfolding. In this way, an infinite series of literals is produced, none of which is a variant of any other.

R_v and/or other computation rules have been used for loop prevention during unfolding in (a.o.) [109], [158], [161]. None of the proposed strategies, however, guarantees automatic finite unfolding in all cases (often human assistance is supposed). Imposing a depth bound on the unfolding process of course presents a solution, but seemingly in a rather ad-hoc way which does not reflect any properties of the given unfolding problem. We therefore develop an alternative approach in this thesis. (Apart from finiteness of unfolding, there is a second termination problem involved in partial deduction. It will be addressed in chapter 7.)

Chapter 6

A Framework for Finite Unfolding

6.1 Introduction

In this chapter, we present a general framework for assuring the construction of finite SLD-trees when unfolding logic programs. It is rooted in well-known techniques for proving termination of programs, based on well-founded sets, as developed by a.o. [59] and [118]. We construct unfolding algorithms and prove that they indeed terminate. We also include some brief comments on how to fully automate algorithms in a sensible way. The latter issue, however, will be addressed in much greater detail in subsequent chapters.

In our work, our basic aim is to push the unfolding of a program to its limits, while preserving finiteness of the generated SLD-tree. The limits we observe are set by the constraint that some form of data consumption should occur at each unfolding. Of course, in the context of partial deduction, maximal unfolding is not always desirable, as it can lead to an explosive growth of the program and therefore possibly deteriorates performance. We return to this and a related issue in section 6.6.

Our presentation below is structured as follows. In section 6.2, we first generalise the notion of an SLD-tree to that of an SLD$^-$-tree. We then introduce the concepts of well-foundedness and subset-wise foundedness and show how they ensure finiteness of SLD$^-$-trees. Next, section 6.3 presents a first framework and algorithm for finite unfolding, based on these ingredients. However, a detailed scrutiny of the algorithm's operation shows that it falls short of treating a large class of programs in a sensible way. A more sophisticated approach is therefore developed in section 6.4. Section 6.5 briefly addresses simulation of existing techniques, includes a discussion of some related work and sets the stage for full automation. Moreover, it can be mentioned that both sections 6.3 and 6.5 contain some reflections on the particular issues involved in unfolding meta-interpreters. Some concluding remarks round off the chapter in section 6.6.

Finally, we point out that the material included in this chapter is largely identical to the contents of [27], an abridged, preliminary version of which can be found in [26].

6.2 Well-Founded and Subset-Wise Founded SLD$^-$-Trees

6.2.1 Well-founded sets and trees

It is clear that, given a program P, an atomic goal $\leftarrow A$ and a computation rule R for $P \cup \{\leftarrow A\}$, there can be (possibly infinitely) many different SLD-trees τ for $P \cup \{\leftarrow A\}$ under R. These are all subtrees of the complete SLD-tree τ_0 for $P \cup \{\leftarrow A\}$ under R. The problem we address is how to select a finite subtree of τ_0 that forms a suitable basis for partial deduction. First, we introduce a concept useful for setting up our general framework.

Definition 6.2.1 Let P be program and $\leftarrow A$ a goal. Then we call any subtree of an SLD-tree τ for $P \cup \{\leftarrow A\}$, having the same root as τ, an *SLD$^-$-tree* (for $P \cup \{\leftarrow A\}$).

The SLD$^-$-tree concept is more general than that of an (incomplete) SLD-tree. Indeed, it allows branching points where some, but not all possible branches are actually included in the tree. For any SLD$^-$-tree τ, we denote by τ^+ the *unique* minimal SLD-tree containing it. Obviously, τ^+ can be obtained from τ by adding to τ the first derivation step from each branch (originating from a non-leaf) missing in τ.

Two basic ingredients of our approach are *strict order relations*, denoted $>$, and *well-founded measures*. A strict order relation is an anti-reflexive, anti-symmetric and transitive binary relation. A (partially) strictly ordered set, $V, >_V$, will be called an *s-poset*, the corresponding order, $>_V$, an *s-order*.

Definition 6.2.2 An s-poset $V, >$ is called *well-founded* if there is no infinite sequence of elements $e_1, e_2, \ldots$ in V such that $e_i > e_{i+1}$, for all $i \geq 1$.

Definition 6.2.3 Let $V, >_V$ be an s-poset. A *well-founded measure, f, on $V, >_V$* is a monotonic function from $V, >_V$ to some well-founded set $W, >_W$.

As discussed in e.g. [49], well-founded sets are a commonly used tool for proving termination of programs. [59] applied them to prove termination of imperative programs, [118] and also [49] itself for dealing with termination of production systems. Some further references (particularly pertaining to the latter issue) can be found in [49].

Let us now return to the context of SLD-tree construction. Given an SLD$^-$-tree τ, an s-poset can be associated with it in a natural way, by taking the goals in the tree as elements of the set and the tree structure itself as a strict ordering on this set. For technical reasons, the actual definition of the associated s-poset will be slightly more complicated. Let τ_0 be the complete (possibly infinite) SLD-tree of which τ forms part. We assume a numbering on the nodes of τ_0 (e.g. left-to-right, top-down and breadth-first). Then we can associate with τ the following set:

$G_\tau = \{(G, i) | G$ is a goal of τ having i as its associated number in $\tau_0\}$

(With slight abuse of notation, we will occasionally also refer to pairs (G, i) as goals or nodes in τ or τ_0 (instead of G_τ or G_{τ_0}).) Considering $(G, i) >_\tau (G', j)$ if node i is an ancestor of node j in τ_0 results in $G_\tau, >_\tau$ being an s-poset. This allows the following definition:

Definition 6.2.4 An SLD$^-$-tree τ is *well-founded* if there exists a well-founded measure f on $G_\tau, >_\tau$.

We are now in a position to formulate a first basic theorem underlying our approach.

Theorem 6.2.5 An SLD$^-$-tree τ is finite iff it is well-founded.

Proof For the if-part, assuming that τ has an infinite derivation, we can construct a sequence $(G_1, i_1) >_\tau (G_2, i_2) >_\tau \ldots$ in $G_\tau, >_\tau$. Applying the well-founded measure f to it, we get an infinite sequence $f(G_1, i_1) >_\tau f(G_2, i_2) >_\tau \ldots$ in $W, >_W$. For the only-if-part, since $G_\tau, >_\tau$ is a well-founded s-poset itself, take f equal to the identity function, 1_{G_τ}. □

Strictly speaking, in the above proof, we should of course have used the notation $f((G, i))$. However, in order not to overload the notation, we have left out the extra pair of brackets, clear from the context. We will proceed similarly on various occasions throughout what follows.

Now, how can we use definition 6.2.4 and theorem 6.2.5 operationally ? Well, for a start, we choose a function $f : \tau_0 \to W, >_W$, where $W, >_W$ is some well-founded s-poset. For an initial $\tau = \{(\leftarrow A, 1)\}$, the restriction of f to the domain τ, $f|_\tau$, is obviously monotonic. Then, we control the construction of the SLD$^-$-tree τ, by imposing that each unfolding must preserve the monotonicity of $f|_\tau$. The resulting SLD$^-$-tree will be the largest SLD$^-$-subtree τ of τ_0 such that $f|_\tau$ is monotonic. Clearly, it is well-founded through f and therefore finite. As final finite SLD-tree serving as a basis for partial deduction, we then take the SLD-tree

τ^+ corresponding to τ. This basic strategy of unfolding in such a way that some given measure function is kept monotonic, is the motivation for introducing the terminology *prefounding* in section 6.3 below.

6.2.2 Subset-wise founded trees

Of course, the problem of obtaining *sensibly* expanded SLD-trees has now been shifted to that of finding sensible functions on τ_0. Although for each finite SLD$^-$-subtree τ, a corresponding function on τ_0 exists, it may be hard to generate useful functions that lead to sensible SLD-trees. Therefore, we introduce a more refined characterisation of finite SLD$^-$-trees, in which several measures can be combined. Each such measure will focus on a separate part of the recursive inference expressed in the SLD$^-$-tree. As a result, it will be more easy to generate them.

The inspiration for these more general measures stems from certain criteria which seem reasonable in practice to control unfolding. When building an SLD-tree, one tends to compare the literal selected for unfolding with the selected literals in the ancestor goals of the same derivation. Especially if two such subgoals are calls to the same predicate, a comparison may allow the avoidance of loops in the derivation. Within the context of our approach, this corresponds to the observation that it is not really necessary to have a measure that decreases at each derivation step, but that the measure should decrease between each two goals in the same derivation, where a literal with a same recursive predicate was selected for unfolding.

A second motivation for allowing more liberal measures is rooted in the application of partial deduction to meta-interpreters. Typically, for vanilla-type meta-interpreters (see definition 4.6.10), one is not really interested in comparing the measure assigned to two goals, unless the selected literals are both of type $solve(A)$. More in particular, one almost never wants to stop the unfolding of a subgoal of type $solve(A\&B)$ (or $solve(empty)$). Therefore, measures assigned to goals where such literals are selected should not interfere with the control of unfolding. Or, more radically, no measure at all should be attached to them.

These two considerations lead to the following generalised notion of foundedness.

Definition 6.2.6 An SLD$^-$-tree τ is *subset-wise founded* if

1. There exists a finite number of sets, $C_0, \ldots, C_N$, such that $G_\tau = \bigcup_{i \leq N} C_i$.

2. For each $i = 1, \ldots, N$, there exists a well-founded measure
   $f_i : C_i, >_\tau \rightarrow W_i, >_i$.

3. For each $(G, k) \in C_0$ and each derivation D in τ containing (G, k):

- either D is finite
- or there exists a descendant (G', j) of (G, k) in D such that $(G', j) \in C_i$ for some $i > 0$.

This definition makes both observations above explicit:

- Not every two goals in a same derivation must be comparable. The measure function only needs to decrease if the goals belong to the same set C_i.
- Some goals (the ones in C_0) are disregarded completely. Imposing condition 3 on C_0 ensures that this can be done safely. Indeed, it demands that goals in C_0 have a descendant within the same derivation, which is either contained in a set with a measure function, $C_i, i > 0$, or which is a leaf.

Throughout what follows, we will denote by $R(G, i)$ the literal selected in a given goal (G, i) in an SLD$^-$-tree τ by the computation rule R, used to construct τ. This being agreed upon, let us consider an example illustrating definition 6.2.6.

Example 6.2.7 Consider the following simple program, defining the relation *contains_pal* where *contains_pal*(X, Y) holds if Y is an initial sublist of a list X and Y is a palindrome.

$$\begin{array}{l}
\mathit{contains\_pal}(X, Y) \leftarrow \mathit{i\_sublist}(Y, X), \mathit{palindrome}(Y) \\
\mathit{i\_sublist}([], Ys) \leftarrow \\
\mathit{i\_sublist}([X|Xs], [X|Ys]) \leftarrow \mathit{i\_sublist}(Xs, Ys) \\
\mathit{palindrome}(X) \leftarrow \mathit{reverse}(X, [], X) \\
\mathit{reverse}([], Zs, Zs) \leftarrow \\
\mathit{reverse}([X|Xs], Ys, Zs) \leftarrow \mathit{reverse}(Xs, [X|Ys], Zs)
\end{array}$$

Choose:

$$\begin{array}{l}
C_0 = \{(G, k) | R(G, k) \text{ contains } \mathit{contains\_pal} \text{ or } \mathit{palindrome} \text{ and } k \in \mathbb{N}\} \\
\qquad \cup \{(\Box, k) | k \in \mathbb{N}\} \\
C_1 = \{(G, k) | R(G, k) \text{ contains } \mathit{i\_sublist} \text{ and } k \in \mathbb{N}\} \\
C_2 = \{(G, k) | R(G, k) \text{ contains } \mathit{reverse} \text{ and } k \in \mathbb{N}\}
\end{array}$$

and:

$f_1 : C_1 \to \mathbb{N}, f_1(G, k) =$ the number of function symbols in the second argument of $R(G, k)$ (this is the input argument of *i_sublist*)

$f_2 : C_2 \to \mathbb{N}, f_2(G, k) =$ the number of function symbols in the first argument of $R(G, k)$ (the input argument of *reverse*)

Finally, assume that R, in each goal, selects the leftmost literal where at least one argument contains a constant. A maximally large subset-wise founded SLD$^-$-tree τ for

$$\leftarrow \mathit{contains\_pal}([3|X], Y)$$

is depicted in figure 6.1. Selected literals are underlined, and predicate names have been abbreviated in an obvious way.

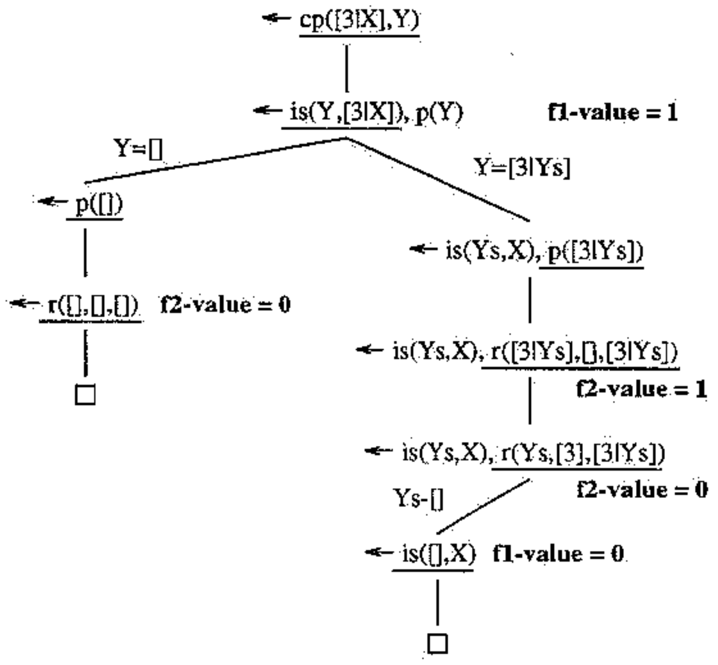


Figure 6.1: A subset-wise founded SLD$^-$-tree.

Observe that every well-founded SLD$^-$-tree τ is subset-wise founded. Just take $N = 1$, $C_1 = G_\tau$, $C_0 = \emptyset$, $f_1 = f$. A second basic theorem follows.

Theorem 6.2.8 An SLD$^-$-tree τ is finite iff it is subset-wise founded.

Proof First, if τ is finite, then it is well-founded and therefore subset-wise founded. Conversely, let τ be subset-wise founded and assume that it is infinite. Since τ contains an infinite derivation D and G_τ is included in the union of $C_0, \ldots, C_N$, there exists a set C_i, such that $C_i \cap D$ is infinite. On $C_1, \ldots, C_N$ we have a well-founded measure. Thus, i must be 0.
Now, since D is infinite, by definition 6.2.6, for each (G, k) in $C_0 \cap D$, there exists a (G', l) in some $C_j \cap D, j > 0$, such that $(G, k) >_\tau (G', l)$. Because $C_0 \cap D$ is infinite and strictly decreasing in $G_\tau, >_\tau$, this allows us to construct an infinite number of elements in $\bigcup_{j>0} C_j$. Thus, one of the sets $C_j \cap D$ must be infinite, which contradicts the existence of f_j. □

Notice that as a consequence, the notions of a finite, a well-founded and a subset-wise founded SLD-tree coincide. This makes us return to the question why it is useful to introduce the subset-wise founded measures at all. The answer is that interesting subsets of goal statements can be defined and that simple well-founded measures can be provided on them. This is especially useful if we aim to use the given theorems operationally, in the sense that we first define sets and measure functions, and consecutively construct the SLD-tree under the constraints imposed by the measures. Finding sensible and useful well-founded measures on the entire tree is often more complex.

So, in the next two sections, we present operational counterparts of the above introduced static notions and unfolding algorithms based on them. Proceeding thus, formal finiteness and termination proofs will be possible. Moreover, we will also explicitly demonstrate the links with the static framework we have established in the present section.

6.3 Using Finite Prefoundings

In this section, we present a first operational derivative of the concepts described above. And we base on it a first algorithm for finite unfolding.

Throughout the rest of this chapter, P denotes a definite program, $\leftarrow A$ a definite goal with one atom, $\mathcal{L}$ the language underlying both P and $\leftarrow A$, R a computation rule for $P \cup \{\leftarrow A\}$ and τ_0 the complete SLD-tree for $P \cup \{\leftarrow A\}$ under R.

6.3.1 Finite prefoundings

We now introduce the notion of a *prefounding*, announced at the end of subsection 6.2.1. The definition of a *finite prefounding*, $((C_0, \ldots, C_N), (f_1, \ldots, f_N))$, *for* τ_0, below, is essentially identical to the conditions imposed on the pair $((C_0, \ldots, C_N), (f_1, \ldots, f_N))$ in the definition of a subset-wise founded SLD$^-$-tree τ (definition 6.2.6), except that:

- τ is replaced by τ_0 (in particular, $(C_0, \ldots, C_N)$ is a covering of τ_0)
- $f_i : C_i \rightarrow W_i, >_i$ $(i > 0)$ is not required to be monotonic

Given such a prefounding, it will be possible to construct a subset-wise founded (and therefore finite) SLD$^-$-tree τ, by unfolding goals as long as the functions $f_i, i > 0$, remain monotonic on the restricted domains $C_i \cap \tau$.

Definition 6.3.1 A pair $((C_0, \ldots, C_N), (f_1, \ldots, f_N))$ is a *finite prefounding for* τ_0 if

1. Each C_i, $i \leq N$, is a set of pairs (G, k), such that G is a goal in $\mathcal{L}$, $k \in I\!N$ and $G_{\tau_0} \subseteq \bigcup_{i \leq N} C_i$.
2. For each $i = 1, \ldots, N$, f_i is a function, $f_i : C_i \to W_i, >_i$, where $W_i, >_i$ is a well-founded s-poset.
3. For each pair $(G, k) \in C_0$ and each derivation D in τ_0 containing (G, k):
   - either D is finite
   - or there exists a descendant (G', j) of (G, k) in D such that $(G', j) \in C_i$ for some $i > 0$.

Before looking at an example of a finite prefounding, we introduce a class of practical measure functions.

Definition 6.3.2 Let *Term* denote the set of terms in $\mathcal{L}$. We define the *functor norm* as the function $|.| : Term \to I\!N$:

If $t = f(t_1, \ldots, t_n), n > 0$
then $|t| = 1 + |t_1| + \cdots + |t_n|$
else $|t| = 0$

The functor norm counts the number of functors in a given term. Alternatively, we could have chosen to count the number of constants as well. However, we do not expect that this leads to an improved measure function with respect to termination properties. (See however section 8.5 for unfolding where constants do matter.)

Definition 6.3.3 Let p be a predicate of arity n and $S = \{a_1, \ldots, a_m\}, 1 \leq a_k \leq n, 1 \leq k \leq m$, a set of argument positions for p. We define the *functor measure* with respect to p and S as the function

$|.|_{p,S} : \{A | A$ is an atom with predicate symbol $p\} \to I\!N$:
$|p(t_1, \ldots, t_n)|_{p,S} = |t_{a_1}| + \cdots + |t_{a_m}|$

Both definition 6.3.2 and definition 6.3.3 will be heavily used in the context of automation throughout the following two chapters. Let us now illustrate in which way natural and useful finite prefoundings can often be constructed.

Example 6.3.4 Suppose the program P has (a finite number of) recursive predicates $p_1, \ldots, p_N$. Given the computation rule R, we take:

$C_i = \{(G, k) | G$ a goal such that $R(G, k)$ contains p_i and $k \in I\!N\}$, for $i > 0$
$C_0 = \{(G, k) | G$ a goal such that $R(G, k)$ contains a non-recursive predicate symbol and $k \in I\!N\} \cup \{(\square, k) | k \in I\!N\}$

Clearly, $G_{\tau_0} \subseteq \bigcup_{i\leq N} C_i$. Also, if (G, k) is in a derivation D of τ_0, then either D is finite or there is a descendant (G', j) of (G, k) in D, such that $R(G', j)$ is an atom with a recursive predicate (independent of whether or not (G, k) is in C_0). So, the conditions 1 and 3 of definition 6.3.1 are fulfilled.
The functor measure induces in a natural way functions f_i on the sets $C_i, i > 0$. For each set C_i, let S_i be a set of argument positions of interest for the predicate p_i. Essentially, these will be the input arguments for p_i. (Later, we will determine these in an automated way.) Then, define f_i to be $|.|_{p_i,S_i}$.
In particular, the pair $((C_0, C_1, C_2), (f_1, f_2))$ as presented in example 6.2.7 is a concrete instance of this generic example. Indeed, *i_sublist* and *reverse* are the program's recursive predicates, $f_1 = |\cdot|_{i\_sublist,\{2\}}$ and $f_2 = |\cdot|_{reverse,\{1\}}$.

6.3.2 A first algorithm

Given a finite prefounding $((C_0, \ldots, C_N), (f_1, \ldots, f_N))$, the following algorithm computes a finite SLD$^-$-subtree τ of τ_0, which is subset-wise founded with respect to the s-posets $C_0 \cap \tau, >_\tau, \ldots, C_N \cap \tau, >_\tau$ and the functions $f_1, \ldots, f_N$ restricted to these s-posets. The basic criterion for unfolding used in the algorithm is that each of the functions $f_1, \ldots, f_N$ should invariantly remain monotonic on its domain.

Algorithm 6.3.5

```
Initialisation
  $\tau := \{\{(\leftarrow A, 1)\}\}$ {* an SLD-tree with a single one-node derivation *}
  $Terminated := \emptyset$
While there exists a derivation $D \in \tau$ such that $D \notin Terminated$ do
  Let $(G, i)$ be the leaf of $D$
  Let $Derive(G, i)$ be the set of all its immediate $>_{\tau_0}$-descendants
  Let $Decrease(G, i)$ be the set of all $(G', j) \in Derive(G, i)$, such that
    for all $k > 0$, such that $(G', j) \in C_k$,
    for all $(G'', j') \in D \cap C_k$:
      $f_k(G'', j') >_k f_k(G', j)$
  If $Decrease(G, i) = \emptyset$
    Then add $D$ to $Terminated$
  Else {* $\tau$ is further extended *}
    Replace $\tau$ by $\tau \setminus \{D\} \cup \{D \cup \{(G', j)\} | (G', j) \in Decrease(G, i)\}$
Endwhile
```

Finally, a finite SLD-tree for $P \cup \{\leftarrow A\}$ under R is obtained by extending τ into τ^+, its minimal containing SLD-tree.

Let us reconsider the above example.

Example 6.3.6 For the *contains_pal* program of example 6.2.7 and using the finite prefounding mentioned in example 6.3.4, the SLD$^-$-tree produced by algorithm 6.3.5 is exactly the one depicted in figure 6.1. The corresponding SLD-tree contains one additional node:

$\leftarrow \mathit{i\_sublist}([Y'|Ys'], X), \mathit{reverse}(Ys', [Y', 3], [3, Y'|Ys'])$

which is a child node of:

$\leftarrow \mathit{i\_sublist}(Ys, X), \mathit{reverse}(Ys, [3], [3|Ys])$

where Ys has been unified with $[Y'|Ys']$. This leads to the following partial deduction for

$\leftarrow \mathit{contains\_pal}([3|X], Y)$

in the *contains_pal* program (definition 5.4.2):

$\mathit{contains\_pal}([3|X], []) \leftarrow$
$\mathit{contains\_pal}([3|X], [3]) \leftarrow$
$\mathit{contains\_pal}([3|X], [3, Y'|Ys']) \leftarrow \mathit{i\_sublist}([Y'|Ys'], X),$
$\qquad \mathit{reverse}(Ys', [Y', 3], [3, Y'|Ys'])$

We have the following theorem:

Theorem 6.3.7 Algorithm 6.3.5 terminates. The resulting finite SLD$^-$-tree τ is subset-wise founded with respect to the sets $C_0 \cap \tau, \ldots, C_N \cap \tau$ and the well-founded measures $f_1, \ldots, f_N$ restricted to these sets.

Proof In case the algorithm would not terminate, it would construct at least one infinite derivation. We can now argue along the same lines as in the proof of theorem 6.2.8. To prove subset-wise foundedness, observe that by the first condition of definition 6.3.1, the sets $C_0 \cap \tau, \ldots, C_N \cap \tau$ cover τ, as required by the first condition of definition 6.2.6. For each of the s-posets $C_i \cap \tau, >_\tau, i > 0$, the corresponding f_i is monotonic by construction, so condition 2 of definition 6.2.6 is satisfied. Finally, its third and last condition is identical to the third condition in definition 6.3.1, restricted to τ. □

It is possible to formulate a similar algorithm that immediately computes an SLD-tree. It suffices to adapt the stopping criterion in such a way that (all possible) unfoldings are carried out as long as the relevant measure function is monotonic *on the node to be unfolded*. The resulting tree would be subset-wise founded with the exception of those of its leaves that are in a $C_i, i > 0$. We return to this issue, and actually exhibit one such algorithm, in subsection 6.5.2.

6.3.3 Unfolding meta-interpreters

The second motivation we mentioned for introducing both the concept of a subset-wise founded SLD$^-$-tree and that of a finite prefounding, was the application of partial deduction to meta-interpreters. As recognised in a.o. [158] and [103], it is certainly desirable that if partial deduction is applied to meta interpreters, it should at least perform the parsing task. This leads to the somewhat ad hoc rule that for interpreters of the vanilla-type, calls to *solve* where the argument is a conjunction of multiple object level atoms, should always be unfolded. On the other hand, for *solve* calls with an argument containing only one object level atom, the control of the unfolding should be based on properties of the atom itself.

This idea fits well within the context of a finite prefounding. Assume that $p_1/n_1, \ldots, p_N/n_N$ are the recursive predicates (with their arities) of the object program. Then we introduce the following sets:

$C_i = \{(G,k) | R(G,k)$ is an atom of type $solve(p_i(t_1, \ldots, t_{n_i}))$,
where $t_j, 1 \leq j \leq n_i$ are any terms, and $k \in I\!N\}$

$C_0 = \{(G,k) | R(G,k)$ is an atom of type $solve(p(t_1, \ldots, t_n))$, where p/n is a non-recursive predicate, $t_j, 1 \leq j \leq n$ terms, and $k \in I\!N\}$
$\cup \{(G,k) | R(G,k)$ is an atom of type $solve(A\&B)$ and $k \in I\!N\}$
$\cup \{(G,k) | R(G,k)$ is an atom of type $solve(empty)$ and $k \in I\!N\}$
$\cup \{(G,k) | R(G,k)$ is an atom of type $clause(A,B)$ and $k \in I\!N\}$
$\cup \{(\Box, k) | k \in I\!N\}$

Again, functor measures are appropriate for the functions f_i on these sets, although we would want more expressivity in the sense that subterms of arguments should be measured instead of just arguments. This is achieved by allowing selector functions. In the following definition, $Atom_P$ and $Term_P$ denote the sets of atoms and terms in $\mathcal{L}$ respectively.

Definition 6.3.8 A *selector function* s (for P), denoted as a finite, non-empty sequence of positive integers connected with slashes, $n_1/n_2/\ldots/n_k$, is a (partial) function: $Atom_P \cup Term_P \rightarrow Term_P$, recursively defined as follows:

```
**If** $s = n$ and $n \leq m$
  **Then** $s(r(t_1, \ldots, t_m)) = t_n$
**Else if** $s = n_1/n_2/\ldots/n_k$, $n_1 \leq m$ and $n_2/\ldots/n_k(t_{n_1})$ is defined
  **Then** $s(r(t_1, \ldots, t_m)) = n_2/\ldots/n_k(t_{n_1})$
**Else** $s(r(t_1, \ldots, t_m))$ is undefined.
```

It is not difficult to generalise definition 6.3.3 of a functor measure with respect to a predicate p and a set of argument positions S to that of a functor measure with respect to p and a set of selector functions S (see definition 8.6.6).

Then, when the set of interesting argument positions for the recursive predicate p_i of the object program is $S_{p_i} = \{a_1, \ldots, a_k\}$, let $S'_{p_i} = \{1/a_1, \ldots, 1/a_k\}$, and define:

$f_i = |\cdot|_{solve, S'_{p_i}}$, for $i = 1, \ldots, N$

In view of full automation of partial deduction applied to meta-interpreters, several other issues need to be addressed, such as:

- how to recognize meta-interpreters
- how to distinguish in general between the meta-predicates that perform the parsing and the arguments within them that contain the object program goals
- how to avoid loops for those predicates in the meta interpreter that perform tasks different from the parsing

We return to these issues in subsection 6.5.3.

6.4 Using Hierarchical Prefoundings

6.4.1 A motivating example

Concrete finite prefoundings of the kind introduced in example 6.3.4, although not difficult to generate and handle automatically, and effective in many cases, have at least one clear disadvantage. The following simple example exposes the problem.

Example 6.4.1 In the program below, the predicate *sums* holds if its second argument is the list of all sums that can recursively be computed for the numbers in the list that is its first argument, in the tail of that list, and so on.

$sums([],[]) \leftarrow$
$sums([X|Xs],[Y|Ys]) \leftarrow sum([X|Xs],Y), sums(Xs,Ys)$
$sum([],0) \leftarrow$
$sum([X|Xs],Y) \leftarrow sum(Xs,Z), add(Z,X,Y)$

where $add(X,Y,Z)$ is supposed to be defined through a number of tuples, and implements the addition of natural numbers. Now, we apply algorithm 6.3.5 with:

$A = sums([1,2],Y)$

and the finite prefounding $((C_0,C_1,C_2),(f_1,f_2))$, where:

$C_0 = \{(G,k)|R(G,k) \text{ has predicate symbol } add\} \cup \{(\Box,k)|k \in I\!N\}$
$C_1 = \{(G,k)|R(G,k) \text{ has predicate symbol } sums\}$
$C_2 = \{(G,k)|R(G,k) \text{ has predicate symbol } sum\}$

and:

$f_1 = |\cdot|_{sums,\{1\}}$
$f_2 = |\cdot|_{sum,\{1\}}$

The resulting subset-wise founded SLD$^-$-tree is shown in figure 6.2. Arrows indicating the structure of the associated proof tree are added.

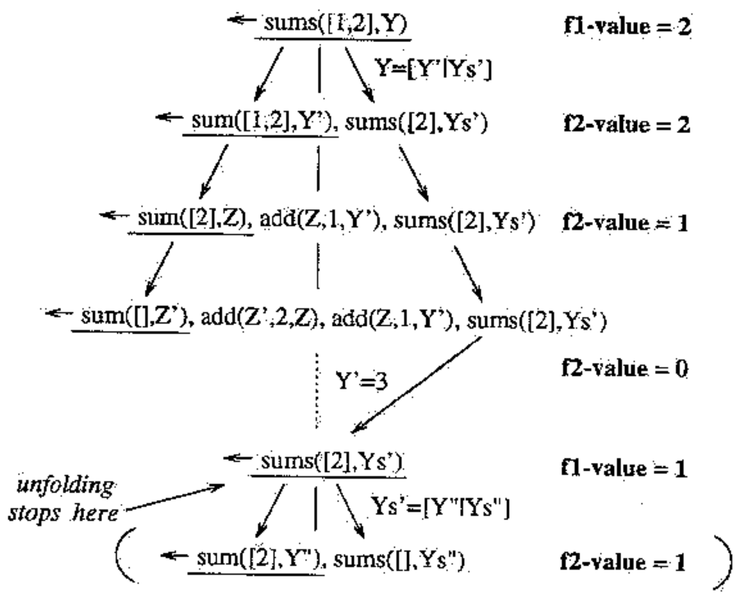

← sums([1,2],Y)
f1-value = 2
Y=[Y'|Ys']
← sum([1,2],Y'), sums([2],Ys')
f2-value = 2
← sum([2],Z), add(Z,1,Y'), sums([2],Ys')
f2-value = 1
← sum([],Z'), add(Z',2,Z), add(Z,1,Y'), sums([2],Ys')
Y'=3
f2-value = 0
← sums([2],Ys')
f1-value = 1
unfolding stops here
Ys'=[Y''|Ys'']
← sum([2],Y''), sums([],Ys'')
f2-value = 1

Figure 6.2: A tree too small.

Clearly, unfolding is stopped prematurely. The reason for this unwanted behaviour is easily discovered. The culprit is the comparison of the f_2-value (1) associated with the goal between brackets with the first one met when mounting the derivation (0). However, although the selected literals in both goals involved obviously contain the same recursive predicate symbol *sum*, it would be appropriate not to compare them. Indeed, the second is not a recursive descendant of the first. It can therefore safely be unfolded, in spite of its larger weight.

The problem occurs systematically when we have clauses of the type:

$p(\ldots) \leftarrow q(\ldots), p(\ldots)$
$q(\ldots) \leftarrow q(\ldots)$

or:

$p(\ldots) \leftarrow p(\ldots), p(\ldots)$

In both cases, the SLD-tree τ_0 contains derivations with selected literals featuring the same predicate symbol in goals that are in different branches of the associated proof tree. Many familiar logic programs are of this type, e.g. *naive_reverse*, *permutation_sort*, *quicksort*, *n_queens*.

For the *sums*-program and -query in example 6.4.1, the erratic behaviour of algorithm 6.3.5 can easily be remedied. It suffices to provide a better tuned finite prefounding. Replace C_2 by:

$C_{21} = \{(G,k) \in C_2 | R(G,k)$ is a descendant of $sum([1,2],Y')$ in the associated proof tree$\}$

$C_{22} = \{(G,k) \in C_2 | R(G,k)$ is a descendant of $sum([2],Y'')$ in the associated proof tree$\}$

However, not only is this technique impractical in view of automation, but the underlying reason why it works in example 6.4.1 is that τ_0 is finite. In general, if we want to cover an SLD-tree with sets C_i such that goals with selected literals originating from different branches in the proof tree are placed in different sets C_i, an infinite number of such sets are needed. This observation motivates the the concept of a *hierarchical prefounding*, introduced in the next subsection.

6.4.2 Hierarchical prefoundings

Definition 6.4.5 below recasts definition 6.3.1 and formally describes the answer to our needs. First, however, we introduce a refined ancestor concept on literals and goals, and prove an important property.

Definition 6.4.2 Let $(G,i) = ((\leftarrow A_1,\ldots,A_j,\ldots,A_n),i)$ be a node in an SLD$^-$-tree τ, let $R(G,i) = A_j$ be the call selected by the computation rule R, suppose that $H \leftarrow B_1,\ldots,B_m$ is a clause whose head unifies with A_j and let $\theta = mgu(A_j,H)$ be their most general unifier. Then (G,i) has a son (G',k) in τ, $(G',k) = ((\leftarrow A_1,\ldots,A_{j-1},B_1,\ldots,B_m,A_{j+1},\ldots,A_n)\theta,k)$. Let (G'',l) be a descendant of (G',k) in τ with $R(G'',l) = B_r\theta\psi$, for some $r \leq m$ and ψ the composition of all mgu's on the subderivation from (G',k) to (G'',l). We say that $B_r\theta\psi$ in (G'',l) is a *direct descendant* of A_j in (G,i) and that A_j in (G,i) is a *direct ancestor* of $B_r\theta\psi$ in (G'',l).

Stated otherwise, the selected literals of two goals in an SLD$^-$-tree τ are in the direct descendant relation if in the proof tree associated to τ, the two nodes labeled by these atoms are connected by an arc.

Definition 6.4.2 (Continued) The binary relations *descendant* and *ancestor*, defined on (selected) literals in goals, are the transitive closures of the direct descendant and direct ancestor relations respectively. For A an atom in (G,i) and B an atom in (G',j), A is an ancestor of B is denoted as $A >_{pr} B$ ("pr" stands for proof tree).
The relations descendant and ancestor on pairs of atoms in goals of τ induce in a natural way descendant and ancestor relations on τ. Let (G,i) and (G',k) be in τ. We call (G,i) a *proper ancestor* of (G',k), if $(G,i) >_{\tau_0} (G',k)$ and $R(G,i) >_{pr} R(G',k)$. Abusing notation, we denote the latter relation between *goals* as $(G,i) >_{pr} (G',k)$.

Definition 6.4.3 Let τ be an SLD$^-$-tree. We call a (possibly infinite) sequence $(G_{i_1}, i_1) >_{pr} (G_{i_2}, i_2) >_{pr} \ldots$ of goals in τ a ***proper achain***, if for each $m \geq 1$, $R(G_{i_m}, i_m)$ in (G_{i_m}, i_m) is a direct ancestor of $R(G_{i_{m+1}}, i_{m+1})$ in $(G_{i_{m+1}}, i_{m+1})$.

The following proposition is fairly immediate, but important background for what follows, and therefore stated explicitly.

Proposition 6.4.4 If a derivation D in an SLD$^-$-tree contains no infinite proper achain, then it contains only a finite number of proper achains.

Proof D can only contain an infinite number of proper achains if D is infinite. But then the associated proof tree contains an infinite branch, implying that D contains an infinite proper achain. □

The notion of a ***hierarchical prefounding for*** τ_0, which we now introduce, differs from that of a ***finite prefounding*** in essentially two ways.

- First, it is *more specific* in the sense that, instead of classifying (with possible overlaps) the goals of τ_0 into a finite number of sets C_i, we now initially partition the selected subgoals of the goals in τ_0 into a finite number of sets R_k and, in a second phase and for each R_k separately, we classify (with possible overlaps) the goals with a selected subgoal in R_k into a number of sets C_i. (Notice that hierarchical prefoundings hardwire into the framework the assumption that comparing selected literals plays a basic role in finite unfolding algorithms.) In this second phase, the classification is completely deterministic: Two goals (G, n) and (G', m) with their selected literal in a particular R_k are placed in the same $C_i, i > 0$, if one of them is a proper ancestor of the other (in other words, if either $(G, n) >_{pr} (G', m)$ or $(G', m) >_{pr} (G, n)$).

- Secondly, in another respect, hierarchical prefoundings are *more general* than finite ones, because the number of sets C_i is allowed to be infinite.

Definition 6.4.5 A pair $((C_0, C_1, C_2, \ldots), (f_1, f_2, \ldots))$ is a ***hierarchical prefounding*** for τ_0 if:

1. There exists a finite partition $R_0, \ldots, R_N$ of $R_{\tau_0} = \{R(G, i) | (G, i) \in \tau_0\}$ such that:

   - $\forall (G, i) \in \tau_0 \setminus C_0$:
     $C_i = \{(G, i)\} \cup$
     $\{(G', j) \in \tau_0 | \exists k : 1 \leq k \leq N$, such that $R(G, i), R(G', j) \in R_k$ and $(G, i) >_{pr} (G', j)$ or $(G', j) >_{pr} (G, i)\}$
   - $C_0 = \{(G, i) \in \tau_0 | R(G, i) \in R_0\} \cup \{(\square, i) \in \tau_0\}$

2. $f_1, f_2, \ldots$ are functions, respectively mapping $C_1, C_2, \ldots$ to one of a finite number of well-founded sets $W_1, >_1, \ldots, W_N, >_N$ such that f_i maps C_i to W_k if the selected literals of the goals in C_i belong to R_k. Moreover, for all $i, j > 0 : f_i|_{C_i \cap C_j} = f_j|_{C_i \cap C_j}$.

3. C_0 contains no infinite proper achain.

Before giving an example, a comment on the relation between the third condition above and condition 3 in definition 6.3.1 seems appropriate. Indeed, in terms of the associated proof tree's structure, the condition above can be reformulated as follows:

3. For each pair $(G, k) \in C_0$ and for each branch B in the proof tree associated with τ_0 containing $R(G, k)$:
   - either B is finite
   - or there exists a descendant (G', j) of (G, k) with $R(G', j)$ a descendant of $R(G, k)$ in B such that $(G', j) \in C_i$, for some $i > 0$.

This formulation is completely similar to the one of condition 3 in definition 6.3.1, with the branch B in the proof tree playing the role formerly allotted to the derivation D in τ_0.

The following example introduces a generic type of hierarchical prefoundings that will prove to be useful in many practical cases.

Example 6.4.6 Assume that the recursive predicate symbols of the program P are $p_1, \ldots, p_N$. Then define:

$R_k = \{R(G, i) \in R_{\tau_0} | R(G, i) \text{ contains } p_k\}$, where $k > 0$

$R_0 = \{R(G, i) \in R_{\tau_0} | R(G, i) \text{ contains a non-recursive predicate symbol}\}$

In this way, C_0 winds up being basically identical to C_0 in example 6.3.4. For each node $(G, i) \in \tau_0 \setminus C_0$, C_i is the set of all ancestors and descendants of (G, i) such that their selected literal under R:

- contains the same (recursive) predicate symbol as $R(G, i)$
- and is a $>_{pr}$-ancestor or -descendant of $R(G, i)$

Now, assume that the functions f_i are functor measures (definition 6.3.3) on the atoms selected in goals $(G', j) \in C_i$. Then it should be clear that, in order to decide on the unfolding of $R(G, i)$, the ancestor goals of (G, i) in C_i (and their selected literals) are precisely what we want to compare with. Example 6.4.13 below will provide a concrete instance of this kind of hierarchical prefoundings.

Definition 6.4.5 itself is more general than the above generic example. In particular, it does not impose the one-to-one correspondence between R_i-sets and recursive predicate symbols. And indeed, in general (e.g. when dealing with meta-interpreters, see subsection 6.3.3), more refined partitions of R_{τ_0} may be useful. We return briefly to this issue in subsection 6.5.3 and more extensively, in a context of full automation, in section 8.6.

The term "*hierarchical* prefounding" refers to the fact that if such a prefounding exists, then the nodes of τ_0 can be partitioned into hierarchical layers. This observation is described formally in the following definitions.

Definition 6.4.7 Let $((C_0, C_1, C_2, \ldots), (f_1, f_2, \ldots))$ be a hierarchical prefounding for τ_0, with associated partition $R_0, \ldots, R_N$ of R_{τ_0}, and let (G, i) and (G', j) be in $\tau_0 \setminus C_0$. We say that (G', j) *covers* (G, i) if the following two conditions are satisfied:

1. $(G', j) >_{pr} (G, i)$
2. $\exists k > 0 : R(G', j), R(G, i) \in R_k$

Note that if (G', j) covers (G, i), then $C_i \subseteq C_j$. Moreover, for each $(G, i) \in \tau_0 \setminus C_0$:
$C_i = \{(G, i)\} \cup \{(G', j) | (G', j) \text{ covers } (G, i) \text{ or } (G, i) \text{ covers } (G', j)\}$
In other words, the C_i-sets are classes of covering goal nodes.

Definition 6.4.8 Let $((C_0, C_1, C_2, \ldots), (f_1, f_2, \ldots))$ be a hierarchical prefounding for τ_0, with associated partition $R_0, \ldots, R_N$ of R_{τ_0}. We define a function *layer* : $\tau_0 \to \{0, 1, \ldots, N\}$ as follows:

```
If $(G, i) \in C_0$
  **Then** $layer(G, i) = 0$
**Else if** $(G', j)$ covers $(G, i)$
  **Then** $layer(G, i) = layer(G', j)$
**Else** $layer(G, i) = max(\{0\} \cup \{layer(G', j) | (G', j) >_{pr} (G, i)\}) + 1$
```

Notice that the range of the *layer* function is indeed $\{0, 1, \ldots, N\}$. There are only $N + 1$ sets R_k and the *layer* value does not increase between two nodes (G', j) and (G, i), such that (G', j) covers (G, i). Moreover, on a given proper achain, *layer* induces a one-to-one correspondence between numbers k such that $R(G, i)$ is in R_k and numbers n such that $layer(G, i) = n$.

It is due to proposition 6.4.3 and the existence of this finite number of hierarchical layers that a procedure for constructing finite SLD$^-$-trees can be formulated on the basis of definition 6.4.5. The underlying idea is that if the sets C_i corresponding to nodes of each layer k are well-founded (through f_i), then (using the hierarchical layer structure) we can prove that only a finite number of mutually non-covered sets of each layer k exist. As a consequence (since "covered" implies "contained in"), we can find a finite number of sets that contain the entire SLD$^-$-tree. So, the tree has become subset-wise founded.

With the exception of the tree construction procedure itself, this reasoning is formalised in the following basic theorem and its proof, serving as main foundation for most material on finite unfolding throughout the ensuing sections and chapters of this thesis.

Theorem 6.4.9 Let $((C_0, C_1, C_2, \ldots), (f_1, f_2, \ldots))$ be a hierarchical prefounding for τ_0 and suppose that τ is an SLD$^-$-subtree of τ_0. If every f_i is a well-founded measure on $C_i \cap \tau, >_\tau$, then there exist a finite number of sets $C_{i_1}, \ldots, C_{i_M}$ among $C_1, C_2, \ldots$, such that τ is subset-wise founded with respect to the pair $((C_0 \cap \tau, C_{i_1} \cap \tau, \ldots, C_{i_M} \cap \tau), (f_{i_1}, \ldots, f_{i_M}))$.

Proof

- We show directly that τ is finite. Assume that it is not and that D is an infinite derivation in τ. Then it follows from proposition 6.4.4 that D contains an infinite proper achain $(G_{i_1}, i_1) >_{pr} (G_{i_2}, i_2) >_{pr} \ldots$. By condition 3 of definition 6.4.5, proper achains in C_0 are of finite length. We may therefore assume that $(G_{i_1}, i_1), (G_{i_2}, i_2), \ldots$ are in $\tau_0 \setminus C_0$. Since there only exist a finite number of sets R_k, a finite number of members of this proper achain, $(G_{j_1}, j_1), \ldots, (G_{j_r}, j_r)$, are maximal elements under the covered-s-order. All other members (G_{i_l}, i_l) are covered by some $(G_{j_s}, j_s), 1 \leq s \leq r$. For each such (G_{j_s}, j_s), the associated set $C_{j_s} \cap \tau$ is well-founded through f_{j_s} by assumption. Furthermore, nodes covered by (G_{j_s}, j_s) are in C_{j_s}. But this means that $(G_{i_1}, i_1) >_{pr} (G_{i_2}, i_2) >_{pr} \ldots$ is finite.

- Since τ is finite, it is subset-wise founded (theorem 6.2.8). It remains to be shown how a finite number of sets $C_{i_1}, \ldots, C_{i_M}$ can be selected from $C_1, C_2, \ldots$, such that τ is subset-wise founded with respect to $((C_0 \cap \tau, C_{i_1} \cap \tau, \ldots, C_{i_M} \cap \tau), (f_{i_1}, \ldots, f_{i_M}))$. Define for any $n, 1 \leq n \leq N$:

  $$\begin{aligned} S_n = \{(G,i) \in \tau \mid & \, layer(G,i) = n \text{ and there exists a } k, 1 \leq k \leq N, \\ & \text{such that } R(G,i) \in R_k \text{ and there does not exist a node} \\ & (G', j) \in \tau, \text{ such that } (G', j) \text{ covers } (G,i)\} \end{aligned}$$

  Now, let $(C_{i_1}, \ldots, C_{i_M})$ be the sequence of C_i-sets in the given hierarchical prefounding such that their corresponding nodes (G_{i_l}, i_l) are in some S_k. Let $(f_{i_1}, \ldots, f_{i_M})$ be the associated sequence of functions $f_{i_l} : C_{i_l} \rightarrow W_{i_l}, >_{i_l}$. Then τ is subset-wise founded with respect to $((C_0 \cap \tau, C_{i_1} \cap \tau, \ldots, C_{i_M} \cap \tau), (f_{i_1}, \ldots, f_{i_M}))$:

  1. Clearly, each of the sets $C_q \cap \tau$ satisfies $C_q \cap \tau \subseteq G_\tau$. To prove that $G_\tau \subseteq \bigcup_{q \in \{0, i_1, \ldots, i_M\}} C_q \cap \tau$, observe that each node $(G,i) \in$

τ which is not in $\bigcup_{k\leq N} S_k$ is either in C_0 or is covered by a node $(G', j) \in \bigcup_{k\leq N} S_k$. If the latter is the case, then $C_i \subseteq C_j$. So, $G_\tau \subseteq \bigcup_{q\in\{0,i_1,\ldots,i_M\}} C_q \cap \tau$ and condition 1 in definition 6.2.6 is fulfilled.

2. Each $C_q \cap \tau, q \neq 0$, has the well-founded measure f_q by assumption, satisfying condition 2 in definition 6.2.6.
3. Finally, condition 3 in definition 6.2.6 on C_0 directly follows from condition 2 in definition 6.4.5 and proposition 6.4.4.

□

6.4.3 A more sophisticated algorithm

Obviously, the pair of potentially infinite sequences $((C_0, C_1, \ldots), (f_1, \ldots))$ can hardly be considered a practical specification of a prefounding. In particular, it can not be used as input to an algorithm computing finite SLD-trees. Instead, we will *specify hierarchical prefoundings* through:

- a *finite* partition $R_0, \ldots, R_N$ of R_{τ_0}
- together with functions $F_1, \ldots, F_N$ such that
  $F_k : \{(G, i) \in \tau_0 | R(G, i) \in R_k\} \rightarrow W_k, >_k$
  where $W_k, >_k, 1 \leq k \leq N$ are well-founded s-posets.

Observe that due to condition 2 of definition 6.4.5, such functions $F_k, 1 \leq k \leq N$ can be associated with any hierarchical prefounding $((C_0, C_1, \ldots), (f_1, \ldots))$. More precisely, for any (G, i) with $R(G, i) \in R_k$, $F_k(G, i) = f_i(G, i)$. Conversely, the pair $((R_0, R_1, \ldots, R_N), (F_1, \ldots, F_N))$ allows the full reconstruction of the hierarchical prefounding, where, for each (G, i) with $R(G, i) \in R_k$, $f_i = F_k|_{C_i}$.

So, let us now proceed to show how the concept of a hierarchical prefounding can be used in a constructive way to produce finite SLD-trees. Before we can actually formulate an algorithm that improves on the unfolding behaviour of algorithm 6.3.5, we need one more definition, singling out a particularly interesting covering ancestor of a goal.

Definition 6.4.10 Let (G, i) and (G', j) be two distinct nodes in $\tau_0 \setminus C_0$. (G', j) is called the *direct covering ancestor* of (G, i) if:

1. (G', j) covers (G, i) and
2. any other (G'', k) that covers (G, i) also covers (G', j)

It follows that the direct covering ancestor of a node, if it exists, is unique.

In algorithm 6.4.11, we assume that the proof tree is constructed simultaneously with the SLD$^-$-tree. Without making this construction explicit, we make use of the $>_{pr}$-relation, derivable from the (partially generated) proof tree. The basic structure of the algorithm is quite similar to the one of algorithm 6.3.5, except that its operation is now controlled through a hierarchical prefounding $((R_0, R_1, \ldots, R_N), (F_1, \ldots, F_N))$, instead of a finite prefounding.

Algorithm 6.4.11

```
Initialisation
  $\tau := \{\{(\leftarrow A,1)\}\}$ {* an SLD-tree with a single one-node derivation *}
  $Pr := \emptyset$ {* in $Pr$, the $>_{pr}$-relation will be constructed *}
  $Terminated := \emptyset$
While there exists a derivation $D \in \tau$ such that $D \notin Terminated$ do
  Let $(G, i)$ be the leaf of $D$
  Let $Derive(G, i)$ be the set of all its immediate $>_{\tau_0}$-descendants
  Let $Decrease(G, i)$ be the set of all $(G', j) \in Derive(G, i)$, such that
    If $(G'', k)$ is the direct covering ancestor of $(G', j)$
    and $R(G', j), R(G'', k) \in R_n (1 \leq n \leq N)$
      Then $F_n(G'', k) > F_n(G', j)$
  If $Decrease(G, i) = \emptyset$
    Then add $D$ to $Terminated$
  Else {* $\tau$ is further extended *}
    Replace $\tau$ by $\tau \setminus \{D\} \cup \{D \cup \{(G', j)\} | (G', j) \in Decrease(G, i)\}$
    Extend the $Pr$-relation accordingly
Endwhile
```

Of course, the resulting SLD$^-$-tree τ again must be completed into τ^+, its minimal containing SLD-tree.

We have the following theorem:

Theorem 6.4.12 Algorithm 6.4.11 terminates. There exist a finite number of sets $C_0, \ldots, C_M$ and functions $f_1, \ldots, f_M$, such that the resulting finite SLD$^-$-tree τ is subset-wise founded with respect to $C_0 \cap \tau, \ldots, C_M \cap \tau$ and the well-founded measures $f_1, \ldots, f_M$ restricted to these sets.

Proof Termination follows in a way completely similar to the finiteness part in the proof of theorem 6.4.9. The subset-wise foundedness of τ is an immediate consequence of theorem 6.4.9. □

Finally, let us illustrate the operation of algorithm 6.4.11 on an example.

Example 6.4.13 We consider a program for transposing a matrix, represented as a list of lists, the latter constituting its rows.

$transpose(X, []) \leftarrow nullrows(X)$
$transpose(X, [Y|Ys]) \leftarrow makerow(X, Y, Z), transpose(Z, Ys)$

$makerow([], [], []) \leftarrow$
$makerow([[X|Xs]|U], [X|V], [Xs|W]) \leftarrow makerow(U, V, W)$
$nullrows([]) \leftarrow$
$nullrows([[]|X]) \leftarrow nullrows(X)$

We wish to construct an SLD-tree for the following query:

$\leftarrow transpose([[a, b]|Xs], Y)$

using algorithm 6.4.11. For the required hierarchical prefounding, we take an instance of the generic one proposed in example 6.4.6. Since all three predicates in the above program are recursive, this gives rise to four R_i-sets. The corresponding functions $F_i, i > 0$ are the functor measures associated with the respective selected literals and their sets of input argument positions. If we abbreviate predicate names, just using their initial character, and label R-sets and $|.|$-functions accordingly, we obtain:

$((R_0, R_t, R_m, R_n), (|.|_{t,\{1\}}, |.|_{m,\{1\}}, |.|_{n,\{1\}}))$

Finally, for the computation rule R, we assume a rule that, from each goal, selects an atom with an associated functor measure value smaller than the functor measure weight of the selected atom in the direct covering ancestor (if the latter exists). If there is a choice of several such atoms, the leftmost with weight larger than 0 is selected, or, if they are all mapped to 0, simply the leftmost. Figure 6.3 depicts the resulting SLD$^-$-tree, which happens to coincide with its minimal containing SLD-tree. As usual, selected literals are underlined. Moreover, each non-trivial goal node is labeled with a triplet, containing, first, its number, second, the number of its direct covering ancestor (if the latter does not exist, the mark "_" has been used), and, third, its weight under the relevant measure function. Notice the finely tuned ancestor selection for nodes with a selected literal containing *makerow*. Obviously, this kind of refined focusing on relevant ancestor goals indeed solves the problem with the *sums* program in example 6.4.1.

From the tree, the following specialised *transpose* clauses can be synthesised (tidying up variable names):

$transpose1([[a, b]], [[a], [b]]) \leftarrow$
$transpose1([[a, b], [X, X'|Xs]|U], [[a, X|Y], [b, X'|Z]]) \leftarrow$
$\qquad makerow(U, Y, V), makerow(V, Z, W), nullrows([Xs|W])$

Partial deduction of the *transpose* program was also studied in e.g. [64] and [13]. The amount of specialisation obtained in the clauses above is slightly less than in the clauses produced by the method in [13], using the R_v computation rule (see also section 5.5), which, in this case, allows some additional useful unfoldings. However, as shown in example 5.5.4, in general, R_v does not guarantee the construction of a finite SLD-tree. More comments on the practical effects of various computation rules can be found in section 7.5 below.

Finally, it is interesting to note that algorithm 6.3.5, using the obvious instance of the generic finite prefounding introduced in example 6.3.4, and assuming the same τ_0 as above, would stop unfolding at node 10, thus deriving a single specialised *transpose* clause:

$$transpose2([[a, b]|X], [[a|Y], [b|Z]]) \leftarrow \\ makerow(X, Y, V), makerow(V, Z, W), nullrows(W)$$

This is precisely the result obtained in [64].

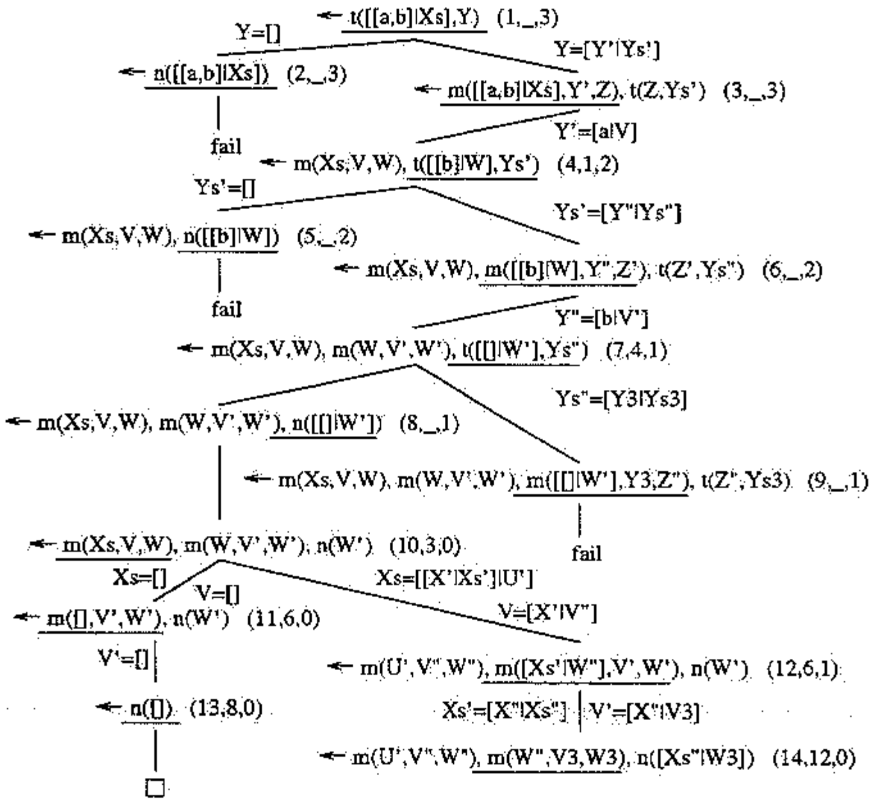

← t([[a,b]|Xs],Y) (1,_,3)
Y=[]
Y=[Y'|Ys']
← n([[a,b]|Xs]) (2,_,3)
fail
← m([[a,b]|Xs],Y',Z), t(Z,Ys') (3,_,3)
Y'=[a|V]
← m(Xs,V,W), t([[b]|W],Ys') (4,1,2)
Ys'=[]
Ys'=[Y"|Ys"]
← m(Xs,V,W), n([[b]|W]) (5,_,2)
fail
← m(Xs,V,W), m([[b]|W],Y",Z'), t(Z',Ys") (6,_,2)
Y"=[b|V']
← m(Xs,V,W), m(W,V',W'), t([[]|W'],Ys") (7,4,1)
Ys"=[Y3|Ys3]
← m(Xs,V,W), m(W,V',W'), n([[]|W']) (8,_,1)
← m(Xs,V,W), m(W,V',W'), m([[]|W'],Y3,Z"), t(Z",Ys3) (9,_,1)
fail
← m(Xs,V,W), m(W,V',W'), n(W') (10,3,0)
Xs=[]
V=[]
Xs=[[X'|Xs']|U']
V=[X'|V"]
← m([],V',W'), n(W') (11,6,0)
V'=[]
← m(U',V",W"), m([Xs'|W"],V',W'), n(W') (12,6,1)
← n([]) (13,8,0)
Xs'=[X"|Xs"]
V'=[X"|V3]
← m(U',V",W"), m(W",V3,W3), n([Xs"|W3]) (14,12,0)
□

Figure 6.3: The resulting tree for example 6.4.13.

6.5 Applicability and Automation

In this section, we first show how our technique can be regarded as a general framework for controlling the unfolding in partial deduction. We illustrate how existing methods can be simulated and briefly discuss the relationship with work on static termination analysis. Next, we address full automation. We point out which issues are still to be settled, and propose a slightly modified version of algorithm 6.4.11 to be used as the actual basis for the development of fully automatic algorithms in subsequent chapters. Finally, we return to the specific issue of unfolding meta-interpreters.

6.5.1 Related techniques

Typical heuristics for finite unfolding as proposed in the literature aim at loop detection and decide not to unfold a goal when a loop is suspected. The four criteria mentioned in [14] do not unfold a literal L when there is, earlier in the same derivation, a node with selected literal L' and respectively, L is a variant of, is an instance of, is more general than, or unifies (has a common instance) with L'. We already mentioned that none of these criteria excludes all loops, as can easily be verified in figure 5.1. In general, problems occur with predicates containing arguments which tend to grow in successive unfolding steps. Such heuristics therefore must be enhanced with the use of a depth bound on generated derivations.

Although the resulting combinations seem quite unsatisfactory (the $>_{pr}$ relation is not taken into account when comparing two nodes and depth bounds are very ad hoc), they can easily be simulated in our framework. Consider e.g. the "variant" rule, R_v. Use of a finite prefounding with a single class is sufficient. The weight to be associated with a goal which is the ith step in a derivation is N if the selected literal is a variant of a previously selected one (i.e. unfolding that literal yields a not well-founded measure) and is $max(0, N-i)$ otherwise (i.e. a derivation is limited to N steps). Alternatively, a slightly more natural simulation can be done with a finite prefounding having one class for each recursive predicate and one for the other predicates. Observe, however, that a direct application of our approach ensures termination in a natural way, without simulating any depth bound.

Several pragmatic loop prevention tests are proposed in [152]. (See also the discussion of [15] below.) They range from a simple restriction on the number of times a literal with the same predicate symbol may be selected in a derivation, to more sophisticated approaches, featuring comparisons of successive calls to the same predicate through the use of some measure function, related to the functor measure of definition 6.3.3. However, the $>_{pr}$ relation is not taken into account when deciding which goals are to be compared.

Bol, Apt and Klop present a systematic and formal account of (run-time) loop checks for logic programs in [16]. They discern between sound and complete checks. The former ensure that no solutions are removed and have mainly been focused on in work on run-time termination analysis; The latter guarantee termination, possibly at the cost of deleting some solutions. In [15], Bol argues that in the context of unfolding for partial deduction, sound loop checks can be helpful to improve the performance of partially deduced programs, but complete loop checks are essential to terminate unfolding. (Since complete checks sometimes remove non-loops as well, this approach is also referred to as *loop prevention* rather than *loop detection*.) Next, a detailed formalisation of the various loop checking strategies employed by Mixtus ([152]) is included. Finally, [26] is discussed and its content briefly compared with the formalised [152] methods. It is stated that its basic framework might be of theoretical interest as a standard for complete loop checks, and natural instances should be easily implementable. We hope chapters 7 and 8 will convince the reader that both conjectures are in fact true.

Somewhat more distantly related is the considerable amount of work on *compile-time* analysis techniques for the detection of (non-)terminating programs and goals. (See e.g. [169], [9], [6], [133], [43]. A recent, extensive survey of the work in this field can be found in [40].) These techniques also use well-founded measures (often called "level mappings" in this context) to prove termination. However, important differences are that compile time termination analysis:

- provides proofs of finiteness *for complete SLD-trees* τ_0 for the given program and goal and is not concerned with characterisation of finiteness for *incomplete* SLD-trees.
- only relies on the *source code* of the program, or at the most, a *finite, abstract* representation of its evaluation (e.g. through abstract interpretation, [38]) to prove termination. We, on the other hand consider *concrete* computation steps.

These differences, in general, represent the typical distinction between compile-time termination analysis and (run-time) loop checking or loop prevention.

Still, static termination analysis seems useful to further optimise our approach. Consider a goal

$\leftarrow p([X, Y, Z])$

and assume that its direct covering ancestor is

$\leftarrow p([X, Y, Z|Xs])$

Then the functor measure maps both goals to the same weight (3). Algorithm 6.4.11 will therefore not include the former goal in the generated SLD-tree τ. Now, assume that in a preprocessing phase, compile time termination analysis has shown that any p query terminates when its single argument is bound to a list of fixed length. Then we can safely add the former goal, and all its descendants, to τ. Such information can perhaps be incorporated in a prefounding by placing such safe goals (and their descendants) in C_0.

Further challenging study material is provided by techniques related to partial deduction, such as the program transformation methodology described in [135] and the work on compiling control ([25], [39], [42]). A detailed comparison of the termination criteria employed by these and various other techniques with the approach presented in this thesis is non-trivial, but might reveal interesting connections.

6.5.2 Setting the stage for automation

We have shown how our approach provides a good framework for finitely unfolding logic programs. Algorithm 6.4.11 in particular incorporates a refined treatment of recursion. However, as it stands, it can not figure as part of a fully automatic partial deduction system for logic programs. Indeed, it presupposes as given a particular hierarchical prefounding (represented by a pair $((R_0, R_1, \ldots, R_N), (F_1, \ldots, F_N)))$ and a computation rule R, together determining one single complete (but possibly infinite) SLD-tree τ_0 for the considered program and query. However, in a fully automatic context, only the query and the program are provided as inputs to the system, which itself should be capable of finding optimal values for $R_0, R_1, \ldots, R_N$, $F_1, \ldots, F_N$ and R.

Various fully automatic algorithms performing these tasks with different degrees of generality and sophistication will be proposed in chapter 7 and, particularly, in chapter 8. In each case, the selected literal in a goal G will be one of the key factors in deciding which weight should be assigned to G. It therefore turns out that a somewhat more elegant treatment of such automatic methods is obtained on the basis of a slightly specialised version of the framework presented above. Indeed, when an incomplete SLD-tree τ is not considered as a subtree of some fixed, complete τ_0, it seems very sensible to regard its non-trivial leaves as goals *without a selected literal*. Not being assigned any weight, such goals can then be included in the SLD-tree without endangering subset-wise foundedness. In this way, the resulting algorithm immediately produces finite, subset-wise founded SLD-trees.

In this subsection, we provide a formal basis for this minor shift of perspective and include a variant of algorithm 6.4.11, to be used as the template for later completely automatic algorithms.

We set out with a useful definition.

Definition 6.5.1 A leaf node in an SLD-tree which is neither a success nor a failure node, but an arbitrary goal statement without selected literal, will be called a *dangling* leaf.

Now, we can *associate one particular SLD$^-$-tree to a given SLD-tree.*

Definition 6.5.2 Suppose τ is an SLD-tree. Then we call the tree τ^-, obtained by deleting from τ its dangling leaves (if any), the *SLD$^-$-tree associated to τ*.

Of course, for any SLD-tree τ, $(\tau^-)^+ = \tau$.

We have the following results:

Proposition 6.5.3 Let τ be an SLD-tree and τ^- its associated SLD$^-$-tree. If τ^- is subset-wise founded with respect to $((C_0, C_1, \ldots, C_N), (f_1, \ldots, f_N))$ than τ is subset-wise founded with respect to $((C'_0, C_1, \ldots, C_N), (f_1, \ldots, f_N))$, where $C'_0 = C_0 \cup \{(G,i) \in \tau | (G,i) \text{ is a dangling leaf of } \tau\}$.

Proof All three conditions of definition 6.2.6 are immediately verifiable. □

Corollary 6.5.4 Let τ be an SLD-subtree of the complete SLD-tree τ_0 and suppose that $((C_0, C_1, C_2, \ldots), (f_1, f_2, \ldots))$ is a hierarchical prefounding for τ_0. If every f_i is a well-founded measure on $C_i \cap \tau^-, >_{\tau^-}$, then there exist a finite number of sets $C_{i_1}, \ldots, C_{i_M}$ among $C_1, C_2, \ldots$ such that τ is subset-wise founded with respect to $((C'_0, C_{i_1} \cap \tau^-, \ldots, C_{i_M} \cap \tau^-), (f_{i_1}, \ldots, f_{i_M}))$ where $C'_0 = (C_0 \cap \tau^-) \cup \{(G,i) \in \tau | (G,i) \text{ is a dangling leaf of } \tau\}$.

Proof The corollary is an immediate consequence of theorem 6.4.9 and proposition 6.5.3. □

In view of this result and our earlier observation that any hierarchical prefounding can be uniquely identified through its pair $((R_0, R_1, \ldots, R_N), (F_1, \ldots, F_N))$, we will, in subsequent chapters, take the liberty to abuse our terminology as follows: We will occasionally call an SLD-tree τ *subset-wise founded with respect to a pair* $((R_0, R_1, \ldots, R_N), (F_1, \ldots, F_N))$.

We can now modify algorithm 6.4.11. Note that its resulting variant still expects a pair $((R_0, R_1, \ldots, R_N), (F_1, \ldots, F_N))$ and a computation rule R to be fixed in advance.

Algorithm 6.5.5

```
Initialisation
  $\tau := \{\{(\leftarrow A, 1)\}\}$ {* an SLD-tree with a single one-node derivation *}
  $Pr := \emptyset$ {* in $Pr$, the $>_{pr}$-relation will be constructed *}
  $Terminated := \emptyset$
While there exists a derivation $D \in \tau$ such that $D \notin Terminated$ do
  Let $(G,i)$ be the leaf of $D$
  Let $Derive(G,i)$ be the set of all its immediate $>_{\tau_0}$-descendants
  If $Derive(G,i) = \emptyset$ {* $(G,i)$ is a success or a failure node *}
```

```
    Then add $D$ to $Terminated$
  Else if there is a direct covering ancestor $(G', j)$ of $(G, i)$
        with $R(G', j), R(G, i) \in R_n$
        such that not$(F_n(G', j) >_n F_n(G, i))$
    Then add $D$ to $Terminated$ {* $(G, i)$ becomes a dangling leaf *}
  Else {* $\tau$ is further extended *}
    Replace $\tau$ by $\tau \setminus \{D\} \cup \{D \cup \{(G'', k)\} | (G'', k) \in Derive(G, i)\}$
    Extend the $Pr$-relation accordingly
Endwhile
```

The difference between this algorithm and algorithm 6.4.11 can be characterised as follows. A node in some $Derive(G, i)$, but not in the corresponding $Decrease(G, i)$, is rejected by algorithm 6.4.11, while it is classified as a dangling leaf by algorithm 6.5.5 and included in τ. In this way, other things equal, algorithm 6.5.5 sometimes (though often not) generates a larger SLD-tree than algorithm 6.4.11 does (after completion of its result into an SLD-tree). Differences only appear at (non-trivial) nodes (G, i) such that $Decrease(G, i) = \emptyset$. Such nodes are unfolded one more level by algorithm 6.5.5, all their descendants being dangling leaves. Reconsider some earlier examples.

Example 6.5.6 The final result for the *contains_pal* program in example 6.2.7 is the same in both cases: The extra leaf directly included by algorithm 6.5.5 is also added after execution of algorithm 6.4.11 to complete the SLD$^-$-tree depicted in figure 6.1 into its minimal containing SLD-tree, as indicated in example 6.3.6.

Example 6.5.7 Algorithm 6.5.5 does produce a larger tree for the *transpose* program and query in example 6.4.13. To the SLD-tree in figure 6.3, two dangling leaves are added as descendants of node 14, as indicated in figure 6.4. Note that the *Decrease* set of node 14 is indeed empty.

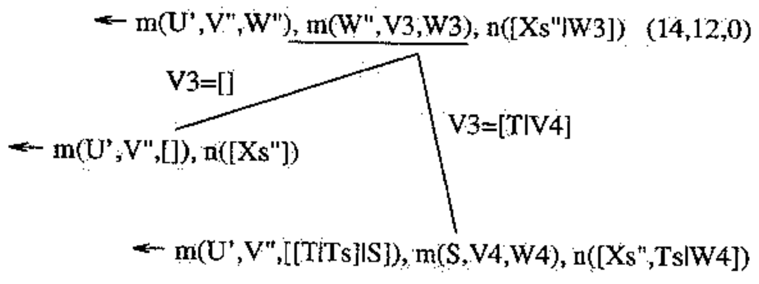

← m(U',V",W"), m(W",V3,W3), n([Xs"|W3]) (14,12,0)
V3=[]
V3=[T|V4]
← m(U',V",[]), n([Xs"])
← m(U',V",[[T|Ts]|S]), m(S,V4,W4), n([Xs",Ts|W4])

Figure 6.4: Two extra dangling leaves.

The reason why algorithm 6.5.5 constitutes a suitable basis for complete automation can now be suspected: An automatic unfolding algorithm will simply terminate an (incomplete) derivation when no literal can be selected in the derivation's leaf. This way of proceeding nicely matches the structure of algorithm 6.5.5.

Finally, the following theorem comes as no surprise:

Theorem 6.5.8 Algorithm 6.5.5 terminates. Let $((C_0, C_1, C_2, \ldots), (f_1, f_2, \ldots))$ be the hierarchical prefounding of the complete SLD-tree τ_0, determined by the pair $((R_0, R_1, \ldots, R_N), (F_1, \ldots, F_N))$ and the computation rule R, underlying an application of algorithm 6.5.5. Let τ be the finite SLD-tree built by algorithm 6.5.5. Then there exist a finite number of sets $C_{i_1}, \ldots, C_{i_M}$ among $C_1, C_2, \ldots$ such that τ is subset-wise founded with respect to $C'_0, C_{i_1} \cap \tau^-, \ldots, C_{i_M} \cap \tau^-$, where $C'_0 = (C_0 \cap \tau^-) \cup \{(G, i) \in \tau | (G, i) \text{ is a dangling leaf of } \tau\}$, and the well-founded measures $f_{i_1}, \ldots, f_{i_M}$ restricted to these sets.

Proof Obviously, termination can again be demonstrated in a way similar to the finiteness part in the proof of theorem 6.4.9. The subset-wise foundedness result follows from corollary 6.5.4 □

6.5.3 Meta-interpreters revisited

To conclude this section, we return to the particular issues involved in handling meta-interpreters.

First, there is the problem of identifying a program as a meta-interpreter. Most meta-interpreters written in Prolog use the built-in predicate *clause* to access a representation of the object program. (This practice is simulated through an explicitly defined *clause* predicate in part I of this thesis.) These are easy to recognise. However, as already discussed in subsection 4.7.2, the object code can also be denoted by meta level terms. In the logic programming language Gödel ([80]), this is standard practice. There, such meta-programs can automatically be told apart from object level term handling programs through the required argument type declarations. In Prolog, however, this is not the case, and therefore, this type of meta-interpreters can not easily be spotted. We return to this issue below.

When a program has been identified as a meta-interpreter, it should receive a special treatment as was illustrated in subsection 6.3.3. The control of the unfolding should be based on the properties of the object level goals, denoted by the meta level arguments. In addition, unfoldings performing only parsing should always be allowed. It seems possible to generalise the approach described in subsection 6.3.3 to this end. Consider for example the following meta-interpreter. It is slightly more complex than the one treated in subsection 6.3.3.

$solve(empty) \leftarrow$
$solve(X\&Xs) \leftarrow clause(X,Y), concat(Y,Xs,Ys), solve(Ys)$

Here, we want to focus attention on the arguments of the first conjunct in $solve(A\&B)$ calls. Assuming that $p_1,\ldots,p_N$ are the recursive predicates of the object program, we define:

$R_k = \{R(G,i) \in R_{\tau_0} | R(G,i)$ is an atom of type $solve(p_k(t_1,\ldots,t_{n_k})\&B)\}$, where $1 \leq k \leq N$

$R_0 = \{R(G,i) \in R_{\tau_0} | R(G,i)$ is an atom of type $solve(p(t_1,\ldots,t_n)\&B)$, where p is a non-recursive predicate$\}$
$\cup\{R(G,i) \in R_{\tau_0} | R(G,i)$ is an atom of type $solve(empty)\}$
$\cup\{R(G,i) \in R_{\tau_0} | R(G,i)$ is an atom of type $clause(A,B)\}$
$\cup\{R(G,i) \in R_{\tau_0} | R(G,i)$ is an atom of type $concat(A,Bs,As)\}$

and, for $1 \leq k \leq N$:

$F_k = |\cdot|_{solve,S_k}$

where S_k contains elements of the form $1/1/n, 1 \leq n \leq n_k$.

Note, however, that such an approach is only correct if the parsing is correct, in the sense that the meta-interpreter terminates when executed with a completely instantiated goal. Moreover the goal with respect to which partial deduction is performed should not contain meta level variables.

Next, we return to the issue of dealing with Prolog meta-interpreters that represent object theories through meta level terms. Here, we would need techniques enabling us to prove at compile time that certain arguments of predicates take their values in a set of terms that can be partitioned in a finite number of classes. Each of these classes should be a set of terms starting with the same functor (denoting an object level predicate). In this way, control on the level of the object program would be feasible. Type inference through abstract interpretation seems promising to provide such information. More generally, even outside a strict meta-programming context, this could prove a useful refinement of the techniques presented in this thesis. Moreover, similar techniques may help to find interesting S_k-sets for meta-interpreters of any kind.

Finally, in a context of full automation, section 8.6 presents a different point of view, not assuming any a priori difference between meta-interpreters and "ordinary" programs. We refer to subsection 8.6.4 for some further comments pertaining to the issues raised in the previous two paragraphs.

6.6 Discussion and Conclusion

In this chapter, we have elaborated a general framework for the control of unfolding during partial deduction of definite logic programs. We have adapted the wellknown concept of a well-founded set to obtain a workable characterisation of finite SLD-trees. We have proposed two different operational derivatives of this characterisation, and shown how they can be used to construct sensible finite SLD-trees. This led to the formulation of two algorithms, the second of which provides a more refined treatment of recursion than the first. Thanks to their roots in the developed framework, both algorithms could be formally shown to terminate on all programs and queries. Next, we briefly discussed some related work and reflected on particularities connected to the treatment of meta-interpreters. Finally, we proposed a generic algorithm to be used as the template for complete automation in subsequent chapters.

We already mentioned that the use of well-founded orderings to prove termination properties is by no means a novel idea. Apart from the applications referred to in subsection 6.2.1, their extensive role in the work on termination of term rewriting systems ([48]) should not go unnoticed. Also within logic programming, many approaches to partial deduction have, implicitly or explicitly, used the notion to impose some control on unfolding. The work on termination analysis of logic programs can also be mentioned here. Particularly [133] explicitly defines a well-founded ordering to be used for supervising unfolding. Finally, similar problems arise with partial evaluation in a functional framework, where one has to decide on unfolding of function calls. We refer to chapter 14 of [85] for some references and a discussion on offline termination analysis in that context.

Our main contributions lie in tailoring the concept to the treatment of logic programs, establishing an elaborate formal framework based on it, and demonstrating the latter's use for the construction of unfolding algorithms with a fine-grained treatment of recursion. Moreover, subsequent chapters will present various concrete instances of our framework, analyse their respective merits, and provide strategies to, within a given category, automatically focus on well-founded measures that are optimal for a given problem. Finally, section 8.5 will reconsider the basic framework and slightly extend it to further increase its power.

We have concentrated on pure definite logic programs, even excluding negation. In [14], it is proposed that during unfolding, negated calls should be executed when ground and remain in the resultant when non-ground. This of course jeopardises termination, since termination of "ordinary" ground query execution is not guaranteed in general. One solution consists in restricting attention to specific subclasses of programs, known to terminate on ground queries (the work on compile-time termination analysis, cited above, is relevant here), but other, more ambitious, approaches seem feasible. We briefly return to this issue, in an overall partial deduction context, in section 7.4.

Next, it has been observed that ensuring *termination* of unfolding does not suffice to provide a good unfolding strategy in the context of partial deduction. Indeed, unlimited unfolding brings along the danger of *code explosion*, possibly resulting in a "specialised" program in fact *less* efficient than the original general program[1]. In other words, good unfolding does not only deliver SLD-trees of finite ***depth***, but also keeps their ***width*** within reasonable bounds. While we have not explicitly addressed the latter issue in our work, a few remarks on this topic can be included:

- In principle, the two issues can be considered independently. When finiteness of the trees is guaranteed, extra measures can be added to ensure reasonable width.
- This being said, it can be observed that "data consumption based" unfolding as carried out by methods based on our framework, goes already a long way towards the latter goal. Since such algorithms tend to only unfold "sufficiently instantiated" atoms, often branching is very low.
- When thoroughly indeterministic programs nevertheless cause wide branching, sophisticated indexing techniques, as available in most Prolog compilers, can often efficiently cope with the resulting large number of highly specialised clauses. Again, this specialisation is caused by the demand that unfolding should only proceed as long as data are consumed.

Summarising, we conclude that it makes sense to pursue strategies for maximal sensible finite unfolding. Further research must settle the question to what extent additional measures to control the width of resulting SLD-trees are necessary. Such measures can be incorporated in developed algorithms, or, perhaps, run in a postprocessing phase on generated SLD-trees. We return to this topic in section 7.5, when we discuss the results of some experiments.

Gallagher mentions a related subtle issue in [65]. Assuming a left-to-right computation rule for logic programs, he points out that unfolding a choice point to the right of another goal can cause duplicated computation and an associated loss of efficiency in the specialised program produced by partial deduction. Again, this issue, or its generalisation to a context without pre-determined computation rule, will not be considered in the rest of this thesis. Methods to handle it can be developed independently. And the overall resulting unfolding strategy will still profit from the presence of a basic methodology that guarantees termination.

[1]See [126] for a treatment of related issues in explanation-based learning.

Chapter 7

Sound and Complete Partial Deduction

7.1 Introduction

In this chapter, we return to the issue raised in section 5.4: We develop an algorithm for automatic partial deduction of definite logic programs and goals, show that its result satisfies the conditions of theorem 5.4.7, and prove that it always terminates. To this end, in section 7.2, we first derive a fully automatic algorithm for finite unfolding, based on the framework laid out in the previous chapter. Next, we use it as a stepping stone for the development of a partial deduction algorithm in section 7.3. We demonstrate the latter's operation on some simple examples and give formal proofs for a number of interesting properties, including certified termination. We discuss some less formal aspects and briefly comment on related work in section 7.4. Next, section 7.5 addresses experiments performed with an implementation of the developed method as well as some related techniques, proposed in the literature for the control of unfolding or of the overall partial deduction. On the basis of the obtained results, we venture some preliminary comparative comments. Finally, the chapter closes with a short overall conclusion in section 7.6.

7.2 Automatic Finite Unfolding

Our first task then, is deriving a fully automatic instance of algorithm 6.5.5. To this end, we have to come up with concrete choices for the the computation rule R and the pair $((R_0, R_1, \ldots, R_N), (F_1, \ldots, F_N))$.

Now, for the $R_0, R_1, \ldots, R_N$ partition, it seems appropriate to proceed as in example 6.4.6. In other words, we associate one R_i with every recursive predicate p_i, and place all calls to a non-recursive predicate in R_0, thus ensuring that they will always be unfolded. Given the (elementary) analysis capacity to automatically identify recursive predicates in a given program, this choice for $R_0, R_1, \ldots, R_N$ is trivial to automate. Moreover, it ensures that condition 3 of definition 6.4.5 is satisfied. For the measure functions $F_1, \ldots, F_N$, functor measures as introduced in definition 6.3.3 can be employed. (To be precise, a goal is mapped to the weight of its selected literal under the relevant functor measure. Here and elsewhere in similar circumstances, we will overload function symbols, and apply to goals functions defined on atoms.) However, this requires additional decisions on the composition of the argument sets S_{p_i} to be used in the $F_i = |.|_{p_i, S_{p_i}}$ functions.

Finding appropriate sets of argument positions and a suitable computation rule R can be regarded as a search problem. Initially, in algorithm 7.2.1 below, for each S_p, the entire set of argument positions of its associated predicate p is taken. This coarse starting value is dynamically refined while the SLD-tree is generated: argument positions that contain "growing" terms are removed. Finally, R is operationally determined by the presence or absence, in goals, of literals that can safely be unfolded.

We obtain the following algorithm:

Algorithm 7.2.1

```
Input
  a definite program $P$
  a definite goal $\leftarrow A$
Output
  a finite SLD-tree $\tau$ for $P \cup \{\leftarrow A\}$
Initialisation
  $\tau := \{(\leftarrow A, 1)\}$ {* an SLD-tree with a single derivation *}
  $Pr := \emptyset$ {* in $Pr$, the $>_{pr}$-relation will be constructed *}
  $Terminated := \emptyset$
  $Failed := \emptyset$
  For each recursive predicate $p/n$ in $P$: $S_p := \{1, \ldots, n\}$
While there exists a derivation $D$ in $\tau$ such that $D \notin Terminated$ do
  Let $(G, i)$ name the leaf of $D$
  If $(G, i) = (\Box, i)$
  Then {* $(G, i)$ is a success node *}
    add $D$ to $Terminated$
  Else
```

```
    {* First, we try to determine $R(G, i)$ *}
    Select the leftmost atom $p(t_1, \ldots, t_n)$ in $G$ such that
    one of the following (mutually exclusive) conditions is satisfied:
      • $(G, i)$ has no direct covering ancestor
      • $(G', j)$ is the direct covering ancestor of $(G, i)$
        and $|R(G', j)|_{p,S_p} > |p(t_1, \ldots, t_n)|_{p,S_p}$
      • $(G', j)$ is the direct covering ancestor of $(G, i)$
        and $|R(G', j)|_{p,S_p} \leq |p(t_1, \ldots, t_n)|_{p,S_p}$
        and $|R(G', j)|_{p,{S_p}^{new}} > |p(t_1, \ldots, t_n)|_{p,{S_p}^{new}}$ where
          ${S_p}^{new} = S_p \setminus \{a_k \in S_p \,|\, |p(t_1, \ldots, t_n)|_{p,\{a_k\}} > |R(G', j)|_{p,\{a_k\}}\} \neq \emptyset$
        and $\tau$ remains subset-wise founded with respect to
          $((R_0, R_1, \ldots, R_N), (|.|_{p_1,S_{p_1}}, \ldots, |.|_{p,{S_p}^{new}}, \ldots, |.|_{p_N,S_{p_N}}))$
    If such an atom $p(t_1, \ldots, t_n)$ cannot be found
    Then {* $(G, i)$ becomes a dangling leaf *}
      Add $D$ to $Terminated$
    Else
      $R(G, i) := p(t_1, \ldots, t_n)$
      If $R(G, i)$ was selected on the basis of the third condition above
        Then $S_p := {S_p}^{new}$
      Let $Derive(G, i)$ name the set of all derivation steps
      that can be performed
      If $Derive(G, i) = \emptyset$
      Then {* $(G, i)$ is a failure node *}
        Add $D$ to $Terminated$ and $Failed$
      Else
        {* Extend the derivation *}
        Expand $D$ in $\tau$ with the elements of $Derive(G, i)$
        Let $Descend(R(G, i), i)$ name the set of all pairs $((R(G, i), i), (B\theta, j))$,
        where
          — $B$ is an atom in the body of a clause
            applied in an element of $Derive(G, i)$
          — $\theta$ is the corresponding m.g.u.
          — $j$ is the number of the corresponding descendant of $(G, i)$
        Apply $\theta$ to the affected elements of $Pr$
        Add the elements of $Descend(R(G, i), i)$ to $Pr$
Endwhile
```

The following theorem is an immediate consequence of theorem 6.5.8.

Theorem 7.2.2 Algorithm 7.2.1 terminates. If a definite program P and a definite goal $\leftarrow A$ are given as inputs, its output τ is a finite (possibly incomplete) SLD-tree for $P \cup \{\leftarrow A\}$.

Proof As long as no S_p changes, algorithm 7.2.1 is an instance of algorithm 6.5.5. The result follows since from a finite set, only finitely many elements can be removed, and there are only finitely many S_p-sets. □

We reconsider some of the examples above.

Example 7.2.3 For the *append* program and query of example 5.5.2, we get the SLD-tree depicted in figure 7.1. (*append* has been abbreviated to *app*.) The set S_{append} does not change: $\{1, 2, 3\}$ is its final as well as its initial value.

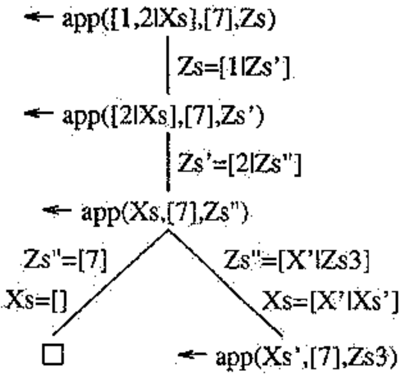


Figure 7.1: The SLD-tree for example 7.2.3.

Example 7.2.4 For the *reverse* program and query of example 5.5.4, the generated SLD-tree is shown in figure 7.2. (Again, *rev* denotes *reverse*.) Notice that, unlike the tree in figure 5.1, it is indeed finite. This time, the algorithm does refine the initial choice for $S_{reverse}$, which is of course $\{1, 2, 3\}$, to the more suitable $\{1, 3\}$: the growing accumulating parameter is removed.

Example 7.2.5 Suppose that the recipe for literal selection in the above algorithm is further refined in such a way that priority is given to non-zero-weight literals (as indicated in example 6.4.13). Then, for the *transpose* program and query of example 6.4.13, it indeed produces the tree depicted in figure 6.3, enhanced with the two dangling leaves shown in figure 6.4. Since none of the three (recursive) predicates involved shows growing argument behaviour, no elements are deleted from any S_p set.

Finally, observe that algorithm 7.2.1, unlike the algorithms in the previous chapter, is indeed fully automatic, in the sense that it requires as inputs from its user only a program and a query, and relies on no further human assistance while

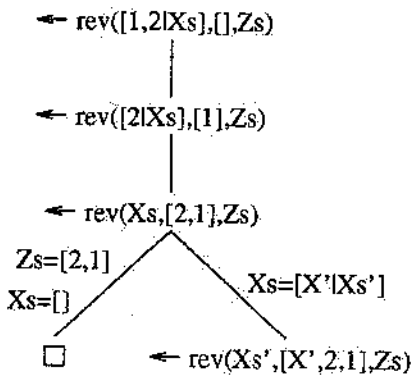


Figure 7.2: The SLD-tree for example 7.2.4.

running. (It of course also needs information on which predicates are recursive in the given program. However, this only requires a static program analysis of a basic kind, and is easy to automate. It amounts to constructing a program's dependency graph.)

7.3 An Algorithm for Partial Deduction

7.3.1 Another termination problem

In the previous section, we presented an algorithm for the automatic construction of (incomplete) finite SLD-trees. In this section, we develop a sound and complete partial deduction method, based on it. Moreover, this method is guaranteed to terminate. The following example shows that this latter property is not obvious, even when termination of the underlying unfolding procedure is ensured. We use the basic partial deduction algorithm from [14], described in section 5.5, together with our unfolding algorithm.

Example 7.3.1 For the *reverse* program and query of example 5.5.4, an infinite number of (finite) SLD-trees is produced (see figure 7.3). This behaviour is caused by the constant generation of "fresh" body-literals which, because of the growing accumulating parameter, are not an instance of any atom that was obtained before. Consequently, they require a separate partial deduction.

In [13], it is remarked that a solution to this kind of problems can be truncating atoms put into A at some fixed depth bound. However, this again seems to have an ad-hoc flavour to it, and the next section therefore proposes a method not relying on any depth bound.

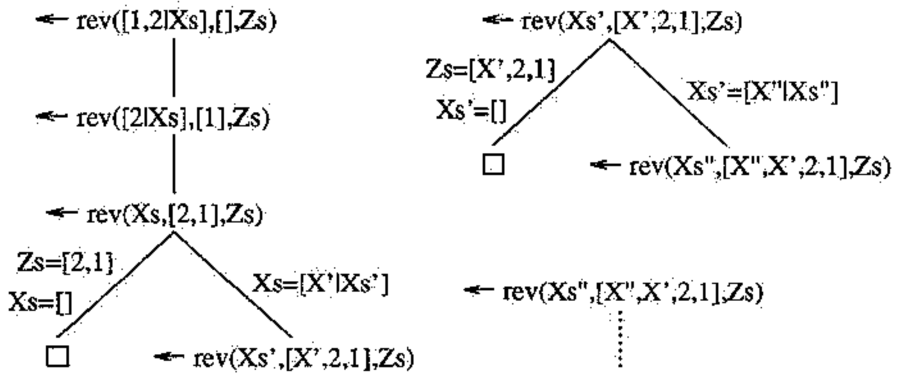


Figure 7.3: An infinite number of (finite) SLD-trees.

7.3.2 A partial deduction algorithm and its properties

We first introduce a useful definition and prove a lemma.

Definition 7.3.2 Let P be a definite program and p a predicate symbol of the language underlying P. Then the *complete pp′-renaming of P* is the program obtained in the following way:

- Take P together with an extra copy of the clauses defining p.
- Replace p in the heads of these new clauses by some new (predicate) symbol p' (of the same arity as p).
- Replace p by p' in all goals in the bodies of clauses.

Example 7.3.3
original program:

$a(X,Y) \leftarrow b(X,Z), a(Z,Y)$
$a(X,Y) \leftarrow c(X,Y)$
$b(X,Y) \leftarrow a(X,Y)$
$b(X,Y) \leftarrow a(X,X), d(Y)$

complete aa'-renaming:

$a(X,Y) \leftarrow b(X,Z), a'(Z,Y)$
$a(X,Y) \leftarrow c(X,Y)$
$a'(X,Y) \leftarrow b(X,Z), a'(Z,Y)$
$a'(X,Y) \leftarrow c(X,Y)$
$b(X,Y) \leftarrow a'(X,Y)$
$b(X,Y) \leftarrow a'(X,X), d(Y)$

Lemma 7.3.4 Let P be a definite program and P_r the complete pp'-renaming of P. Let G be a definite goal in the language underlying P. Then the following hold:

- $P_r \cup \{G\}$ has an SLD-refutation with computed answer θ iff $P \cup \{G\}$ does.
- $P_r \cup \{G\}$ has a finitely failed SLD-tree iff $P \cup \{G\}$ does.

Proof There is an obvious equivalence between SLD-derivations and -trees for P and P_r. □

We can now formulate (let $\mathcal{L}_P$ be P's underlying language):

```
Algorithm 7.3.5
Input
  a definite program $P$
  a definite goal $\leftarrow A = \leftarrow p(t_1, \ldots, t_n)$ in $\mathcal{L}_P$
  a predicate symbol $p'$, of the same arity as $p$ and not in $\mathcal{L}_P$
Output
  a set of atoms $\mathbf{A}$
  a partial deduction $P_r{}'$ of $P_r$, the complete $pp'$-renaming of $P$, wrt $\mathbf{A}$
Initialisation
  $P_r$ := the complete $pp'$-renaming of $P$
  $\mathbf{A} := \{A\}$ and label $A$ unmarked
While there is an unmarked atom $B$ in $\mathbf{A}$ do
  Apply algorithm 7.2.1 with $P_r$ and $\leftarrow B$ as inputs
  Let $\tau_B$ name the resulting SLD-tree
  Synthesise $P_{rB}$, a partial deduction for $B$ in $P_r$, from $\tau_B$
  Label $B$ marked
  Let $\mathbf{A}_B$ name the set of body literals in $P_{rB}$
  For each predicate $q$ appearing in an atom in $\mathbf{A}_B$
    Let $msg_q$ name an msg of all atoms in $\mathbf{A}$ and $\mathbf{A}_B$
    having $q$ as predicate symbol
    If there is an atom in $\mathbf{A}$ having $q$ as predicate symbol,
    less general than $msg_q$,
      Then remove this atom from $\mathbf{A}$
    If now there is no atom in $\mathbf{A}$ having $q$ as predicate symbol
      Then add $msg_q$ to $\mathbf{A}$ and label it unmarked
  Endfor
Endwhile
Finally, construct the partial deduction $P_r{}'$ of $P_r$ wrt $\mathbf{A}$,
using the partial deductions for the elements of $\mathbf{A}$ in $P_r$.
```

We show how algorithm 7.3.5 deals with the two simple examples, addressed above.

Example 7.3.6
complete renaming of the *append* program:

$append([], Y, Y) \leftarrow$
$append([X|Xs], Y, [X|Zs]) \leftarrow append'(Xs, Y, Zs)$
$append'([], Y, Y) \leftarrow$
$append'([X|Xs], Y, [X|Zs]) \leftarrow append'(Xs, Y, Zs)$

partial deduction for $\leftarrow append([1, 2|Xs], [7], Zs)$:

$append([1, 2], [7], [1, 2, 7]) \leftarrow$
$append([1, 2, X|Xs], [7], [1, 2, X|Zs]) \leftarrow append'(Xs, [7], Zs)$

partial deduction for $\leftarrow append'(Xs, [7], Zs)$:

$append'([], [7], [7]) \leftarrow$
$append'([X|Xs], [7], [X|Zs]) \leftarrow append'(Xs, [7], Zs)$

resulting set A: $\{append([1, 2|Xs], [7], Zs), append'(Xs, [7], Zs)\}$
resulting partial deduction:

$append([1, 2], [7], [1, 2, 7]) \leftarrow$
$append([1, 2, X|Xs], [7], [1, 2, X|Zs]) \leftarrow append'(Xs, [7], Zs)$
$append'([], [7], [7]) \leftarrow$
$append'([X|Xs], [7], [X|Zs]) \leftarrow append'(Xs, [7], Zs)$

Notice that this result is equal to the one obtained in example 5.5.3.

Example 7.3.7
complete renaming of the *reverse* program:

$reverse([], Zs, Zs) \leftarrow$
$reverse([X|Xs], Ys, Zs) \leftarrow reverse'(Xs, [X|Ys], Zs)$
$reverse'([], Zs, Zs) \leftarrow$
$reverse'([X|Xs], Ys, Zs) \leftarrow reverse'(Xs, [X|Ys], Zs)$

partial deduction for $\leftarrow reverse([1, 2|Xs], [], Zs)$:

$reverse([1, 2], [], [2, 1]) \leftarrow$
$reverse([1, 2, X|Xs], [], Zs) \leftarrow reverse'(Xs, [X, 2, 1], Zs)$

partial deduction for $\leftarrow reverse'(Xs, [X, 2, 1], Zs)$:

$reverse'([], [X, 2, 1], [X, 2, 1]) \leftarrow$
$reverse'([X'|Xs], [X, 2, 1], Zs) \leftarrow reverse'(Xs, [X', X, 2, 1], Zs)$

msg of $reverse'(Xs, [X, 2, 1], Zs)$ and $reverse'(Xs, [X', X, 2, 1], Zs)$:

$reverse'(Xs, [X, Y, Z|Ys], Zs)$

partial deduction for $\leftarrow reverse'(Xs, [X, Y, Z|Ys], Zs)$:

$reverse'([], [X, Y, Z|Ys], [X, Y, Z|Ys]) \leftarrow$
$reverse'([X'|Xs], [X, Y, Z|Ys], Zs) \leftarrow reverse'(Xs, [X', X, Y, Z|Ys], Zs)$

resulting set A: $\{reverse([1,2|Xs],[],Zs), reverse'(Xs,[X,Y,Z|Ys],Zs)\}$
resulting partial deduction:

$reverse([1,2],[],[2,1]) \leftarrow$
$reverse([1,2,X|Xs],[],Zs) \leftarrow reverse'(Xs,[X,2,1],Zs)$
$reverse'([],[X,Y,Z|Ys],[X,Y,Z|Ys]) \leftarrow$
$reverse'([X'|Xs],[X,Y,Z|Ys],Zs) \leftarrow reverse'(Xs,[X',X,Y,Z|Ys],Zs)$

Of course, the obtained specialised programs can be further optimised by eliminating redundant functors, using techniques as proposed in [67] or [12].

We can prove the following interesting properties of algorithm 7.3.5.

Theorem 7.3.8 Algorithm 7.3.5 terminates.

Proof Termination of the For-loop follows from the finiteness of A and A_B and from the fact that for a finite number of terms, an msg can be computed in a finite amount of time (see e.g. [105] for an algorithm).

We now show termination of the While-loop (and therefore of the entire algorithm). Termination of a single execution of the While-loop is immediate from the above and theorem 7.2.2. That it will only be executed a finite number of times is shown by the following argument. We first notice that whenever an atom is added to A, A does not contain any atom with the same predicate symbol as the one to be added. In other words, for each predicate symbol occurring in $P_r \cup \{\leftarrow A\}$, there can be at most one atom in A (and none for any other predicate symbol). This means that at any time, the number of atoms in A is not only finite, but is bounded by the (finite) number of predicate symbols occurring in $P_r \cup \{\leftarrow A\}$. Secondly, the (initially unmarked) atom considered in a specific iteration, is always marked. Finally, whenever a marked atom in A is replaced by an unmarked one, the latter is a generalisation of the former. This entails that this process of replacement will stop after a finite amount of iterations, at the latest when for this particular predicate, a most general atom (all terms being equal to uninstantiated variables) has been reached. These three considerations taken together ensure that after a finite amount of iterations of the While-loop, there will be no unmarked atoms in A. So, execution of the While-loop terminates. □

Theorem 7.3.9 Let P be a definite program, $\leftarrow p(t_1,\ldots,t_n)$ be a goal and p' be a predicate symbol used as inputs to algorithm 7.3.5. Let A be the (finite) set of atoms and $P_r{}'$ be the program output by algorithm 7.3.5. Then the following hold:

- A is independent.
- For any goal $G = \leftarrow A_1,\ldots,A_m$ consisting of atoms that are instances of atoms in A, $P_r{}' \cup \{G\}$ is A-covered.

Proof

- We first prove that **A** is independent.
  In the proof of the previous theorem we have already argued that no two atoms in **A** contain the same predicate symbol. Independence of **A** is an immediate consequence of this.

- To prove the second part of the theorem, let $P_r{}^*$ be the subprogram of $P_r{}'$ consisting of the definitions of the predicates in $P_r{}'$ upon which G depends. We show that $P_r{}^* \cup \{G\}$ is **A**-closed.
  Let A be an atom in **A**. Then the For-loop in algorithm 7.3.5 ensures there is in **A** a generalisation of any body literal in the computed partial deduction for A in $P_r{}'$. The **A**-closedness of $P_r{}^* \cup \{G\}$ now follows from the following two facts :

  1. $P_r{}'$ is a partial deduction of a program (P_r) wrt **A**.
  2. All atoms in G are instances of atoms in **A**.

□

Corollary 7.3.10 Let P be a definite program, $\leftarrow p(t_1, \ldots, t_n)$ be a goal and p' be a predicate symbol used as inputs to algorithm 7.3.5. Let **A** be the set of atoms and $P_r{}'$ be the program output by algorithm 7.3.5. Let $G = \leftarrow A_1, \ldots, A_m$ be a goal in the language underlying P, consisting of atoms that are instances of atoms in **A**. Then the following hold:

- $P_r{}' \cup \{G\}$ has an SLD-refutation with computed answer θ iff $P \cup \{G\}$ does.
- $P_r{}' \cup \{G\}$ has a finitely failed SLD-tree iff $P \cup \{G\}$ does.

Proof The corollary is an immediate consequence of lemma 7.3.4 and theorems 5.4.7 and 7.3.9. □

Proposition 7.3.11 Let P be a definite program and $\leftarrow A$ be an atomic goal used as inputs to algorithm 7.3.5. Let **A** be the set of atoms output by algorithm 7.3.5. Then $A \in$ **A**.

Proof A is put into **A** in the initialisation phase. From definition 7.3.2, it follows that no clause in P_r contains a condition literal with the same predicate symbol as A. Therefore, A will never be removed from **A**. □

This proposition ensures us that algorithm 7.3.5 does not suffer from the kind of specialisation loss illustrated in example 5.5.2: The definition of the predicate which appears in the query $\leftarrow A$, used as starting input for the partial deduction, will indeed be replaced by a partial deduction for A in P in the program output by the algorithm.

Finally, we have :

Corollary 7.3.12 Let P be a definite program, $\leftarrow A = \leftarrow p(t_1, \ldots, t_n)$ be a goal and p' be a predicate symbol used as inputs to algorithm 7.3.5. Let $P_r{}'$ be the program output by algorithm 7.3.5. Then the following hold for any instance A' of A:

- $P_r{}' \cup \{\leftarrow A'\}$ has an SLD-refutation with computed answer θ iff $P \cup \{\leftarrow A'\}$ does.
- $P_r{}' \cup \{\leftarrow A'\}$ has a finitely failed SLD-tree iff $P \cup \{\leftarrow A'\}$ does.

Proof The corollary immediately follows from corollary 7.3.10 and proposition 7.3.11. □

Theorem 7.3.8 and corollary 7.3.12 are particularly interesting. In words, their content can be stated as follows: Given a program and a goal, algorithm 7.3.5 produces a program which provides the same answers as the original program to the given query and any instances of it. Moreover, computing this (hopefully more efficient) program terminates in all cases.

7.4 Discussion, Some Related Work

Let us first briefly compare algorithm 7.3.5 with the partial deduction procedure with static renaming presented in [14] (see section 5.5). First, we showed above that our procedure terminates for all definite programs and queries while the latter does not. The culprit of this difference in behaviour is (apart from the unfolding strategy used) the way in which msg's are taken. We do this predicatewise, while the authors of [14] only take an msg when this is necessary to keep **A** independent. This may keep more specialisation, but causes non-termination whenever an infinite, independent set **A** is generated (as illustrated in example 7.3.1). The use of algorithm 7.2.1 (or further refinements, see below) guarantees that all sensible unfolding (and therefore specialisation) is obtained. The way in which algorithm 7.3.5, in addition, ensures soundness and completeness, takes care that none of the obtained specialisation is undone. Therefore, in principle, it does not seem worthwhile to consider more than one msg per predicate. See however subsection 7.5.3 for some further comments on this issue.

Next, the method in [14] is formulated in a somewhat more general framework than the one presented here. A reformulation of the latter incorporating the concept of L-selectability (i.e. some predicates can be declared "not to be unfolded") and allowing more than one literal in the starting query is quite straightforward (see also the subsequent section). Not at all immediate, however, is a generalisation to *normal programs and queries* and SLDNF-resolution, while retaining the termination property. In e.g. [14], it is proposed that during unfolding, negated calls can be executed when ground and remain in the resultant when non-ground. This of course jeopardises termination, since termination of "ordinary" ground logic program execution is not guaranteed in general. One solution is restricting attention to specific subclasses of programs (e.g. acyclic or acceptable programs, see [6], [9]). Another might be using an adapted version of our unfolding criterion in the evaluation of the ground negative call, and keeping the latter one in the resultant whenever the produced SLD(NF)-tree is not complete. Yet a third way might be offered by the use of more powerful techniques related to constructive negation (see [32], [33]). [20] can also be mentioned here. It presents a specialisation method for normal programs, based on the transformational approach to negation taken in [11] and constructive negation. It will be interesting to investigate the relationship with extensions of our method. Finally, [10] is another recent paper that treats negation in partial deduction. In the context of well-founded model semantics ([172]), it re-establishes a theoretical framework for partial deduction, including provisions for unfolding non-ground negative literals and applying loop checks.

Returning to [14], we can mention that its partial deduction method has been further extended with supporting transformations in [12]. Two separate optimisation techniques are introduced. The first is called *dynamic renaming* and constitutes a generalisation of the renaming techniques described above. Indeed, these only apply to the predicate symbol(s) occurring in the initial goal for which a partial deduction is computed. However, it might be advantageous to similarly avoid specialisation loss for other predicates, appearing in SLD-tree leaves (and thus in **A**). To this end, instead of statically introducing renamed definitions for some predicates, one performs renaming dynamically, during partial deduction. Atoms that are added to **A** and are unifiable with one already present, but not a variant (modulo renaming) of any such atom, obtain a fresh predicate symbol. In this way, no specialisation loss at all occurs. A more modest strategy, only renaming atoms that are, again modulo renaming, not an *instance* of one already present, is also mentioned. It does not prevent all specialisation loss, but since it more moderately introduces new predicates and their definitions, it is less prone to cause code explosion. Further details and formal correctness proofs can be found in [12].

A second optimisation technique proposed in [12], can be termed *argument filtering*. In essence, it is a postprocessing procedure that removes common constants, structures and multiple occurrences of variables from atoms with the same predicate symbol in the resulting program and goal. Similar techniques have been extensively studied in [67]. The beneficial influence on size and speed of transformed programs is often considerable.

Next, an interesting approach to program specialisation is proposed in [66] and further elaborated in [70]. Its general flavour differs from that of partial deduction as presented above, and its overall aims are more ambitious. We will not attempt a full discussion, but rather limit ourselves to some issues that are immediately relevant to our work in the present context. A first important ingredient is the use of *determinism* to control unfolding. In the context of pure, definite logic programs, this reduces to unfolding atoms that match the head of only one program clause. Proceeding thus avoids code explosion and other phenomena that effectively diminish instead of increase a program's execution efficiency (see also section 6.6). Several enhancements to this basic strategy are also proposed, one of the most noteworthy of which is a "look-ahead" facility, which essentially relaxes the "match only one head" criterion into "start only one non-failing branch". We briefly return to the importance of the latter in the next section. It is argued that determinism suffices to ensure termination of unfolding for realistic programs. However, a depth bound is necessary to safely deal with exceptional cases.

A second issue we would like to discuss, pertains to the overall control of partial deduction (i.e. the manipulation of the **A** set). The method in [70] first constructs **A** during a separate *flow analysis* phase. The central ingredient of the latter is an abstraction operation, relying on the notion of *characteristic path*. Such a path formally characterises how an SLD(NF)-derivation "looks". Every concrete atom that produces a derivation with the same "look" is represented by the same atom in **A**. In this way, a good compromise is achieved between having not enough atoms in **A** (and missing out on important specialisation opportunities) and having too many of them (with code explosion as a result). Especially when combined with restrictive unfolding strategies, this issue seems a crucial one. In general, certified termination of such a flow analysis requires a fixed unfolding depth bound.

Finally, another recent, quite closely related work is presented in [75]. It constitutes an automatic partial deduction system for Gödel programs, and is also based on finite unfolding and the framework in [114]. Especially the specialisation of meta-programs using the ground representation stands in the focus of attention ([76]). Moreover, the system is self-applicable. Particularly interesting in the present context, is its control strategy, both to ensure the termination of unfolding and to handle the composition of **A**. The former issue is dealt with through a static termination analysis, dividing predicates into safe and unsafe, based on the behaviour of their arguments during an abstract partial deduction. During concrete partial deduction, calls to unsafe predicates are not unfolded. Moreover, the abstract analysis also returns msgs for calls to unsafe predicates. In this way, both the computation rule and composition of **A** are *statically* determined. This might result in less specialised programs, but facilitates self-application. And, as test results show, the system's overall performance is quite satisfactory.

7.5 Experiments

7.5.1 Setup

Horváth ([83]) has implemented the above presented procedure for partial deduction, as well as several techniques drawn from the work discussed in section 7.4. The resulting system provides a good testbed for various approaches to the control of partial deduction and unfolding. In this section, we present some results obtained for five well-known benchmark programs, used to compare partial deduction systems in [104].

The five examples we will consider are fairly small, definite Prolog programs. Some of them contain calls to built-in comparison predicates (that are, of course, free of side effects). The current implementation does not touch such atoms during partial deduction. Since their role is all in all quite modest in the considered programs, this turns out not to be too big a limitation, and the programs can simply be assumed pure. (In fact, we decided not to discuss results for the *ssupply* pogram in [104] for this reason, since that particular program does rely heavily on such comparison predicates.) The source code of the five benchmark programs, together with the goals for which partial deductions were computed, as well as the goal instances used for run-time tests, can be found in appendix B.

Horváth's system allows the transformation of programs using various *combinations of control techniques*. We will present results for the following overall methods:

Sw: Basically algorithm 7.3.5. However, the implementation is founded on the work in [124] and [125], the essence of which coincides with the content of sections 7.2 and 7.3. Some details vary, mainly in the unfolding algorithm used (see e.g. subsection 8.2.4 below), but these do not affect the discussed example programs.

Swv: As above but with one important modification to algorithm 7.2.1: An atom can be selected for unfolding if the resulting goal weight *equals* the weight of the direct covering ancestor, unless a *variant* of that atom was selected in some covering ancestor. In other words, the unfolding strategy of *Swv* amalgamates those of *Sw* above and *Sv* below.
Used in this crude form, *Swv* no longer guarantees finite unfolding. However, in section 8.5, we show how the basic framework of chapter 6 can be extended to enable *safe* modifications of automatic unfolding along this line. Again, the example programs below are not affected by the more subtle aspects involved and the current implementation of *Swv* can be used.

Sv: Partial deduction is again performed as in algorithm 7.3.5, only now R_v (see page 87) governs unfolding.

Swvd: As *Swv*, but, with the possible exception of the first step, unfolding the root of an SLD-tree, only *deterministic* unfoldings are performed. Here, determinism is to be understood in the strict sense: No look-ahead is available.

Dwv: The unfolding strategy of *Swv* is combined with an overall partial deduction method that performs *dynamic renaming* (according to the eager "not a variant" heuristic).

Dwvd: As *Dwv*, but unfoldings are again *deterministic*, as in *Swvd*.

Diwvd: As the previous one, but with the *more moderate* "not an instance" *renaming* strategy.

Finally, in all cases, the gross result programs are further optimised, filtering arguments and removing obviously useless clauses, in a postprocessing phase.

For the original (*Orig*) and the various transformed programs produced by the methods described above, we present the following data:

- *ClauseNr*: the number of clauses in the program
- *Int-Time*: the approximate CPU-time (in microseconds) of *interpreted* execution with the run-time goal
- *Comp-Time*: the approximate CPU-time (in microseconds) of *compiled* execution with the run-time goal

CPU-times are not very accurate (fluctuations of 20 to 50 microseconds are common), but do provide a correct overall picture of major tendencies. Experiments were run in *ProLog by BIM* on a Sun4.

7.5.2 Results

Our first example concerns the already familiar *transpose* program (page 236). Table 7.1 displays the obtained test data. Each program is labeled with the name of the method used to produce it.

Table 7.1 shows that algorithm 7.3.5 (*Sw*) performs very well on this example. It succeeds in completely unfolding the source program and obtains a single fact, for transposing a 3×9 matrix, as its result. Of course, both *Swv* and *Dwv* lead to exactly the same outcome. Interesting is the behaviour of *Sv*. It lacks the facility to focus on covering ancestors, which turns out to be a serious handicap for dealing with this example (see also example 6.4.13). Of course, R_v can be modified so that it does restrict comparisons to chains of covering ancestors, but currently, this is not implemented. Finally, we can also observe that the methods with deterministic unfolding are somewhat too cautious. The method without

Program	ClauseNr	Int-Time	Comp-Time
Orig	6	1440	310
Sw	1	310	80
Swv	1	310	80
Sv	6	1280	210
Swvd	2	400	80
Dwv	1	310	80
Dwvd	10	450	80
Diwvd	10	440	90

Table 7.1: Test results for *transpose*.

dynamic renaming (*Swvd*) can not sufficiently specialise. It produces a (quite elegant) program for transposing a matrix with 3 rows, containing just a single predicate defined through one base and one recursive clause. Dynamic renaming prevents specialisation loss, but leads to a rather large program, with ten clauses for ten different predicates, each processing one element in 3 increasingly shorter rows. The culprit is the non-deterministic recursive *transpose* call. Obviously, a one-level look-ahead, deciding whether *nullrows*(*Xmatrix*) holds, would solve the problem.

Let us now consider *relative* (page 237), a program without any function symbols. Test results are exhibited in table 7.2.

Program	ClauseNr	Int-Time	Comp-Time
Orig	15	6300	1090
Sw	60	2010	280
Swv	21	40	10
Sv	53	610	150
Swvd	15	6270	1100
Dwv	21	30	20
Dwvd	18	5110	900
Diwvd	18	5110	950

Table 7.2: Test results for *relative*.

Not surprisingly, the performance of *Sw* is totally insufficient. But *Swv* behaves excellently: It produces a program with only ground facts, registering all relatives of *john*, most of which occur multiple times. Deterministic unfolding either does nothing, or, when combined with dynamic renaming, produces two slightly specialised definitions of *ancestor*: one finds ancestors of *john*, the other is general. A somewhat unexpected result is found for *Sv*. That method again suffers from the lack of something equivalent to hierarchical prefoundings, due to the presence of two *ancestor* calls in the clause for *relative*. (Note that this does not immediately fit either of the two patterns on page 101. We could therefore add a third case there.) Moreover, the current implementation of *Sv* does not distinguish between recursive and non-recursive predicates. As a result, not only some *ancestor* goals, but also various calls to *parent* are not unfolded.

Our third example involves meta-interpretation, computing the depth of a proof through a vanilla-like meta-program (page 238). Table 7.3 shows the test data.

Program	ClauseNr	Int-Time	Comp-Time
Orig	9	1380	300
Sw	12	90	20
Swv	12	70	20
Sv	12	100	20
Swvd	10	1430	310
Dwv	13	690	30
Dwvd	13	1390	320
Diwvd	12	1395	310
Sw^-	10	1030	260

Table 7.3: Test results for *depth*.

At first sight, the results for this example are quite surprising. In subsections 6.3.3 and 6.5.3, we discussed how properly unfolding meta-programs requires sophisticated measure functions, focusing on subarguments. And yet, algorithm 7.2.1, which lacks such a capacity, performs very well. *Sw* (and *Swv*) produces twelve ground facts, one for each element in the input list. For the given goal, this result is optimal. (Less surprisingly, *Sv* reaches the same result.) However, further reflection leads to the conclusion that this excellent result is an artefact, created by the extremely simple nature of the object program at hand: It contains no growing arguments, and no clause bodies with more than one goal. The latter feature implies that it can be meta-interpreted completely without parsing; *depth*'s second clause is never used. The former cancels the need for focusing on subarguments, as long as the same object predicate is dealt with. So the only remaining difficulty for *Sw*'s unfolding is the (single !) transfer from *member* to *append*. And this particular increase in goal weight happens to be handled by the static renaming. Without such a renaming, the result is far less satisfactory, as can be seen in table 7.3's Sw^- entry. It comes as no surprise then that dynamic renaming (*Dwv*) improves on the performance of Sw^-: Specialisation is now obtained through the construction of separate SLD-trees. However, the optimal *Sw* result is not reached: More arguments and structures are left in the twelve facts, and an extra "translation" top-level clause is produced. Finally, it can be observed that deterministic unfolding is again too weak. We return to the specific issue of unfolding meta-interpreters in subsection 8.6.4 below.

Next, let us address a classic example from partial evaluation and deduction literature: the (in)famous *match* program (page 239). The obtained test results are depicted in table 7.4.

Program	ClauseNr	Int-Time	Comp-Time
Orig	4	1560	470
Sw	8	1590	480
Swv	8	1560	470
Sv	20	4120	850
Svo	12	2630	610
Swvd	4	1820	480
Dwv	31	1530	480
Dwvd	13	1060	400
Diwvd	7	1080	400
Kmp	7	970	350

Table 7.4: Test results for *match*.

The first striking aspect of this example is the apparent breakdown of *Sv*. Part of the problem, however, seems to be the untouchable nature of the \== built-in, even when both of its arguments are ground. As a result, a number of obviously useless clauses are left in the transformed program. A postprocessed, hand-optimised version, not including such nonsensical clauses nor evidently true \== calls, was therefore produced and tested. As the *Svo* entry in table 7.4 shows, this turns out to be only a partial remedy. So, it seems that R_v really performs too many unfoldings on this example.

Sw, *Swv* and *Dwv* do not succeed in transforming *match* in a useful way. It can be noted that the same was true for the partial deduction systems tested in [104]. However, deterministic unfolding combined with dynamic renaming does produce interesting results. Particularly *Diwvd*, using a moderate renaming strategy, leads to the elegant and concise program in figure 7.4. As can be observed in table 7.4, its execution times improve noticeably upon those of the

$$
\begin{array}{l}
\mathit{match}(X) \leftarrow \mathit{match1}(X)\\[4pt]
\mathit{match1}([X|Xs]) \leftarrow a \backslash{==}\, X, \mathit{match1}(Xs)\\
\mathit{match1}([a|Xs]) \leftarrow \mathit{match2}(Xs)\\[4pt]
\mathit{match2}([X|Xs]) \leftarrow a \backslash{==}\, X, \mathit{match1}([X|Xs])\\
\mathit{match2}([a|Xs]) \leftarrow \mathit{match3}(Xs)\\[4pt]
\mathit{match3}([X|Xs]) \leftarrow b \backslash{==}\, X, \mathit{match1}([a, X|Xs])\\
\mathit{match3}([b|Xs]) \leftarrow
\end{array}
$$

Figure 7.4: *match* program produced by *Diwvd*.

original *match* program. However, we also find that the basic "upon a non-match, shift one position in the string and restart the search" strategy of *match* is left untouched. A smart "Knuth-Morris-Pratt"-like (see the comments and references on page 239) pattern matcher would pass smaller arguments to the "*match*" call in the first clauses for *match2* and *match3*. A program operating in that way is depicted in figure 7.5. We also submitted this program to a test-run, using the

$$
\begin{array}{l}
\mathit{match}(X) \leftarrow \mathit{match1}(X)\\[4pt]
\mathit{match1}([X|Xs]) \leftarrow a \backslash{==}\, X, \mathit{match1}(Xs)\\
\mathit{match1}([a|Xs]) \leftarrow \mathit{match2}(Xs)\\[4pt]
\mathit{match2}([X|Xs]) \leftarrow a \backslash{==}\, X, \mathit{match1}(Xs)\\
\mathit{match2}([a|Xs]) \leftarrow \mathit{match3}(Xs)\\[4pt]
\mathit{match3}([X|Xs]) \leftarrow b \backslash{==}\, X, \mathit{match2}([X|Xs])\\
\mathit{match3}([b|Xs]) \leftarrow
\end{array}
$$

Figure 7.5: KMP-like *match* program.

same goal as before. The *Kmp* entry in table 7.4 shows the results: A further performance improvement is indeed attained, in spite of the fact that the given goal does not elicit *Kmp*'s major talents.

A close inspection of its operation reveals why *Diwvd* stops short of producing *Kmp*. Unfolding the unsatisfactory calls one further step, and being able to handle ground \== atoms would do the trick. In fact, the result would be the program obtained by Gallagher in [65], from which *Kmp* can be derived via duplicate call removal. And indeed, [65] uses deterministic unfolding with a look-ahead facility.

As a final example, we consider a more intricate pattern matching program, named *contains* (page 240). Test data for some methods with weight based unfolding can be found in table 7.5.

First, unlike for the *match* program just considered, methods with determin-

Program	ClauseNr	Int-Time	Comp-Time
Orig	7	3230	680
Sw	24	5090	600
Swv	29	4190	480
Swvd	7	3150	660
Dwv	32	930	320
Dwvo	27	820	270
Dwvd	13	3080	660
Diwvd	11	3130	630
Dwvo-ma	27	1860	550
Orig-ma	7	7230	1440

Table 7.5: Test results for *contains*.

istic unfolding return the original program virtually unchanged. This agrees with [70], where it is also observed that deterministic unfolding seems unable to deal properly with the profoundly non-deterministic *contains* example. *Sw* and *Swv* show signs of code explosion: the result programs are fairly large and the speed of interpreted execution is in fact considerably worse than that of the original program. In section 6.6, we conjectured that in such cases, compiled execution will perform much better. Clearly, the present example confirms this. It can be observed that of *Sw* and *Swv*, the latter produces a better result: The larger program is in fact faster because, through deeper unfolding, more specialisation can be kept. However, both methods suffer from the "one msg per predicate" strategy which inhibits proper specialisation of *con*. Dynamic renaming offers salvation: *Dwv* produces the largest program, but also the fastest. In fact, the improper treatment of \== again leaves some obviously discardable calls and clauses. The *Dwvo* entry in table 7.5 shows the results after manual postprocessing.

Finally, we thought it might be amusing to run *Dwvo* with the (more difficult) run-time pattern matching task used in the *match* example. The outcome of this experiment is included in the *Dwvo-ma* entry. It can be compared with the performance of the original *contains* program on the same query (*Orig-ma*) and the results for the various *match* versions. Of course, as noted above, for neither of the two run-time goals *contains'* more clever repositioning strategy is of much use. A fair comparison should therefore include tests where the string does contain (initial) parts of the search pattern. However, such further explorations are not of immediate interest in the present context.

This concludes our presentation of experimental results. Further material on various comparative tests can be found in [83].

7.5.3 Discussion

Of course, the above experiments constitute a somewhat shallow basis for a general comparative discussion. However, we nevertheless endeavour to formulate some tentative conclusions which may be the subject of future experimental verification.

- First, *weight based unfolding*, particularly within the framework of *hierarchical* prefoundings, seems to provide a good foundation for partial deduction. Not only does it enable unfolding that terminates in a non ad-hoc way, but its behaviour is also quite reasonable from a practical point of view. This can be contrasted with non-variant based unfolding which, apart from possible non-termination, is also plagued by a tendency to unfold too deeply. (For *contains*, partial deduction using *Sv* completely explodes.) Moreover, the *covering* ancestor concept seems of prime practical importance.

- However, *algorithm 7.2.1 itself is not powerful enough* to provide proper unfolding in all cases. A combination with non-variant unfolding is obviously an improvement. It also lacks sophistication to deal with complex meta-programs. Other issues beyond its reach include unfolding obeying co-routining computation rules, and handling back propagation of variable instantiation. These topics are extensively studied in chapter 8 of this thesis.

- *Deterministic* unfolding in its elementary "match-only-one-clause" form is often *too restrictive*. A *look-ahead* facility seems indispensable. Of course, taken to its limit, this reverts to the weight based construction of entire (incomplete) SLD-trees and the subsequent pruning of "inappropriate" branches. However, deciding which branches should in fact be cut, is by no means straightforward. For the *transpose* and the *relative* programs above, both giving rise to *complete* SLD-trees, it would be easy to see that no such trimming should in fact take place. But as *match* and *contains* show, the general picture is not so simple. We conjecture that the optimal choice depends on the amount of non-determinism inherent in the program, but clearly a challenging area of research lies here, to the best of our knowledge virtually unexplored.

- Finally, having *one msg in* A *per predicate symbol*, though of theoretical interest because it guarantees termination, and, combined with powerful unfolding, not at all practically useless, is probably *insufficient*. First of all, as could be expected, restrictive unfolding strategies obviously need more flexible approaches: in most examples *Swvd* does virtually nothing. More importantly, also in most examples, a method with dynamic renaming produces the best result, or at least equals it. Interesting in this respect is the one exception: *depth*. It reveals a drawback inherent in the construction of several smaller SLD-trees: specialisation can not pass from subsequent trees to earlier ones. Repeated processing provides a remedy: for *depth*, running *Dwv* on its output leads to the optimal program produced by *Sw*.

- Next, it can be noted that it is not so much the dynamic *renaming* that is responsible for the good behaviour of *Dwv* (or, for *match*, *Diwvd*). In fact, very little renaming is going on in most of the above experiments, and simply using the A-control strategy of [14] produces basically the same results. We nevertheless decided to concentrate on the dynamic renaming strategies, because these are the more general. Of course, neither of the two approaches guarantees *termination*. In spite of the excellent test results above, this shortcoming should be judged unacceptable. A possible way out might be provided by the application of weight based methods, not unlike our approach to unfolding, to govern the composition of A. The basic idea is as follows. If an atom appears in the leaf of some SLD-tree, and it is a candidate for addition to A, one would first check whether it "descends" (through a series of SLD-trees) from some atom in A with the same predicate symbol. If so, the latter should be "heavier". Of course, bookkeeping might get quite complex. An alternative is provided by a safe (i.e. sufficiently abstract) preliminary analysis to decide the composition of A as proposed in [70]. A combination, improving the precision of the analysis, seems also feasible. Further research is necessary.

7.6 Conclusion

In this chapter, we derived a first fully automatic algorithm for finite unfolding, based on the framework laid out in chapter 6. We then used it as a building block in an overall partial deduction method for definite logic program and queries. The latter was shown to be sound and complete in the sense of [114], and to always terminate.

We also sketched some of the main ingredients found in related work on partial deduction. Experiments comparing various blends were performed and their results briefly discussed.

Weight based unfolding of the kind proposed in chapter 6 emerges as a promising approach, also from a practical point of view. However, quite a few issues require further study, in the control of automatic finite unfolding, as well as in the guidance of overall partial deduction. We will leave the latter as a topic for possible future research, but extensively address the former in the next chapter.

Chapter 8

Advanced Techniques in Finite Unfolding

8.1 Introduction

In section 7.2, we presented a first fully automatic unfolding algorithm, based on the framework laid out in chapter 6. We discussed its operational merits in section 7.5, and concluded that it seems to provide a good basis for the construction of operationally viable unfolding strategies. However, we also pointed out some of its inherent limitations, and in fact already applied some modifications, not yet formally justified. This final chapter of part II therefore reconsiders the problem of how to finitely unfold definite logic programs and queries. Detailed formal developments and concrete algorithms are presented. Proceeding in this way, we strive to achieve several aims:

- First, we present (at least partial) solutions to the unfolding issues left open in the preceding chapter. We also provide formal justifications for already mentioned modifications and enhancements to algorithm 7.2.1.

- Throughout most of what follows, we will particularly concentrate on issues pertaining to full automation, presenting extensive formalisations of the technicalities involved at a gradually increasing level of generality. In this way, it will be possible to uncover common principles underlying our fully automatic techniques.

- Elaborating several concrete instances of the framework in chapter 6, we hope to illustrate its generality and flexibility.

- Finally, in a wider perspective, this chapter can be regarded as a study on what seems possible and what not in online loop prevention, not relying on any substantial offline analysis techniques.

This leads to the following sketch of the present chapter's further layout. Section 8.2 describes a first generalisation of the earlier work. More general measure functions are introduced that incorporate lexicographical priorities among arguments in a selected literal. We present an automatic unfolding algorithm relying on such functions, as well as a noteworthy optimisation. In section 8.3, we build on this work and expand our horizon to also consider parts of a goal, not belonging to the literal to be unfolded, while deciding on unfolding. We show how this enables a treatment of co-routining, as well as variable instantiation back propagation. Section 8.4 contains an explicit formalisation of the underlying issues in the search for optimal measure functions, carried out by automatic unfolding algorithms. Next, section 8.5 forges a unified approach, incorporating both weight based unfolding and the "checking for a variant ancestor" technique. In this way, it presents a formal justification for the "combined" unfolding rule used in section 7.5. Next, in section 8.6, we propose yet another refinement of the basic unfolding method in algorithm 7.2.1: We show that the framework in chapter 6 is sufficiently powerful to allow automatic focusing on subarguments. We illustrate how the resulting algorithm improves unfolding of meta-programs. We touch on remaining problems in that context and suggest possible solutions as interesting directions for further research. Finally, a brief overall discussion again constitutes the chapter's end.

8.2 Lexicographical Priorities

8.2.1 Introduction

In this section, we consider a first generalisation of the automatic unfolding method proposed in section 7.2. Indeed, algorithm 7.2.1 relies on measures considering a subset of the selected literal's argument positions, as introduced in definition 6.3.3. It turns out that solving some of the problems mentioned above, requires measures that also take into account arguments of other literals in the goal, and moreover, are capable of imposing a priority between different (subsets of) arguments.

A method incorporating these facilities will be presented in the next section. First we prepare the way by elaborating an intermediary extension of the preceding work. We will in this section still limit ourselves to measures based on the arguments of a goal's selected literal, but we will *impose priorities among different argument positions*. We will show that this first advance already results in increased unfolding power.

Below, we first introduce measure functions based on partitions of a predicate's set of argument positions and show that they can be used as well-founded measures on an SLD-tree, resulting in increased unfolding potential. Next, we present a fully automatic unfolding algorithm, capable of discovering optimal partition based measures. Finally, we show that an important simplification of the algorithm keeps the refined control and the guaranteed termination while efficiency is improved.

8.2.2 More powerful measures

We set out by introducing the following bits of notation. Let V be a set.

- Then $\mathcal{P}(V)$ denotes the powerset (or set of subsets) of V.
- Then V^n denotes the n-fold Cartesian product $V \times V \times \ldots \times V$ (n copies) of V.

This allows us to define the following:

Definition 8.2.1 Let V be a set and let $S_1, \ldots, S_k$ be k mutually disjoint, non-empty subsets of V, together forming a partition of V. Then the k-tuple $(S_1, \ldots, S_k) \in \mathcal{P}(V)^k$ is called an *ordered k-partition* of V.

In the sequel, we will often simply use the term *ordered partition* when the dimension (k) is clear from the context, or unimportant. Moreover, our attention will focus on ordered partitions of the set of argument positions of predicate symbols. In this context, we will refer to a predicate symbol and an *associated* ordered (k-)partition, without explicit mention of the fact that the latter is a partition of the former's *set of argument positions*.

Definition 8.2.2 Let p be a predicate of arity n with an associated ordered k-partition $O = (\{i_{11}, \ldots, i_{1j}\}, \ldots, \{i_{k1}, \ldots, i_{kl}\})$. We define $|.|_{p,O} : \{A|A$ is an atom with predicate symbol $p\} \to I\!N^k$ as follows:

$$|p(t_1, \ldots, t_n)|_{p,O} = (|t_{i_{11}}| + \cdots + |t_{i_{1j}}|, \ldots, |t_{i_{k1}}| + \cdots + |t_{i_{kl}}|)$$

where $|.|$ is the functor norm as defined in definition 6.3.2.

Example 8.2.3
$|p([a,b,c], f(g(a)), b)|_{p,(\{1,2,3\})} = (3+2+0) = (5)$
$|p([a,b,c], g(a), b)|_{p,(\{2\},\{1,3\})} = (1,3)$

We intend to use such partition based measure functions instead of the subset based ones introduced in definition 6.3.3. However, an atom is no longer mapped into an element of $I\!N$, but into a tuple in $I\!N^k$. (k of course can be 1, as in the first line of example 8.2.3 above.) We must therefore first establish that we can indeed define an order on $I\!N^k$, such that it becomes a well-founded set.

Definition 8.2.4 Let V be a set with an order $>$ (and equality $\equiv$). Then we define the *lexicographical order* $\succ_k$ on V^k as follows:

$$\begin{gathered}(v_1,\ldots,v_k) \succ_k (w_1,\ldots,w_k)\\ \text{iff}\\ \forall 1 \leq i \leq k : v_i \equiv w_i \text{ or } v_i > w_i \text{ or } w_i > v_i\\ \text{and}\\ \exists 1 \leq i \leq k : v_i > w_i \text{ and } \forall 1 \leq j < i : v_j \equiv w_j\end{gathered}$$

Notice that the relation $\succ_k$ thus defined, is indeed a strict order relation. The first condition in the above definition is necessary in the context of partial orders, to impose the restriction that the components of both tuples should be pairwise comparable. For $I\!N$ and $I\!N^k$, this condition is of course trivially satisfied since we have a total order on $I\!N$.

Proposition 8.2.5 Let $V, >$ be a well-founded set, and $\succ_k$ the associated lexicographical order on V^k. Then $V^k, \succ_k$ is a well-founded set.

Proof The proposition follows from the well-foundedness of $V, >$, definition 8.2.4 and the fact that a tuple in V^k has only a finite number, k, of components. □

In particular, for each k, $I\!N^k, \succ_k$ is a well-founded set. This means that functions as introduced in definition 8.2.2 can indeed be used as measure functions to control unfolding.

Proposition 8.2.6 Let O be an ordered k-partition associated to a predicate p. Let τ be an SLD-tree and S_τ a subset of G_τ such that all goals in S_τ have a selected literal with predicate p. Suppose F_p is defined as follows:

$F_p : S_\tau, >_\tau \rightarrow I\!N^k, \succ_k : (G,i) \in S_\tau \mapsto |R(G,i)|_{p,O}$

Then F_p is a well-founded measure on $S_\tau, >_\tau$ iff F_p is monotonic.

Proof This immediately follows from definition 6.2.3 and proposition 8.2.5. □

Doing so can entail a significant gain in unfolding capacity, as the following prototypical example shows.

Example 8.2.7 Consider the following program:

$produce\_consume([X|Xs],[]) \leftarrow produce\_consume(Xs,[X])$
$produce\_consume(X,[Y|Ys]) \leftarrow produce\_consume(X,Ys)$

And query:

$\leftarrow produce\_consume([1,2|Xs],[])$

Now apply algorithm 6.5.5 using a single R-class for all selected literals, with associated measure function $|.|_{p\_c,O}$ where $O = (\{1\},\{2\})$ is an ordered 2-partition

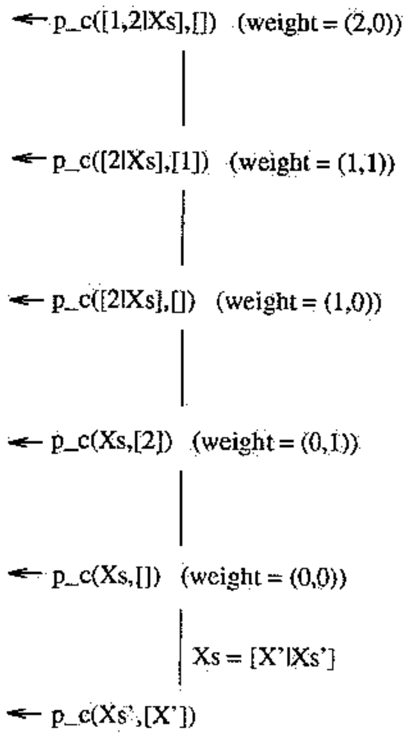

← p_c([1,2|Xs],[]) (weight = (2,0))
← p_c([2|Xs],[1]) (weight = (1,1))
← p_c([2|Xs],[]) (weight = (1,0))
← p_c(Xs,[2]) (weight = (0,1))
← p_c(Xs,[]) (weight = (0,0))
Xs = [X'|Xs']
← p_c(Xs',[X'])

Figure 8.1: Unfolding with weights in $\mathbb{N}^2$.

associated to the predicate *produce_consume*, abbreviated to *p_c*. The resulting incomplete SLD-tree is depicted in figure 8.1. Nodes are annotated with their weight, except the last one, which is a dangling leaf.
Notice that, when a simple one-component weight of the kind introduced in definition 6.3.3 is used, only a trivial single step is possible if both arguments (or only the second) are considered, while focusing on the first argument causes termination after two steps.

8.2.3 An automatic unfolding algorithm

In this subsection, we present a detailed, fully automatic algorithm for unfolding, based on the ingredients introduced above. It is a first extension of algorithm 7.2.1. We would like to obtain a concise and clear formulation of the algorithm, including the automatic search for optimal measure functions. To make this possible, we first include some helpful definitions and prove a few relevant properties about the concepts they introduce.

Setting the scene

We set out with some straightforward definitions related to the behaviour of measure functions on a pair of atoms.

Definition 8.2.8 Let p be a predicate of arity n. Let $P_1 = p(t_1, \ldots, t_n)$ and $P_2 = p(s_1, \ldots, s_n)$ be two atoms and F a mapping from the set of terms in the language underlying P_1 and P_2 to an s-poset $V, >$. Then an *argument position* i $(1 \leq i \leq n)$ is:

- (P_1, P_2)-*decreasing* for F iff $F(t_i) > F(s_i)$
- (P_1, P_2)-*increasing* for F iff $F(s_i) > F(t_i)$
- (P_1, P_2)-*stable* for F iff $F(t_i) \equiv F(s_i)$

Example 8.2.9 Let $|.|$ be the functor norm, counting functors in terms as defined in definition 6.3.2. Take
$P_1 = p([a, b, c], f(g(a)), b)$
and
$P_2 = p([d, e], g(g(b)), h(b))$
Then:

- 1 is (P_1, P_2)-decreasing for $|.|$
- 2 is (P_1, P_2)-stable for $|.|$
- 3 is (P_1, P_2)-increasing for $|.|$

In the present section, we will always take F equal to $|.|$. We will therefore usually omit the explicit "for $|.|$" addition. Notice that, since $|.|$ maps to the *totally* (strictly) ordered $\mathbb{N}, >$, an argument position is either decreasing or increasing or stable for $|.|$. A similar remark pertains to definitions 8.2.10 and 8.2.12 below. The next definition focuses not on argument positions, but one level higher up the scale, in the restricted context of partition based measure functions.

Definition 8.2.10 Let p be a predicate of arity n and O an associated ordered k-partition. Let P_1 and P_2 be two atoms such that $|P_1|_{p,O} = (v_1, \ldots, v_k)$ and $|P_2|_{p,O} = (w_1, \ldots, w_k)$. Then the i-th $(1 \leq i \leq k)$ *component* of O is:

- (P_1, P_2)-*decreasing* iff $v_i > w_i$
- (P_1, P_2)-*increasing* iff $w_i > v_i$

- (P_1, P_2)-*stable* iff $v_i = w_i$

We will use the notation $O[i]$ to denote the i-th component of an ordered partition O.

Example 8.2.11 Take P_1 and P_2 as in example 8.2.9.

- Let $O = (\{1, 2, 3\})$ then its single component $\{1, 2, 3\}$ is (P_1, P_2)-stable.
- For $O' = (\{1, 2\}, \{3\})$, we have:
  - $O'[1] = \{1, 2\}$ is (P_1, P_2)-decreasing.
  - $O'[2] = \{3\}$ is (P_1, P_2)-increasing.

Finally, at the level of complete measure functions, we can introduce:

Definition 8.2.12 Let M be a *mapping* from a set S of atoms to an s-poset $V, >$. Let P_1 and P_2 be two atoms in S. Then M is:

- (P_1, P_2)-*decreasing* iff $M(P_1) > M(P_2)$
- (P_1, P_2)-*increasing* iff $M(P_2) > M(P_1)$
- (P_1, P_2)-*stable* iff $M(P_1) \equiv M(P_2)$

Of course, also definition 8.2.12 will mainly be applied to $|.|_{p,O}$-like measure functions.

Example 8.2.13 Take P_1, P_2, O and O' as above.

- $|.|_{p,O}$ is (P_1, P_2)-stable.
- $|.|_{p,O'}$ is (P_1, P_2)-decreasing, since $(5, 0) \succ_2 (4, 1)$.
- If $O'' = (\{3\}, \{1, 2\})$ then $|.|_{p,O''}$ is (P_1, P_2)-increasing, since $(1, 4) \succ_2 (0, 5)$.

In the sequel, we will occasionally drop the (P_1, P_2) annotation while using the terminology introduced above (and below), when it is clear which couple of atoms is intended.

The above definitions will be useful in the context of comparing the weight of a goal with the weight of its direct covering ancestor. If we find that the weight increases, we will try to replace the ordered partition in use by one that does result in a decrease. The next few definitions further prepare the way for this operation.

Definition 8.2.14 Let p be a predicate of arity n and O an associated ordered k-partition. Let P_1 and P_2 be two atoms with predicate symbol p. Then $O[i]$ is *O's leftmost* (P_1, P_2)-*increasing component* if

1. it is (P_1, P_2)-increasing
2. there is no $1 \leq j < i$ such that $O[j]$ is (P_1, P_2)-increasing

We will occasionally use "leftmost" and "rightmost" in similar contexts without explicitly including a precise definition as the one above.

Definition 8.2.15 Let p be a predicate of arity n. Let P_1 and P_2 be two atoms with predicate symbol p and O an associated ordered k-partition. Then we call a component $O[i]$ (P_1, P_2)-*sensitive* if the following two conditions are satisfied:

1. $O[i]$ contains at least one decreasing argument position.
2. If $|.|_{p,O}$ is (P_1, P_2)-increasing and $O[l]$ is O's leftmost increasing component then $i \leq l$.

Below, we will be interested in replacing a non-decreasing measure by one that does decrease, through a more detailed partitioning of the set of argument positions. In particular, this can be obtained by splitting a sensitive component in, first, a decreasing and, second, an increasing part. Note that, for non-decreasing measures, both parts will be non-empty. However, splitting a component in this way only produces the desired effect if it is not preceded by an increasing component. This is the reason for the second condition above. Finally, note that, while the *leftmost* is therefore the focus of attention among the *increasing* components, below we will be interested in the *rightmost* among the *sensitive* components. Indeed, splitting that one will result in the least drastic, useful weight change (see point (2b) in the proof of proposition 8.2.24). The following definition focuses on *non-decreasing* measures.

Definition 8.2.16 Let p be a predicate of arity n and O an associated ordered k-partition. Let P_1 and P_2 be two atoms with predicate symbol p, such that $|.|_{p,O}$ is not (P_1, P_2)-decreasing. Then we say that $|.|_{p,O}$ has (P_1, P_2)-*potential* iff O has at least one (P_1, P_2)-sensitive component.

Example 8.2.17 Take P_1, P_2, O, O' and O'' as above.

- $O[1]$ is (P_1, P_2)-sensitive. Since $|.|_{p,O}$ is (P_1, P_2)-stable, this implies that $|.|_{p,O}$ has (P_1, P_2)-potential.
- $O'[1]$ is (P_1, P_2)-sensitive.
- O'' has no (P_1, P_2)-sensitive components.

We have the following property:

Proposition 8.2.18 Let p be a predicate of arity n and O an associated ordered k-partition. Let P_1 and P_2 be two atoms with predicate symbol p, such that $|.|_{p,O}$ has (P_1, P_2)-potential. Let $O[i]$ be a (P_1, P_2)-sensitive component. Then $\forall 1 \leq j < i : O[j]$ is (P_1, P_2)-stable.

Proof $|.|_{p,O}$ has (P_1, P_2)-potential. It is therefore either (P_1, P_2)-increasing or (P_1, P_2)-stable. In the latter case, the result is immediate. In the former case, it follows from the second condition in definition 8.2.15. □

We now formally introduce a refinement operation for ordered partitions and partition based measures.

Definition 8.2.19 Let $O = (C_1, \ldots, C_k)$ be an ordered k-partition of some set V. Then the ordered $k+1$-partition $O' = (C'_1, \ldots, C'_{k+1})$ is called an *i-refinement* of O $(1 \leq i \leq k)$ if:

- $C_i = C'_i \cup C'_{i+1}$
- $\forall 1 \leq j < i : C_j = C'_j$
- $\forall i < j \leq k : C_j = C'_{j+1}$

Example 8.2.20 When defined as above, both O' and O'' are 1-refinements of O.

As in the following definition, we will occasionally use the term "refinement" in contexts where the actual i index does not matter.

Definition 8.2.21 Let p be a predicate of arity n and O an associated ordered partition. Let O' be a refinement of O. Then we call $|.|_{p,O'}$ a *refinement* of $|.|_{p,O}$.

Finally, we are in a position to introduce the following key concept:

Definition 8.2.22 Let p be a predicate of arity n and $O = (C_1, \ldots, C_k)$ an associated ordered k-partition. Let P_1 and P_2 be two atoms with predicate symbol p, such that $|.|_{p,O}$ has (P_1, P_2)-potential. Let C_l be O's rightmost (P_1, P_2)-sensitive component. Then $|.|_{p,O'}$ is the *tight (P_1, P_2)-decreasing refinement* of $|.|_{p,O}$ if O' is an l-refinement of O and:

- $O'[l] = \{i \in C_l | i$ is (P_1, P_2)-decreasing or (P_1, P_2)-stable$\}$
- $O'[l+1] = \{i \in C_l | i$ is (P_1, P_2)-increasing$\}$

Example 8.2.23 In our running example, $|.|_{p,O'}$ is the tight (P_1, P_2)-decreasing refinement of $|.|_{p,O}$. Notice that no refinement of $|.|_{p,O''}$ is (P_1, P_2)-decreasing.

Definition 8.2.22 is central in the automatic search for good measure functions. Indeed, when a given measure function does not result in a weight decrease, we will (try to) replace it by its tight decreasing refinement, thus enabling further unfolding. Before we present the details of the procedure in the algorithm below, we first establish that tight decreasing refinements are well-defined.

Proposition 8.2.24 Let p be a predicate of arity n and $O = (C_1, \ldots, C_k)$ an associated ordered k-partition. Let P_1 and P_2 be two atoms with predicate symbol p, such that $|.|_{p,O}$ has (P_1, P_2)-potential. Then $|.|_{p,O'}$, the tight (P_1, P_2)-decreasing refinement of $|.|_{p,O}$, exhibits the following properties:

1. $|.|_{p,O'}$ is (P_1, P_2)-decreasing.
2. Any other (P_1, P_2)-decreasing refinement, $|.|_{p,O''}$, of $|.|_{p,O}$, obtained by splitting one of O's components in two subsets, one of which contains all its increasing argument positions, has the following property:

   There is no atom P with predicate symbol p such that $|P|_{p,O''} \succ_{k+1} |P|_{p,O'}$.

Proof Definition 8.2.16 assures that O has a rightmost (P_1, P_2)-sensitive component. Let l be its index. Then we can argue as follows:

1. First, we note that $O'[l]$ is (P_1, P_2)-decreasing. Next, proposition 8.2.18 implies that all O' components to the left of $O'[l]$ are (P_1, P_2)-stable. It follows that $|.|_{p,O'}$ is (P_1, P_2)-decreasing.
2. The desired property follows from the following considerations:
   (a) Splitting a component which is *not* (P_1, P_2)-sensitive, if possible, does not result in a (P_1, P_2)-decreasing measure.
   (b) Suppose we split a (P_1, P_2)-sensitive component $O[i]$ with $i \neq l$. It follows that $i < l$. Suppose that $|P|_{p,O} = (v_1, \ldots, v_k)$.
   Let
   $$|P|_{p,O'} = (v'_1, \ldots, v'_i, \ldots, v'_l, v'_{l+1}, \ldots, v'_{k+1})$$
   and
   $$|P|_{p,O''} = (v''_1, \ldots, v''_i, v''_{i+1}, \ldots, v''_l, \ldots, v''_{k+1}).$$
   Then $\forall 1 \leq j < i : v_j = v'_j = v''_j$. Moreover $v'_i = v_i$ and $v_i \geq v''_i$.
   - If $v_i > v''_i$ then obviously $|P|_{p,O'} \succ_{k+1} |P|_{p,O''}$.
   - If $v_i = v''_i$ then $v''_{i+1} = 0$. Moreover, $\forall i+1 \leq j < l : v'_j = 0 \Rightarrow v''_{j+1} = 0$. Finally, $v'_l = 0 \Rightarrow v'_{l+1} = v_l = v''_{l+1}$ while, of course, $\forall l+1 < j \leq k+1 : v'_j = v_{j-1} = v''_j$. So, either $v'_j \neq 0$ for some $i < j \leq l$, from which it follows that $|P|_{p,O'} \succ_{k+1} |P|_{p,O''}$. Or $|P|_{p,O'} = |P|_{p,O''}$.

(c) The property is immediate for any refinement where also some non-increasing argument positions are put into $O''[l+1]$.

□

The second property above is the motivation for including "tight" in the naming done in definition 8.2.22. It ensures that we are relatively "conservative" when taking refinements. Bigger weights are generally better than smaller because they allow a longer decrease, which means more unfolding potential. It would of course be possible to further "tighten" definition 8.2.22 by splitting off sufficiently large *subsets* of the set of increasing argument positions in the rightmost increasing component. We have decided not to do this because it would complicate the method both conceptually and computationally. Moreover, we conjecture that there would almost never be a substantial gain in unfolding capacity.

The algorithm

Above, we developed partition based measure functions, thus generalising the subset based functions introduced in definition 6.3.3. We now use them as a basic ingredient of an advanced automatic algorithm for sensible finite unfolding.

In algorithm 8.2.25 below, we choose $R_0, R_1, \ldots, R_N$ as indicated in section 7.2. This means that a goal (G', j) such that $(G', j) >_{pr} (G, i)$, *covers* (G, i) if their *selected literals contain the same recursive predicate symbol.* For $F_1, \ldots, F_N$, we take $|.|_{p,O_p}$-like functions, one per recursive predicate, where, as before, we associate with a *goal* the value of the relevant measure function on its *selected literal.* The optimal partitions O_p to be used for each recursive predicate p, and the computation rule R are dynamically fixed while executing the algorithm.

Algorithm 8.2.25

```
Input
  a definite program $P$
  a definite goal $\leftarrow A$
Output
  a finite SLD-tree $\tau$ for $P \cup \{\leftarrow A\}$
Initialisation
  $\tau := \{(\leftarrow A, 1)\}$ {* an SLD-tree with a single derivation *}
  $Pr := \emptyset$ {* in $Pr$, the $>_{pr}$-relation will be constructed *}
  $Terminated := \emptyset$
  $Failed := \emptyset$
  For each recursive predicate $p/n$ in $P$: $O_p := (\{1, \ldots, n\})$
  {* We set out with 1-partitions
  grouping all argument positions in a single component *}
While there exists a derivation $D$ in $\tau$ such that $D \notin Terminated$ do
  Let $(G, i)$ name the leaf of $D$
  If $(G, i) = (\Box, i)$
  Then {* $(G, i)$ is a success node *}
    add $D$ to $Terminated$
  Else
    {* First, we try to determine $R(G, i)$ *}
    Select the leftmost atom $p(t_1, \ldots, t_n)$ in $G$ such that
    one of the following (mutually exclusive) conditions is satisfied:
      • $(G, i)$ has no direct covering ancestor
      • $(G', j)$ is the direct covering ancestor of $(G, i)$ and
        $|\cdot|_{p,O_p}$ is $(R(G', j), p(t_1, \ldots, t_n))$-decreasing
      • $(G', j)$ is the direct covering ancestor of $(G, i)$ and
        $|\cdot|_{p,O_p}$ has $(R(G', j), p(t_1, \ldots, t_n))$-potential and
        $|\cdot|_{p,O'_p}$ is its tight $(R(G', j), p(t_1, \ldots, t_n))$-decreasing refinement and
        $\tau$ remains subset-wise founded with respect to
          $((R_0, R_1, \ldots, R_N), (|\cdot|_{p_1,O_{p_1}}, \ldots, |\cdot|_{p,O'_p}, \ldots, |\cdot|_{p_N,O_{p_N}}))$          $(*)$
    If such an atom $p(t_1, \ldots, t_n)$ cannot be found
    Then {* $(G, i)$ becomes a dangling leaf *}
      Add $D$ to $Terminated$
    Else
      $R(G, i) := p(t_1, \ldots, t_n)$
      If $R(G, i)$ was selected on the basis of the third condition above
      Then {* Register the new partition *}
        $O_p := O'_p$
      Let $Derive(G, i)$ name the set of all derivation steps
      that can be performed
      If $Derive(G, i) = \emptyset$
      Then {* $(G, i)$ is a failure node *}
        Add $D$ to $Terminated$ and $Failed$
      Else
        {* Extend the derivation *}
        Expand $D$ in $\tau$ with the elements of $Derive(G, i)$
        Let $Descend(R(G, i), i)$ name the set of all pairs $((R(G, i), i), (B\theta, j))$,
        where
          — $B$ is an atom in the body of a clause
            applied in an element of $Derive(G, i)$
          — $\theta$ is the corresponding m.g.u.
        — $j$ is the number of the corresponding descendant of $(G, i)$
    Apply $\theta$ to the affected elements of $Pr$
    Add the elements of $Descend(R(G, i), i)$ to $Pr$
Endwhile
```

Example 8.2.26 For the *produce_consume* program and query of example 8.2.7, algorithm 8.2.25 produces exactly the SLD-tree depicted in figure 8.1. Indeed, the first unfolding is possible on the basis of the "no covering ancestor" condition. When we try to continue, we notice that $|.|_{p\_c,(\{1,2\})}$ is not decreasing. But it has potential, and its tight decreasing refinement, $|.|_{p\_c,(\{1\},\{2\})}$, is the actual measure function used to construct the SLD-tree displayed above.

Theorem 8.2.27 Algorithm 8.2.25 terminates. The resulting SLD-tree τ is finite.

Proof We first note that, as long as changes in the measure functions do not occur, the computation rule R is developed in a way that renders algorithm 8.2.25 an instance of algorithm 6.5.5. The result then follows from theorem 6.5.8. It remains to be shown that the operation of replacing a measure function by its tight decreasing refinement occurs only a finite amount of times.
First note there is only a finite number of distinct measure functions considered at any moment, one for each recursive predicate in the input progam P. Next, the operation of taking a tight decreasing refinement involves splitting a non-empty set of argument positions in two disjunct non-empty sets of argument positions. The result follows from the fact that any predicate symbol has only a finite number of argument positions. □

Concluding, we have proposed, formalised and proven correct (terminating) an automatic unfolding method for definite logic programs. This method generalises our previous approach where increasing argument positions were *deleted* from the considered set. Instead of employing this straightforward, but drastic, technique, we now *shift* such argument positions to the right in the considered partition, where they determine a lexicographically less important component of the resulting weight. Unlike before, doing so allows *future consideration* of the corresponding arguments. This creates extra unfolding opportunities, as example 8.2.7 illustrates. We conjecture that partition based measures are strictly more powerful (i.e., other things equal, produce at least equally large SLD-trees) than those based on sets. (But see subsection 8.5.3.) Intuitively, this seems obvious. Indeed, set based unfolding is equivalent to considering only the leftmost component in partition based unfolding. We do not include a formal analysis of this issue, but instead proceed to exhibit an important simplification of the above algorithm.

8.2.4 Relaxing monotonicity

We will not present a complete formal complexity analysis of algorithm 8.2.25. However, one important observation should be made: it is possible to implement such weight based unfolding algorithms in such a way that their behaviour is ***linear in the size of the generated SLD-tree***, as long as no change in measure function is required. Indeed, as shown below, efficient labeling techniques ensure that for any literal, deciding whether it should be compared with the selected literal of an ancestor goal, and, if so, carrying out the comparison, can be done without any search.

However, the requirement that, upon refining a measure function, the whole SLD-tree generated thus far, should be checked to verify whether it remains subset-wise founded, destroys this linearity property. It was already pointed out in [125] that in practice this re-checking can safely be ignored, without damaging termination. Experiments in [83] confirmed this conjecture, but no formal proof was given. In the present subsection, we do present a brief formal development on this issue.

First, we generalise the notion of a well-founded measure.

Definition 8.2.28 Let $V, >_V$ be an s-poset. A *nearly-founded measure, f, on* $V, >_V$ is a function from $V, >_V$ to some well-founded set $W, >_W$ such that the following holds for only a *finite* number of pairs of elements v_i and $v_j \in V$:
$v_i >_V v_j$ and not$(f(v_i) >_W f(v_j))$.

Unlike a well-founded measure (definition 6.2.3), a nearly-founded measure does not have to be monotonic. But it is "almost" monotonic: There are only a finite number of offending pairs in the mapped set. We now introduce the notion of a subset-wise *nearly*-founded SLD-tree, and show that it still guarantees finiteness.

Definition 8.2.29 An SLD-tree τ is *subset-wise nearly-founded* if

1. There exists a finite number of sets, $C_0, \ldots, C_N$, such that $G_\tau = \bigcup_{i \leq N} C_i$.
2. For each $i = 1, \ldots, N$, there exists a nearly-founded measure
   $f_i : C_i, >_\tau \rightarrow W_i, >_i$.
3. For each $(G, k) \in C_0$ and each derivation D in τ containing (G, k):
   - either D is finite
   - or there exists a descendant (G', j) of (G, k) in D such that $(G', j) \in C_i$ for some $i > 0$.

Theorem 8.2.30 An SLD-tree τ is finite iff it is subset-wise nearly-founded.

The proof is a straightforward adaptation of the proof for theorem 6.2.8. We include it for completeness.

Proof

- If τ is finite, then it is subset-wise founded and therefore subset-wise nearly-founded.
- Conversely, suppose that τ is subset-wise nearly-founded and infinite. Then it contains an infinite derivation D. From the first condition in definition 8.2.29, it follows that there must be a C_i such that $C_i \cap D$ is infinite. In other words, there is some C_i containing an infinite sequence $(G_0, i_0) >_\tau (G_1, i_1) >_\tau \ldots$.
  - Suppose $i > 0$. Then $f_i((G_{j_1}, i_{j_1})) >_i f_i((G_{j_2}, i_{j_2})) >_i \ldots$ is an infinite sequence in $W_i, >_i$, contradicting the well-foundedness of $W_i, >_i$.
  - This leaves $i = 0$ as the only possibility. But then condition 3 of definition 8.2.29 implies that $D \cap \bigcup_{i>0} C_i$ is infinite, which again requires the existence of some $C_i, i > 0$ such that $C_i \cap D$ is infinite.

□

We modify algorithm 8.2.25.

Algorithm 8.2.31 Algorithm 8.2.25 remains almost completely unchanged. We just delete the fourth conjunct (∗) from the third condition enabling a literal selection. We do not reproduce the whole remaining algorithm description here.

Proposition 8.2.32 Algorithm 8.2.31 constructs a subset-wise nearly-founded SLD-tree.

Proof Most of the reasoning is identical to what has been presented in the subset-wise founded case. We only point out that measure functions are indeed nearly founded, since when refining a partition the resulting measure function:

1. might be non-monotonic on the *finite* subtree constructed thus far
2. will be (subset-wise) strictly decreasing on newly added nodes

Furthermore, as argued in the proof for theorem 8.2.27, replacing a measure function by its tight decreasing refinement occurs only a finite number of times. Together, these considerations imply that (subset-wise) non-monotonicity will hold for only a finite amount of pairs in the overall tree. □

We therefore obtain:

Theorem 8.2.33 Algorithm 8.2.31 terminates. The resulting SLD-tree τ is finite.

Having established its termination, we briefly return to the performance properties of algorithm 8.2.31. It is important to realise that checking the unfoldability of a certain literal can be done *without searching branches* in the SLD-tree, thus giving rise to the above mentioned linearity property. Indeed, there is just *a single test* involved: a weight comparison with the selected literal of the direct covering ancestor. Finding out whether such a direct covering ancestor exists, and if so, spotting it (in other words, maintaining and using the $>_{pr}$-relation), can be efficiently implemented through the use of lists of relevant node numbers. With an atom in a goal, one such list is associated. It registers per atom for every recursive predicate the closest $>_{pr}$-ancestor node where the selected literal contains that predicate symbol.

Example 8.2.34 Consider the following schematic program with three recursive predicates:

$p \leftarrow q, p$
$q \leftarrow r$
$r \leftarrow p$

An annotated SLD-derivation for $\leftarrow p$ is depicted in figure 8.2.
Since the program contains three recursive predicates, the length of the lists associated to the atoms in goals is 3. In each list, the first position corresponds to p, the second to q, the third to r. Elements of the list associated to an atom A, are the indices of the most recent goal node where a $>_{pr}$-ancestor literal of A with the corresponding predicate symbol was selected. A few concrete examples will clarify this somewhat complex description:

- In (2), the two descendant literals of the selected p-atom in (1) both get a 1 on the first list position. The other list positions of course remain "vacant".
- In (4), the left p-literal has $>_{pr}$-ancestors of every kind, the right one does not.
- In (5), the q-literal and the left p-literal both descend from the p-literal selected in (4). The right p-literal, however, does not.

In this way, direct covering ancestors can immediately be spotted. Consider goal (5) above:

- Selecting the q-literal requires a weight comparison with the selected literal of (2).
- The left p-literal must be compared with the selected one in (4).

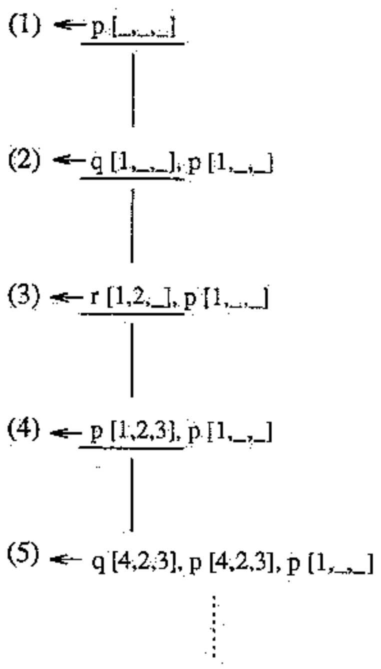

(1) ← p [_,_,_]
(2) ← q [1,_,_], p [1,_,_]
(3) ← r [1,2,_], p [1,_,_]
(4) ← p [1,2,3], p [1,_,_]
(5) ← q [4,2,3], p [4,2,3], p [1,_,_]

Figure 8.2: Direct covering ancestor annotation.

- Finally, the right p-literal, not descending from (4)'s selected literal, can be unfolded if its weight is less than the one associated to the original p goal, selected in (1).

It is clear that the above results do not depend on the use of partition based measures; they carry over straightforwardly to methods using set based measures, thus providing the formal justification for the version of algorithm 7.2.1 that was actually implemented and used in the context of [83].

At this point, it is interesting to include a brief comparison with some other criteria to control unfolding as they have been proposed in the literature. First, for all four possible tests mentioned in [14], apart from the fact that they are not safe (or not "complete", using the terminology proposed in [16] and [15]), another drawback, more immediately relevant to the discussion at hand, can be observed. Indeed, any such criterion unavoidably necessitates searching through the ancestor goals, thus destroying the above mentioned linearity property. Next, consider the increasingly sophisticated criteria, proposed in [152]. The most simple minded one just involves counting the number of occurrences of the various predicate symbols in selected literals throughout the derivation. This can obviously be implemented such that linear behaviour results. For the more advanced methods, this seems less straightforward. Finally, [134] proposes a method for partial deduction which involves not only unfoldings, but also the introduction of new predicates and foldings. A detailed discussion leads too far, but it can be noted that the method behaves linearly with respect to a parameter related to the size of the SLD-tree upon unfolding. Our analysis shows that similar results can be obtained through the use of weight based unfolding.

Theoretically, algorithm 8.2.31 might build larger trees than algorithm 8.2.25. Indeed, its third condition for literal unfoldability is more easily satisfied, since it does not contain (∗). However, switching to the tight decreasing refinement of the measure under consideration, and thus extending the given derivation, might be at the cost of diminished unfolding capabilities in other derivations. The latter possibility would be excluded through the use of separate partitions per chain of covering nodes. In practice, it turns out that all such technicalities are of very minor importance: arguments of most programs behave in a way sufficiently regular to mask these details. (Condition (∗) e.g. is usually satisfied when taking a tight decreasing refinement.) We therefore simply note that we have eliminated a possible source of considerable inefficiency from algorithm 8.2.25, while still guaranteeing termination. And we will do likewise for all weight based unfolding algorithms to be presented below.

8.3 Considering the Context

8.3.1 Introduction

Above, we introduced partitions of the set of, and priorities among argument positions of *a goal's selected literal*. We illustrated how this generalisation of section 7.2 brings extra unfolding power. In the present section, we will take yet another step towards increased power.

Indeed, we will return to our original objective, mentioned in the introduction to section 8.2: Taking into account arguments of several/all literals in the goal, not only the candidate for selection. In other words, the basic idea underlying our approach is kept intact: The weight of successive nodes with the same selected literal in a chain of covering goals should decrease. But, weights will no longer be assigned solely on the basis of the selected literal. A recasting of example 8.2.7 shows what we have in mind.

Example 8.3.1 Consider the following program:

$produce([\,], [\,]) \leftarrow$
$produce([X|Xs], [X|Ys]) \leftarrow produce(Xs, Ys)$

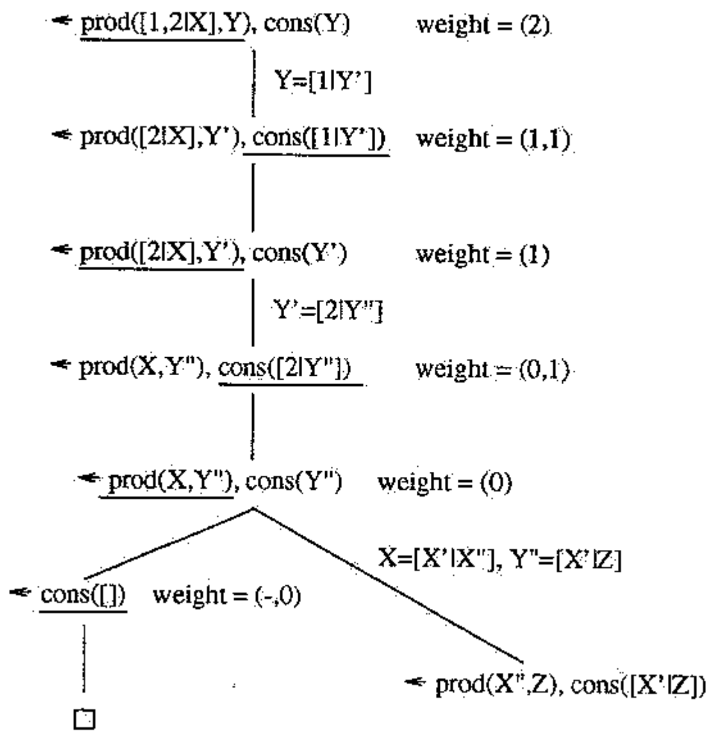

← prod([1,2|X],Y), cons(Y) weight = (2)
Y=[1|Y']
← prod([2|X],Y'), cons([1|Y']) weight = (1,1)
← prod([2|X],Y'), cons(Y') weight = (1)
Y'=[2|Y"]
← prod(X,Y"), cons([2|Y"]) weight = (0,1)
← prod(X,Y"), cons(Y") weight = (0)
X=[X'|X"], Y"=[X'|Z]
← cons([]) weight = (-,0)
← prod(X",Z), cons([X'|Z])

Figure 8.3: Unfolding when considering other literals.

$consume([]) \leftarrow$
$consume([X|Xs]) \leftarrow consume(Xs)$

and query:

$\leftarrow produce([1,2|X],Y), consume(Y)$

We apply algorithm 6.5.5 *imposing a coroutining computation rule* and choosing the pair $((R_0, R_1, R_2), (F_1, F_2))$ as follows:

- $R_0 = \emptyset$
- $R_1 = \{R(G,i)$ containing *produce*$\}$
- $R_2 = \{R(G,i)$ containing *consume*$\}$
- $F_1 = |\cdot|_{produce,(\{1,2\})}$
- $F_2 = (|\cdot|_{produce,\{1,2\}}, |\cdot|_{consume,\{1\}})$

The resulting SLD-tree is depicted in figure 8.3. Selected literals are underlined and nodes are annotated with their weight according to F_1 or F_2.
It can be noted that the second *consume*-unfolding is not allowed using a measure function just taking the *consume*-argument into account.

In this extended context, a major difficulty is the dynamic nature of literal occurrences in goals. This is illustrated by the left branch of the SLD-tree in figure 8.3. As long as the goals to be measured basically look the same, containing one *produce* and one *consume* literal, a measure function like F_2 indeed makes sense. But this is not the case for the last non-empty goal in the above left branch, where the *produce* call has "disappeared" (indicated by a "-" in the associated weight couple). Worse even, in general more than one literal with the same predicate symbol might appear in one goal, and it is by no means immediately clear what should be compared with what in such circumstances.

Our work in this section will be presented as follows. First, we exhibit in detail a rather straightforward solution to the problems indicated above, and show that the resulting algorithms possess sufficient power to handle typical producer-consumer coroutining applications. Secondly, we demonstrate how a more satisfactory treatment of an annoying issue in "standard" unfolding is also enabled. Finally, we briefly describe a more sophisticated approach.

8.3.2 Handling coroutining

We present two algorithms suitable for unfolding under coroutining-like computation rules. The first one, algorithm 8.3.11, is simply a properly specialised version of algorithm 6.5.5. Algorithm 8.3.19 on the other hand, is a semi-automatic generalisation of algorithm 8.2.31. Along the way, we need to generalise a number of concepts introduced above.

Tuning the basic algorithm

First, we want to allow argument positions of non-selected literals among those determining a goal's weight. On the other hand, we will stick to comparing a goal with its direct covering ancestor. So, the selected literal will continue to play a major role, as actually seems quite natural in the context of unfolding logic programs. (See also [15].) We set out with the following definition.

Definition 8.3.2 Let P be a program containing a (recursive) predicate symbol p. A *context considering ordered k-partition (cco-k-partition) associated to p in P* is a k-tuple $O = (\{i_{11}, \ldots, i_{1j}\}, \ldots, \{i_{k1}, \ldots, i_{kl}\})$ satisfying the following conditions:

1. O has *two kinds of components*. Some, which we will call *p-components*, consist of argument positions of p. The others contain argument positions

of recursive predicate symbols in P, and will occasionally be named *non-p-components*.

2. The p-components together form an *ordered partition of p's set of argument positions*.

3. Argument positions of recursive literals in P (including p) can appear in at most one non-p-component.

Example 8.3.3 For the program in example 8.3.1, the following are some cco-2-partitions associated to *consume*:

- $(\{1_{produce}, 2_{produce}\}, \{1\})$
- $(\{1_{produce}\}, \{1\})$
- $(\{1\}, \{2_{produce}, 1_{consume}\})$
- $(\{1\}, \{1_{consume}\})$

Some further examples, this time of cco-partitions associated to *produce* are:

- $(\{1, 2\})$
- $(\{1\}, \{1_{consume}\}, \{2\})$

While the next two tuples are *not* legitimate *produce* associated cco-partitions:

- $(\{1\}, \{1_{consume}\})$
- $(\{1_{consume}\}, \{1, 2\}, \{1_{consume}\})$

Note that definition 8.3.2 is a conservative extension of the notion "associated ordered k-partition" introduced above. Indeed, we extend the earlier partition of a predicate's set of argument positions with "components" containing some argument positions of other relevant predicates. Although possible, we have decided against requiring the presence of *all* argument positions of *all* (recursive) predicates, since this would often introduce a great amount of irrelevant information in a goal's measure function. The same consideration is the motivation for only considering *recursive* predicate symbols. Finally, note that we anticipate a distinction between p in the selected literal, and the same predicate symbol occurring in non-selected literals: Argument positions of p appearing in non-p-components refer to such occurrences.

Before we can actually introduce the generalised measure functions, we need to cater for "absent" arguments.

Definition 8.3.4 We define $\mathbb{N}_b = \mathbb{N} \cup \{\perp\}$. We extend the usual order relation, $>$, on $\mathbb{N}$ with $0 > \perp$. We extend the usual addition, $+$, on $\mathbb{N}$ with $x + \perp = \perp + x = \perp$.

$\perp$ is an extra "bottom" element. It will serve as the image of a measure function on an absent argument. Notice that we have overloaded the $>$, $+$ and $=$ symbols, and that the result of an addition involving $\perp$ is $\perp$. Of course, $\mathbb{N}_b, >$ is a well-founded set, and so is ${\mathbb{N}_b}^k, \succ_k$ for any $k > 1$.

We need an operation on finite subsets of $\mathbb{N}_b$, delivering their maximum element.

Definition 8.3.5 We define $max : \mathcal{P}(\mathbb{N}_b) \to \mathbb{N}_b$ as follows:

- $max(\emptyset) = \perp$
- $max(\{v_1, \ldots, v_n\}) = v_i$ such that $\forall 1 \leq j \leq n : j \neq i \Rightarrow v_i > v_j$

Note that, since $>$ is a *total* order on $\mathbb{N}_b$, max is well-defined on finite sets and its result uniquely determined. We continue:

Definition 8.3.6 Let G be a goal consisting of a number of atoms, one of which has been selected for unfolding, p an n-ary predicate symbol and $1 \leq i \leq n$. Then we define:

$$M(G, p, i) = max(\{|t_i| \mid t_i \text{ is the term occurring as } i\text{th argument of some } \textit{non-selected} \text{ atom } p(t_1, \ldots, t_n) \text{ in } G\})$$

Finally, we can formulate the following generalisation of definition 8.2.2:

Definition 8.3.7 Let P be a program and p an n-ary (recursive) predicate symbol appearing in P. Let O be a cco-k-partition associated to p in P. Then we define $\|\cdot\|_{P,O} : \{G \mid G$ is a goal in the language underlying P whose selected literal has predicate symbol $p\} \to {\mathbb{N}_b}^k, G \mapsto (v_1, \ldots, v_k)$ as follows:

- If $O[r] = \{i_{r1}, \ldots, i_{rs}\}$ is a *p-component* and $p(t_1, \ldots, t_n)$ is G's selected literal, then $v_r = |t_{i_{r1}}| + \cdots + |t_{i_{rs}}|$.
- If $O[r] = \{i_{r1,p_{r1}}, \ldots, i_{rj,p_{rj}}\}$ where the p_{rl}-subscripts denote recursive predicate symbols in P, then $v_r = M(G, p_{r1}, i_{r1}) + \cdots + M(G, p_{rj}, i_{rj})$.

Example 8.3.8 The weight annotations in figure 8.3 correspond to:

- $O_{produce} = (\{1, 2\})$
- $O_{consume} = (\{1_{produce}\}, \{1\})$
  (taking $O_{consume}[1] = \{1_{produce}, 2_{produce}\}$ gives the same results)

Example 8.3.9 Suppose

- p, q and r are recursive predicates
- $G = \leftarrow q(f(a)), p(f(f(a)), f(a)), p(f(a), a), q(f(f(a)))$
- in G, the first p-atom is selected
- $O = (\{1_q, 2_p\}, \{1, 2\}, \{1_p, 1_r\})$

Then $||G||_{p,O} = (2 + 0, 2 + 1, 1 + \perp) = (2, 3, \perp)$

We have the following equivalent of proposition 8.2.6:

Proposition 8.3.10 Let P be a program and p a predicate symbol appearing in P. Let O be a cco-k-partition associated to p in P. Let τ be an SLD-tree for P and S_τ a subset of G_τ such that all goals in S_τ have a selected literal with predicate p. Then $||.||_{p,O}$ is a well-founded measure on $S_\tau, >_\tau$ iff it is monotonic.

Proof The proposition follows immediately from definitions 6.2.3 and 8.3.7 and the well-foundedness of ${I\!N_b}^k, \succ_k$. □

Suppose now that a (coroutining) computation rule R is fixed, to be used while unfolding a goal with respect to a program P. Moreover, also given are cco-partitions O_p, one for every recursive predicate p in P. Then we can introduce the following specialised version of algorithm 6.5.5:

Algorithm 8.3.11 As before, we associate one R_k to every recursive predicate. If p_k is a recursive predicate with associated class R_k, then $F_k = ||.||_{p_k, O_{p_k}}$. Adapting the code of the algorithm is straightforward; we do not reproduce it here.

Theorem 8.3.12 Algorithm 8.3.11 terminates. The resulting SLD-tree τ is finite.

Proof $((R_0, R_1, \ldots), (F_1, \ldots))$ is chosen such that it indeed determines a hierarchical prefounding of the complete SLD-tree τ_0, resulting for the program P and a given goal under the computation rule R. The result follows immediately from theorem 6.5.8. □

Example 8.3.13 We have formalised the intuitions underlying example 8.3.1. Indeed, instead of the ad hoc couple of "measure" functions used for F_2 there, we can now take $F_2 = ||.||_{consume,(\{1_{produce}\},\{1\})}$ (and $F_1 = ||.||_{produce,(\{1,2\})}$). And also the disappearing *produce* call can now be dealt with properly within our framework. An application of algorithm 8.3.11 using these ingredients, reproduces the SLD-tree depicted in figure 8.3. Goal weights result as indicated, except $(\_, 0)$ which becomes $(\perp, 0)$.

Example 8.3.14 As a concrete example of coroutining behaviour, we consider the well-known permutation sort program:

(1) $sort(X, Y) \leftarrow perm(X, Y), ord(Y)$
(2) $perm([], []) \leftarrow$
(3) $perm([X|Xs], [Y|Ys]) \leftarrow del(Y, [X|Xs], Z), perm(Z, Ys)$
(4) $del(X, [X|Xs], Xs) \leftarrow$
(5) $del(X, [Y|Ys], [Y|Z]) \leftarrow del(X, Ys, Z)$
(6) $ord([]) \leftarrow$
(7) $ord([X]) \leftarrow$
(8) $ord([X, Y|Z]) \leftarrow X \leq Y, ord([Y|Z])$

We do not include explicit clauses for $\leq$, and simply assume that a $X \leq Y$ call can be evaluated when both arguments are ground. We want to build an SLD-tree for the query:

$\leftarrow sort([5, 2|X], Y)$

using the following coroutining computation rule (the actions are listed according to priority, "possible" refers to associated weight behaviour):

1. Evaluate $X \leq Y$ if both arguments are ground.
2. Unfold a *del* call if possible.
3. Unfold an *ord* call if possible, provided its argument is not an uninstantiated variable.
4. Unfold a *perm* call if possible.

We will use the following R_i classes:

- $R_0 = \{R(G, i) \text{ containing } \leq\} \cup \{R(G, i) \text{ containing } sort\}$
- $R_1 = \{R(G, i) \text{ containing } del\}$
- $R_2 = \{R(G, i) \text{ containing } ord\}$
- $R_3 = \{R(G, i) \text{ containing } perm\}$

and as associated cco-partitions:

- $O_{del} = (\{1, 2, 3\})$
- $O_{ord} = (\{1_{perm}\}, \{1\})$
- $O_{perm} = (\{1\}, \{2\})$

Rather than reproducing the complete SLD-tree resulting from an application of algorithm 8.3.11, we select one SLD-derivation and depict it in figure 8.4. Each node is annotated with a label showing its identifier, the one of its direct covering ancestor and its weight under the relevant measure function. Links are enhanced with a number indicating the clause used in that particular unfolding. Selected literals are underlined.

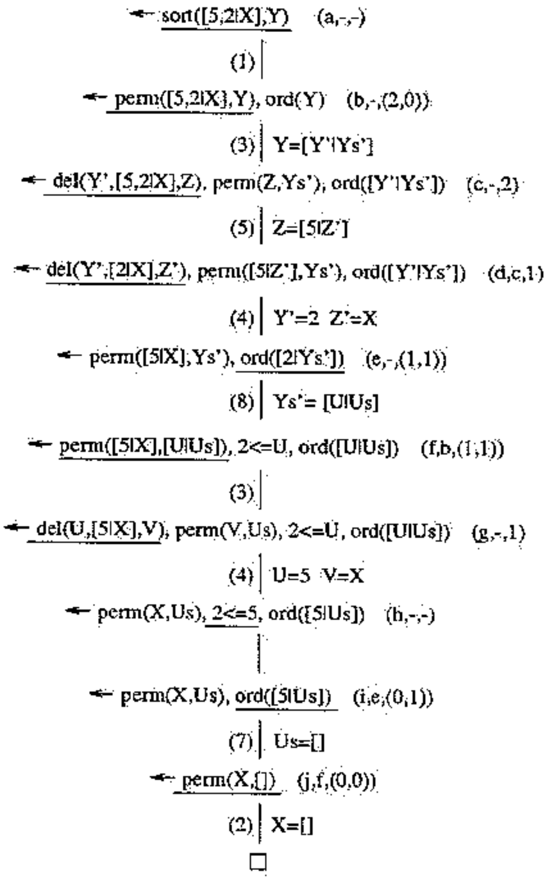

← sort([5,2|X],Y) (a,-,-)
(1)
← perm([5,2|X],Y), ord(Y) (b,-,(2,0))
(3) Y=[Y'|Ys']
← del(Y',[5,2|X],Z), perm(Z,Ys'), ord([Y'|Ys']) (c,-,2)
(5) Z=[5|Z']
← del(Y',[2|X],Z'), perm([5|Z'],Ys'), ord([Y'|Ys']) (d,c,1)
(4) Y'=2 Z'=X
← perm([5|X],Ys'), ord([2|Ys']) (e,-,(1,1))
(8) Ys'= [U|Us]
← perm([5|X],[U|Us]), 2<=U, ord([U|Us]) (f,b,(1,1))
(3)
← del(U,[5|X],V), perm(V,Us), 2<=U, ord([U|Us]) (g,-,1)
(4) U=5 V=X
← perm(X,Us), 2<=5, ord([5|Us]) (h,-,-)
← perm(X,Us), ord([5|Us]) (i,e,(0,1))
(7) Us=[]
← perm(X,[]) (j,f,(0,0))
(2) X=[]
□

Figure 8.4: A coroutining SLD-derivation.

Especially noteworthy is the unfolding carried out between nodes (i) and (j). It would be prohibited when using a measure function based solely on the *ord* literal itself.

Automatically refining measure functions

Having established that algorithm 8.3.11 can cope with coroutining when apt cco-partitions are provided, we want to take a further step. Indeed, it seems appropriate to relieve the user from the burden of choosing the right partitions. In other words, we want to obtain an algorithm similar to algorithm 8.2.31 (or algorithm 8.2.25), with the difference that some computation rule preferences are specified by the user, as was done in example 8.3.14 above.

First, we introduce proper variants of the definitions in subsection 8.2.3.

Definition 8.3.15 Let P be a program containing an n-ary predicate symbol p. Let O be a cco-k-partition associated to p in P. Let G_1 and G_2 be two goals in the language underlying P. Let the selected literal of G_1 and G_2 be $p(t_1, \ldots, t_n)$ and $p(s_1, \ldots, s_n)$ respectively. Finally, let $||G_1||_{p,O} = (v_1, \ldots, v_k)$ and $||G_2||_{p,O} = (w_1, \ldots, w_k)$. Then we define the following:

- An *argument position i in a p-component* of O is
  - (G_1, G_2)-*decreasing* iff $|t_i| > |s_i|$
  - (G_1, G_2)-*increasing* iff $|s_i| > |t_i|$
  - (G_1, G_2)-*stable* iff $|t_i| = |s_i|$
- An *argument position i_q* is
  - (G_1, G_2)-*context-decreasing* iff $M(G_1, q, i) > M(G_2, q, i)$
  - (G_1, G_2)-*context-increasing* iff $M(G_2, q, i) > M(G_1, q, i)$
  - (G_1, G_2)-*context-stable* iff $M(G_1, q, i) = M(G_2, q, i)$

  When we consider arguments as elements of a non-p-component of a cco-partition, we will occasionally describe their behaviour without the explicit "context" addition. In such cases, it is clear that we refer to M- and not $|.|$-values.
- The *i-th component of O* is
  - (G_1, G_2)-*decreasing* iff $v_i > w_i$
  - (G_1, G_2)-*increasing* iff $w_i > v_i$
  - (G_1, G_2)-*stable* iff $v_i = w_i$
- $||.||_{p,O}$ is
  - (G_1, G_2)-*decreasing* iff $||G_1||_{p,O} \succ_k ||G_2||_{p,O}$
  - (G_1, G_2)-*increasing* iff $||G_2||_{p,O} \succ_k ||G_1||_{p,O}$

  - (G_1, G_2)-*stable* iff $||G_1||_{p,O} = ||G_2||_{p,O}$

- $O[i]$ is O's *leftmost* (G_1, G_2)-*increasing component* if
  1. it is (G_1, G_2)-increasing
  2. there is no $1 \leq j < i$ such that $O[j]$ is (G_1, G_2)-increasing

- We call a component $O[i]$ (G_1, G_2)-*sensitive* if the following two conditions are satisfied:
  1. $O[i]$ contains at least one (G_1, G_2)-decreasing argument position.
  2. If $||.||_{p,O}$ is (G_1, G_2)-increasing and $O[l]$ is O's leftmost increasing component then $i \leq l$.

Definition 8.3.16 Let P be a program containing an n-ary predicate symbol p. Let O be a cco-k-partition associated to p in P. Let G_1 and G_2 be two goals in the language underlying P whose selected literal contains p. Suppose $||.||_{p,O}$ is *not* (G_1, G_2)-decreasing. Then we say that $||.||_{p,O}$ has

- *internal* (G_1, G_2)-*potential* iff O has at least one (G_1, G_2)-sensitive component.

- *external* (G_1, G_2)-*potential* iff there is at least one argument position i_q of some recursive predicate q in P for which the following two conditions hold:
  1. i_q is (G_1, G_2)-context-decreasing
  2. For every non-p-component $O[j]$ of O: $i_q \notin O[j]$
     (We will call i_q *context-absent in* O.)

We will occasionally say that a measure function $||.||_{p,O}$ has (G_1, G_2)-*potential* if either of the above two conditions is satisfied.

Just like before, we are interested in refining obsolete (i.e. non-decreasing) measure functions. The notion of *internal* potential with respect to a pair of goals is an immediate generalisation of the *potential* with respect to a pair of atoms as it was introduced in definition 8.2.16. New is the capability of *adding* argument positions of non-selected literals to the "partition". A measure function with *external* potential can be refined into a decreasing one, doing just that. At this point, we will not include explicit generalisations of definitions 8.2.19 and 8.2.21. Instead, we immediately define an extended notion of *tight decreasing refinement*.

Definition 8.3.17 Let P be a program containing an n-ary predicate symbol p. Let O be a cco-k-partition associated to p in P. Let G_1 and G_2 be two goals in the language underlying P whose selected literal contains p. Let $||.||_{p,O}$ have (G_1, G_2)-potential. Then $||.||_{p,O'}$ is the *tight* (G_1, G_2)-*decreasing refinement* of $||.||_{p,O}$ if O' is defined as follows:

- If $||.||_{p,O}$ has *internal* (G_1, G_2)-potential and $O[l]$ is O's rightmost (G_1, G_2)-sensitive component, then:
  - $\forall 1 \leq j < l : O'[j] = O[j]$
  - $\forall l < j \leq k : O'[j+1] = O[j]$
  - If $O[l]$ is a p-component, then:
    - $O'[l] = \{i \in O[l] | i \text{ is } (G_1, G_2)\text{-decreasing or } (G_1, G_2)\text{-stable}\}$
    - $O'[l+1] = \{i \in O[l] | i \text{ is } (G_1, G_2)\text{-increasing}\}$
  - Else:
    - $O'[l] = \{i_q \in O[l] | i_q \text{ is } (G_1, G_2)\text{-context-decreasing or } (G_1, G_2)\text{-context-stable}\} \setminus \{i_q | M(G_1, q, i) = \perp\}$
    - $O'[l+1] = \{i_q \in O[l] | i_q \text{ is } (G_1, G_2)\text{-context-increasing}\} \cup \{i_q | M(G_1, q, i) = \perp\}$
- *Else* if $||.||_{p,O}$ has *external* (G_1, G_2)-potential, then:
  - $O'[1] = \{i | i \text{ is } (G_1, G_2)\text{-context-decreasing and context-absent in } O\}$
  - $\forall 1 \leq j \leq k : O'[j+1] = O[j]$

Again, the first part of the above definition generalises definition 8.2.22. Notice the special treatment for $\perp$-valued arguments. This is necessary since $\perp$ is an absorbing element for $+$ in $I\!N_b$. The second part is extra. It can be noted that our particular choice for what might be termed a *tight* (G_1, G_2)*-decreasing external refinement* is in fact not so tight, in the sense that a reformulation of the second property in proposition 8.2.24 does not hold. Several "more tight" variants can be imagined, but we do not believe they would significantly improve the behaviour of algorithm 8.3.19 on a meaningful class of programs. In consequence, we have preferred intuitive appeal and simplicity of formulation. We do have the following property:

Proposition 8.3.18 Let P, p, O, G_1, G_2 and O' be as in definition 8.3.17. Then $||.||_{p,O'}$ is (G_1, G_2)-decreasing.

Proof

- The result is immediate when O' is an *external* refinement.
- The proof for the *internal* refinement case is similar to the proof for point (1) in proposition 8.2.24. We just need the extra observation that any (G_1, G_2)-context-decreasing argument i_q in $O[l]$ (in case it is a non-p-component) certainly has $M(G_1, q, i) \neq \perp$, thus guaranteeing that $O'[l]$ is non-empty and (G_1, G_2)-decreasing.

□

We can now formulate a first algorithm for automatic maximal sensible unfolding under some computation rule preferences. The algorithm assumes that a computation rule R is partially specified, in the sense that for any goal a (partial) priority order among literals, candidate for selection, is known (see e.g. example 8.3.14 above).

Algorithm 8.3.19

```
Input
  a definite program $P$
  a definite goal $\leftarrow A$
Output
  a finite SLD-tree $\tau$ for $P \cup \{\leftarrow A\}$
Initialisation
  $\tau := \{(\leftarrow A, 1)\}$ {* an SLD-tree with a single derivation *}
  $Pr := \emptyset$ {* in $Pr$, the $>_{pr}$-relation will be constructed *}
  $Terminated := \emptyset$
  $Failed := \emptyset$
  For each recursive predicate $p/n$ in $P$: $O_p := (\{1, \ldots, n\})$
  {* We set out with cco-1-partitions
  grouping all $p$'s argument positions in a single p-component
  and without any non-$p$-components *}
While there exists a derivation $D$ in $\tau$ such that $D \notin Terminated$ do
  Let $(G, i)$ name the leaf of $D$
  If $(G, i) = (\square, i)$
  Then {* $(G, i)$ is a success node *}
    add $D$ to $Terminated$
  Else
    {* First, we try to determine $R(G, i)$ *}
    Select the leftmost R-preferred atom $p(t_1, \ldots, t_n)$ in $G$ such that
    one of the following (mutually exclusive) conditions is satisfied:
      • $(G, i)$ has no direct covering ancestor
      • $(G', j)$ is the direct covering ancestor of $(G, i)$ and
        $||\cdot||_{p,O_p}$ is $(G', G)$-decreasing
      • $(G', j)$ is the direct covering ancestor of $(G, i)$ and
        $||\cdot||_{p,O_p}$ has $(G', G)$-potential and
        $||\cdot||_{p,O_p^t}$ is its tight $(G', G)$-decreasing refinement
    If such an atom $p(t_1, \ldots, t_n)$ cannot be found
    Then {* $(G, i)$ becomes a dangling leaf *}
      Add $D$ to $Terminated$
    Else
```

```
        $R(G,i) := p(t_1,\ldots,t_n)$
        If $R(G,i)$ was selected on the basis of the third condition above
        Then {* Register the new cco-partition *}
          $O_p := O'_p$
        Let $Derive(G,i)$ name the set of all possible derivation steps
        If $Derive(G,i) = \emptyset$
        Then {* $(G,i)$ is a failure node *}
          Add $D$ to $Terminated$ and $Failed$
        Else
          {* Extend the derivation *}
          Expand $D$ in $\tau$ with the elements of $Derive(G,i)$
          Let $Descend(R(G,i),i)$ name the set of all pairs $((R(G,i),i),(B\theta,j))$,
          where
            — $B$ is an atom in the body of a clause
              applied in an element of $Derive(G,i)$
            — $\theta$ is the corresponding m.g.u.
            — $j$ is the number of the corresponding descendant of $(G,i)$
          Apply $\theta$ to the affected elements of $P\tau$
          Add the elements of $Descend(R(G,i),i)$ to $P\tau$
Endwhile
```

Theorem 8.3.20 Algorithm 8.3.19 terminates. The resulting SLD-tree τ is finite.

Proof The only substantially new element is the facility to externally refine a cco-partition. It can be seen that this does not jeopardise termination, since:

- A program P contains only a finite amount of recursive predicates, all of finite arity.
- Any argument position of any recursive predicate can occur in at most one non selected literal component of a cco-partition, and once included in the partition, is never removed.

□

Example 8.3.21 Applying algorithm 8.3.19 to the programs, algorithms and computation rules discussed in examples 8.3.1, 8.3.13 and 8.3.14 produces the results presented there.

8.3.3 Back propagation of instantiations

At this point, it is interesting to have a look at the behaviour of algorithm 8.3.19 when *no* computation rule preferences are in fact specified. In that case, the search for an unfoldable literal reduces to the basic "take the leftmost whose (possibly refined) measure function allows it" technique, used above. It turns out that, thanks to its context considering capabilities, algorithm 8.3.19 can often improve upon an annoying deficiency of algorithms solely focusing on selected literals. Consider the following prototypical example.

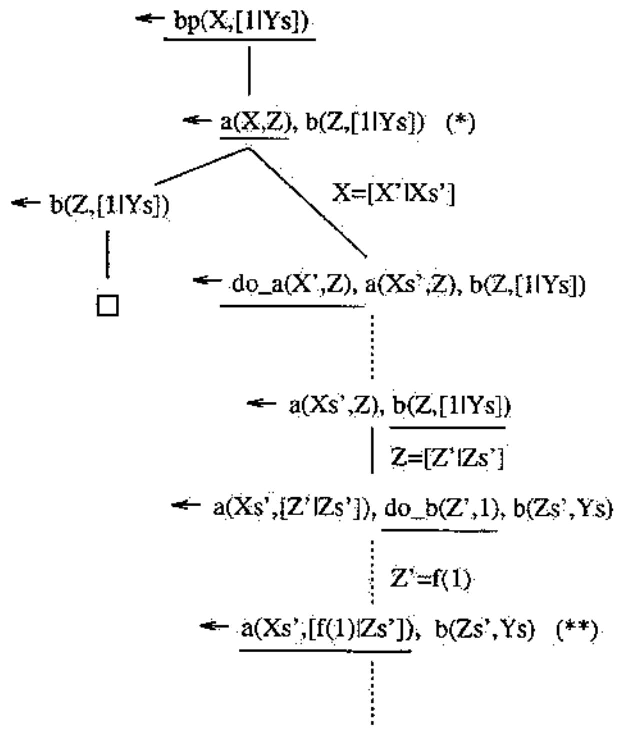

← bp(X,[1|Ys])
← a(X,Z), b(Z,[1|Ys]) (*)
← b(Z,[1|Ys])
X=[X'|Xs']
← do_a(X',Z), a(Xs',Z), b(Z,[1|Ys])
← a(Xs',Z), b(Z,[1|Ys])
Z=[Z'|Zs']
← a(Xs',[Z'|Zs']), do_b(Z',1), b(Zs',Ys)
Z'=f(1)
← a(Xs',[f(1)|Zs']), b(Zs',Ys) (**)

Figure 8.5: Handling back propagation.

Example 8.3.22

$bp(X,Y) \leftarrow a(X,Z), b(Z,Y)$
$a([], Y) \leftarrow$
$a([X|Xs], Y) \leftarrow do\_a(X,Y), a(Xs,Y)$
$b([], []) \leftarrow$
$b([X|Xs], [Y|Ys]) \leftarrow do\_b(X,Y), b(Xs,Ys)$

(The definitions for *do_a* and *do_b* do not matter here. Just suppose they can be fully resolved at unfolding time.)

We are interested in unfolding the following query:

$\leftarrow bp(X, [1|Ys])$

Part of the SLD-tree generated by algorithm 8.3.19 is depicted in figure 8.5. The unfolding carried out in node $(**)$ is particularly interesting. At that point, $||\cdot||_{a,(\{1,2\})}$ is increasing with respect to the pair formed by $(**)$ and its direct covering ancestor $(*)$. Moreover, it has no internal potential. But $||\cdot||_{a,(\{2_b\},\{1,2\})}$ is its *external* $(*, **)$-decreasing refinement. It maps $(*)$ into $(1,0)$ and $(**)$ into $(0,1)$, and therefore allows the unfolding of the a-goal. No measure function solely based on a-arguments can do likewise.

The above example illustrates a general phenomenon: In unfolding, back propagation of information (i.e. in the reverse direction as the "scan" for unfoldable literals) is very similar to coroutining. (Observe that a further instantiated second argument in the bp starting goal would lead to further little by little passing of information chunks from the b to the a goal.) It creates similar problems, and requires similar solutions. Therefore, the work in this section also improves on earlier algorithms with respect to this previously rather disturbing issue. We include a final example, illustrating that the mere capacity to register the "disappearance" of literals from the context has already beneficial effects.

Example 8.3.23 Consider the well-known "naive reverse" program:

$rev([], []) \leftarrow$
$rev([X|Xs], Y) \leftarrow rev(Xs, Z), app(Z, [X], Y)$
$app([], X, X) \leftarrow$
$app([X|Xs], Y, [X|Zs]) \leftarrow app(Xs, Y, Zs)$

Together with the following query:

$\leftarrow rev([1|Xs], Y)$

From an SLD-tree produced by algorithm 7.2.1, we synthesise:

$rev([1], [1]) \leftarrow$
$rev([1, X|Xs], [X, 1]) \leftarrow rev(Xs, [])$
$rev([1, X|Xs], [Y, 1]) \leftarrow rev(Xs, [Y|Ys]), app(Ys, [X], [])$
$rev([1, X|Xs], [Y, Z|Zs]) \leftarrow$
$\qquad rev(Xs, [Y|Ys]), app(Ys, [X], [Z|Zs']), app(Zs', [1], Zs)$

Using algorithm 8.3.19 for unfolding, applying a simple "select the leftmost suitable literal" computation rule, we obtain the more sensible:

$rev([1], [1]) \leftarrow$
$rev([1, X], [X, 1]) \leftarrow$
$rev([1, X|Xs], [Y, Z|Zs]) \leftarrow$
$\qquad rev(Xs, [Y|Ys]), app(Ys, [X], [Z|Zs']), app(Zs', [1], Zs)$

The resulting cco-partitions are:

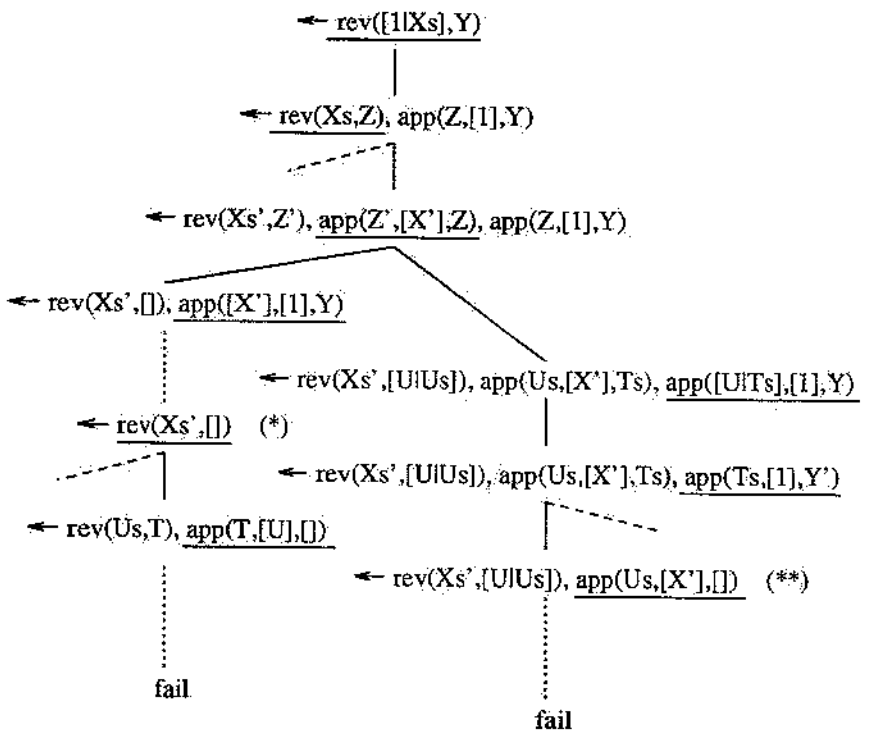

← rev([1|Xs],Y)
← rev(Xs,Z), app(Z,[1],Y)
← rev(Xs',Z'), app(Z',[X'],Z), app(Z,[1],Y)
← rev(Xs',[]), app([X'],[1],Y)
← rev(Xs',[U|Us]), app(Us,[X'],Ts), app([U|Ts],[1],Y)
← rev(Xs',[]) (*)
← rev(Xs',[U|Us]), app(Us,[X'],Ts), app(Ts,[1],Y')
← rev(Us,T), app(T,[U],[])
← rev(Xs',[U|Us]), app(Us,[X'],[]) (**)
fail
fail

Figure 8.6: Part of the SLD-tree generated for example 8.3.23.

$O_{rev} = (\{1_{app}, 2_{app}, 3_{app}\}, \{1, 2\})$
$O_{app} = (\{1_{app}, 2_{app}, 3_{app}\}, \{1, 2, 3\})$

We do not include the complete generated SLD-tree, but an interesting portion can be found in figure 8.6. Non-depicted branches are indicated by a dashed link at their origin. Particularly noteworthy are the unfoldings at node (*) and node (**): They cause the differences between the first and the second set of resulting clauses above.

Summarising, we can state that we generalised previous concrete methods for weight-based finite unfolding, by allowing (also) the consideration of non-selected literals in a goal. We have presented two algorithms incorporating this feature, the second one automatically focusing on sensible measure functions. We have shown how they handle coroutining and the related issue of instantiation back propagation.

Numerous variants of and/or enhancements to algorithm 8.3.19 can be considered.

- Argument positions of non-recursive context literals can be included in cco-partitions.
- Priority can be given to unfolding literals which cause little branching in the tree.
- Using separate cco-partitions for different chains of covering nodes seems reasonable, and will probably have a more profound influence than is the case in approaches solely based on measuring selected literals.
- It seems possible that, in larger applications, some extra limitation should be imposed, restricting unfolding literals on account of a contextual weight decrease to those who have actually been "influenced" by those context unfoldings. It is however not immediately clear how to pin down this notion of "influenced". Demanding a higher weight, or simply not being a variant of the call selected in the direct covering ancestor, are too restrictive conditions, as an inspection of the coroutining examples 8.3.1 and 8.3.14 reveals. A comparison with the state of the corresponding literal in some intermediate goal node seems more appropriate.
- A further step towards more sophistication can involve an offline analysis phase to derive useful supporting information about (mutual) influences among argument positions. This might help to resolve the problem indicated above. It might also lead to more "sensible" tight decreasing refinements. Reconsider example 8.3.23 above. Rather than the resulting O_{rev}, the partition $(\{1\}, \{1_{app}\}, \{2\})$ (or $(\{1\}, \{1_{app}, 2_{app}, 3_{app}\}, \{2\})$) might be considered as the natural one to be used for the *rev* predicate. In fact, with these partitions, the whole tree is subset-wise founded, while this is not the case with O_{rev}. The development of relevant analysis techniques and an assessment of their value are challenging topics for future research.

Rather than delving into the above sketched issues, in the next subsection, we will address another topic glossed over by the development so far. Indeed, both algorithm 8.3.11 and 8.3.19, relying on definition 8.3.7, are incapable of distinguishing between several non-selected literals with the same predicate symbol in one goal. This is fine for most producer-consumer coroutining applications, where the basic structure is typically of the simple linearly recursive type as exhibited by the programs in examples 8.3.1 and 8.3.14. But, in general, they might fall short of dealing properly with instantiation back propagation in logic programs exhibiting a more complex structure.

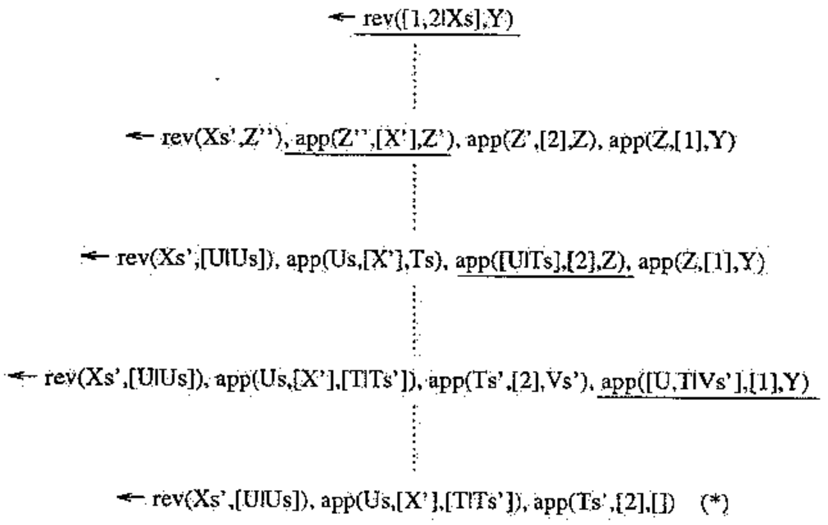
8.3.4 Focusing on ancestor literals

We restrict ourselves to a somewhat informal discussion of the issue and its most striking aspects, omitting a complete technical development.

Example 8.3.24 Let us unfold the naive reverse program with respect to the following query:

$\leftarrow \mathit{rev}([1, 2|Xs], Y)$

using the cco-partitions produced in example 8.3.23. Some fragments from an interesting SLD-derivation are shown in figure 8.7.



← rev([1,2|Xs],Y)
← rev(Xs',Z''), app(Z'',[X'],Z'), app(Z',[2],Z), app(Z,[1],Y)
← rev(Xs',[U|Us]), app(Us,[X'],Ts), app([U|Ts],[2],Z), app(Z,[1],Y)
← rev(Xs',[U|Us]), app(Us,[X'],[T|Ts']), app(Ts',[2],Vs'), app([U,T|Vs'],[1],Y)
← rev(Xs',[U|Us]), app(Us,[X'],[T|Ts']), app(Ts',[2],[]) (*)

Figure 8.7: A case for yet more powerful measures.

In (∗), no literal can be selected for unfolding. Our measure functions are too coarse; they do not register the "disappearance" of the last *app* literal.

A solution for the problem illustrated in example 8.3.24 is obvious: Introduce more fine grained comparisons among context literals. The second part of definition 8.3.7 does not distinguish between literals in different ancestor-descendant chains. Redefining cco-partitions and their associated measure functions in this sense is possible: compare (maximum) argument weights of context literals in the descendant goal with the argument weights of the corresponding literal in the ancestor goal, situated on the same branch of the associated proof tree. (Of course, often the comparison will simply be between a literal and a possibly less instantiated version of the same literal. This is the case when the considered literal itself has not been the subject of unfoldings carried out between the two inspected goals.) It can be noted that such an approach would indeed allow further unfoldings in example 8.3.24.

Rather than going through the complete technical development, rephrasing definitions and results in subsection 8.3.2, we conclude this section by pointing out an intriguing additional difficulty emerging in this context. Consider the following schematic example.

Example 8.3.25 Suppose we unfold the program:

$a(X) \leftarrow a(Y), b(X, Y)$
$b(X, X) \leftarrow$
some other (recursive) clauses for b

and the query:

$\leftarrow a([1|X])$

using more refined measure functions of the kind sketched above. Part of a possible SLD-derivation is depicted in figure 8.8.

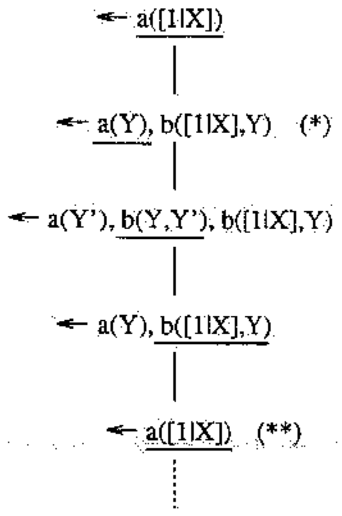


Figure 8.8: An infinite SLD-derivation.

The unfolding in (**) is allowed on account of b's disappearance, compared with (*). In this way, continuously appearing and disappearing "fresh" b literals create an infinite series of tight decreasing refinements. As a result, *unfolding does not terminate.*

Example 8.3.25 shows that an unrestricted application of fine grained context considering unfolding techniques may lead to non-termination. Imposing a *bound on the number of components* allowed in a cco-partition is an obvious remedy. More refined variants of this basic idea can of course be imagined, e.g. predicate-wise bounds. Moreover, it seems likely that offline analysis can be helpful, perhaps even to the extent of reducing the choice of measure functions to a restricted range, again safely guaranteeing termination.

However, we must leave these and related considerations as a subject for future research. Indeed, example 8.3.25 also sheds further light on the *principles underlying automatic finite unfolding.* It is this latter issue that, in the next section, we wish to reconsider in proper detail. Meanwhile, we believe that sections 8.2 and 8.3, as they stand, give a good impression of the power and generality of partition based unfolding, using lexicographical priorities among arguments occurring in goals.

8.4 Refining Measure Functions: A Generic Treatment

8.4.1 Introduction

At this point, it seems a good idea to interrupt our presentation of sophisticated weight based unfolding methods for a brief moment of reflection. Indeed, now that we have worked our way through quite a few algorithms, a clarified picture of common underlying principles emerges. Formalising this understanding is our business in the present section. We will resume the development of concrete algorithms afterwards.

First, we notice that the framework laid out in chapter 6 has proved to be quite general. We were able to deal both with example 8.2.7 and examples 8.3.13 and 8.3.14, using suitably specialised versions of algorithm 6.5.5. In the next section, we *will* find occasion to relax some of its inherent limitations. For the moment, however, we concentrate on our increased insight in the structure of automation.

Compared with algorithm 6.5.5, fully automatic algorithms presented or mentioned in sections 7.2, 8.2 and 8.3:

- Dynamically fix the computation rule, possibly keeping count of statically fixed preferences among selectable literals.

- All use the same particular recipe for choosing $R_0, R_1, \ldots, R_N$, assigning one R_i per recursive predicate symbol in the program, to remain unchanged throughout the execution of the algorithm.
- Each incorporate one particular basic strategy to assign to a goal a weight, i.e. an element in some well-founded (until now always totally ordered) set.
- Use measure functions based on this strategy, one function per R_i, to associate concrete weights to goals. Initial choices for these measure functions are built into the algorithms. Measure functions are refined dynamically, if desirable and possible, to enable further unfolding.

In this section, we will concentrate on a generic formalisation of the latter point, treating the other issues at about the same level of generality as was adhered to in chapter 6.

8.4.2 A generic algorithm

We start with some definitions.

Definition 8.4.1 Let P be a definite program, than we denote by $Atom_P$ the set of atoms that can be formed in the language underlying P.

Definition 8.4.2 Let P be a definite program, $\mathcal{L}_P$ its underlying language and suppose that $R_0, R_1, \ldots, R_N$ is a partition of $Atom_P$. Let $W, >_W$ be some well-founded set. Then we call a function

$$F : \{G | G \text{ is a definite goal in } \mathcal{L}_P \text{ with selected literal} \in R_k\} \to W, >_W$$

an (R_k, P)*-applicable measure function with target set* W.

Definition 8.4.3 A set $\{F | F \text{ is an } (R_k, P)\text{-applicable measure function}\}$ is called an (R_k, P)*-applicable measure space.*

In the sequel, when P (or $\mathcal{L}_P$) is clear from the context, we will often denote an (R_k, P)-applicable measure space, by $\mathcal{F}_k$. Note that different elements of a measure space may have different target sets. Finally, we demand that some *partial order relation* be defined on measure spaces. For a space $\mathcal{F}_k$, we will denote it by $\gg_k$.

We can now formulate a generic algorithm for automatic, weight-based, finite unfolding.

Algorithm 8.4.4

Input
a definite program P
a definite goal $\leftarrow A$

```
Output
  an SLD-tree $\tau$ for $P \cup \{\leftarrow A\}$
Initialisation
  $\tau$ is initialised as the SLD-tree that contains a single derivation,
    consisting of the goal $\leftarrow A$, without selected literal.
  Initial choices are made for the measure functions $F_1, \ldots, F_N$.
While there exists a non-terminated derivation $D$ in $\tau$ do
  If $D$ is succesful, Then terminate $D$
  Else If $D$'s leaf-node contains no selectable atom, Then terminate $D$
  Else
    select an $R$-preferred selectable atom
    If no derivation steps are possible, Then terminate and fail $D$
    Else extend $D$
  Where an atom $p(t_1, \ldots, t_n) \in R_k$ in a goal $G$ is selectable
  if one of the following (mutually exclusive) conditions holds,
  in case it is actually selected:
    • $G$ has no direct covering ancestor
    • $G'$ is the direct covering ancestor of $G$ and
      $F_k(G') >_k F_k(G)$
    • $G'$ is the direct covering ancestor of $G$ and
      not$(F_k(G') >_k F_k(G))$ and
      $\exists F \in \mathcal{F}_k$:   (1) $F_k \gg_k F$
                      (2) $F(G') >_k F(G)$
  If an atom $p(t_1, \ldots, t_n)$ has been selected on the basis of the third condition,
    Then replace $F_k$, in the set of measure functions in use,
    by some $F'_k$ satisfying the conditions (1) and (2) above.
Endwhile
```

In order to produce an executable instance of the above algorithm, the following is necessary:

- Computation rule preferences can be stipulated, resulting in a concrete meaning of the term "R-preferred". If none are given, all atoms are equally R-preferred.

- $R_0, R_1, \ldots, R_N$ should be chosen such that they form a partition of $Atom_P$ and guarantee the satisfaction of the third condition in definition 6.4.5.

- For every k, $1 \leq k \leq N$, we must specify $\mathcal{F}_k, \gg_k$ and choose an initial F_k in $\mathcal{F}_k$.

- Finally, if so desired, a subset-wise foundedness check like condition $(*)$ in algorithm 8.2.25 can be added.

We will call any executable instance of algorithm 8.4.4, thus determined, *proper*.

We have the following theorem:

Theorem 8.4.5 A proper instance of algorithm 8.4.4 terminates for a definite program P and goal $\leftarrow A$, producing a finite SLD-tree τ for $P \cup \{\leftarrow A\}$, if $\forall k, 1 \leq k \leq N : \mathcal{F}_k, \gg_k$ is a well-founded set. The resulting τ is subset-wise nearly founded with respect to the final $((R_0, R_1, \ldots, R_N), (F_1, \ldots, F_N))$.

Proof (Sketch)
The well-foundedness condition ensures that a change in the set of measure functions used, can occur only finitely many times. The rest of the proof is analogous to what was presented before. □

It can be verified that algorithms 7.2.1, 8.2.25, 8.2.31 and 8.3.19 are proper instances of algorithm 8.4.4, and therefore terminate. Indeed:

- Only algorithm 8.3.19 caters for user-specifiable computation rule preferences; The others do not. All eliminate remaining non-determinism in literal selection through a "choose the leftmost selectable" strategy.

- The choice for $R_0, R_1, \ldots, R_N$ is always along the same basic line: one R_i per recursive predicate symbol in P, R_0 for atoms featuring a non-recursive predicate symbol. Obviously, this satisfies the above specified conditions on $R_0, R_1, \ldots, R_N$.

- The measure spaces used are of increasing complexity:

  - Algorithm 7.2.1 uses set-based measures, universally mapping to $\mathbb{N}, >$.
  - The measure functions in algorithms 8.2.25, 8.2.31 and 8.3.19 are respectively based on partitions and cco-partitions of argument positions. They map to some $\mathbb{N}^i, \succ_i$ or $\mathbb{N}_b{}^i, \succ_i$ respectively, i varying within measure spaces.

- Definition 8.2.21 can serve as a basis for the required order relation on partition based measure spaces. Indeed, transitively closing the inverse of the "is a refinement of" relation results in a strict partial order on such spaces. Introducing a similar notion for set and cco-partition based measures is straightforward, and implicit in the work presented.

- *All measure spaces involved are finite, and therefore (trivially) well-founded.*

- Remaining non-determinism in the choice of a refined, properly decreasing measure function (i.e. satisfying conditions (1) and (2) in algorithm 8.4.4) has always been removed by imposing extra conditions (see e.g. definition 8.2.22 and proposition 8.2.24).
- Some book-keeping instructions related to the maintenance of the covering relationship have not been included in the generic algorithm 8.4.4.

Finally, it is interesting to reconsider subsection 8.3.4. The underlying reasons for non-termination in example 8.3.25 now become clear:

1. Measure spaces are no longer guaranteed to be finite.
2. And the adapted notion of refinement no longer guarantees that the resulting ordering is well-founded.

Imposing a bound on the number of components re-establishes finiteness, while more sophisticated order relations, inspired by an offline analysis, could perhaps guarantee well-foundedness of measure spaces, left infinite.

8.5 Incorporating Variant Checking

8.5.1 Introduction

Structure based weights, as presented above, are obviously not a good basis to control unfolding of datalog (i.e. functor-free) programs. Consider the following example, borrowed from [13].

Example 8.5.1

$reach(X, X) \leftarrow$
$reach(X, Z) \leftarrow reach(X, Y), edge(Y, Z)$
$edge(a, b) \leftarrow$
$edge(b, d) \leftarrow$
$edge(f, g) \leftarrow$

For the query

$\leftarrow reach(a, X)$

partial deduction produces the following specialised clauses from the SLD-tree built according to any of the methods discussed above:

$reach(a, a) \leftarrow$
$reach(a, b) \leftarrow reach(a, a)$
$reach(a, d) \leftarrow reach(a, b)$
$reach(a, g) \leftarrow reach(a, f)$

If instead we *unfold the leftmost literal without an unfolded variant* on the same derivation, we are able to derive:

$reach(a, a) \leftarrow$
$reach(a, b) \leftarrow$
$reach(a, d) \leftarrow$

However, as already illustrated in example 5.5.4 (page 88), for logic programs *with functors*, this non-variant based unfolding in general does not guarantee termination. One possible solution seems identifying datalog predicates in a program. Abstract interpretation e.g. should be able to uncover predicates whose arguments will certainly be free from functors. These can then be treated in a special way during unfolding, ignoring their (0) weight, but applying the above mentioned non-variant check. However, we have opted for a more global approach, unifying the two criteria in one generally applicable unfolding methodology. The basic idea can be described as follows: Selecting a literal A in a goal G is permitted, even when this results in *equal weights* for G and its direct covering ancestor, if no goal covering G has a selected literal which is a variant of A. However, it is clear that a straightforward application of this coarse rule does not always guarantee finiteness. Let us reconsider example 5.5.4.

Example 8.5.2 Under the set-based measure function $|.|_{rev,\{1,3\}}$, all non-empty goals from the third downwards in figure 5.1 have weight 0, and yet selected literals are not variants. However, restricting the attention to the first and third argument, i.e. those "measured" by $|.|_{rev,\{1,3\}}$, we find that they *are* variants.

Example 8.5.2 shows that welding together weight and non-variant based unfolding requires some care, but it also suggests that the enterprise is not hopeless. The rest of this section formally develops this issue. We first adapt/specialise the framework underlying algorithm 6.5.5. In a second step, we briefly address full automation.

8.5.2 Reconsidering the framework

Throughout this section, we will be dealing with definite goals, containing a finite amount of literals (which all are, of course, atoms). This allows us to suppose a numbering on the literals in a goal, e.g. from left to right, starting with 1. Each argument position in a goal is then unambiguously determined by a pair of natural numbers: (*literal_number_within_goal*, *argument_number_within_literal*).

In order to simplify the ensuing presentation, we agree on the following for the rest of subsection 8.5.2:

- Given a definite program P, $\mathcal{L}_P$ will denote its underlying language, *enhanced with an arbitrary function symbol, if P contains none.* The reason for this slight deviation from convention will become clear below. Note that a finite program's underlying language contains only *finitely many constant and function symbols.*

- A goal to which some measure function F can be applied, will be called *F-suitable*. For instance, a goal in some language $\mathcal{L}_P$, such that $R_0, R_1, \ldots, R_N$ is a partition of $Atom_P$ is suitable for some (R_k, P)-applicable measure function if its selected literal is in R_k.

We can now introduce the following:

Definition 8.5.3 Let F be a measure function and G an F-suitable goal. Then we call an argument position (i, j) of G *F-measured* if there exists an F-suitable goal G' such that:

1. $F(G') \not\equiv F(G)$
2. G' and G are identical, except at argument position (i, j).

We call the set of all F-measured argument positions of G its *F-measured argument set*, denoted $MS_F(G)$, and the multi-set of terms occurring on these argument positions in G, its *F-measured part*, denoted $MP_F(G)$. Finally, we will use the notation $t_G(i, j)$ to denote the term corresponding to an argument position (i, j) in a goal G.

The intuition behind this definition is simple: Measured argument positions indicate terms whose structure influences the weight associated to the goal. The reason for demanding the presence of at least one functor in the language now becomes clear: In a language without any function symbols, the first condition above would be unsatisfiable for the structure based measure functions in the focus of our interest. Definition 8.5.3 in that case no longer correctly formalises the intuitive notion just described.

Example 8.5.4

- With a constant measure function, mapping any suitable goal to the same element in its target set, goals obviously have empty measured argument sets.
- A measure function, merely counting the number of atoms (possibly containing a given predicate symbol) in a goal, likewise results in empty measured argument sets.
- A suitable goal's measured argument set under a *set based* measure $|.|_{p,S}$ contains exactly the argument positions corresponding to the selected literal's argument positions in S.

- All argument positions of the selected literal, and none other, are in the measured argument set under a *partition based* measure.
- Finally, all argument positions corresponding to elements occurring in the underlying cco-partition are in a goal's measured argument set under a *cco-partition based* measure function. Consider e.g. the second goal from the top, included in figure 8.7 on page 177.
  Let O_{app} be $(\{1_{app}, 2_{app}, 3_{app}\}, \{1, 2, 3\})$. Then that goal has the following $||.||_{app,O_{app}}$-measured argument set: $\{(i,j) | 2 \leq i \leq 4, 1 \leq j \leq 3\}$.

We will be interested in verifying whether the measured parts of two goals are "variants" of each other. The following definition lays down an exact content of the variant notion in this specialised context.

Definition 8.5.5 Suppose F is a measure function and let G and G' be two F-suitable goals. Then we say that *the F-measured parts of G and G' are variants*, denoted $MP_F(G) \sim MP_F(G')$ iff there is a one-to-one correspondence C between $MS_F(G)$ and $MS_F(G')$ such that $((i,j),(i',j')) \in C$ implies:

- $j = j'$
- The ith literal in G and the i'th literal in G' have the same predicate symbol.
- The ith literal in G is selected iff the i'th literal in G' is.
- There exist (renaming) substitutions θ and θ' such that
  $\forall((i,j),(i',j')) \in C : t_G(i,j)\theta = t_{G'}(i',j') \wedge t_{G'}(i',j')\theta' = t_G(i,j)$

We will also use the following notation: $G \sim_F G'$. Obviously, $\sim_F$ is an equivalence relation on the set of F-suitable goals.

The first three items above demand "equality" of the two measured argument sets. In that case, we can properly compare the terms in the corresponding measured parts, and require that they indeed be variants, as expressed in the fourth item.

Example 8.5.6 Consider the following goals (selected literals are underlined):

- $G = \leftarrow \underline{p(f(X), X)}, q(g(Y))$
- $G' = \leftarrow q(g(X')), q(g(Y')), \underline{p(f(Z'), Z'')}$

and the following measure functions:

- $F_1 = |.|_{p,\{1\}}$

- $F_2 = |\cdot|_{p,(\{1\},\{2\})}$
- $F_3 = ||\cdot||_{p,(\{1_q\},\{1,2\})}$

Then we have:

- $MS_{F_1}(G) = \{(1,1)\}$
- $MS_{F_1}(G') = \{(3,1)\}$
- We can take: $C = \{((1,1),(3,1))\}$, $\theta = \{X/Z'\}$, $\theta' = \{Z'/X\}$.
  And therefore: $G \sim_{F_1} G'$

And:

- $MS_{F_2}(G) = \{(1,1),(1,2)\}$
- $MS_{F_2}(G') = \{(3,1),(3,2)\}$
- $C = \{((1,1),(3,1)),((1,2),(3,2))\}$ (uniquely) satisfies the first three conditions in definition 8.5.5. But no θ and θ', as required in the last part of the above definition, can be constructed.
  So: $G \not\sim_{F_2} G'$

Finally:

- $MS_{F_3}(G) = \{(1,1),(1,2),(2,1)\}$
- $MS_{F_3}(G') = \{(1,1),(2,1),(3,1),(3,2)\}$
- There is no one-to-one correspondence between $MS_{F_3}(G)$ and $MS_{F_3}(G')$.
  This means: $G \not\sim_{F_3} G'$

Notice that all three measure functions assign equal weights to G and G': 1, $(1,0)$ and $(1,1)$ respectively.

We can now adapt algorithm 6.5.5 (we only include an updated version of its main loop):

Algorithm 8.5.7

```
While there exists a non-terminated derivation $D \in \tau$ do
  Let $(G,i)$ be the leaf of $D$
  Let $Derive(G,i)$ be the set of all its immediate $>_{\tau_0}$-descendants
  If $Derive(G,i) = \emptyset$
    Then add $D$ to $Terminated$
  Else if there is a direct covering ancestor $(G',j)$ of $(G,i)$
          with $R(G',j), R(G,i) \in R_n$
          such that none of the following is satisfied:
             1) $F_n(G', j) >_n F_n(G, i)$
             2) $F_n(G', j) \equiv F_n(G, i) \wedge$
                $\neg\exists (G'', k) \in D : (G'', k)$ covers $(G, i) \wedge$
                                    $F_n(G'', k) \equiv F_n(G, i) \wedge$
                                    $(G'', k) \sim_{F_n} (G, i)$
       Then add $D$ to $Terminated$
     Else
       Replace $\tau$ by $\tau \setminus D \cup \{D \cup \{(G^*, l)\} | (G^*, l) \in Derive(G, i)\}$
Endwhile
```

Algorithm 8.5.7 differs from algorithm 6.5.5 by the presence of condition 2). It allows extending a derivation whose leaf has a weight identical to the weight of its direct covering ancestor, but contains a measured part which is not a variant of the measured part found in any covering ancestor of equal weight.

Example 8.5.8 Consider the program and query in example 8.5.1. Take:

- $R_0 = \{$atoms containing $edge\}$
- $R_1 = \{$atoms containing $reach\}$
- $F_1 = |\cdot|_{reach, \{1,2\}}$

The SLD-tree produced by algorithm 8.5.7 can be found in figure 8.9. Notice that all measured goals have the same weight: 0. However, condition 2) in algorithm 8.5.7 enables proper unfolding.

Example 8.5.9 For the *reverse* program and query treated in example 5.5.4, algorithm 8.5.7 stops unfolding after the first branching in figure 5.1, (producing the tree in figure 7.2) both when applied with $|\cdot|_{rev,\{1,3\}}$ and $|\cdot|_{rev,(\{1,3\},\{2\})}$. In the former case, goal $(*)$ and its direct covering ancestor, $(**)$, have equal weights (0), but also variant measured parts, $\{Xs', Z\}$ and $\{Xs, Z\}$ respectively. $|\cdot|_{rev,(\{1,3\},\{2\})}$ on the other hand is simply $(**, *)$-increasing.

Finally, we address the question whether algorithm 8.5.7 always terminates. Unfortunately, this is not the case.

Example 8.5.10 Consider the following program fragment:

$p(X) \leftarrow q(f(X)), r(g(X))$
$q(X) \leftarrow q(X), r(X)$

And unfold the goal

$\leftarrow p(a)$

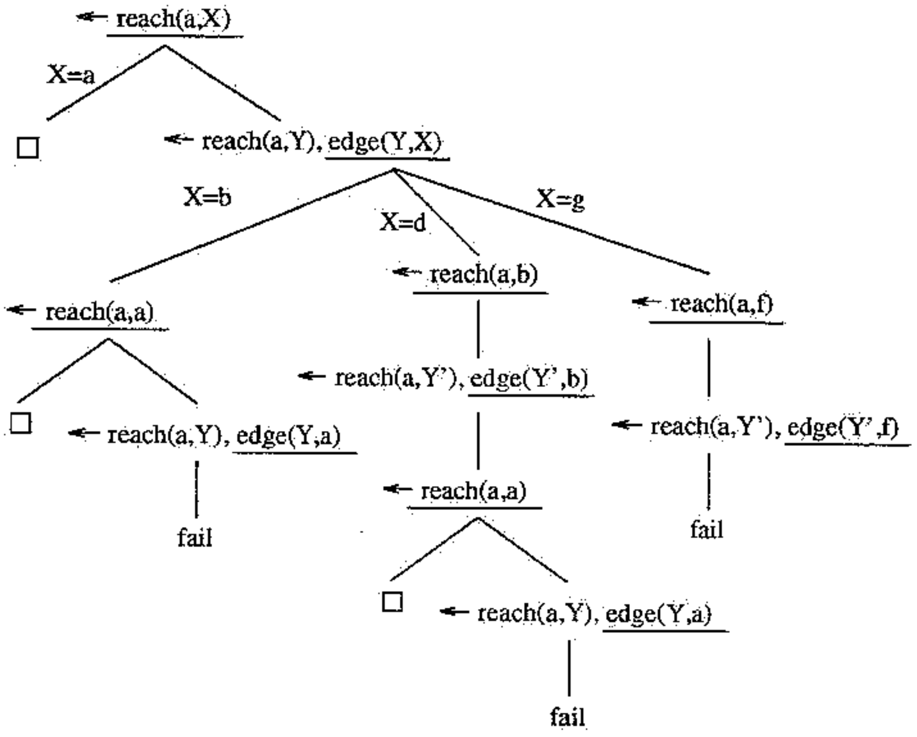


Figure 8.9: A properly unfolded datalog tree.

using the cco-partition based measure function $||.||_{q,(\{1,\},\{1\})}$. The first few nodes of the resulting infinite SLD-derivation are depicted in figure 8.10. Measured nodes are annotated with their weight. Note that the measure function is stable, but the measured argument set changes with each unfolding. Condition 2) in algorithm 8.5.7 is therefore always satisfied and the derivation never terminated.

So, algorithm 8.5.7 may fail to terminate. We need a notion to characterise "safe" measure functions.

Definition 8.5.11 A measure function F with target set W is called ***finitely measuring*** for a language $\mathcal{L}_P$, if for any weight $w \in W$, the quotient set $\{G \text{ in } \mathcal{L}_P | F(G) \equiv w\}/\sim_F$ is finite.

In other words, only a finite number of goals, expressed in the given language, with non-variant measured part, have the same weight under F.

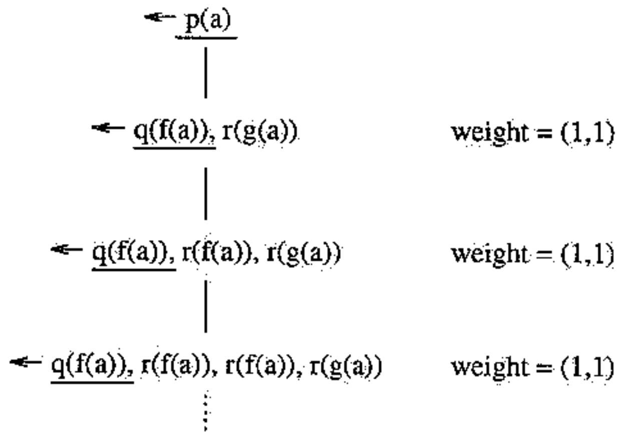


Figure 8.10: Infinitely many goals with equal weight but non-variant measured part.

Theorem 8.5.12 Algorithm 8.5.7 terminates for a definite program P and goal $\leftarrow A$, using a given computation rule R and pair $((R_0, R_1, \ldots, R_N), (F_1, \ldots, F_N))$, if $F_1, \ldots, F_N$ are finitely measuring for $\mathcal{L}_P$. The resulting finite SLD-tree τ is subset-wise nearly founded with respect to $((R_0, R_1, \ldots, R_N), (F_1, \ldots, F_N))$.

Proof (Sketch)
The proof is similar to the proofs for analogous theorems above, now relying on definition 8.5.11 and condition 2) in algorithm 8.5.7, to argue that the measure functions are *nearly* founded on chains of covering nodes. □

Theorem 8.5.12 shows that algorithm 8.5.7 can safely be used with finitely measuring measure functions. The remaining question is whether we can characterise classes of measure functions as finitely measuring. In particular, are measure functions of the kinds introduced in this thesis, finitely measuring ? We have the following results (overloading function symbols as was implicitly done before):

Proposition 8.5.13 Let P be a definite program and $\mathcal{L}_P$ its (enhanced) underlying language. Let p be a predicate symbol of arity n in $\mathcal{L}_P$. Let S be a set of argument positions of p and O an ordered k-partition associated to P. Then:

1. The measure function $|.|_{p,S}$:
   $\{G|G \text{ is a definite goal in } \mathcal{L}_P \text{ with selected literal } p(t_1, \ldots, t_n)\} \to I\!N$,
   $G \mapsto |p(t_1, \ldots, t_n)|_{p,S}$
   is finitely measuring for $\mathcal{L}_P$.

2. The measure function $|.|_{p,O}$:

   $\{G|G$ is a definite goal in $\mathcal{L}_P$ with selected literal $p(t_1,\ldots,t_n)\} \rightarrow I\!N^k$,

   $G \mapsto |p(t_1,\ldots,t_n)|_{p,O}$

   is finitely measuring for $\mathcal{L}_P$.

Proof Since $\mathcal{L}_P$ contains only a finite amount of constant and function symbols, there is only a finite number of non-variant terms in $\mathcal{L}_P$ to which the functor norm $|.|$, defined in definition 6.3.2, assigns the same natural number. The results now follow from definition 8.5.11 and

1. definition 6.3.3 together with the observation under the third point of example 8.5.4 for case (1).
2. definition 8.2.2 together with the observation under the fourth point of example 8.5.4 for case (2).

□

So, indeed, *set and partition based* measure functions are finitely measuring for languages underlying a finite program. Algorithm 8.5.7 will certainly terminate when using them. Their common characteristic, guaranteeing these results, is the fact that all goals suitable for such a measure function have essentially the *same measured argument set* under that measure function. Example 8.5.6 already shows that this is not always the case for *cco-partition based* measure functions. And indeed, example 8.5.10 demonstrates that such measure functions are in general *not* finitely measuring. Of course, safely combining their use with some form of "equal weight but non-variant" unfolding remains possible. It suffices to focus on the selected literal for the variant test. Alternatively, a specially tuned version of the "measured part" notion, to some extent also incorporating context information, can probably be developed. However, we will not devote a detailed study to this issue. Rather, we shift our attention to automation of the "safe" cases in the next subsection.

8.5.3 Issues in automation

Basically, it is very straightforward to adapt algorithm 8.2.31 for fully automatic partition based unfolding along the lines of the previous subsection.

Algorithm 8.5.14 We add one more item to the list of conditions enabling the selection of a literal for unfolding:

- (G',j) is the direct covering ancestor of (G,i) and
  $|.|_{p,O_p}$ is $(R(G',j),p(t_1,\ldots,t_n))$-stable and
  $\neg\exists(G'',k)\in D : [(G'',k)$ covers $(G,i)\ \wedge$
  $\qquad |.|_{p,O_p}$ is $(R(G'',k),p(t_1,\ldots,t_n))$-stable $\wedge$
  $\qquad (G'',k) \sim_{|.|_{p,O_p}} (G,i)]$

The rest of algorithm 8.2.31 remains unchanged (except for one detail addressed below).

Example 8.5.15 Algorithm 8.5.14, applied to the program and query in example 8.5.1, produces the SLD-tree depicted in figure 8.9. Throughout the whole unfolding process, O_{reach} is never changed, and keeps its initial value: $(\{1,2\})$. In general, it is obvious that no partition based measure, as introduced in section 8.2, ever has potential in a datalog context.

It is not difficult to realise that algorithm 8.5.14 still terminates, and builds a finite, subset-wise nearly founded SLD-tree. One detail remains to be settled. Indeed, the enhanced list of conditions for literal selection *no longer contains mutually exclusive cases*: $|.|_{p,O_p}$ can at the same time be stable and have potential. Let us once more have a look at example 5.1. The measure function $|.|_{rev,(\{1,2,3\})}$ is stable all along the top three goal nodes of the tree in figure 5.1. Moreover, the *rev* literals are not variants of each other. Therefore, algorithm 8.5.14 allows unfolding without changing the measure function to $|.|_{rev,(\{1,2\},\{3\})}$ as would be done by algorithm 8.2.31. (Note, however, that both terminate unfolding at node (*).) Such a behaviour might be considered in conflict with our basic philosophy. It can easily be avoided by imposing a priority among the unfoldability conditions: first try a refinement, only if that fails, try unfolding on the "equal weight but non-variant" basis.

A second point, and an additional motive for imposing the just mentioned priority, is the *potential inefficiency of the non-variant check*. Indeed, the latter does require searching through a list of (equal weight) covering ancestors. So, the linearity property, established in subsection 8.2.4, is lost. It might therefore be a good idea to restrict as much as possible the cases in which such scans are undertaken. Observe, however, that the implementation technique proposed in subsection 8.2.4, provides excellent support for scanning chains of covering ancestors, if so desired. Reconsider figure 8.2 on page 159. If we select e.g. the left p literal in node (5), the covering ancestors can easily be spotted. The literal is annotated with the list [4, 2, 3] (remember that the first element of these lists refers to p). So, node (4) is the direct covering ancestor. Its selected p-literal has annotation [1, 2, 3]. This means that the next covering ancestor is node (1), and the "_" on the first position of its annotation list marks the top of the chain.

Summarising, we can state that an integration of variant checking with *partition* based unfolding is relatively straightforward, even in a fully automatic context. And, since the measured part for these measure functions invariably coincides with the whole selected literal, the full generality of the development in the previous subsection is, in this restricted context, probably somewhat of an overkill. Open issues requiring further (experimental) research are a.o.:

- How big is the gain in sensible unfolding capacity outside a strict datalog context ?

- How severe is the mentioned efficiency problem ?

Finally, we briefly address *set* based unfolding. Since, now, part of a selected literal can be "disregarded", our integration enterprise involves some more subtle issues. Consider the following example:

Example 8.5.16
$p(X,Y) \leftarrow q(X,Z), p(Z,[X|Y])$
$q(a,b) \leftarrow$
Part of an SLD-derivation for $\leftarrow p(a,Y)$ can be found in figure 8.11.

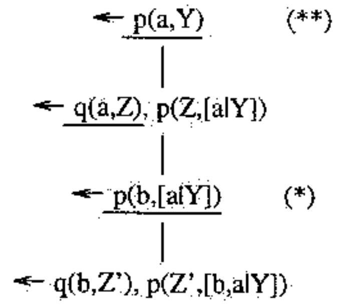


Figure 8.11: Combining set based and non-variant unfolding.

In node $(*)$, we would like to be able to carry out the indicated unfolding step. This is possible with combined unfolding using $|.|_{p,\{1\}}$ as measure function. However, setting out with the initial $|.|_{p,\{1,2\}}$, and adapting the notion of tight decreasing refinement to the context of set based measures in the obvious way, the above measure function is *not* a $(**,*)$-*decreasing* refinement of $|.|_{p,\{1,2\}}$. In fact, there is none.

So, we need to revise the automatic unfolding algorithm in such a way that measure functions are refined into *non-increasing* ones, possibly leading to useful stable unfolding on the basis of the newly added non-variant condition. Doing so *can make set based unfolding more powerful than partition based* in some cases, thus outdating our conjecture at the end of subsection 8.2.3. Of course, blending the two approaches is possible, through *partitioning subsets* of a selected literal's argument positions.

8.6 Focusing on Subterms

8.6.1 Introduction

All concrete unfolding strategies presented above rely on atom and goal weights determined by one or more arguments in the considered atoms. (As can be observed in definitions 6.3.3, 8.2.2 and 8.3.7.) And indeed, we have shown how this provides a good basis for unfolding most logic programs. However, in some cases, it seems more appropriate not to consider the overall structure of entire arguments, but instead to focus attention on specific parts of complex terms. The following example indicates what we have in mind.

Example 8.6.1 Consider a clause as follows:

$p(f([X|Y], Z)) \leftarrow p(f(Y, [X, X|Z]))$

And take as query to be unfolded:

$\leftarrow p(f([a, b, c|Y], Z))$

An (incomplete) SLD-derivation is depicted in figure 8.12. Nodes are annotated with the weight assigned to p's single argument by the functor norm introduced in definition 6.3.2.

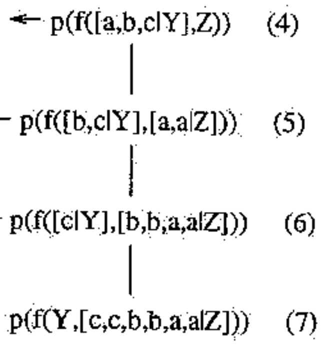

← p(f([a,b,c|Y],Z)) (4)
← p(f([b,c|Y],[a,a|Z])) (5)
← p(f([c|Y],[b,b,a,a|Z])) (6)
← p(f(Y,[c,c,b,b,a,a|Z])) (7)

Figure 8.12: Interesting subargument behaviour.

It seems quite reasonable to expect that (at least) this amount of unfolding would be performed at partial deduction time. However, under any concrete measure function introduced in this thesis, goal weights continuously *increase* throughout the derivation. The unfolding algorithms based on them will therefore halt after just one step. The first f-argument is what we need to measure, rather than the whole p-argument.

The rest of the present section contains a formal treatment of this issue. Below, we first show how algorithm 6.5.5 can cope with argument behaviour as in example 8.6.1, provided it is run with a suitable $((R_0, R_1, \ldots, R_N), (F_1, \ldots, F_N))$ pair. Next, subsection 8.6.3 addresses automation. It includes a formal development of more flexible instruments, needed in this more sophisticated context, as well as an algorithm that is capable of automatically focusing on subarguments. Finally, meta-interpreters already surfaced as a class of programs requiring sophisticated measuring of this kind. Some examples and preliminary considerations were included in subsections 6.3.3 and 6.5.3 above. Our development here sheds new light on the issues involved in (automatically) unfolding such programs. We therefore explicitly revisit the subject in subsection 8.6.4.

Throughout this section, in order to concentrate on the major novel issues specific to subterm focusing, we will use as a basis the simple framework of *set-based* measures.

8.6.2 More detailed measures

As pointed out above, definition 6.3.3 lacks sophistication to properly serve our needs in the present context. We need more powerful measure functions. In fact, definition 6.3.8 provides a suitable basis. For convenience, we repeat it here (let $Term_P$ and $Atom_P$ respectively denote the set of terms and atoms in the first order language $\mathcal{L}_P$, used to define a given theory P):

Definition 8.6.2 A *selector function* s (for P), denoted as a finite, non-empty sequence of positive integers connected with slashes, $n_1/n_2/\ldots/n_k$, is a (partial) function: $Atom_P \cup Term_P \rightarrow Term_P$, recursively defined as follows:

```
**If** $s = n$ and $n \leq m$
  **Then** $s(r(t_1, \ldots, t_m)) = t_n$
**Else if** $s = n_1/n_2/\ldots/n_k$, $n_1 \leq m$ and $n_2/\ldots/n_k(t_{n_1})$ is defined
  **Then** $s(r(t_1, \ldots, t_m)) = n_2/\ldots/n_k(t_{n_1})$
**Else** $s(r(t_1, \ldots, t_m))$ is undefined.
```

We include the following definition for later use.

Definition 8.6.3 Let P_1 and P_2 be two atoms in $Atom_P$ and s be a selector function such that both $s(P_1)$ and $s(P_2)$ are defined. Then s is:

- (P_1, P_2)-*decreasing* iff $|s(P_1)| > |s(P_2)|$
- (P_1, P_2)-*increasing* iff $|s(P_2)| > |s(P_1)|$
- (P_1, P_2)-*stable* iff $|s(P_1)| = |s(P_2)|$

We continue:

Definition 8.6.4 Let V be a subset of $Atom_P \cup Term_P$. Then a selector function s for P is called V-*applicable* if $s(v)$ is defined for every $v \in V$.

Example 8.6.5
Consider

$s_1 = 1/1$
$s_2 = 1/2$
$s_3 = 1/2/2$

And

$A = p(f([a, b, c|Y], Z))$
$B = p(f([b, c|Y], [a, a|Z]))$

Then s_1 and s_2 are $\{A, B\}$-applicable, but s_3 is not.
Furthermore:

$s_1(A) = [a, b, c\|Y]$	$s_1(B) = [b, c\|Y]$
$s_2(A) = Z$	$s_2(B) = [a, a\|Z]$
$s_3(A)$ is undefined	$s_3(B) = [a\|Z]$

Definition 8.6.6 Let V be a subset of $Atom_P$. Let $S = \{s_1, \ldots, s_m\}$ be a (non-empty) set of V-applicable selector functions. We define $|.|_{V,S} : V \to I\!N$ as follows:

$|v|_{V,S} = |s_1(v)| + \cdots + |s_m(v)|$

where $|.|$ is the functor norm introduced in definition 6.3.2.

Example 8.6.7 Elaborating example 8.6.5, we have:

$|A|_{\{A,B\},\{s_1,s_2\}} = |[a, b, c|Y]| + |Z| = 3$
$|B|_{\{A,B\},\{s_1,s_2\}} = |[b, c|Y]| + |[a, a|Z]| = 4$
$|B|_{\{B\},\{s_2,s_3\}} = |[a, a|Z]| + |[a|Z]| = 3$

Using the terminology introduced in definition 8.4.2, a function $|.|_{V,S}$ thus defined, can be considered a (V, P)-applicable measure function with target set $I\!N$ (often simply to be called a V-applicable measure function). Indeed, throughout this entire section, a *goal's* weight will always be taken equal to the weight of its selected literal. Therefore, we will (again) overload our language and our mathematical symbols, and use identical notation and terminology to refer to (measure) functions defined on atoms on the one hand, and on entire goals (with a selected literal) on the other hand.

It turns out that definition 8.6.6 is slightly too general for our purposes. Indeed, as can be observed in the above example, some parts of an atom may be considered more than once in determining its weight. This is remedied as follows:

Definition 8.6.8 Let s and s' be two selector functions, denoted by the strings t and t'. Then the string t/t' also denotes a selector function, called a ***subselector*** of s.

If a selector function s is applicable to some set V, than all selector functions of which it is a subselector, are too. For any element of V, they will select larger subterms than s. Hence the name "subselector" for s. We will also need the following, more specific, concept:

Definition 8.6.9 Let s be a selector function, denoted by t. Let n be a positive integer. Then t/n denotes a *first level subselector* of s.

Example 8.6.10
In example 8.6.5, s_3 is a (first level) subselector of s_2.

Definition 8.6.11 Let V be a subset of $Atom_P$ and S a (non-empty) set of V-applicable selector functions. Then we define $|.|_{V,S}$ to be *singularly measuring* if no element of S is a subselector of another element of S.

Example 8.6.12
$|.|_{\{A,B\},\{s_1,s_2\}}$ is singularly measuring, but $|.|_{\{B\},\{s_2,s_3\}}$ is not.

The notion of a singularly measuring measure function is the generalisation of definition 6.3.3 that we will actually use throughout this section. Note that all measure functions of the kind introduced in definition 6.3.3 are indeed singularly measuring.

We now have the right tools to apply algorithm 6.5.5 succesfully to example 8.6.1.

Example 8.6.13
Running the algorithm with

$R_0 = \emptyset$
$R_1 = \{\text{instances of } p(f(X,Y))\}$
$F_1 = |.|_{R_1,\{1/1\}}$

produces the desired result: the derivation in figure 8.12 with one extra (dangling) leaf. The root goal's weight equals 3. The goal weight decreases with 1 at each unfolding until it finally reaches 0.

The following is a slightly more complicated example.

Example 8.6.14 Consider the following clauses:

$p(f([X|Y], Z)) \leftarrow p(g(X, Y, Z))$
$p(g(X, Y, Z)) \leftarrow p(f(Y, [X, X|Z]))$

together with the query from example 8.6.1. There might be other ways to handle this situation, but obvious choices are:

$R_1 = \{\text{instances of } p(f(X,Y))\}$
$F_1 = |.|_{R_1,\{1/1\}}$
$R_2 = \{\text{instances of } p(g(X,Y,Z))\}$
$F_2 = |.|_{R_2,\{1/2\}}$ or $F_2 = |.|_{R_2,\{1/1,1/2\}}$

The above examples, brief as they may be, clearly announce the main issues to be considered in the context of automation:

- Taking one R_i per recursive predicate symbol, as was standard practice until now, is no longer satisfactory. Rather, the R_i classes must be dynamically determined, together with the associated measure functions.
- An extended notion of tight decreasing refinement is required. Instead of just removing elements from S as proposed in section 7.2, we might want to replace them by one or more of their subselectors.
- We must ensure that each F_i is indeed R_i-applicable.
- Finally, new issues in termination arise.

8.6.3 Automation

Preliminaries

Automatic unfolding with a "focusing on subterms" facility involves a number of quite subtle issues. A detailed technical treatment of all aspects is beyond the scope of this thesis. Rather than including a high level, somewhat vague, descriptive discussion, we do present a precise technical development and a (nearly) concrete algorithm. *Let it be clear however, that this subsection is meant to serve as one example study of how subterms can be dealt with in the context of online logic program unfolding, much more than as a final treatise on the subject.*

In section 8.4, we presented a generic treatment of automatic weight based unfolding, highlighting the common principles underlying most of our algorithms. However, in the pair $((R_0, R_1, \ldots, R_N), (F_1, \ldots, F_N))$ of atom classes and associated measure functions, underlying the template algorithm 8.4.4, the R_i-classes are fixed, and only the F_i-functions are dynamically tuned to reach optimal values. It is clear that this choice is no longer sufficient if we want to dynamically focus on subterms. We must have the ability to *refine the atom class partition* as well as the associated measure functions.

Let us first introduce some general terminology.

Definition 8.6.15 A pair $((R_0, R_1, \ldots, R_N), (F_1, \ldots, F_N))$ where $R_0, R_1, \ldots, R_N$ is a partition of $Atom_P$ and $F_1, \ldots, F_N$ a series of associated measure functions, will be called a *class-measure pair (cmp)* (for P).

Definition 8.6.16 Let G_1 and G_2 be two goals in $\mathcal{L}_P$ with selected literals P_1 and P_2. Let $((R_0, R_1, \ldots, R_i, \ldots, R_N), (F_1, \ldots, F_i, \ldots, F_N))$ be a cmp such that $P_1, P_2 \in R_i$. Then we call this cmp (G_1, G_2)-*decreasing*, respectively *-increasing* or *-stable*, iff F_i is.

Before we can actually describe the required refinement notions, we need a formal symbolic apparatus enabling us to handle the cmps that are of immediate interest in the context of this section. In the previous subsection, we already completed this task for the measure functions to be considered. Let us now address atom classes. (Throughout the following definitions, we assume that every function symbol and every predicate symbol in $\mathcal{L}_P$ has a single associated arity, which we can then safely leave implicit.)

Definition 8.6.17 A *term flag* is recursively defined as follows:

- A variable symbol is a term flag.
- A set of function symbols is a term flag.
- A function symbol applied to arguments which are term flags, together not containing any variable symbol more than once, is a term flag.

Clearly, no variable symbol can occur more than once in any term flag. In the sequel, we will refer to term flags of the above introduced varieties as *var flags*, *set flags* and *functor flags* respectively. A term flag appearing at some level inside a (functor) flag, will be called a *subflag* of the latter. (Notice that only functor flags have subflags.)

Definition 8.6.18 Let φ be a term flag (in $\mathcal{L}_P$). Then we define its *covered term set* $\mathcal{T}_\varphi$ as follows:

- If φ is a var flag, then $\mathcal{T}_\varphi$ is the set of all terms in $\mathcal{L}_P$.
- If φ is a set flag, then $\mathcal{T}_\varphi$ is the set of all terms in $\mathcal{L}_P$ whose top level functor (if any) does *not* occur in φ.
- If $\varphi = f(\varphi_1, \ldots, \varphi_n)$ and $\mathcal{T}_{\varphi_1}, \ldots, \mathcal{T}_{\varphi_n}$ are the term sets respectively covered by $\varphi_1, \ldots, \varphi_n$, then $\mathcal{T}_\varphi = \{f(t_1, \ldots, t_n) | t_1 \in \mathcal{T}_{\varphi_1}, \ldots, t_n \in \mathcal{T}_{\varphi_n}\}$

Definition 8.6.19 A predicate symbol applied to arguments which are term flags, together not containing any variable symbol more than once, is an *atom flag*.

Again observe that repeated occurrences of the same variable symbol within an atom flag are not allowed.

Definition 8.6.20 Let ψ be an atom flag $p(\varphi_1, \ldots, \varphi_n)$. Then its *covered atom set* is defined as follows:

$\mathcal{A}_\psi = \{p(t_1, \ldots, t_n) | t_1 \in \mathcal{T}_{\varphi_1}, \ldots, t_n \in \mathcal{T}_{\varphi_n}\}$

It can be observed that flags identical up to variable symbol renaming, cover the same set (of atoms or terms). We will therefore not distinguish between such flags and consider for example $p(X)$ and $p(Y)$ as one and the same atom flag. The following property is immediate:

Proposition 8.6.21 If ψ is an atom flag then $\mathcal{A}_\psi \neq \emptyset$.

Example 8.6.22

- $p(X)$ is an atom flag, covering the set of all atoms with predicate symbol p.
- $p(f(X,Y))$ is an atom flag and its covered atom set contains all atoms which are instances of $p(f(X,Y))$.
- In general, when an atom flag contains no term flag of the "set" type, its covered atom set comprises exactly all instances of the atom which is syntactically identical to the given flag.
- $p(\{f\})$ covers the set of all p-atoms that do not have f as the top level function symbol of their argument.
- $p(f(X,\{g,h\}))$ covers the subset of $\mathcal{A}_{p(f(X,Y))}$ that contains all its atoms with neither g nor h as the top level functor of the second f argument. Thus:
  - $p(f(X,Y)) \in \mathcal{A}_{p(f(X,\{g,h\}))}$
  - $p(f(X,[g(X,a,h(b)),b])) \in \mathcal{A}_{p(f(X,\{g,h\}))}$
  - $p(f(X,g(X,a,h(b)))) \notin \mathcal{A}_{p(f(X,\{g,h\}))}$

The above introduced formalism could be slightly extended to allow more flexibility at the level of atoms. That would for example be of immediate use in the description of R_0. However, for R_0, we will, also in this section, always make the same static choice: it will contain all atoms with a non-recursive predicate symbol. As a consequence, if $R_1, \ldots, R_N$ forms a partition of the rest of $Atom_P$, the required partition property as well as the third condition in definition 6.4.5 will be satisfied for $R_0, R_1, \ldots, R_N$. Which can then be safely used as a basis for unfolding, given proper measure functions. In the sequel, we will therefore *no longer explicitly refer to* R_0, and characterise cmps by $((R_1, \ldots, R_N), (F_1, \ldots, F_N))$ pairs.

Definition 8.6.23 A cmp $((R_1, \ldots, R_N), (F_1, \ldots, F_N))$ (for P) is *well-structured* iff:

- $R_1, \ldots, R_N$ is a partition of the set of atoms in $\mathcal{L}_P$ with a recursive predicate symbol.
- $\forall 1 \leq i \leq N : F_i$ is R_i-applicable.

Example 8.6.24 Suppose that, in the given language, p is the only (recursive) predicate symbol. Then the cmp used in example 8.6.13 can be represented as:

$$((\mathcal{A}_{p(f(X,Y))}, \mathcal{A}_{p(\{f\})}), (|\cdot|_{\mathcal{A}_{p(f(X,Y))},\{1/1\}}, |\cdot|_{\mathcal{A}_{p(\{f\})},\{1\}}))$$

To simplify notation somewhat, we will henceforth drop the $\mathcal{A}$ from the measure function subscripts. Doing so, we reconsider example 8.6.14:

Example 8.6.25 Taking the second choice for F_2, the cmp becomes:

$$((\mathcal{A}_{p(f(X,Y))}, \mathcal{A}_{p(g(X,Y,Z))}, \mathcal{A}_{p(\{f,g\})}),$$
$$(|\cdot|_{p(f(X,Y)),\{1/1\}}, |\cdot|_{p(g(X,Y,Z)),\{1/1,1/2\}}, |\cdot|_{p(\{f,g\}),\{1\}}))$$

Clearly, both this cmp and the one in example 8.6.24 are well-structured.

The following definition will be convenient:

Definition 8.6.26 Let $\psi_1, \ldots, \psi_N$ be N atom flags such that $\mathcal{A}_{\psi_1}, \ldots, \mathcal{A}_{\psi_N}$ is a partition of the set of atoms with a recursive predicate symbol. Let $S_1, \ldots, S_N$ be N sets of selector functions, such that each S_i contains only $\mathcal{A}_{\psi_i}$-applicable elements, and each $|\cdot|_{\psi_i,S_i}$ is singularly measuring. Then we call the cmp

$$((\mathcal{A}_{\psi_1}, \ldots, \mathcal{A}_{\psi_i}, \ldots, \mathcal{A}_{\psi_N}), (|\cdot|_{\psi_1,S_1}, \ldots, |\cdot|_{\psi_i,S_i}, \ldots, |\cdot|_{\psi_N,S_N}))$$

a *selector based cmp (sbcmp)*.

Proposition 8.6.27 A selector based cmp is well-structured.

Proof Immediate from definitions 8.6.23 and 8.6.26. □

Two refinement notions

Definition 8.6.30 below presents *the first of two refinement notions on sbcmps.* It is a straightforward reformulation of the already familiar one on measure functions.

Definition 8.6.28 Let χ be an sbcmp, $((\ldots, \mathcal{A}_{\psi_i}, \ldots), (\ldots, |\cdot|_{\psi_i,S_i}, \ldots))$. Let G_1 and G_2 be two goals with respective selected literals P_1 and P_2, both covered by ψ_i, such that χ is not (G_1, G_2)-decreasing. Then we say that χ has (G_1, G_2)-*m-potential* iff S_i contains at least one (P_1, P_2)-decreasing selector function.

Example 8.6.29 Let G_1 be a goal with selected literal $P_1 = p(f([a, b, c|Y], Z))$ and G_2 be a goal with selected literal $P_2 = p(f([b, c|Y], [a, a|Z]))$. Then the sbcmp

$$((\mathcal{A}_{p(f(X,Y))}, \mathcal{A}_{p(\{f\})}), (|\cdot|_{p(f(X,Y)),\{1/1,1/2\}}, |\cdot|_{p(\{f\}),\{1\}}))$$

has (G_1, G_2)-m-potential.

Definition 8.6.30 Let χ, G_1, G_2, P_1 and P_2 be as in definition 8.6.28. Let χ have (G_1, G_2)-m-potential. Then we define the (G_1, G_2)-*m-refinement* of χ to be the pair

$$((\ldots, \mathcal{A}_{\psi_i}, \ldots), (\ldots, |\cdot|_{\psi_i, S'_i}, \ldots))$$

identical to χ, except at S'_i, for which the following holds:

$$S'_i = S_i \setminus \{s \in S_i | s \text{ is } (P_1, P_2)\text{-increasing}\}$$

Example 8.6.31 The (G_1, G_2)-m-refinement of the sbcmp in example 8.6.29 is:

$$((\mathcal{A}_{p(f(X,Y))}, \mathcal{A}_{p(\{f\})}), (|\cdot|_{p(f(X,Y)),\{1/1\}}, |\cdot|_{p(\{f\}),\{1\}}))$$

The following proposition shows that m-refining is a well-defined operation.

Proposition 8.6.32 Let χ, G_1, G_2, P_1 and P_2 be as in definition 8.6.30. Let χ' be the (G_1, G_2)-m-refinement of χ. Then:

1. $\chi' \neq \chi$
2. χ' is an sbcmp.
3. χ' is (G_1, G_2)-decreasing.

Proof

1. χ is (G_1, G_2)-stable or -increasing and has (G_1, G_2)-m-potential. This means that there is at least one (P_1, P_2)-increasing selector in S_i. So, $S'_i \neq S_i$.

2. No change is made to the atom classes, so they keep the required partition property. Furthermore, deleting one or more selectors from S_i obviously does not affect the $\mathcal{A}_{\psi_i}$-applicability of the remaining ones. Nor does it damage the singularly measuring property of $|.|_{\psi_i, S_i}$. Finally, all other measure functions remain unchanged.

3. There is at least one (P_1, P_2)-decreasing element in S'_i and none which is (P_1, P_2)-increasing. The result follows.

□

This concludes the construction of the first refinement notion. Obviously, it does not bring anything substantially new. We now turn our attention to the description of *a second refinement operation* which does present a major innovation. It incorporates the possibility to shift attention towards subterm structure. This not only requires changes in the measure functions of an sbcmp, but also necessitates adapting the atom classes. It is this operation which we will not formalise in its fullest generality. Rather, we will limit the complexity of the ensuing presentation by hard-wiring a number of particular choices into our definitions. We will briefly indicate some possible alternatives at the end of this subsection.

First, we need some additional tools for manipulating flags. Generalising definition 8.6.2 such that selector functions can also be applied to flags is straightforward, and will not be done explicitly. Then we can define the following:

Definition 8.6.33 Let ϕ be an (atom or term) flag and s a selector function. We call s *leaf selecting* for ϕ if $s(\phi)$ is a var or a set flag.

Definition 8.6.34 Let ψ be an atom flag, s a selector function which is leaf selecting for ψ and f a function symbol (with arity n), not occurring in $s(\psi)$. Then the (s,f)*-refinement* of ψ is the atom flag obtained from ψ by replacing $s(\psi)$ with $f(X_1,\ldots,X_n)$, where $X_1,\ldots,X_n$ are distinct variable symbols, not appearing in ψ.

Occasionally, we will simply speak about a "refinement" in contexts where the actual s and f do not matter.

Definition 8.6.35 Let ψ be an atom flag, s a selector function which is leaf selecting for ψ and f a function symbol. Let ψ_1 be the (s,f)-refinement of ψ. Then the ψ*-complement* of ψ_1 is the atom flag ψ_2 defined as follows:

- If $s(\psi)$ is a var flag, then ψ_2 results from ψ by replacing $s(\psi)$ with the set flag $\{f\}$.
- If $s(\psi)$ is a set flag, then ψ_2 results from ψ by adding the function symbol f to $s(\psi)$.

Example 8.6.36 Let f and g be function symbols with arity 2 and 3 respectively. Then:

- $p(f(X,Y))$ is the $(1,f)$-refinement of $p(X)$; it has $p(\{f\})$ as its $p(X)$-complement.
- The $(1,g)$-refinement of $p(\{f\})$ is $p(g(X,Y,Z))$, the $p(\{f\})$-complement of which is $p(\{f,g\})$.
- Finally, the $(1/1,g)$-refinement of $p(f(X,Y))$ is $p(f(g(X_1,X_2,X_3),Y))$. Here the $p(f(X,Y))$-complement is $p(f(\{g\},Y))$.

The following proposition motivates definition 8.6.35:

Proposition 8.6.37 Let ψ be an atom flag, ψ_1 a refinement of ψ and ψ_2 the ψ*-complement* of ψ_1. Then the following hold:

- $\mathcal{A}_{\psi_1} \cap \mathcal{A}_{\psi_2} = \emptyset$
- $\mathcal{A}_{\psi_1} \cup \mathcal{A}_{\psi_2} = \mathcal{A}_{\psi}$

Proof Obvious from the definitions. □

We need one more concept:

Definition 8.6.38 Let ψ be an atom flag and S a set of $\mathcal{A}_\psi$-applicable selector functions. Then we call $|.|_{\psi,S}$ *leaf measuring* if every $s \in S$ is leaf selecting for ψ.

Proposition 8.6.39 Let ψ be an atom flag and S a set of $\mathcal{A}_\psi$-applicable selector functions. If $|.|_{\psi,S}$ is leaf measuring then $|.|_{\psi,S}$ is singularly measuring.

Proof A selector function which is leaf measuring for ψ has no $\mathcal{A}_\psi$-applicable subselectors. □

Finally, we can present the second refinement notion.

Definition 8.6.40 Let χ be an sbcmp, $((\ldots,\mathcal{A}_{\psi_i},\ldots),(\ldots,|.|_{\psi_i,S_i},\ldots))$, such that $|.|_{\psi_i,S_i}$ is leaf measuring. Let G_1 and G_2 be two goals with respective selected literals P_1 and P_2, both covered by ψ_i, such that χ is not (G_1,G_2)-decreasing. Let $s \in S_i$. Then we say that χ has (G_1,G_2,s)-*c-potential* iff the following two conditions are both satisfied:

- There is a function symbol f such that the (s,f)-refinement of ψ_i covers both P_1 and P_2.
- At least one first level subselector of s is (P_1,P_2)-decreasing.

Note that f, if it exists, is uniquely determined by s, P_1 and P_2 (and thus by s, G_1 and G_2).

Example 8.6.41 Consider the first and the second goal in figure 8.12. In other words:

$G_1 = \leftarrow p(f([a,b,c|Y],Z))$
$G_2 = \leftarrow p(f([b,c|Y],[a,a|Z]))$
$P_1 = p(f([a,b,c|Y],Z))$
$P_2 = p(f([b,c|Y],[a,a|Z]))$

Let

$\chi = ((\mathcal{A}_{p(X)}),(|.|_{p(X),\{1\}}))$

Then:

- χ is an sbcmp.
- $|.|_{p(X),\{1\}}$ is leaf measuring.

- $p(X)$ covers P_1 and P_2.
- χ is (G_1, G_2)-increasing.

It follows that χ has $(G_1, G_2, 1)$-c-potential, since:

- $p(f(X,Y))$, the $(1, f)$-refinement of $p(X)$ covers both P_1 and P_2.
- $1/1$ is a (P_1, P_2)-decreasing first level subselector of 1.

Definition 8.6.42
Take $\chi = ((\ldots, \mathcal{A}_{\psi_{i-1}}, \mathcal{A}_{\psi_i}, \mathcal{A}_{\psi_{i+1}}, \ldots), (\ldots, |\cdot|_{\psi_{i-1}, S_{i-1}}, |\cdot|_{\psi_i, S_i}, |\cdot|_{\psi_{i+1}, S_{i+1}}, \ldots))$, G_1, G_2, P_1, P_2 and s as in definition 8.6.40. Let χ have (G_1, G_2, s)-c-potential. Let f be the function symbol satisfying the first condition in definition 8.6.40. Then we define the (G_1, G_2, s)-*c-refinement* of χ to be the pair

$$((\ldots, \mathcal{A}_{\psi_{i-1}}, \mathcal{A}_{\psi'_i}, \mathcal{A}_{\psi''_i}, \mathcal{A}_{\psi_{i+1}}, \ldots),$$
$$(\ldots, |\cdot|_{\psi_{i-1}, S_{i-1}}, |\cdot|_{\psi'_i, S'_i}, |\cdot|_{\psi''_i, S_i}, |\cdot|_{\psi_{i+1}, S_{i+1}}, \ldots))$$

where:

- ψ'_i is the (s, f)-refinement of ψ_i.
- ψ''_i is the ψ_i-complement of ψ'_i.
- $S'_i = S_i \setminus \{s\} \setminus \{s' \in S_i | s' \text{ is } (P_1, P_2)\text{-increasing}\}$
  $\cup \{s'' | s'' \text{ is a } \mathcal{A}_{\psi'_i}\text{-applicable first level subselector of } s$
  $\text{which is not } (P_1, P_2)\text{-increasing}\}$.

Example 8.6.43 Continuing example 8.6.41, we obtain

$$((\mathcal{A}_{p(f(X,Y))}, \mathcal{A}_{p(\{f\})}), (|\cdot|_{p(f(X,Y)), \{1/1\}}, |\cdot|_{p(\{f\}), \{1\}}))$$

as the $(G_1, G_2, 1)$-c-refinement of χ.

Just like m-refining (see proposition 8.6.32), c-refining is well-defined.

Proposition 8.6.44 Let χ, G_1, G_2, P_1, P_2 and s be as in definition 8.6.42. Let χ' be the (G_1, G_2, s)-c-refinement of χ. Then:

1. $\chi' \neq \chi$
2. χ' is an sbcmp.
3. χ' is (G_1, G_2)-decreasing.

Proof

1. Obvious from the definition.

2. Let χ and χ' be denoted as above in definition 8.6.42. We have to address three properties of an sbcmp (definition 8.6.26):

   - $\ldots, \mathcal{A}_{\psi_{i-1}}, \mathcal{A}_{\psi'_i}, \mathcal{A}_{\psi''_i}, \mathcal{A}_{\psi_{i+1}}, \ldots$ is a partition of the set of atoms with a recursive predicate symbol. This follows from the fact that χ is an sbcmp, and from propositions 8.6.21 and 8.6.37.
   - We know that any S_j in χ contains only $\mathcal{A}_{\psi_j}$-applicable elements. Now, definition 8.6.35 implies that the selector functions in S_i are also $\mathcal{A}_{\psi''_i}$-applicable. Moreover, it follows from definition 8.6.34 that they are $\mathcal{A}_{\psi'_i}$-applicable, which means that the latter property holds for any element in S'_i.
   - Finally, the second part of the proof for proposition 8.6.45 below shows that all measure functions in χ' are leaf measuring. So, they are singularly measuring (proposition 8.6.39).

3. Since P_1 and P_2 are covered by ψ', we have to show:
   $|P_1|_{\psi'_i,S'_i} > |P_2|_{\psi'_i,S'_i}$
   This follows from the fact that S'_i contains no (P_1, P_2)-increasing and at least one (P_1, P_2)-decreasing selector function.

$\square$

Finally, the following property holds:

Proposition 8.6.45 Let χ be an sbcmp, χ' an m- and χ'' a c-refinement of χ. If each measure function in χ is leaf measuring, then so is each measure function in χ' and each measure function in χ''.

Proof

- In definition 8.6.30, S'_i is derived from S_i through the deletion of one or more selectors. Obviously, the remaining ones are still leaf selecting for ψ_i. This proves the result for χ'.

- For χ'', the proof is slightly more complex. Let ψ be an atom flag, s a selector function which is leaf selecting for ψ and f a function symbol. Let ψ' be the (s, f)-refinement of ψ and ψ'' its ψ-complement. Let $|.|_{\psi,S}$ be the measure function in χ which is replaced by $|.|_{\psi',S'}$ and $|.|_{\psi'',S}$ in χ'', where S' is derived from S as stipulated in definition 8.6.42. Then we have to show that $|.|_{\psi',S'}$ and $|.|_{\psi'',S}$ are leaf measuring.

  - ψ' is identical to ψ, except at $s(\psi')$, where a var or a list flag has been replaced by a functor flag. This implies that all elements of $S \setminus \{s\}$ are leaf selecting for ψ'. Finally, since $s(\psi')$ is a functor flag with only var subflags, $\mathcal{A}_{\psi'}$-applicable first level subselectors of s are likewise leaf selecting for ψ'.

- Since ψ'' is derived from ψ by replacing a var flag by a set flag, or by extending a set flag, any selector function which is leaf selecting for ψ is also leaf selecting for ψ''.

□

An algorithm

Algorithm 8.6.46 below incorporates an automatic unfolding method with a built-in "focusing on subterms" facility. It is presented in the same overall style as algorithm 8.4.4 in subsection 8.4.2, for ease of comparison. However, in the present complex context, it seems very reasonable to provide, as has been suggested before, a separate sbcmp for each proper achain (see definition 6.4.3 on page 103). In other words, below, sbcmps as well as sbcmp refinements are local to proper achains. Moreover, the whole algorithm is designed in such a way that it is guaranteed to terminate. Some variants, not enjoying this property, will be discussed below.

Before we proceed, we introduce some further terminology:

- It can be seen from definition 6.4.7 that the notion of one goal covering another depends on the atom class partition in use. Until now, in our algorithms, this could be safely left implicit because this partition was stable throughout the algorithm (and even always chosen in the same way). However, this is no longer the case. So, for clarity, below, we will explicitly mention the considered sbcmp and refer to a *χ-covering* ancestor.

- Suppose χ is an sbcmp and χ' its (G_1, G_2, s)-c-refinement with $\mathcal{A}_{\psi,''}$ as freshly introduced complement class. Then we say that G_1 *pseudo covers* proper descendant goals with selected literal in $\mathcal{A}_{\psi,''}$. (But no longer goals with selected literal in any refinements or refinement complements of $\mathcal{A}_{\psi,''}$.) Moreover, the weight of G_1 under χ will be called the *master weight* associated with $\mathcal{A}_{\psi,''}$. It follows that a goal has at most one pseudo covering ancestor and associated master weight. Both notions play an important role in restricting the behaviour of algorithm 8.6.46 in such a way that its termination is ensured.

- Selectability condition (2) in algorithm 8.6.46 abuses terminology: It demands that an sbcmp should be (W, G)-decreasing, where W is not a goal, but a master weight. The intention is obvious: the weight of G under χ should be strictly smaller than W.

```
Algorithm 8.6.46
Input
  a definite program $P$
  a definite goal $\leftarrow A$
Output
  an SLD-tree $\tau$ for $P \cup \{\leftarrow A\}$
Initialisation
  $\tau$ is initialised as the SLD-tree that contains a single derivation,
    consisting of the goal $\leftarrow A$, without selected literal.
  One starts with a single sbcmp, composed as follows:
    For each recursive predicate $p/n$ in $P$:
      – an atom class $\mathcal{A}_{p(X_1,\ldots,X_n)}$
      – a measure function $|.|_{p(X_1,\ldots,X_n),\{1,\ldots,n\}}$
    No other atom classes or measure functions.
While there exists a non-terminated derivation $D$ in $\tau$ do
  If $D$ is succesful, Then terminate $D$
  Else If $D$'s leaf-node contains no selectable atom, Then terminate $D$
  Else
    select the leftmost selectable atom
    If no derivation steps are possible, Then terminate and fail $D$
    Else extend $D$
  Where an atom $p(t_1,\ldots,t_n)$ in a goal $G$ is selectable
  if one of the following conditions holds in case it is actually selected:
    (Let $\chi$ name the sbcmp currently controlling the proper achain
    to which $G$ belongs if $p(t_1,\ldots,t_n)$ is selected.)
    (1)  $G$ has neither a $\chi$-covering nor a pseudo covering ancestor.
    (2)  $G$ has no $\chi$-covering ancestor and
         $G'$ is its pseudo covering ancestor and
         $W$ is its associated master weight and
         $\chi$ is $(W, G)$-decreasing.
    (3)  $G'$ is the direct $\chi$-covering ancestor of $G$ and
         $\chi$ is $(G', G)$-decreasing.
    (4)  $G'$ is the direct $\chi$-covering ancestor of $G$ and
         $\chi$ has $(G', G)$-m-potential.
    (5)  There are a goal $G'$ in $D$ and a selector function $s$ such that
           $\chi$ has $(G', G, s)$-c-potential and
           $G'$ is the direct $\chi'$-covering ancestor of $G$,
             where $\chi'$ is the $(G', G, s)$-c-refinement of $\chi$.
  If an atom $p(t_1,\ldots,t_n)$ has been selected on the basis of condition (4) or (5),
    Then replace $\chi$ by its corresponding refinement.
Endwhile
```

The following proposition shows that algorithm 8.6.46 is well constructed in an important sense.

Proposition 8.6.47 Throughout algorithm 8.6.46, the pairs of atom classes and measure functions underlying its operation, are sbcmps, all of whose measure functions are leaf measuring.

Proof

- The property is obviously satisfied after initialisation.
- Propositions 8.6.32, 8.6.44 and 8.6.45 guarantee that the property is preserved by m- as well as c-refinements.

□

So, χ in the above algorithm is sufficiently well-behaved; it always satisfies the basic preconditions for both refinement operations.

We include two simple examples, illustrating some basic aspects of algorithm 8.6.46's operation.

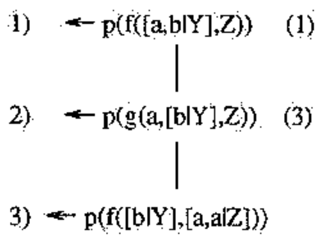


Figure 8.13: Initial unfolding.

Example 8.6.48
Reconsider:

$p(f([X|Y], Z)) \leftarrow p(g(X, Y, Z))$
$p(g(X, Y, Z)) \leftarrow p(f(Y, [X, X|Z]))$

with the query:

$\leftarrow p(f([a, b|Y], Z))$

Algorithm 8.6.46 builds an SLD-tree with one SLD-derivation that contains just a single proper achain. Let χ name its sbcmp, then initially:

$\chi = ((\mathcal{A}_{p(X)}), (|\cdot|_{p(X),\{1\}}))$

A first part of the SLD-derivation produced by algorithm 8.6.46 is shown in figure 8.13. In this and the following figures, goal nodes are labeled by a number to

their left. The labels at the right indicate the selectability condition (according to the list in algorithm 8.6.46) satisfied upon unfolding by the (single) literal in the goal.
If we select the sole literal in goal 3), its direct χ-covering ancestor is goal 2). We observe that χ is not $(2,3)$-decreasing, nor does it have $(2,3)$-m-potential. However, there is $(1,3,1)$-c-potential, and after c-refinement, we obtain:

$$\chi = ((\mathcal{A}_{p(f(X,Y))}, \mathcal{A}_{p(\{f\})}), (|\cdot|_{p(f(X,Y)),\{1/1\}}, |\cdot|_{p(\{f\}),\{1\}}))$$

This enables further unfolding as indicated in figure 8.14.

$$\begin{tikzcd}[row sep=small] \vdots \arrow[d, dash, dotted] \\ 3)\ \leftarrow p(f([b|Y],[a,a|Z]))\quad (5) \arrow[d, dash] \\ 4)\ \leftarrow p(g(b,Y,[a,a|Z])) \end{tikzcd}$$

Figure 8.14: Continued unfolding.

Now, goal 2) is the direct χ-covering ancestor of goal 4). Again, c-refining is the only way to continue unfolding. We replace χ by its $(2,4,1)$-c-refinement and obtain:

$$\chi = ((\mathcal{A}_{p(f(X,Y))}, \mathcal{A}_{p(g(X,Y,Z))}, \mathcal{A}_{p(\{f,g\})}), \\ (|\cdot|_{p(f(X,Y)),\{1/1\}}, |\cdot|_{p(g(X,Y,Z)),\{1/1,1/2\}}, |\cdot|_{p(\{f,g\}),\{1\}}))$$

The resulting additional unfolding steps are depicted in figure 8.15.

$$\begin{tikzcd}[row sep=small] \vdots \arrow[d, dash, dotted] \\ 4)\ \leftarrow p(g(b,Y,[a,a|Z]))\quad (5) \arrow[d, dash] \\ 5)\ \leftarrow p(f(Y,[b,b,a,a|Z]))\quad (3) \arrow[d, dash] \\ 6)\ \leftarrow p(g(X',Y',[b,b,a,a|Z])) \end{tikzcd}$$

Figure 8.15: Concluding unfolding.

Goal 6), constituting the leaf in figure 8.15, contains no selectable literal. Unfolding stops and algorithm 8.6.46 terminates.

Example 8.6.49
Consider the following 2 goals:

- $G_1 = \leftarrow p(f([a|X]), Y)$
- $G_2 = \leftarrow p(f(X), [a|Y])$

And suppose that
$\chi = ((\mathcal{A}_{p(X,Y)}), (|\cdot|_{p(X,Y),\{1,2\}}))$
Then χ is not (G_1, G_2)-decreasing, but it has both (G_1, G_2)-m-potential and $(G_1, G_2, 1)$-c-potential. Refining gives the following results:

m: $((\mathcal{A}_{p(X,Y)}), (|\cdot|_{p(X,Y),\{1\}}))$

c: $((\mathcal{A}_{p(f(X),Y)}, \mathcal{A}_{p(\{f\},X)}), (|\cdot|_{p(f(X),Y),\{1/1\}}, |\cdot|_{p(\{f\},Y),\{1,2\}}))$

Example 8.6.49 indicates that algorithm 8.6.46 is non-deterministic; its conditions for literal selectability are not mutually exclusive. More precisely, conditions (4) and (5) might both be satisfied upon selection of a certain literal in a given goal. Moreover, it is also possible that there are more than one G' and s such that χ has (G', G, s)-c-potential. We present some brief remarks on these and various other issues below.

Let us first prove the termination of algorithm 8.6.46.

Theorem 8.6.50 Algorithm 8.6.46 terminates. The resulting SLD-tree τ is finite.

Proof We must prove that algorithm 8.6.46 can not produce an infinite SLD-derivation. To this end, it suffices to show that every proper achain must necessarily be finite (proposition 6.4.4).
So, consider a proper achain C constructed by algorithm 8.6.46. Every non-leaf goal in this chain has a selected literal. We will call such a goal *of kind* 1 *to respectively* 5 if its selected literal was chosen on the basis of condition (1) to respectively (5) in algorithm 8.6.46. The proofs of the following two statements are analogous to what was presented above:

- If there is goal G of kind 4 or 5 in C which has no descendant goal in C of either kind 4 or 5 (in other words, literal selection in G caused the last sbcmp refinement), then G has only a finite number of descendants in C.
- Between two kind 5 goals, C contains only a finite number of kind 4 goals. (In other words, only a finite amount of m-refinements is possible between two c-refinements.)

It remains to be shown that C can contain only finitely many goals of kind 5 (i.e. c-refinement can occur only finitely often).
We introduce some terminology. Consider an atom flag ψ. Then we can associate a *level* with term flags occurring in ψ as follows. The arguments of ψ's predicate symbol are its level 1 term flags. If ϕ is a level n functor flag in ψ, then the arguments of its function symbol are level $n+1$ term flags in ψ. Then let the ***depth of an atom flag*** be the maximum among the levels of its term flags, and ***the depth of a set of atom flags*** be the maximum among the depths of its atom flags. In particular, we can ***associate a depth with the atom class partition (acp)*** $(\ldots, A_{\psi_i}, \ldots)$ of an sbcmp.
Now, there are only finitely many predicate and function symbols in the program and goal serving as input to algorithm 8.6.46, each of which is of finite arity. Therefore, there can be only finitely many different acps of a certain depth. Next, we observe that of the two sbcmp changing operations in algorithm 8.6.46, m-refinement does not influence the acp, and c-refinement changes it into a fresh (not used before in C) acp, the depth of which is either equal or larger than that of the current acp. So, it suffices to prove that there is ***an upper bound to the possible depth of the acp in C's sbcmp.*** In particular, since a flag's depth can only increase through c-refinement, it suffices to establish an upper bound for the depth of the refined flag after c-refinement in C.
We in fact show that for any recursive predicate symbol, there is one such upper bound for atom flags with this predicate symbol. Clearly, the maximum of such predicate-wise upper bounds constitutes an upper bound for the overall acp depth. So, let p be a recursive predicate symbol. Call goals in C the selected literal of which contains p, p-goals, and atom flags with p, p-flags. If there is no p-goal, than the required result follows trivially: the depth of the sole p-flag occurring in any sbcmp generated for C is simply 0. So, let us assume that there is at least one p-goal. Now every p-goal is assigned a weight under any sbcmp for the given language. However, one weight is particularly interesting: ***the weight which resulted under the sbcmp in force when the literal was actually selected for goals of kind*** 1 ***to*** 3, ***or the sbcmp produced by refinement for goals of kind*** 4 ***and*** 5.
We introduce some notation. Suppose that G is a p-goal, then, in the context of this proof, it will be convenient to denote the weight of G under some sbcmp χ as $\chi(G)$. The above characterised sbcmps and weights will be denoted by χ_G and $\chi_G(G) = w_G$ respectively. Next, the atom flag in χ that covers the selected literal in G, will be referred to as ψ_G and its depth as d_G. We show that ***there is a bound on the d_G values.***
Let H be the first (i.e. top-most in C) p-goal. We prove that for any p-goal G:

$$d_G + w_G \leq w_H \qquad (*)$$

thus exhibiting w_H as such a bound. Since H is the first p-goal, there is just a single p-flag in χ_H and its depth equals 0. It follows that $d_H = 0$ and we obtain

$(*)$ for $G = H$. Let now $(*)$ be satisfied for all p-goals occurring before a certain p-goal $G \neq H$, then we show that it also holds for G. The proof proceeds through a case analysis on the kind of G.

1. *G can not be of kind* 1. Indeed, if G has a selected literal in some atom class $\mathcal{A}$ of χ_G, then since C is a proper achain, any ancestor p-goal with a selected literal in $\mathcal{A}$ χ_G-covers G. So, in order to be of kind 1, G must be the first p-goal with a selected literal in $\mathcal{A}$. Moreover, $\mathcal{A}$ can not be a class created as a complement class at some c-refinement, since then there would at least be a pseudo covering ancestor. But neither can $\mathcal{A}$ be a non-complement class created at c-refinement, because that operation ensures that some goals with selected literal in such a class are already present. So, H is the only kind 1 p-goal.

2. If G is of kind 2, it has a pseudo covering ancestor G' and associated master weight W, such that $w_G < W$. Moreover, the definition of c-refinement entails that $W = w_{G'}$ (and therefore $w_G < w_{G'}$) as well as $d_G = d_{G'}$. For G, $(*)$ now follows from $(*)$ for G'.

3. A kind 3 goal G has a direct χ_G-covering ancestor G' such that $w_G < \chi_G(G')$. Moreover, $\psi_G = \psi_{G'}$ and $\chi_G(G') = \chi_{G'}(G') = w_{G'}$. So, again $w_G < w_{G'}$ and $d_G = d_{G'}$.

4. In case G is a kind 4 goal, there again is a χ_G-covering ancestor G' for which $\psi_G = \psi_{G'}$ (and therefore $d_G = d_{G'}$) holds. Moreover, $w_G < \chi_G(G') \leq \chi_{G'}(G') = w_{G'}$.

5. Finally, suppose G is a kind 5 goal. Then there is a χ_G-covering ancestor G' such that $w_G < \chi_G(G') \leq w_{G'}$. Moreover, $d_{G'} \leq d_G \leq d_{G'} + 1$. Since $(*)$ holds for G', it again follows for G.

□

Assorted remarks

A number of brief comments on various topics conclude section 8.6.3 on automation.

First, in our study of automation, we have remained within the general line of development in this thesis. Indeed, we have formalised an approach not relying on any substantial offline analysis. This led to the introduction of the c-refinement notion which enables dynamically focusing on interesting subarguments. However, c-refining or dynamically changing a cmp's atom class partition in general are quite complex operations that involve a number of subtleties. Alternative approaches to automation, avoiding these complications, are conceivable. Offline analysis techniques can perhaps statically determine interesting (sub)arguments, thus resulting in a smarter cmp initialisation and eliminating the need for (c-)refinements.

Next, definition 8.6.26 demands a cmp to contain only singularly measuring measure functions. This limitation reduces complexity and seems very reasonable. However, definition 8.6.40 imposes the more stringent condition that the relevant measure function be *leaf measuring*. And proposition 8.6.47 shows that algorithm 8.6.46 ensures this property for any measure function actually used while running it. Moreover, definition 8.6.42 only caters for replacement of a selector function by one or more of its *first level* subselectors. Finally, *only one* selector function can be replaced by subselectors. These choices are debatable. Their main underlying motivation is our wish to reduce the overall complexity of the presentation above. Less restrictive definitions require more complex atom class manipulation, neatly splitting a class in a refined class and its complement no longer being universally sufficient.

We have already pointed out that algorithm 8.6.46 is *non-deterministic*. Indeed, more than one c-refinement option might be available for the same literal, candidate for selection. Again, this is caused by our wish to keep the definitions simple. Furthermore, it is not completely obvious that all freedom of choice in this context can be eliminated in a reasonable way. Another source of non-determinism is the lack of mutual exclusivity between the conditions for m- and c-refinement. (It can even be observed that an sbcmp with m-potential usually also has c-potential, e.g. in a list processing context.) Here, the obvious solution seems attaching priority to the more straightforward operation of m-refinement. However, this does entail a loss of generality in the processing of any atoms which c-refinement might have put into the resulting complement atom class: for this class, no selector functions are removed from the measure function.

Next, algorithm 8.6.46 has been carefully designed in such a way that its termination is ensured. Some *more liberal variants* can be considered.

- First, the notion of a pseudo covering ancestor can be left out, as in algorithm 7.50 in [123]. Whether this implies possible non-termination is as yet not clear to us.

- A further step leads to a variant of algorithm 8.6.46 which is obviously *not* guaranteed to terminate. Indeed, a definition of c-refinement can be imagined without imposing the condition that the resulting sbcmp should be decreasing on the considered goal with respect to its new direct covering ancestor. Instead, not having a covering ancestor would also be all right. This modification fits neatly into the overall framework, but, in general, it destroys termination. Having a clause as simple as the following suffices:

  $p(X) \leftarrow p(f(X))$

  Imposing a depth bound on flag structure is an obvious solution, but would allow ad hoc termination behaviour, not always preventing unfolding beyond reasonable limits. The latter phenomenon is avoided by the algorithm as it stands, which can of course be enhanced with one or more depth bounds, if so desired. However, in some cases, it seems to be too weak. (See example 8.6.53 below.)

- A third possible way to proceed, perhaps offering both termination and more flexible unfolding, is briefly outlined towards the end of the next subsection.

Finally, it is clear that checking for c-potential and carrying out c-refinements can be quite costly operations. They require scanning and adapting complete chains of covering ancestors.

8.6.4 Unfolding meta-interpreters

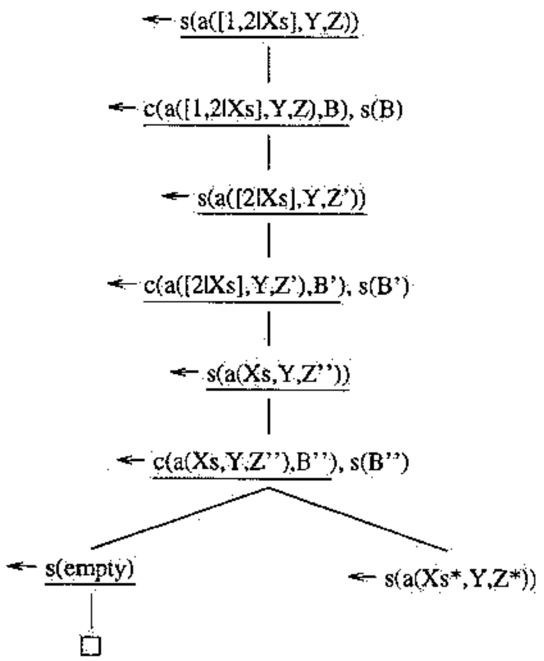

← s(a([1,2|Xs],Y,Z))
← c(a([1,2|Xs],Y,Z),B), s(B)
← s(a([2|Xs],Y,Z'))
← c(a([2|Xs],Y,Z'),B'), s(B')
← s(a(Xs,Y,Z''))
← c(a(Xs,Y,Z''),B''), s(B'')
← s(empty)
← s(a(Xs*,Y,Z*))

Figure 8.16: Meta-interpreting *append*.

To conclude section 8.6, we return to the issue of unfolding meta-programs, and discuss the operation of algorithm 8.6.46 on some examples. In particular, we consider the vanilla meta-interpreter for definite programs, introduced in definition 4.6.10 on page 61. We encode various simple object programs and consider the resulting unfolding behaviour for relevant meta level queries. For convenience, we will abbreviate predicate and function names in figures.

Example 8.6.51 Our first example concerns the standard *append* program for concatenating two lists. It is encoded at the meta level as follows:

$clause(append([], X, X), empty) \leftarrow$

$clause(append([X|Xs], Y, [X|Z]), append(Xs, Y, Z)) \leftarrow$

Consider now as starting query:

$\leftarrow solve(append([1, 2|Xs], Y, Z))$

Then algorithm 8.6.46 produces the SLD-tree depicted in figure 8.16. Comparing this tree with figure 7.1, we conclude that the result is entirely satisfactory. It is obtained without executing any refinement. So, all proper achains have an associated sbcmp, identical to the one created at initialisation:

$((\mathcal{A}_{solve(X)}), (|\cdot|_{solve(X),\{1\}}))$

Example 8.6.52 Next, reconsider the *reverse* program for reversing a list, using an accumulating parameter:

$clause(reverse([], X, X), empty) \leftarrow$

$clause(reverse([X|Xs], Y, Z), reverse(Xs, [X|Y], Z)) \leftarrow$

and the following query:

$\leftarrow solve(reverse([1, 2|Xs], [], Z))$

The initial sbcmp is of course:

$((\mathcal{A}_{solve(X)}), (|\cdot|_{solve(X),\{1\}}))$

Using such an sbcmp, we can perform the unfoldings shown in figure 8.17.

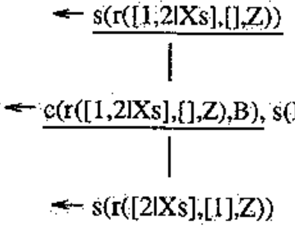


Figure 8.17: The need for c-refinement.

At this point, c-refinement is the only way to continue. It produces the following refined sbcmp:

$$((\mathcal{A}_{solve(reverse(X,Y,Z))}, \mathcal{A}_{solve(\{reverse\})}), \\ (|\cdot|_{solve(reverse(X,Y,Z)),\{1/1,1/3\}}, |\cdot|_{solve(\{reverse\}),\{1\}}))$$

Proceeding with this sbcmp, algorithm 8.6.46 carries out all desired unfoldings, and then terminates.

Algorithm 8.6.46 deals aptly with example 8.6.52. However, let us consider, in the following example, a slightly more complicated object program, the so-called "naive" *reverse* program. It differs from the above two examples in that it requires non-trivial parsing.

Example 8.6.53

$clause(reverse([],[]), empty) \leftarrow$
$clause(reverse([X|Xs], Y), reverse(Xs, Ys)\&append(Ys, [X], Y)) \leftarrow$
$clause(append([], X, X), empty) \leftarrow$
$clause(append([X|Xs], Y, [X|Z]), append(Xs, Y, Z)) \leftarrow$
$\leftarrow solve(reverse([1,2|Xs], Y))$

Using the same initial sbcmp again, we halt at the leaf in figure 8.18.

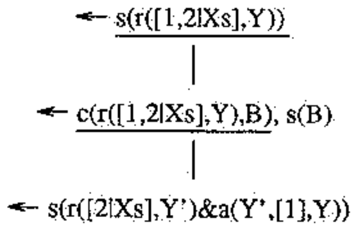


Figure 8.18: Algorithm 8.6.46 is too weak.

The performance of algorithm 8.6.46 on example 8.6.53 is clearly unacceptable. Its above discussed second variant (allowing c-refining also in the absence of a properly covering ancestor) gives better results. However, in general, it requires an (ad hoc) depth bound. Again, it seems possible that some form of program analysis can infer, for a given program (and query), sensible depth bounds. When e.g. abstract interpretation shows that for the given query, all terms in subsidiary calls necessarily are of bounded complexity, no depth bounds are necessary, and algorithm 8.6.46's more powerful variant can safely be used. Such methods might also provide valuable information in a context where meta-programs more complex than the above vanilla interpreter are to be dealt with. (See e.g. subsection 6.5.3.)

An alternative online approach, offering guaranteed termination as well as sufficiently flexible unfolding, might also be feasible. It would combine the following two novel ingredients:

- No notion of pseudo covering ancestor would be introduced and creating atom classes would be allowed even in the absence of a covering ancestor. However, such atom classes, lacking an initial covering ancestor, should only be further refined if the depth of their flag (see the proof of theorem 8.6.50 for the definition of this notion) does not increase. In this way, it seems likely that termination can be recovered.

- Additional flexibility in unfolding would be provided by exploiting mutual recursion. Indeed, one could perhaps within a proper achain require global weight decrease for just one (or more) classes of goals and demand decreases for other classes only "locally" if the goal's selected literal descends directly (see definition 6.4.2) from that of its direct covering ancestor.

In further work, we hope to investigate whether safe and powerful online unfolding of meta-interpreters can actually be based on the above ideas.

Finally, it can be mentioned that the particular problem of controlling unfolding during partial deduction of meta-programs has been explicitly considered in [103]. The proposed approach relies on annotations to certainly allow unfolding sufficient to remove the parsing overhead. This however presupposes a distinction between meta-interpreters and "ordinary" programs and a clearly distinguishable parsing component, immune to non-terminating unfolding. These characteristics can indeed be observed in (correctly written) vanilla-like meta-interpreters, but it is not obvious that they will hold for any program a general partial deduction system might be applied to. We have therefore investigated another approach, not singling out meta-interpreters as being special in any sense. Our succint development above shows some successes, but also leaves open questions, to be addressed in future research.

8.7 Discussion and Conclusion

In this chapter, we have elaborated in detail the issue of finite unfolding. We have built on the basic framework presented in chapter 6 and shown various instances of it, capable of dealing with some advanced unfolding issues. We have consistently avoided ad hoc solutions as much as possible, always trying to concentrate on meaningful properties of the task at hand. Particular emphasis was put on the development of fully automatic unfolding algorithms, requiring no user assistance apart from providing a program and a query to be unfolded. Moreover, we have opted for a general, highly formalised presentation, trusting that this choice provides better chances for uncovering common underlying principles. In this way, we believe that we have enhanced the basic framework with a good understanding of at least some issues involved in automatic, maximal, sensible, finite unfolding. Finally, we have shown that weight based unfolding often can be made to enjoy good complexity properties, its execution effort being linear in the size of the generated SLD-tree when a clever implementation scheme is used.

A first major part of this chapter concentrates on methods relying on structure based weights assigned to arguments upon which lexicographic priorities were imposed. Much of the inspiration for this approach came from work on termination of rewrite systems (see e.g. [48]). Well-founded and related orderings also play a major role in that context, and the use of lexicographic priorities is common practice. However, since there are some important differences between a set of rewrite rules and a logic program, and most of the work in the former context is on *static* termination analysis, we have decided against an attempt to literally translate approaches. Extensive comments on the relationship between rewrite systems and logic programs, seen from the perspective of (static) termination analysis, can be found in [40].

The possible use of lexicographic well-founded orderings to control the unfolding of *logic programs* is already mentioned in [60]. To the best of our knowledge, however, our work is the first to carry out complete formal developments and present concrete algorithms based on this idea. We also believe that the work in section 8.3 is the first to present concrete algorithms basing unfolding decisions in logic programs not solely on the shape of a goal's selected literal, but also on contextual information in the rest of the goal. We have shown how our techniques can deal with co-routining and, to a certain extent, with back propagation of variable instantiations. To be sure, a number of challenging issues remain as subjects for further research; we refer to subsections 8.3.3 and 8.3.4 for more detailed comments.

Next, in section 8.5, we have integrated into our framework, the well-known heuristic that a literal can be unfolded if it is not a variant of one already unfolded in the same derivation. As a stand-alone method, this heuristic does not provide safe unfolding. Combined approaches guarantee termination when specific conditions are fulfilled. Our work can be extended to reach more generally satisfied conditions for termination, or specific classes of instances of the framework can be studied in further detail, thus perhaps establishing termination in spite of the fact that the above mentioned general conditions are not satisfied. Further experimental work is needed to assess unfolding power in complex cases, comparing different instances of the framework. Another issue that merits further attention are the performance characteristics of integrated methods. We already pointed out that the above mentioned linearity property is lost. It may be that this has little practical consequences, but re-establishing linearity would certainly be of interest. An attempt in this direction might start from an idea mentioned in [21], where it is suggested that some order could be inferred among constants in a datalog program on the basis of the program's atom dependency graph. Such an offline program analysis might enable the definition of a measure function again just requiring a comparison with the direct covering ancestor to decide on unfolding. Whether this idea actually holds water, is as yet unclear to us. More generally, it seems very likely that various sophisticated offline analysis techniques might provide useful supporting information for online unfolding as studied in this thesis.

Section 8.6, finally, explores the boundaries of what is possible within our basic framework for online weight based finite unfolding. It presents an algorithm, not only refining measure functions dynamically, but also occasionally recalibrating the partition of SLD-derivations in chains of covering nodes. It blurs the distinction between predicate and function symbols, in this way closing part of the gap between logic programming on the one hand, and functional programming as well as term rewriting on the other. Our main motivation for this part of the work was the desire to handle meta-interpreters properly. Unfolding behaviour for such programs should be determined as much as possible by the evolution of subterms, representing the arguments of object level predicates. Since our ultimate aim is a fully general, completely automated partial deduction system, we have investigated an approach not making any a priori distinction between meta- and other programs. However, as we already mentioned, a number of open issues remain as topics for further research.

Finally, we wish to point out that, in the formulation of the algorithms (except algorithm 8.6.46), we have consistently reduced indeterminism to a minimum: summing argument weights per component in partitions, choosing the leftmost unfoldable literal, imposing the choice of one particular decreasing refinement, limiting the range of possible measure function switches through a "narrow" definition of the refinement concept. Only the choice of a non-terminated derivation, candidate for extension is left open. We have not studied in detail whether some of these decisions sometimes reduce unfolding potential. However, we conjecture that this will hardly ever be the case. We therefore preferred to formulate algorithms in such a way that their implementation requires a minimum of extra design decisions. (On the other hand, the development of the basic framework and the generic treatment in section 8.4 have both been kept very general.) An exception is clearly algorithm 8.6.46, which has been presented more or less in the same style as algorithm 8.4.4. The former incorporates automatic unfolding of a kind outside the generic framework built in section 8.4. Ease of comparison with algorithm 8.4.4 was therefore a major concern while formulating it. Moreover, we also feel that some subtle design decisions are involved in further refining algorithm 8.6.46 (or a more powerful variant). So, at this early stage, we preferred to leave them open.

Chapter 9

General Discussion and Conclusion

In this thesis, we have addressed the semantics of meta-programming and the control of partial deduction in logic programming. Both issues meet in the practical, efficient use of meta-interpreters.

In part I, we have studied the Herbrand semantics of vanilla-like, untyped, non-ground meta-programs. First, we have shown that the vanilla meta-program associated with a stratified normal object program is weakly (but not locally) stratified, in this way identifying an important application area for the notions of a weakly stratified program and a weakly perfect model. We contend that weak stratification is not only a more general, but also a much more natural extension of stratification than local stratification is. In that context, lemma 3.3.12 and its proof in appendix A are interesting. Indeed, we conjecture that they provide the essential ingredients for a completely general proof showing that a weakly stratified program possesses a unique weakly perfect model, independent of the particular (successful) "weak stratification" used to obtain it. In other words, definition 3.3.7 can be reformulated to allow the use of non-maximal layers as in proposition 3.3.11. This is an important issue from a practical point of view, and one that was, to the best of our knowledge, not addressed before.

Returning to the subject of meta-programming, we have introduced language independence as a generalisation of range restriction, and shown that it plays a key role in the classical ground Herbrand semantics of untyped meta-programs using the non-ground representation for object level variables. Indeed, for language independent object programs, there is a natural correspondence between the perfect model of the object program and the weakly perfect model of its associated vanilla meta-program.

Next, we have investigated to what extent these results can be generalised for a class of meta-programs that are (more useful) extensions of *vanilla*. It turned out that, in general, language independence again ensures good results for definite programs, but that the concept is too lax to deal properly with normal programs. Range restriction rectifies the situation. Of course, as we mentioned, further research can refine the classification of meta-programs and strengthen the results obtained in this thesis.

We have also considered several kinds of amalgamation, welding together object and meta-program through a very simple and straightforward overloading technique for function and predicate symbols. It is mainly in applications of this kind that the non-ground approach to meta-programming might be preferable to the ground approach, since the latter tends to impose more strict separations between various "layers" of reasoning. Our work indicates a sensible semantics for some limited, but nevertheless interesting forms of amalgamation.

Finally, in the context of S-semantics, an extended "non-ground" Herbrand semantics for definite logic programs, we were able to generalise our basic result beyond the class of language independent object programs. But language independence did resurface in the treatment of extended meta-interpreters.

Various topics for further research were mentioned throughout chapters 3 and 4, and particularly in section 4.9. Let us just mention that we judge the development of an elegant and powerful semantical framework, allowing both full amalgamation and more general object level formulas, to be a particularly challenging and intriguing issue. With the work in this thesis, we hope to have contributed to the quest for this ultimate goal.

Subsequently, in part II of the thesis, we turned our attention to partial deduction of (definite) logic programs and the control of unfolding in that context.

A first contribution on the latter topic is the development of a general framework within which algorithms for finite unfolding can be formulated. Tuning the notion of a well-founded set and introducing that of a well-founded measure, we provided a general and practically useful characterisation of finite SLD-trees. Building on that foundation, we presented two operational approaches of increasing sophistication. In particular, the notion of a hierarchical prefounding provides a refined treatment of recursion and was therefore used as the basis for a semi-automatic general algorithm for finite unfolding.

In section 7.2, but mainly in chapter 8, several fully automatic unfolding algorithms were derived from this template. First, measure functions were introduced that base unfolding decisions on (sub)sets of the selected literal's argument positions. And it was shown how these can be tuned dynamically during the construction of an SLD-tree. Next, more refined measuring used ordered partitions, first solely concentrating on the literal, candidate for selection, but subsequently enlarging the perspective to also take into account context information in the remainder of a considered goal. The latter capability allows a treatment of phenomena such as coroutining and instantiation back propagation. We also investigated a modification of the basic framework that enables a combination of weight based unfolding with the classic non-variant checking. Experiments showed that the practical importance of this enhancement is considerable: The tested combined method performed very well. Finally, we explored automatic focusing on subarguments.

Throughout, we have particularly aimed at a precise mathematical formalisation of the occasionally quite complex operations involved. As a result, we were able to formally investigate interesting (termination) properties of the various algorithms. Moreover, we were also able to gain a good insight into the underlying issues involved in the search for optimal measure functions, used to control unfolding.

Most of chapter 7 is devoted to the topic of overall partial deduction, of which constructing finite SLD-trees is one important component. An algorithm was presented that, using finite unfolding with set based measures, performs sound and complete partial deduction and always terminates. Moreover, we performed experiments, blending in different ways various techniques to control partial deduction and unfolding, and compared the performance of the resulting overall methods on some benchmark programs.

An important aspect of our work is the following: virtually all control decisions are taken online, during partial deduction/unfolding. Apart from an elementary (syntactic) identification of recursive predicates, no preliminary offline analysis of any kind is included. We have, however, on several occasions discerned the limits of such an approach. Further work can therefore clearly be mounted in that direction. On the other hand, in subsection 8.6.4, we outlined some further ideas that might enable more powerful, and yet still safe, online unfolding of meta-programs. Other interesting topics for future research were mentioned in subsection 7.5.3. They include a refined control of the set of atoms to be partially deduced and techniques to obtain operationally optimal amounts of unfolding. Of course, also the extension of our work to programs including negation stands as a non-trivial task.

Finally, a promising application, drawing together the two major threads in this thesis, lies in the area of update related integrity checking. Effective methods for verifying integrity constraints in deductive databases upon an update have been extensively studied; see e.g. [44], [115], [150], [120], [29], [30]. Gallagher ([63]) suggested how the whole process can be seen as an application of partial deduction to meta-programs. However, the respective formalisations offered in [115] and [114] are quite distinct. It will be interesting to elaborate this subject, both from a formal and an operational point of view.

Appendix A

Proof of Lemma 3.3.12

Throughout this appendix, P will denote some normal program. Observe that, for any normal program P, $Ground(P)$ is a (possibly infinite, but countable) set of ground clauses. It is then possible and convenient to associate a particular natural number with each clause in $Ground(P)$. So, we will occasionally refer to a clause in $Ground(P)$ as being a couple (C, n_C) where no two clauses in $Ground(P)$ have the same associated n_C. In this way, we can in the construction in proposition 3.3.11, always identify the unique clause in $Ground(P)$ from which a clause in a given P_i is derived (through the deletion of true body literals): they have identical labels. In the same way, correspondences between clauses in P_i-sets in two different such constructions can be established. Finally, when we refer to some series $V_1, \ldots$ below, we always mean a series of sets chosen to serve as V_i sets in the context of the construction in proposition 3.3.11. We will occasionally use the notation $\mathcal{V}$ (or other characters in a similar way) to denote such a series. And we will use superscripts to distinguish P_i, L_i, H_i and H_P sets in different constructions. Observe that for any series $\mathcal{V}$: ${P_1}^V = Ground(P)$. And also: $B_{{P_1}^V} = B_{Ground(P)} = B_P$.

We start with a fairly basic lemma:

Lemma A.0.1 Let $\mathcal{V} = V_1, \ldots$ be a series such that ${P_i}^V$ is defined. Let ${N_i}^V = \{n_C \in I\!N \mid \text{there exists a clause } (C, n_C) \text{ in } {P_i}^V\}$. Then:

- ${N_1}^V, \ldots$ is monotonically decreasing.
- $B_{{P_1}^V}, \ldots$ is monotonically decreasing.

Proof In the construction in proposition 3.3.11, each ${P_{i+1}}^V$ is obtained from ${P_i}^V$ by:

1. deleting some clauses

2. removing some literals from the remaining clauses

Both statements above trivially follow. □

Lemma A.0.1 enables the following definition:

Definition A.0.2 Let $\mathcal{V} = V_1, \ldots$ be a series such that $P_i^{\mathcal{V}}$ is defined. Then for each atom $A \in B_P \setminus B_{P_i^{\mathcal{V}}}$, we define the *$\mathcal{V}$-layer* of A, $l_{\mathcal{V}}(A)$, to be the smallest $j(1 \leq j < i)$ such that $A \notin B_{P_{j+1}^{\mathcal{V}}}$.

Notice that $l_{\mathcal{V}}(A)$ is well-defined in the sense that it does not depend on i. In particular, if $\mathcal{V}$ terminates successfully, then each $A \in B_P$ has a unique associated layer $l_{\mathcal{V}}(A)$.

The layer concept is interesting, since we can easily show:

Lemma A.0.3 Let $A \in B_P$ such that $i = l_{\mathcal{V}}(A)$. Then for any $j \neq i$ such that $L_j^{\mathcal{V}}$ is defined, there are no clauses with head A in $L_j^{\mathcal{V}}$.

Proof For $j < i$, $A \in B_{P_{j+1}^{\mathcal{V}}}$. Therefore $A \notin V_j$, from which the result follows. On the other hand, $A \notin B_{P_{i+1}^{\mathcal{V}}}$, so that the desired result for $j > i$ follows from lemma A.0.1. □

So, when an atom is "consumed" by a series, the unique definite program corresponding to its layer is the only one that possibly contains clauses defining A. It is therefore this program which decides whether A is in the resulting model (if any).

Another useful property of the layer concept is the following:

Lemma A.0.4 Let $A \in B_P$ such that $l_{\mathcal{V}}(A) = i$. For any $B \in B_{P_i^{\mathcal{V}}}$, such that $B \leq A$, we have: $l_{\mathcal{V}}(B) = i$.

Proof By the construction in proposition 3.3.11, $B \in V_i$. Thus, $B \notin B_{P_{i+1}^{\mathcal{V}}}$ (which exists since $l_{\mathcal{V}}(A) = i$ exists). Obviously, $B \in B_{P_i^{\mathcal{V}}}$, so that the result follows. □

We can now prove a quite powerful result, basically establishing equality of truth according to two different series for the same P.

Lemma A.0.5 Let $\mathcal{V}$ and $\mathcal{W}$ be two series such that $P_i^{\mathcal{V}}$ and $P_j^{\mathcal{W}}$ are defined. Let A be an atom in $B_P \setminus (B_{P_i^{\mathcal{V}}} \cup B_{P_j^{\mathcal{W}}})$. Then:

$$A \in \bigcup_{m<i} H_m^{\mathcal{V}} \iff A \in \bigcup_{m<j} H_m^{\mathcal{W}}$$

Proof Notice that the result trivially holds if either i or j equals 1, since in that case $B_P \setminus (B_{P_i^{\mathcal{V}}} \cup B_{P_j^{\mathcal{W}}}) = \emptyset$. Suppose therefore that they are both bigger than 1. It follows from lemma A.0.3 that for any $A \in B_P \setminus (B_{P_i^{\mathcal{V}}} \cup B_{P_j^{\mathcal{W}}})$:

$A \in \bigcup_{m<i} H_m{}^V \Longleftrightarrow A \in H_{l_{\mathcal{V}}(A)}{}^V$

and

$A \in \bigcup_{m<j} H_m{}^W \Longleftrightarrow A \in H_{l_{\mathcal{W}}(A)}{}^W$

So, it is sufficient to show that:

$A \in H_{l_{\mathcal{V}}(A)}{}^V \Longleftrightarrow A \in H_{l_{\mathcal{W}}(A)}{}^W$

The proof proceeds through *induction on $l_{\mathcal{V}}(A)$ and $l_{\mathcal{W}}(A)$*.

We first comment on the *structure of the induction proof*, which is non-standard. Let us denote by $Equi(k,l)$ the following formula:

$\forall A \in B_P \setminus (B_{P_i{}^V} \cup B_{P_j{}^W})$:

$$(l_{\mathcal{V}}(A) \leq k \wedge l_{\mathcal{W}}(A) \leq l) \Longrightarrow (A \in H_{l_{\mathcal{V}}(A)}{}^V \Longleftrightarrow A \in H_{l_{\mathcal{W}}(A)}{}^W)$$

Notice that, since for any $A \in B_P \setminus (B_{P_i{}^V} \cup B_{P_j{}^W})$, $(l_{\mathcal{V}}(A) \leq i-1 \wedge l_{\mathcal{W}}(A) \leq j-1)$ trivially holds, we need to prove $Equi(i-1, j-1)$. In order to achieve this, we will show:

1. $Equi(1,l)$, for all $l \leq j-1$.

2. $Equi(k-1,l) \wedge Equi(k,l-1) \Longrightarrow Equi(k,l)$, for all $k \leq i-1$ and $l \leq j-1$.

From 1, due to symmetry, $Equi(k,1)$, for all $k \leq i-1$, follows. Then, by repeatedly applying 2, $Equi(i-1,j-1)$ is implied by a finite conjunction of formulas of the type $Equi(1,l)$, $l \leq j-1$ and $Equi(k,1)$, $k \leq i-1$, which completes the proof.

The *proof of* 1 is by induction on l. For the base case, $l=1$, $A \in H_1{}^V \Longleftrightarrow A \in H_1{}^W$ follows immediately, because the subprograms of $L_1{}^V$ and $L_1{}^W$ on which A depends are identical. This is due to the construction in proposition 3.3.11, which ensures that:

- $L_1{}^V$ and $L_1{}^W$ contain *all* clauses of $Ground(P)$ with head A.

- V_1 and W_1 contain all atoms B with $B \leq A$.

- $L_1{}^V$ and $L_1{}^W$ also contain *all* clauses of $Ground(P)$ with such B's as their head.

Next, assume that $Equi(1,l-1)$ holds, with $l \leq j-1$. We prove $Equi(1,l)$. The increment of $Equi(1,l)$ over $Equi(1,l-1)$ is:

$\forall A \in B_P \setminus (B_{P_i{}^V} \cup B_{P_j{}^W})$:

$$(l_{\mathcal{V}}(A) = 1 \wedge l_{\mathcal{W}}(A) = l) \Longrightarrow (A \in H_1{}^V \Longleftrightarrow A \in H_l{}^W)$$

So, let A be an atom with $l_{\mathcal{V}}(A) = 1$ and $l_{\mathcal{W}}(A) = l$. From the way clauses are transformed and/or deleted in the construction procedure, we obtain the following correspondence between clauses with head A in $L_1{}^V$ and $L_l{}^W$:

1. Every such clause $(C, n) \in {L_1}^V$ such that there is no clause $(C^W, n) \in {L_l}^W$, has a body with at least one atom in $\bigcup_{m<l} W_m$, but not in $\bigcup_{m<l} {H_m}^W$.
2. There is a one-to-one correspondence between all other clauses $(A \leftarrow B^V, n) \in {L_1}^V$ and $(A \leftarrow B^W, n) \in {L_l}^W$, where B^V is identical to B^W except for the possible occurrence of atoms in B^V that are not in B^W but in $\bigcup_{m<l} W_m$ and in $\bigcup_{m<l} {H_m}^W$.

We show by induction that:

$\forall A \in B_P \setminus (B_{{P_i}^V} \cup B_{{P_j}^W})$:

$(l_V(A) = 1 \wedge l_W(A) = l) \Longrightarrow (\exists m : A \in T_{{L_1}^V}\uparrow m \Longleftrightarrow \exists m' : A \in T_{{L_l}^W}\uparrow m')$

First, if $A \in T_{{L_1}^V}\uparrow 1$, then ${L_1}^V$ contains a fact $(A\leftarrow, n)$. Only 2 is applicable and, since the ${L_1}^V$-clause can only have *more* body atoms than the corresponding ${L_l}^W$-clause, $(A\leftarrow, n) \in {L_l}^W$, so that $A \in T_{{L_l}^W}\uparrow 1$. Conversely, if $A \in T_{{L_l}^W}\uparrow 1$, then $(A\leftarrow, n) \in {L_l}^W$. Again, only 2 is applicable. Thus, ${L_1}^V$ contains a clause $(A \leftarrow B^V, n)$, where all atoms in B^V are in $\bigcup_{m<l} W_m$ and in $\bigcup_{m<l} {H_m}^W$. Let B be such an atom. Since $L_V(B) = 1$ and $L_W(B) < l$, we can apply the induction hypothesis $Equi(1, l-1)$. It states that $B \in {H_1}^V \Longleftrightarrow B \in {H_{l_W(B)}}^W$. But since $B \in \bigcup_{m<l} {H_m}^W$, the right hand side holds, so that $B \in {H_1}^V$. Therefore, there exists an m_B such that $B \in T_{{L_1}^V}\uparrow m_B$. Taking $m_A = max(\{m_B | B \text{ in } B^V\}) + 1$, we get $A \in T_{{L_1}^V}\uparrow m_A$.

For the induction step and the left-to-right implication, let $A \in T_{{L_1}^V}\uparrow m$ and assume that this implication holds for all $B \in T_{{L_1}^V}\uparrow(m-1)$. So, there exists a clause $(A \leftarrow B^V, n) \in {L_1}^V$, such that $B \in T_{{L_1}^V}\uparrow(m-1)$, for each $B \in B^V$.

First, we prove that $(A \leftarrow B^V, n)$ must be of type 2. Assume that it is of type 1, then there exists a B in B^V such that $B \in \bigcup_{m<l} W_m \setminus \bigcup_{m<l} {H_m}^W$. Clearly, $l_V(B) = 1$ and $l_W(B) < l$, so that the induction hypothesis $Equi(1, l-1)$ applies. So, $B \in {H_1}^V \Longleftrightarrow B \in {H_{l_W(B)}}^W$. But since $B \in {H_{l_W(B)}}^W \Longleftrightarrow B \in \bigcup_{m<l} {H_m}^W$ and $B \notin \bigcup_{m<l} {H_m}^W$, we obtain $B \notin {H_1}^V$. This contradicts $B \in T_{{L_1}^V}\uparrow(m-1)$. So, $(A \leftarrow B^V, n)$ is of type 2 and there exists a clause $(A \leftarrow B^W, n)$ in ${L_l}^W$ containing a subset of the body atoms in B^V. For each such atom, B in B^W, $B \in T_{{L_1}^V}\uparrow(m-1)$ holds. By the induction hypothesis, $B \in T_{{L_l}^W}\uparrow m_B$ for some m_B. Taking $m_A = max(\{m_B | B \text{ in } B^W\}) + 1$, we get $A \in T_{{L_l}^W}\uparrow m_A$.

Finally, for the induction step on the right-to-left implication, let $A \in T_{{L_l}^W}\uparrow m$ and assume that the implication holds for all $B \in T_{{L_l}^W}\uparrow(m-1)$. Again, there exists a clause $(A \leftarrow B^W, n)$ in ${L_l}^W$, such that $B \in T_{{L_l}^W}\uparrow(m-1)$, for all B in B^W. Only case 2 can apply. Thus there exists a clause $(A \leftarrow B^V, n)$ in ${L_1}^V$, identical to $(A \leftarrow B^W, n)$, except that it may contain some additional body atoms B, where $B \in \bigcup_{m<l} W_m \cap \bigcup_{m<l} {H_m}^W$. For the body atoms B which are both in B^V and B^W, we can apply the right-to-left induction hypothesis: $B \in T_{{L_1}^V}\uparrow m_B$, for some m_B. For the atoms B not in B^W, we again have $l_V(B) = 1$, $l_W(B) < l$

and $B \in H_{l_{\mathcal{W}}(B)}{}^W$. So, we apply $Equi(1, l-1)$, obtaining $B \in H_1{}^V$, which means that $B \in T_{L_1{}^V} \uparrow m_B$, for some m_B. Again, $m_A = max(\{m_B | B \text{ in } B^V\}) + 1$ allows the conclusion of the proof.

Next, *we prove the implication:*

$\forall k < i, \forall l < j : Equi(k-1, l) \wedge Equi(k, l-1) \Longrightarrow Equi(k, l)$

The proof is completely similar to the previous step, except that more different correspondence cases between clauses in $L_k{}^V$ and $L_l{}^W$ can be distinguished. Let A be an atom with $l_{\mathcal{V}}(A) = k$ and $l_{\mathcal{W}}(A) = l$. From the way clauses are transformed and/or deleted in the construction in proposition 3.3.11, we can now distinguish the following possible cases for clauses in $L_k{}^V$ or in $L_l{}^W$, having A as their head:

1. Every such clause $(C^V, n) \in L_k{}^V$ such that there is no clause $(C^W, n) \in L_l{}^W$, contains at least one body atom B, such that B is in $\bigcup_{m<l} W_m$, but not in $\bigcup_{m<l} H_m{}^W$.
   Notice that this is less trivial than the corresponding statement for $L_1{}^V$ and $L_l{}^W$ above. Although it is clear that the clause (C, n) in $Ground(P)$ contains at least one such body atom B, in (C^V, n) this body atom could have been removed. However, since $Equi(k, l-1)$ is given and since $B \notin \bigcup_{m<l} H_m{}^W$, we have $B \notin \bigcup_{m<k} H_m{}^V$. Thus B has not been removed from (C^V, n).
   Furthermore, $Equi(k, l-1)$ also implies that this atom B is not in $H_k{}^V$.

2. Every such clause $(C^W, n) \in L_l{}^W$ such that there is no clause $(C^V, n) \in L_k{}^V$, contains at least one atom B such that B is in $\bigcup_{m<k} V_m$, but not in $\bigcup_{m<k} H_m{}^V$. (Here, we have used $Equi(k-1, l)$.) Now, $Equi(k-1, l)$ also implies that $B \notin H_l{}^W$.

3. There is a one-to-one correspondence between other clauses $(A \leftarrow B^V, n) \in L_k{}^V$ and $(A \leftarrow B^W, n) \in L_l{}^W$, where B^V is identical to B^W except that:

   (a) B^V might contain atoms not occurring in B^W.
   All such atoms are in $\bigcup_{m<l} H_m{}^W$, but then also in $H_k{}^V$ on account of $Equi(k, l-1)$.

   (b) B^W might contain atoms not occurring in B^V.
   All such atoms are in $\bigcup_{m<k} H_m{}^V$, but then also in $H_l{}^W$ on account of $Equi(k-1, l)$.

The proof again proceeds through induction, proving the statement:

$\forall A \in B_P \setminus (B_{P_i{}^V} \cup B_{P_j{}^W}) :$

$(l_{\mathcal{V}}(A) = k \wedge l_{\mathcal{W}}(A) = l) \Longrightarrow (\exists m : A \in T_{L_k{}^V} \uparrow m \Longleftrightarrow \exists m' : A \in T_{L_l{}^W} \uparrow m')$

It is very similar to the proof of the induction step for $Equi(1, l)$. The main differences are:

- Due to symmetry, we only need to prove one of the implications.
- We occasionally use lemma A.0.4 to establish that any atom B occurring in ${L_k}^V$ (resp. ${L_l}^W$), on which A depends, also has $l_{\mathcal{V}}(B) = k$ (resp. $l_{\mathcal{W}}(B) = l$).

We omit the details. □

Let us now take a first look at the maximal choice series for some P and its relationship with a successful other series.

Lemma A.0.6 Let $V_1, \ldots$ be a series such that the construction in proposition 3.3.11 terminates successfully for P with resulting model ${H_P}^V$. Let $S_1, \ldots$ be the maximal choice series for P. Then, for every i such that ${P_i}^S$ is defined:

$\bigcup_{j<i} {H_j}^S = {H_P}^V \cap (B_P \setminus B_{{P_i}^S})$

Proof Let i be such that ${P_i}^S$ is defined. First, if $A \in B_{{P_i}^S}$, then both $A \notin \bigcup_{j<i} {H_j}^S$ and $A \notin B_P \setminus B_{{P_i}^S}$ are trivially satisfied. So, suppose that $A \notin B_{{P_i}^S}$. Then $A \in {H_P}^V \iff A \in \bigcup_{j<l_{\mathcal{V}}(A)+1} {H_j}^V$, and, of course, $A \notin B_{{P_{l_{\mathcal{V}}(A)+1}}^V}$. The result now follows from lemma A.0.5. □

We need one more lemma before we can actually complete the proof of lemma 3.3.12.

Lemma A.0.7 Let $\mathcal{V} = V_1, \ldots$ be a series such that the construction in proposition 3.3.11 terminates successfully for P with resulting model ${H_P}^V$. Let $\mathcal{S} = S_1, \ldots$ be the maximal choice series for P. Then, for every i such that ${P_i}^V$ is defined, one of the following holds:

1. Either there is a $j \leq i$ such that ${P_j}^S = \emptyset$ and

   $\bigcup_{k<i} {H_k}^V = {H_P}^S \cap (B_P \setminus B_{{P_i}^V})$

2. Or ${P_i}^S$ is defined and non-empty and the following both hold:

   (a) $\bigcup_{k<i} {H_k}^V = \bigcup_{k<i} {H_k}^S \cap (B_P \setminus B_{{P_i}^V})$

   (b) $\forall (C^S, n_C) \in {P_i}^S : \exists (C^V, n_C) \in {P_i}^V$, where C^V is identical to C^S except for the possible presence of extra body literals, with an atom in $\bigcup_{k<i} S_k$, which are satisfied in $\bigcup_{k<i} {H_k}^S$.

Proof Not surprisingly, the proof proceeds through induction on i. Again, the base case ($i = 1$) is immediate (either $Ground(P) = \emptyset$ and 1 holds or $Ground(P) \neq \emptyset$ and 2 holds). So, let us prove the induction step. In other words, we assume that the property is satisfied for every $i \leq n$. If ${P_n}^V = \emptyset$, then ${P_i}^V$ is not defined for $i > n$ and we are done. Suppose therefore that ${P_n}^V \neq \emptyset$, then ${P_{n+1}}^V$ is defined and we have to show that the property holds for $i = n+1$. Now, if there is a $j \leq n+1$ such that ${P_j}^S = \emptyset$, then it follows from lemma A.0.5 that:

$\forall A \in B_P \setminus B_{P_{n+1}V} : A \in {H_P}^S \iff A \in \bigcup_{k<n+1} {H_k}^V$

Moreover, obviously:

$\forall A \in B_{P_{n+1}V} : A \notin \bigcup_{k<n+1} {H_k}^V$

and the result follows.

This leaves us with the case that ${P_n}^S$ is defined and non-empty. We have to show that also ${P_{n+1}}^S$ is defined, and that 2*a* and 2*b* hold for $i = n+1$ in case it is non-empty.

- We first show that ${P_n}^S$ has at least one minimal component. Suppose that this is not the case. Then every atom in $B_{P_n S}$ is member of some infinite series of atoms $A_1, A_2, \ldots$ such that $A_1 > A_2 > \ldots$. Now, the clauses in *Ground*(P) (for the atoms in $B_{P_n S}$) that correspond to the clauses in ${P_n}^S$ might contain extra body literals whose atom is not in $B_{P_n S}$ and which are satisfied according to $\bigcup_{k<n} {H_k}^S$. Lemma A.0.5 ensures that these literals can never be falsified in the $\mathcal{V}$-construction. So, the $\mathcal{V}$-construction can delete none of these clauses without first having an atom from $B_{P_n S}$ in a bottom component of some $B_{P_l V}$. This, however, is not possible and therefore the $\mathcal{V}$-construction can not terminate successfully. Which contradicts the assumption that it does.

- Our next task is showing that ${L_n}^S$ is definite. Suppose that it is not. Then there is a bottom component B of $B_{P_n S}$ containing at least one atom with a non-definite defining clause C in ${P_n}^S$. But then, if any atom in B is defined in terms of an atom not in B, B would not be a bottom component of $B_{P_n S}$. Therefore, all clauses in ${P_n}^S$ with a head in B only contain literals with an atom in B. Again, it follows that none of the corresponding clauses in *Ground*(P) can be deleted by the $\mathcal{V}$-construction without first having B as part of some V_l. But then also ${L_l}^V$ would be non-definite, which again contradicts the assumption that $\mathcal{V}$ terminates successfully.

- So, ${P_{n+1}}^S$ is indeed defined. Above, we have already dealt with the case that it is empty. Assume therefore that it is non-empty. We prove 2*a* and 2*b* for $i = n+1$.

  - We start with 2*b* and first show that $V_n \cap B_{P_n S} \subseteq S_n$. Let $A \in V_n \cap B_{P_n S}$. We show that the subprogram of ${P_n}^S$ on which A depends is either empty or definite. In both cases, A does not depend negatively

on any atom in $B_{P_n{}^S}$, so that by definition, $A \in S_n$.
If $P_n{}^S$ contains no clause with A as its head, we are done. Otherwise, let $(A \leftarrow B^S, n_C) \in P_n{}^S$. By the induction hypothesis 2*b*, there exists a clause $(A \leftarrow B^V, n_C) \in P_n{}^V$, such that every literal in B^S is also in B^V. Because $A \in V_n$ and $L_n{}^V$ is definite, $(A \leftarrow B^V, n_C)$ is definite and so is $(A \leftarrow B^S, n_C)$. It remains to be shown that for all other atoms $B \in B_{P_n{}^S}$, such that $B \leq A$, the same holds. Let B be such an atom. Using induction hypothesis 2*b* again, $B \leq A$ also holds in $P_n{}^V$: for any sequence of clauses $(C_k{}^S, n_{C_k})_k$ in $P_n{}^S$, establishing the dependency of A on B in $P_n{}^S$, the corresponding sequence $(C_k{}^V, n_{C_k})_k$ in $P_n{}^V$ establishes the dependency in $P_n{}^V$. Therefore, $B \in V_n \cap B_{P_n{}^S}$, and, by the same argument as used for A above, the clauses in $P_n{}^S$ with head B are definite. Thus, the subprogram of $P_n{}^S$ on which A depends is definite, and $A \in S_n$.
Next, we show that:

$$\forall (C^S, n_C) \in P_{n+1}{}^S : \exists (C^V, n_C) \in P_{n+1}{}^V$$

Since it is given that:

$$\forall (C^S, n_C) \in P_n{}^S : \exists (C^V, n_C) \in P_n{}^V$$

it suffices to prove that if a clause $(C^S, n_C) \in P_n{}^S$ is not pruned during the construction of $P_{n+1}{}^S$ from $P_n{}^S$, then neither is the corresponding clause $(C^V, n_C) \in P_n{}^V$ pruned while constructing $P_{n+1}{}^V$ from $P_n{}^V$. Suppose that such a $(C^V, n_C) \in P_n{}^V$ is pruned. One possible cause is that $(C^V, n_C) \in L_n{}^V$. Since (C^S, n_C) has the same head, say A, as (C^V, n_C) and $A \in V_n \cap B_{P_n{}^S}$, by $V_n \cap B_{P_n{}^S} \subseteq S_n$, we have $A \in S_n$. So, $(C^S, n_C) \in L_n{}^S$ and is pruned too.
Alternatively, $(C^V, n_C) \in P_n{}^V \backslash L_n{}^V$ can be pruned because it contains a body literal B, with atom $B^a \in V_n$, such that B is falsified in $H_n{}^V$. By induction hypothesis 2*b*, the only body literals of (C^V, n_C) that do not occur in the body of (C^S, n_C) are satisfied in $\bigcup_{k<n} H_k{}^S$. Furthermore, by lemma A.0.5:

$$\forall A \in B_P \setminus (B_{P_{n+1}{}^S} \cup B_{P_{n+1}{}^V}) :$$
$$A \in \bigcup\nolimits_{k<n+1} H_k{}^V \Longleftrightarrow A \in \bigcup\nolimits_{k<n+1} H_k{}^S \qquad (*)$$

so that B is not satisfied in $\bigcup_{k<n} H_k{}^S$ and is a body literal of (C^S, n_C). Finally, $B^a \in S^n$, because $B^a \in V_n \cap B_{P_n{}^S}$, and B is falsified in $H_n{}^S$, because it is falsified in $H_n{}^V$ and $(*)$ holds. Thus, (C^S, n_C) would also be pruned in the S-construction.
It remains to be proved that C^S and C^V are related as stated in 2*b*. To see this, let $(C^S, n_C) \in P_n{}^S$, $(C^V, n_C) \in P_n{}^V$, $(C'^S, n_C) \in P_{n+1}{}^S$ and $(C'^V, n_C) \in P_{n+1}{}^V$. We know that C^S and C^V are related in the

proper way. Moreover, C'^S is identical to C^S except for the possible removal of body literals with an atom in S^n which are satisfied in ${H_n}^S$. And C'^V is identical to C^V except for the possible removal of body literals with an atom in V^n which are satisfied in ${H_n}^V$. The result now again follows from $(*)$ and $V_n \cap B_{P_n{}^S} \subseteq S_n$, since we obtain that no literal which occurs in the bodies of both C^S and C^V, can be removed from the latter without likewise being removed from the former.

– Finally, we prove $2a$ for $i = n + 1$.
First, $2b$ for $i = n + 1$ implies:
$$B_{P_{n+1}{}^S} \subseteq B_{P_{n+1}{}^V}$$
Therefore, $(*)$ can be rewritten as:
$$\forall A \in B_P \setminus B_{P_{n+1}{}^V} : A \in \bigcup_{k<n+1} {H_k}^V \iff A \in \bigcup_{k<n+1} {H_k}^S$$
Moreover, from the definition of the construction in proposition 3.3.11:
$$\bigcup_{k<n+1} {H_k}^V \cap (B_P \setminus B_{P_{n+1}{}^V}) = \bigcup_{k<n+1} {H_k}^V$$
so that $2a$ for $i = n + 1$ follows.

□

We can now complete the proof for lemma 3.3.12:

Proof (of lemma 3.3.12) First, it is clear that S will also terminate successfully. Indeed, suppose there is some i such that ${P_i}^V = \emptyset$, then case 2 in lemma A.0.7 can not hold for i, due to $2b$. Therefore, case 1 holds and S terminates successfully. So, suppose ${P_i}^V \neq \emptyset$ for all i. If there is an i such that case 1 in lemma A.0.7 holds, we again obtain successful termination of S. When case 2 holds for every i, ${P_i}^S$ is defined for every $i < \omega$. But since $\bigcap_{i<\omega} CH({P_i}^V) = \emptyset$, $2b$ implies that also $\bigcap_{i<\omega} CH({P_i}^S) = \emptyset$
Next, lemma A.0.6 implies that for every j such that ${P_j}^S$ is defined, $\bigcup_{k<j} {H_k}^S \subseteq {H_P}^V$. It follows that ${H_P}^S \subseteq {H_P}^V$. Similarly, lemma A.0.7 guarantees that for every i such that ${P_i}^V$ is defined, $\bigcup_{k<i} {H_k}^V \subseteq {H_P}^S$ and we obtain ${H_P}^V \subseteq {H_P}^S$. Therefore, the two models are equal. □

Appendix B

Benchmarks for Partial Deduction

In this appendix, we include the code of the five definite logic programs on which the tests in section 7.5 were run. Together with each program, we also present the goals for which partial deductions were computed, as well as their further instantiated versions that were used in run-time comparisons. As mentioned in section 7.5, these programs and goals are taken from [104], where they function as benchmarks for the comparison of five partial deduction systems. Finally, all variable names in the programs below start with X. To convert the programs into code that can actually be run on a *ProLog by BIM* system, these X's must be replaced by underscores, $\leftarrow$'s by $: -$'s, and dots must be added to signal clause ends.

transpose

We already met the first test program in chapter 6. It is the program for transposing a matrix, studied in example 6.4.13. But the PD-time goal below is further instantiated than the top-level goal used for unfolding in that example.

$transpose(Xmatrix, []) \leftarrow$
$\qquad nullrows(Xmatrix)$
$transpose(Xmatrix, [Xtransrow|Xtransremmatrix]) \leftarrow$
$\qquad makerow(Xmatrix, Xtransrow, Xremmatrix),$
$\qquad transpose(Xremmatrix, Xtransremmatrix)$

$makerow([], [], []) \leftarrow$
$makerow([[Xel|Xrrow]|Xmatrix], [Xel|Xtrrow], [Xrrow|Xrmatrix]) \leftarrow$
$\qquad makerow(Xmatrix, Xtrrow, Xrmatrix)$

$nullrows([]) \leftarrow$
$nullrows([[]|Xremmatrix]) \leftarrow$
$\qquad nullrows(Xremmatrix)$

PD-time goal:

$\leftarrow transpose([[X1, X2, X3, X4, X5, X6, X7, X8, X9], Xr2, Xr3], Xtrm)$

Run-time goal:

$\leftarrow transpose([[1, 2, 3, 4, 5, 6, 7, 8, 9], [2, 3, 4, 5, 6, 7, 8, 9, 10],$
$\qquad [3, 4, 5, 6, 7, 8, 9, 10, 11]], Xtrm)$

relative

The second example program is a (functor free) database application. It involves the standard *parent-ancestor* transitive closure, with a non-recursive predicate *relative* defined on top of *ancestor*.

```
relative(Xperson1, Xperson2) ←
          ancestor(Xcommonancestor, Xperson1),
          ancestor(Xcommonancestor, Xperson2)

ancestor(Xancestor, Xperson) ←
          parent(Xancestor, Xperson)
ancestor(Xancestor, Xperson) ←
          parent(Xancestor, Xcloserancestor),
          ancestor(Xcloserancestor, Xperson)

parent(Xparent, Xchild) ←
          father(Xparent, Xchild)
parent(Xparent, Xchild) ←
          mother(Xparent, Xchild)

father(zach, jonas) ←
father(zach, carol) ←
father(jonas, maria) ←

mother(carol, paulina) ←
mother(carol, albertina) ←
mother(albertina, peter) ←
mother(maria, mary) ←
mother(maria, jose) ←
mother(mary, anna) ←
mother(mary, john) ←
```

PD-time goal: ← *relative(john, Xwho)*

Run-time goal: ← *relative(john, peter)*

depth

Next, *depth* constitutes a simple vanilla like meta-interpreter, registering the number of steps in a proof.

```
depth(empty, 0) ←
depth((Xgoal, Xgoals), Xdepth) ←
            depth(Xgoal, Xdepthgoal),
            depth(Xgoals, Xdepthgoals),
            max(Xdepthgoal, Xdepthgoals)
depth(Xgoal, s(Xdepth)) ←
            pclause(Xgoal, Xbody),
            depth(Xbody, Xdepth)

max(Xnum, 0, Xnum) ←
max(0, Xnum, Xnum) ←
max(s(Xnum1), s(Xnum2), s(Xmax)) ←
            max(Xnum1, Xnum2, Xmax)

pclause(member(Xel, Xl), append(Xany1, [Xel|Xany2], Xl)) ←
pclause(append([], Xl, Xl), empty) ←
pclause(append([Xh|Xt], Xl, [Xh|Xnt]), append(Xt, Xl, Xnt)) ←
```

PD-time goal: ← *depth*(*member*(*Xel*, [*a*, *b*, *c*, *m*, *d*, *e*, *m*, *f*, *g*, *m*, *i*, *j*]), *Xdepth*)

Run-time goal: ← *depth*(*member*(*i*, [*a*, *b*, *c*, *m*, *d*, *e*, *m*, *f*, *g*, *m*, *i*, *j*]), *Xdepth*)

match

The *match* program below is a classic in partial evaluation (see e.g. section 12.1 in [85]) and partial deduction (see e.g. [156] and section 3 of [65]). It serves to match a certain pattern with part of a given string, using a straightforward strategy. Notice that *match*, upon discovering a non-matching symbol, simply restarts the search for the whole pattern in the string minus its first symbol. Clever partial deduction can produce, for a known first argument, an efficient "Knuth-Morris-Pratt"-like ([91]) linear pattern matcher.

$match(Xpat, Xstr) \leftarrow$
$\qquad match1(Xpat, Xstr, Xpat, Xstr)$

$match1([], Xany1, Xany2, Xany3) \leftarrow$
$match1([Xterm|Xany1], [Xdifterm|Xany2], Xpat, [Xany3|Xremstr]) \leftarrow$
$\qquad Xterm \backslash == Xdifterm,$
$\qquad match1(Xpat, Xremstr, Xpat, Xremstr)$
$match1([Xterm|Xrempat], [Xterm|Xremstr], Xpat, Xstr) \leftarrow$
$\qquad match1(Xrempat, Xremstr, Xpat, Xstr)$

PD-time goal:

$\leftarrow match([a, a, b], Xstring)$

Run-time goal:

$\leftarrow match([a, a, b], [a, b, c, d, e, f, g, h, i, j, k, l, m, n, o,$
$\qquad p, q, r, s, t, u, v, a, a, b, w, x, y, z])$

contains

The final benchmark program also performs pattern matching, but in a more sophisticated way than *match*. The two arguments of the predicate *con* are used as follows. The first argument registers the string to be searched through. The second argument is a couple of lists. The first list contains the part of the search pattern that was already found on the string positions immediately preceding the current initial string symbol. The second registers the remainder of the search pattern: the part that still has to be found (as initial part of the remaining string). *new* tests for identity of the first symbols in the remaining string and the part of the pattern still to be found. If they indeed turn out to be the same, (trivially) adapted values of the "(found,notyetfound)"-pair are returned to *con* for further search. If not, a more complex operation returns a value for *con*'s second argument with a (non-deterministically choosen) valid new "already found" part. As a result, unlike *match*1 which occasionally retraces its steps, *con* processes 1 element of the input string in each recursive step. On the other hand, it is non-deterministic.

```
contains(Xpat, Xstr) ←
        con(Xstr, ([], Xpat))

con(Xany1, (Xany2, [])) ←
con([Xterm|Xremstr], Xpatinfo) ←
        new(Xterm, Xpatinfo, Xnewpatinfo),
        con(Xremstr, Xnewpatinfo)

new(Xterm, (Xfound, [Xterm|Xrempat]), (Xnewfound, Xrempat)) ←
        append(Xfound, [Xterm], Xnewfound)
new(Xterm, (Xfound, [Xdifterm|Xrempat]), (Xnewfound, Xnewrempat)) ←
        Xterm \== Xdifterm,
        append(Xfound, [Xterm], Xprecstr),
        append(Xnewfound, Xrest, Xfound),
        append(Xany, Xnewfound, Xprecstr),
        append(Xrest, [Xdifterm|Xrempat], Xnewrempat)

append([], Xlist, Xlist) ←
append([Xhead|Xtail], Xlist, [Xhead|Xnewtail]) ←
        append(Xtail, Xlist, Xnewtail)
```

PD-time goal:

$\leftarrow contains([a, a, b], Xstring)$

Run-time goal:

$\leftarrow contains([a, a, b], [a, b, c, d, e, f, g, h, a, a, b, i, j, k, l,$
$m, n, o, p, q, r, s, t, u, v, w, x, y, z])$

Bibliography

[1] H. Abramson and M. H. Rogers, editors. *Meta-Programming in Logic Programming, Proceedings of Meta'88*. MIT Press, 1989.

[2] ACM, editor. *Proceedings PEPM'91, Sigplan Notices, 26(9)*, New Haven, Connecticut, USA, 1991.

[3] ACM, editor. *Proceedings PEPM'93*, Copenhagen, Denmark, 1993.

[4] L. C. Aiello, D. Nardi, and M. Schaerf. Reasoning about knowledge and ignorance. In *Proceedings FGCS'88*, pages 618–627, Tokyo, 1988. ICOT.

[5] K. R. Apt. Logic programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B, Formal Models and Semantics*, pages 493–574. Elsevier Science Publishers B.V., 1990.

[6] K. R. Apt and M. Bezem. Acyclic programs. In D. H. D. Warren and P. Szeredi, editors, *Proceedings ICLP'90*, pages 617–633, Jerusalem, June 1990. MIT Press. Revised version in *New Generation Computing*, 9(3&4):335–364.

[7] K. R. Apt, H. Blair, and A. Walker. Towards a theory of declarative knowledge. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 89–148. Morgan-Kaufmann, 1988.

[8] K. R. Apt, E. Marchiori, and C. Palamidessi. A theory of first-order built-in's of Prolog. In H. Kirchner and G. Levi, editors, *Proceedings of the 3rd International Conference on Algebraic and Logic Programming*, pages 69–83. Springer-Verlag, LNCS 632, 1992.

[9] K. R. Apt and D. Pedreschi. Studies in pure prolog: Termination. In J. W. Lloyd, editor, *Proceedings of the Esprit Symposium on Computational Logic*, pages 150–176, Brussels, November 1990. Springer-Verlag.

[10] C. Aravindan and P. M. Dung. Partial deduction of logic programs wrt well-founded semantics. In H. Kirchner and G. Levi, editors, *Proceedings of the 3rd International Conference on Algebraic and Logic Programming*, pages 384–402. Springer-Verlag, LNCS 632, 1992.

[11] R. Barbuti, P. Mancarella, D. Pedreschi, and F. Turini. A transformational approach to negation in logic programming. *Journal of Logic Programming*, 8(3):201–228, 1990.

[12] K. Benkerimi and P. M. Hill. Supporting transformations for the partial evaluation of logic programs. *Journal of Logic and Computation*, 3(5):469–486, 1993.

[13] K. Benkerimi and J. W. Lloyd. A procedure for the partial evaluation of logic programs. Technical Report TR-89-04, Department of Computer Science, University of Bristol, Great-Britain, May 1989.

[14] K. Benkerimi and J. W. Lloyd. A partial evaluation procedure for logic programs. In S. Debray and M. Hermenegildo, editors, *Proceedings NACLP'90*, pages 343–358, Austin, Texas, October 1990. MIT Press.

[15] R. Bol. Loop checking in partial deduction. *Journal of Logic Programming*, 16(1&2):25–46, 1993.

[16] R. N. Bol, K. R. Apt, and J. W. Klop. An analysis of loop checking mechanisms for logic programs. *Theoretical Computer Science*, 86(1):35–79, 1991.

[17] P. A. Bonatti. Model theoretic semantics for Demo. In A. Pettorossi, editor, *Proceedings Meta'92*, pages 220–234. Springer-Verlag, LNCS 649, 1992.

[18] K. A. Bowen. Meta-level programming and knowledge representation. *New Generation Computing*, 3(4):359–383, 1985.

[19] K. A. Bowen and R. A. Kowalski. Amalgamating language and metalanguage in logic programming. In K. L. Clark and S.-Å. Tärnlund, editors, *Logic Programming*, pages 153–172. Academic Press, 1982.

[20] K. Broda and B. T. Ng. Partially evaluating the completed program. Technical Report DoC 91/43, Department of Computing, Imperial College, London, Great-Britain, December 1991.

[21] A. Brodsky and Y. Sagiv. Inference of monotonicity constraints in datalog programs. In *Proceedings PODS'89*, pages 190–199, Philadelphia, Pennsylvania, USA, March 1989. ACM.

[22] A. Brogi, P. Mancarella, D. Pedreschi, and F. Turini. Composition operators for logic theories. In J. W. Lloyd, editor, *Proceedings of the Esprit Symposium on Computational Logic*, pages 117–134. Springer-Verlag, November 1990.

[23] A. Brogi, P. Mancarella, D. Pedreschi, and F. Turini. Meta for modularising logic programming. In A. Pettorossi, editor, *Proceedings Meta'92*, pages 105–119. Springer-Verlag, LNCS 649, 1992.

[24] M. Bruynooghe, editor. *Meta'90, Proceedings of the Second Workshop on Meta-Programming in Logic.* K.U.Leuven, 1990.

[25] M. Bruynooghe, D. De Schreye, and B. Krekels. Compiling control. *Journal of Logic Programming*, 6(1&2):135–162, 1989.

[26] M. Bruynooghe, D. De Schreye, and B. Martens. A general criterion for avoiding infinite unfolding during partial deduction of logic programs. In V. Saraswat and K. Ueda, editors, *Proceedings ILPS'91*, pages 117–131, San Diego, October 1991. MIT Press.

[27] M. Bruynooghe, D. De Schreye, and B. Martens. A general criterion for avoiding infinite unfolding during partial deduction. *New Generation Computing*, 11(1):47–79, 1992.

[28] F. Bry. Query evaluation in recursive databases: Bottom-up and top-down reconciled. *Data & Knowledge Engineering*, 5(4):289–312, 1990.

[29] F. Bry, H. Decker, and R. Manthey. A uniform approach to constraint satisfaction and constraint satisfiability in deductive databases. In *Proceedings EDBT'88*, March 1988.

[30] F. Bry, R. Manthey, and B. Martens. Integrity verification in knowledge bases. In A. Voronkov, editor, *Proceedings 1st and 2nd Russian Conference on Logic Programming*, pages 114–139. Springer-Verlag, LNAI 592, 1992.

[31] S. Ceri, G. Gottlob, and L. Tanca. *Logic Programming and Databases.* Springer-Verlag, 1990.

[32] D. Chan. Constructive negation based on the completed database. In R. A. Kowalski and K. A. Bowen, editors, *Proceedings ICSLP'88*, pages 111–125, Seattle, August 1988. MIT Press.

[33] D. Chan and M. Wallace. A treatment of negation during partial evaluation. In H. D. Abramson and M. H. Rogers, editors, *Proceedings Meta'88*, pages 299–318. MIT Press, 1989.

[34] W. Chen, M. Kifer, and D. S. Warren. HiLog: A foundation for higher-order logic programming. *Journal of Logic Programming*, 15(3):187–230, 1993.

[35] K. L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, 1978.

[36] K. L. Clark and S.-Å. Tärnlund, editors. *Logic Programming*. Academic Press, 1982.

[37] S. Costantini and G. A. Lanzarone. A metalogic programming language. In G. Levi and M. Martelli, editors, *Proceedings ICLP'89*, pages 218–233, Lisbon, Portugal, June 1989. MIT Press.

[38] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(2&3):103–179, 1992.

[39] D. De Schreye and M. Bruynooghe. On the transformation of logic programs with instantiation based computation rules. *Journal of Symbolic Computation*, 7(2):125–154, 1989.

[40] D. De Schreye and S. Decorte. Termination of logic programs: the never-ending story. Technical Report CW182, Departement Computerwetenschappen, K.U.Leuven, Belgium, October 1993. To appear in the Journal of Logic Programming.

[41] D. De Schreye and B. Martens. A sensible least Herbrand semantics for untyped vanilla meta-programming and its extension to a limited form of amalgamation. In A. Pettorossi, editor, *Proceedings Meta'92*, pages 192–204. Springer-Verlag, LNCS 649, 1992.

[42] D. De Schreye, B. Martens, G. Sablon, and M. Bruynooghe. Compiling bottom-up and mixed derivations into top-down executable logic programs. *Journal of Automated Reasoning*, 7(3):337–358, 1991.

[43] D. De Schreye, K. Verschaetse, and M. Bruynooghe. A framework for analysing the termination of definite logic programs with respect to call patterns. In ICOT, editor, *Proceedings FGCS'92*, pages 481–488, Tokyo, June 1992. Omsha Ltd.

[44] H. Decker. Integrity enforcement on deductive databases. In L. Kerschberg, editor, *Proceedings Expert Database Systems '86*, Charleston, USA, 1986.

[45] R Demolombe. Syntactical characterization of a subset of domain independent formulas. *Journal of the ACM*, 39(1):71–94, 1992.

[46] M. Denecker and D. De Schreye. Justification semantics: A unifying framework for the semantics of logic programs. In A. Nerode and L. Pereira, editors, *Proceedings LPNMR'93*, pages 365–379, Lisbon, Portugal, June 1993. MIT Press.

[47] M. Denecker, D. De Schreye, and Y. D. Willems. Terms in logic programs : a problem with their semantics and its effect on the programming methodology. *CCAI, Journal for the Integrated Study of Artificial Intelligence, Cognitive Science and Applied Epistemology*, 7(3&4):363–383, 1990.

[48] N. Dershowitz. Termination of rewriting. *Journal of Symbolic Computation*, 3(1&2):69–115, 1987. Corrigendum in 4(3), pages 409–410.

[49] N. Dershowitz and Z. Manna. Proving termination with multiset orderings. *Communications of the ACM*, 22(8):465–476, 1979.

[50] H. B. Enderton. *A Mathematical Introduction to Logic*. Academic Press, 1972.

[51] A. P. Ershov. Mixed computation: Potential applications and problems for study. *Theoretical Computer Science*, 18:41–67, 1982.

[52] A. P. Ershov. Opening key-note speech. *New Generation Computing*, 6(2&3):79–86, 1988.

[53] A. P. Ershov, D. Bjørner, Y. Futamura, K. Furukawa, A. Haraldson, and W. Scherlis, editors. *New Generation Computing, 6(2&3): Special Issue with Selected Papers from the Workshop on Partial Evaluation and Mixed Computation, 1987*, 1988.

[54] K. Eshghi. *Meta-Language in Logic Programming*. PhD thesis, Department of Computing, Imperial College, London, Great-Britain, 1986.

[55] M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. Declarative modeling of the operational behaviour of logic programs. *Theoretical Computer Science*, 69:289–318, 1989.

[56] M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. A model-theoretic reconstruction of the operational semantics of logic programs. *Information and Computation*, 103(1):86–113, 1993.

[57] J. A. Fernández and J. Minker. Semantics of disjunctive deductive databases. In *Proceedings ICDT'92*, 1992.

[58] M. Fitting. *First-Order Logic and Automated Theorem Proving*. Springer-Verlag, 1990.

[59] R. W. Floyd. Assigning meaning to programs. In *Proceedings Symp. in Applied Math., 19*, pages 19–32, Providence, R.I., 1967. Amer Math Soc.

[60] H. Fujita and H. Furukawa. A self-applicable partial evaluator and its use in incremental compilation. *New Generation Computing*, 6(2&3):91–118, 1988.

[61] K. Furukawa. Logic programming as the integrator of the Fifth Generation Computer Systems Project. *Communications of the ACM*, 35(3):82–92, 1992.

[62] Y. Futamura. Partial evaluation of a computation process — an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.

[63] J. Gallagher. Personal Communication.

[64] J. Gallagher. Transforming logic programs by specialising interpreters. In *Proceedings ECAI'86*, pages 109–122, 1986.

[65] J. Gallagher. Tutorial on specialisation of logic programs. In *Proceedings PEPM'93*, pages 88–98, Copenhagen, June 1993. ACM.

[66] J. Gallagher and M. Bruynooghe. The derivation of an algorithm for program specialisation. In D. H. D. Warren and P. Szeredi, editors, *Proceedings ICLP'90*, pages 732–746, Jerusalem, June 1990. MIT Press. Revised version in *New Generation Computing*, 9(3&4):305–334.

[67] J. Gallagher and M. Bruynooghe. Some low-level source transformations for logic programs. In M. Bruynooghe, editor, *Proceedings Meta'90*, pages 229–244, Leuven, April 1990.

[68] J. Gallagher, M. Codish, and E. Shapiro. Specialisation of Prolog and FCP programs using abstract interpretation. *New Generation Computing*, 6(2&3):159–186, 1988.

[69] J. Gallagher and D. A. de Waal. Deletion of redundant unary type predicates from logic programs. In K.-K. Lau and T. Clement, editors, *Proceedings LOPSTR'92*. Springer-Verlag, LNCS, 1993.

[70] J. P. Gallagher. A system for specialising logic programs. Technical Report TR-91-32, Computer Science Department, University of Bristol, Great-Britain, November 1991.

[71] H. Gallaire and C. Lasserre. Metalevel control for logic programs. In K. L. Clark and S.-Å. Tärnlund, editors, *Logic Programming*, pages 173–185. Academic Press, 1982.

[72] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R. A. Kowalski and K. A. Bowen, editors, *Proceedings ICSLP'88*, pages 1070–1080, 1988.

[73] F. Giunchiglia and P. Traverso. Plan formation and execution in a uniform architecture of declarative metatheories. In M. Bruynooghe, editor, *Proceedings Meta'90*, pages 306–322, Leuven, April 1990.

[74] J. Grant and J. Minker. The impact of logic programming on databases. *Communications of the ACM*, 35(3):66–81, 1992.

[75] C. A. Gurr. *A Self-Applicable Partial Evaluator for the Logic Programming Language Gödel*. PhD thesis, Department of Computer Science, University of Bristol, Great-Britain, 1993. Submitted August 1993.

[76] C. A. Gurr. Specialising the ground representation for the logic programming language Gödel. In Y. Deville, editor, *Proceedings LOPSTR'93*, Louvain-la-Neuve, Belgium, 1994. Springer-Verlag, Workshops in Computing Series.

[77] P. Hammond. Micro-PROLOG for expert systems. In K. L. Clark and F. G. McCabe, editors, *Micro-PROLOG : Programming in Logic*, pages 294–319. Prentice-Hall, 1984.

[78] P. J. Hayes. The logic of frames. In D. Metzing, editor, *Frame Conceptions and Text Understanding*, pages 46–61. Walter de Gruyter and Co., Berlin, 1979. Reprinted in "Readings in Knowledge Representation", R. J. Brachman and H. J. Levesque, editors, Morgan Kaufmann, 1985.

[79] T. J. Hickey and D. A. Smith. Toward the partial evaluation of CLP languages. In *Proceedings PEPM'91, Sigplan Notices, 26(9)*, pages 43–51, New Haven, Connecticut, 1991. ACM.

[80] P. Hill and J. Lloyd. *The Gödel Programming Language*. MIT Press, 1993. To Appear.

[81] P. M. Hill and J. W. Lloyd. Meta-programming for dynamic knowledge bases. Technical Report CS-88-18, Computer Science Department, University of Bristol, Great-Britain, 1988.

[82] P. M. Hill and J. W. Lloyd. Analysis of meta-programs. In H. D. Abramson and M. H. Rogers, editors, *Proceedings Meta'88*, pages 23–51. MIT Press, 1989.

[83] T. Horváth. Experiments in partial deduction. Master's thesis, Departement Computerwetenschappen, K.U.Leuven, Leuven, Belgium, July 1993.

[84] Y. J. Jiang. On the semantics of real metalogic programming — a preliminary report. Technical report, Department of Computing, Imperial College, London, Great-Britain, July 1993.

[85] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall, 1993.

[86] N. D. Jones, P. Sestoft, and H. Søndergaard. Mix: A self-applicable partial evaluator for experiments in compiler generation. *Lisp and Symbolic Computation*, 2(1):9–50, 1989.

[87] M. Kalsbeek. The vanilla meta-interpreter for definite logic programs and ambivalent syntax. Technical Report CT-93-01, Department of Mathematics and Computer Science, University of Amsterdam, The Netherlands, January 1993.

[88] M. Kalsbeek and F. van Harmelen. Some more questions concerning "A metalogic programming approach to multi-agent knowledge and belief". In *Proceedings Benelog'92*, Luxembourg, September 1992. CRP-CU.

[89] J.-S. Kim and R. A. Kowalski. An application of amalgamated logic to multi-agent belief. In M. Bruynooghe, editor, *Proceedings Meta'90*, pages 272–283, Leuven, April 1990.

[90] S. C. Kleene. *Introduction to Meta-Mathematics.* North-Holland, 1952.

[91] D. E. Knuth, J. H. Morris, and V. R. Pratt. Fast pattern matching in strings. *SIAM Journal of Computation*, 6(2):323–350, 1977.

[92] J. Komorowski. *A Specification of an Abstract Prolog Machine and its Application to Partial Evaluation.* PhD thesis, Department of Computer and Information Science, Linköping University, Linköping, Sweden, 1981.

[93] J. Komorowski. An introduction to partial deduction. In A. Pettorossi, editor, *Proceedings Meta'92*, pages 49–69. Springer-Verlag, LNCS 649, 1992.

[94] J. Komorowski. Guest editor's introduction. *Journal of Logic Programming*, 16(1&2):1–3, 1993.

[95] J. Komorowski, editor. *Journal of Logic Programming, 16(1&2): Special Issue on Partial Deduction*, 1993.

[96] R. A. Kowalski. Predicate calculus as a programming language. In *Proceedings of the Sixth IFIP Congress*, pages 569–574. North Holland, 1974.

[97] R. A. Kowalski. *Logic for Problem Solving.* North-Holland, 1979.

[98] R. A. Kowalski. The early years of logic programming. *Communications of the ACM*, 31:38–43, 1988.

[99] R. A. Kowalski. Problems and promises of computational logic. In J. W. Lloyd, editor, *Proceedings of the Esprit Symposium on Computational Logic*, pages 1–36. Springer-Verlag, November 1990.

[100] R. A. Kowalski and J.-S. Kim. A metalogic programming approach to multi-agent knowledge and belief. In V. Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation*, pages 231–246. Academic Press, 1991.

[101] R. A. Kowalski and D. Kühner. Linear resolution with selection function. *Artificial Intelligence*, 2:227–260, 1971.

[102] A. Lakhotia. To PE or not to PE. In M. Bruynooghe, editor, *Proceedings Meta'90*, pages 218–228, Leuven, April 1990.

[103] A. Lakhotia and L. Sterling. How to control unfolding when specializing interpreters. *New Generation Computing*, 8(1):61–70, 1990.

[104] J. K. K. Lam and A. J. Kusalik. A comparative analysis of partial deductors for pure Prolog. Technical report, Department of Computational Science, University of Saskatchewan, Saskatoon, Saskatchewan, Canada, May 1990. Revised April 1991.

[105] J.-L. Lassez, M. J. Maher, and K. Marriott. Unification revisited. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 587–625. Morgan-Kaufmann, 1988.

[106] K.-K. Lau and T. Clement, editors. *Logic Program Synthesis and Transformation, Proceedings of LOPSTR'92*. Springer-Verlag, LNCS, 1993.

[107] M. Leuschel. Self-applicable partial evaluation in Prolog. Master's thesis, Departement Computerwetenschappen, K.U.Leuven, Leuven, Belgium, September 1993.

[108] G. Levi and D. Ramundo. A formalization of metaprogramming for real. In D. S. Warren, editor, *Proceedings ICLP'93*, pages 354–373, Budapest, June 1993. MIT Press.

[109] G. Levi and G. Sardu. Partial evaluation of metaprograms in a multiple worlds logic language. *New Generation Computing*, 6(2&3):227–247, 1988.

[110] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1987.

[111] J. W. Lloyd. Directions for meta-programming. In *Proceedings FGCS'88*, pages 609–617. ICOT, 1988.

[112] J. W. Lloyd. Designing logic programming languages. In P. Dewilde and J. Vandewalle, editors, *Computer Systems and Software Engineering*, pages 263–285. Kluwer Academic Publishers, 1992.

[113] J. W. Lloyd. Personal Communication, September 1993.

[114] J. W. Lloyd and J. C. Shepherdson. Partial evaluation in logic programming. *Journal of Logic Programming*, 11(3&4):217–242, 1991.

[115] J. W. Lloyd, E. A. Sonenberg, and R. W. Topor. Integrity checking in stratified databases. *Journal of Logic Programming*, 4(4):331–343, 1987.

[116] J. Lobo, J. Minker, and A. Rajasekar. *Foundations of Disjunctive Logic Programming*. MIT Press, 1992.

[117] P. Maes and D. Nardi, editors. *Meta-Level Architectures and Reflection*. North-Holland, 1988.

[118] Z. Manna and S. Ness. On the termination of Markov algorithms. In *Proceedings 3rd Hawaii Int. Conf. on Syst. Sci.*, pages 784–792, Honolulu, Hawaii, 1970.

[119] R. Manthey and F. Bry. SATCHMO: a theorem prover implemented in Prolog. In E. Lusk and R. Overbeek, editors, *Proceedings CADE'88*, pages 415–434. Springer-Verlag, LNCS 310, May 1988.

[120] B. Martens and M. Bruynooghe. Integrity constraint checking using a rule/goal graph. In L. Kerschberg, editor, *Proceedings Expert Database Systems '88*, pages 297–310, Tysons Corner, Virginia, April 1988.

[121] B. Martens and D. De Schreye. Why untyped non-ground meta-programming is not (much of) a problem. *Journal of Logic Programming*. To Appear.

[122] B. Martens and D. De Schreye. A perfect Herbrand semantics for untyped vanilla meta-programming. In K. Apt, editor, *Proceedings JICSLP'92*, pages 511–525, Washington, November 1992. MIT Press.

[123] B. Martens and D. De Schreye. Advanced techniques in finite unfolding. Technical Report CW180, Departement Computerwetenschappen, K.U.Leuven, Belgium, October 1993.

[124] B. Martens, D. De Schreye, and M. Bruynooghe. Sound and complete partial deduction with unfolding based on well-founded measures. In ICOT, editor, *Proceedings FGCS'92*, pages 473–480, Tokyo, June 1992. Omsha Ltd.

[125] B. Martens, D. De Schreye, and T. Horváth. Sound and complete partial deduction with unfolding based on well-founded measures. *Theoretical Computer Science*, 122(1-2):97–117, 1994.

[126] S. Minton. Quantitative results concerning the utility of explanation-based learning. *Artificial Intelligence*, 42:363–391, 1990.

[127] T. Mogensen and A. Bondorf. Logimix: A self-applicable partial evaluator for Prolog. In K.-K. Lau and T. Clement, editors, *Proceedings LOPSTR'92*. Springer-Verlag, LNCS, 1993.

[128] J.-M. Nicolas. Logic for improving integrity checking in relational databases. *Acta Informatica*, 18(3):227–253, 1982.

[129] S. Owen. Issues in the partial evaluation of meta-interpreters. In H. D. Abramson and M. H. Rogers, editors, *Proceedings Meta'88*, pages 319–339. MIT Press, 1989.

[130] A. Pettorossi, editor. *Meta-Programming in Logic, Proceedings of Meta'92*. Springer-Verlag, LNCS 649, 1992.

[131] F. Pfenning, editor. *Types in Logic Programming*. MIT Press, 1992.

[132] G. Plotkin. A note on inductive generalization. In B. Meltzer and D. Michie, editors, *Machine Intelligence 5*, pages 153–163, 1970.

[133] L. Plümer. *Termination Proofs of Logic Programs*. Springer-Verlag, LNCS 446, 1990.

[134] S. Prestwich. Online partial deduction of large programs. In *Proceedings PEPM'93*, pages 111–118, Copenhagen, Denmark, June 1993. ACM.

[135] M. Proietti and A. Pettorossi. The loop absorption and the generalization strategies for the development of logic programs and partial deduction. *Journal of Logic Programming*, 16(1&2):123–161, 1993.

[136] H. Przymusinska and T. C. Przymusinski. Weakly perfect model semantics for logic programs. In R. A. Kowalski and K. A. Bowen, editors, *Proceedings ICSLP'88*, pages 1106–1120, 1988.

[137] H. Przymusinska and T. C. Przymusinski. Semantic issues in deductive databases and logic programs. In R. B. Banerji, editor, *Formal Techniques in Artificial Intelligence*, pages 321–367. Elsevier Science Publishers B.V., 1990.

[138] H. Przymusinska and T. C. Przymusinski. Weakly stratified logic programs. *Fundamenta Informaticae*, XIII:51–65, 1990.

[139] T. C. Przymusinski. On the declarative semantics of deductive databases and logic programs. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 193–216. Morgan-Kaufmann, 1988.

[140] T. C. Przymusinski. Every logic program has a natural stratification and an iterated least fixed point model. In *Proceedings PODS'89*, pages 11–21, Philadelphia, Pennsylvania, USA, March 1989. ACM.

[141] T. C. Przymusinski. On the declarative and procedural semantics of logic programs. *Journal of Automated Reasoning*, 5(2):167–205, 1989.

[142] A. Ramsay. *Formal Methods in Artificial Intelligence*. Cambridge University Press, 1988.

[143] J. Reynolds. Transformational systems and the algebraic structure of atomic formulas. In B. Meltzer and D. Michie, editors, *Machine Intelligence 5*, pages 135–152, 1970.

[144] B. Richards. A point of reference. *Synthese*, 28:361–454, 1974.

[145] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12:23–41, 1965.

[146] J. A. Robinson. Logic and logic programming. *Communications of the ACM*, 35(3):40–65, 1992.

[147] K. A. Ross. A procedural semantics for well founded negation in logic programs. In *Proceedings PODS'89*, pages 22–33, Philadelphia, Pennsylvania, USA, March 1989. ACM.

[148] K. A. Ross. Modular stratification and magic sets for DATALOG programs with negation. In *Proceedings PODS'90*, pages 161–171, Nashville, Tennessee, April 1990. ACM.

[149] P. Roussel. Contribution to the pannel on the history of Prolog. JICSLP'92, November 1992.

[150] F. Sadri and R. Kowalski. A theorem-proving approach to database integrity. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 313–362. Morgan-Kaufmann, 1988.

[151] S. Safra and E. Shapiro. Meta interpreters for real. In H.-J. Kugler, editor, *Information Processing 86*, pages 271–278, 1986.

[152] D. Sahlin. The Mixtus approach to automatic partial evaluation of full Prolog. In S. Debray and M. Hermenegildo, editors, *Proceedings NACLP'90*, pages 377–398, Austin, Texas, October 1990. MIT Press.

[153] D. Sahlin. *An Automatic Partial Evaluator for Full Prolog*. PhD thesis, Kungliga Tekniska Högskolan, Stockholm, Sweden, 1991.

[154] T. Sato. Meta-programming through a truth predicate. In K. Apt, editor, *Proceedings JICSLP'92*, pages 526–540, Washington, November 1992. MIT Press.

[155] P. Sestoft and A. V. Zamulin. Annotated bibliography on partial evaluation and mixed computation. *New Generation Computing*, 6(2&3):309–354, 1988.

[156] D. A. Smith. Partial evaluation of pattern matching in constraint logic programming languages. In *Proceedings PEPM'91, Sigplan Notices, 26(9)*, pages 62–71, New Haven, Connecticut, 1991. ACM.

[157] D. A. Smith and T. J. Hickey. Partial evaluation of a CLP language. In S. Debray and M. Hermenegildo, editors, *Proceedings NACLP'90*, pages 119–138, Austin, Texas, October 1990. MIT Press.

[158] L. Sterling and R. D. Beer. Meta interpreters for expert system construction. *Journal of Logic Programming*, 6(1&2):163–178, 1989.

[159] L. Sterling and E. Shapiro. *The Art of Prolog*. MIT Press, 1986.

[160] V. S. Subrahmanian. A simple formulation of the theory of metalogic programming. In H. D. Abramson and M. H. Rogers, editors, *Proceedings Meta'88*, pages 65–101. MIT Press, 1989.

[161] A. Takeuchi and K. Furukawa. Partial evaluation of Prolog programs and its application to metaprogramming. In H.-J. Kugler, editor, *Information Processing 86*, pages 415–420, 1986.

[162] H. Tamaki and T. Sato. Unfold/fold transformation of logic programs. In S-Å. Tärnlund, editor, *Proceedings ICLP'84*, pages 127–138, Uppsala, July 1984.

[163] R. W. Topor and E. A. Sonenberg. On domain independent databases. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 217–240. Morgan-Kaufmann, 1988.

[164] J. L. Träff and S. D. Prestwich. Meta-programming for reordering literals in deductive databases. In A. Pettorossi, editor, *Proceedings Meta'92*, pages 280–293. Springer-Verlag, LNCS 649, 1992.

[165] V. F. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):292–325, 1986.

[166] D. Turi. Extending S-models to logic programs with negation. In K. Furukawa, editor, *Proceedings ICLP'91*, pages 397–411, Paris, June 1991. MIT Press.

[167] J. D. Ullman. *Database and Knowledge-Base Systems, Volume I.* Computer Science Press, 1988.

[168] J. D. Ullman. *Database and Knowledge-Base Systems, Volume II.* Computer Science Press, 1989.

[169] J. D. Ullman and A. Van Gelder. Efficient tests for top-down termination of logical rules. *Journal of the ACM*, 35(2):345–373, 1988.

[170] M. H. van Emden and R. A. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM*, 23(4):733–742, 1976.

[171] A. Van Gelder. The alternating fixpoint of logic programs with negation (extended abstract). In *Proceedings PODS'89*, pages 1–10, Philadelphia, Pennsylvania, USA, March 1989. ACM.

[172] A. Van Gelder, K. A. Ross, and J. S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, 1991.

[173] F. van Harmelen. The limitations of partial evaluation. In P. Jackson, H. Reichgelt, and F. van Harmelen, editors, *Logic-Based Knowledge Representation*, pages 87–111. MIT Press, 1989.

[174] F. van Harmelen and A. Bundy. Explanation based generalisation = partial evaluation. *Artificial Intelligence*, 36(3):401–412, 1988.

[175] R. Venken. A Prolog meta interpreter for partial evaluation and its application to source to source transformation and query optimization. In T. O'Shea, editor, *Advances in Artificial Intelligence, Proceedings ECAI'84*, pages 347–356. North-Holland, 1984.

[176] R. Venken and B. Demoen. A partial evaluation system for Prolog : Some practical considerations. *New Generation Computing*, 6(2&3):279–290, 1988.

[177] R. W. Weyhrauch. Prolegomena to a theory of mechanized formal reasoning. *Artificial Intelligence*, 13(1&2):133–170, 1980.

Over de semantiek van meta-progamma's en het beheersen van partiële deductie in logisch programmeren

Bernhard Martens

Departement Computerwetenschappen, K.U.Leuven

Samenvatting

In logisch programmeren is meta-programmeren verdedigd als een krachtige en makkelijk realiseerbare techniek om de mogelijkheden voor kennisvoorstelling en automatisch redeneren te vergroten. Vele praktische toepassingen leken echter een duidelijke semantische grondslag te ontberen.

In het eerste deel van het proefschrift wordt daarom een semantiek bestudeerd voor vanille meta-programma's zonder types, waarin veranderlijken die figureren op het object-niveau, worden voorgesteld door middel van veranderlijken. Niet alleen de elementaire vanille vertolker is voorwerp van studie, maar ook enkele interessante uitbreidingen, met inbegrip van programma's die een beperkte vorm van versmelting tussen object- en meta-niveau toelaten. Dergelijke meta-programma's blijken zwak gelaagd indien hun onderliggende object-programma gelaagd is. Voor een grote klasse van object-programma's stelt men een natuurlijke overeenkomst vast tussen hun perfecte Herbrand model en het zwak perfecte meta-model. Zodoende is voor dergelijke programma's een zinnige meta-semantiek bepaald. Voor positieve programma's kan men deze resultaten bovendien veralgemenen, gebruik makend van een uitgebreide Herbrand semantiek, ontworpen om het operationeel gedrag van logische programma's beter te vatten.

Meta-programmeren wordt ook belemmerd door een ondermaatse uitvoeringsefficiëntie. Partiële deductie kan dit euvel verhelpen. Bovendien is deze techniek ter specialisering van programma's ook relevant in andere contexten.

Daarom behandelt dit proefschrift in een tweede deel partiële deductie van positieve logische programma's. Veel aandacht gaat uit naar het beheersen van ontvouwen. In het bijzonder worden technieken onderzocht om ervoor te zorgen dat deze laatste operatie steeds eindigt, en wel op een manier die strukturele eigenschappen van de beschouwde vragen en programma's weerspiegelt. Een algemeen, formeel kader wordt voorgesteld, evenals diverse volledig automatische algoritmen. Bovendien wordt een eveneens geheel automatische methode voor partiële deductie behandeld. Een aantal interessante eigenschappen van de beschouwde algoritmen blijken formeel bewijsbaar, ondermeer het feit dat ze altijd eindigen. Tenslotte komen enige resultaten van een proefondervindelijke studie kort aan bod.

Inhoudsopgave

1 Inleiding

In logisch programmeren is meta-programmeren verdedigd als een krachtige en in principe makkelijk realiseerbare techniek om de mogelijkheden voor kennisvoorstelling en automatisch redeneren te vergroten. In het algemeen kan "redeneren op een meta-niveau" gezien worden als "redeneren betreffende redeneren". Vertaald naar logisch programmeren resulteert zulks in programma's die andere programma's als gegevens behandelen, als invoer nemen, analyseren, transformeren, en/of als uitvoer produceren. Talrijke toepassingen (zie de verwijzingen in sectie 2.1) illustreren het praktisch nut en de conceptuele eenvoud van "logisch meta-programmeren".

Menige toepassing leek echter een goede semantiek te ontberen. Vooral de logische status van programma's met "versmelting" (*amalgamation*) van object- en meta-niveau is veelal erg onduidelijk. Een degelijke logische semantiek is nu net één van de belangrijkste voordelen van een logisch programma, en helderheid op dit vlak is dan ook gewenst. In hoofdstuk 2 bestuderen we daarom de semantiek van meta-programma's behorend tot een bepaald, wijd verspreid type, waarbij object-veranderlijken op het meta-niveau eveneens door veranderlijken worden voorgesteld. We blijven zo dicht mogelijk bij de algemeen gangbare Herbrand semantiek voor logische programma's en bewijzen interessante overeenkomsten tussen de semantiek op het object- en die op het meta-niveau.

Logisch programmeren verschaft niet enkel een werktuig voor de voorstelling van kennis. Het is ook (en in zekere zin vooral) een programmeerparadigma dat de gelegenheid biedt programma's te schrijven en uit te voeren. Maar wie programmeert, dient aandacht te schenken aan overwegingen in verband met efficiëntie. Een niet onaanzienlijk deel van het onderzoek aangaande logisch programmeren behandelt technieken en methodes om de programmeur enigszins van deze taak te ontlasten. Het computersysteem zelf zou dan in grote mate verantwoordelijk zijn voor het bepalen van efficiënte uitvoeringsstrategieën, eventueel via (automatische) programmatransformaties en -specialisaties.

Naast zijn tweevoudige natuur als kennisvoorstellings- en programmeerwerktuig, blijkt ook de formele semantiek van logisch programmeren een onschatbaar hulpmiddel bij een dergelijke onderneming. Inderdaad, dit laatste aspect maakt het mogelijk programma's te vergelijken volgens hun "betekenis", afstand nemend van eventuele operationele verschillen. Zo worden belangrijke stellingen omtrent de semantische equivalentie van oorspronkelijke en getransformeerde en/of gespecialiseerde programma's formuleerbaar. Een welbekend voorbeeld van die aard verschaft de stelling van Lloyd en Sherpherdson omtrent de correctheid en de volledigheid van partiële deductie (zie stelling 4.5).

Precies partiële deductie, een specifieke techniek voor programmaspecialisatie, ook wel bekend onder de naam "partiële evaluatie", wordt behandeld in hoofdstukken 3 tot 5. De centrale doelstelling in partiële deductie is het (automatisch) specialiseren van een gegeven programma met betrekking tot gedeeltelijk bekende invoer. Het aldus verkregen meer specifieke programma zou konkrete waarden voor de rest van de invoer sneller moeten kunnen verwerken dan de oorspronkelijke algemene versie.

Een belangrijke toepassing van partiële deductie betreft het specialiseren van meta-vertolkers voor verschillende gegeven object-programma's. De aldus geproduceerde programma's zijn vrij van de extra bewerkingen die samenhangen met de meta-vertolking, hetgeen vrijwel altijd een zeer aanzienlijke winst aan efficiëntie oplevert. Zelfs honderdvoudige verbeteringen werden verkregen voor Gödel meta-vertolkers ([75]). Praktische aanwending van meta-programmatie in logisch programmeren lijkt dan ook niet denkbaar zonder dergelijke specialisatietechnieken.

Hiermee is meteen het verband aangegeven tussen hoofdstuk 2 enerzijds en hoofdstukken 3 tot 5 anderzijds. Inderdaad, zoals hoger reeds vermeld werd, gaat vanaf hoofdstuk 3 onze aandacht niet langer in de eerste plaats naar semantische aspecten van meta-programmatie, maar bestuderen we meer operationele onderwerpen. We behandelen partiële deductie van logische programma's (niet enkel meta-vertolkers) waarbij veel aandacht uitgaat naar één specifiek deelprobleem: het beheersen van ontvouwen (*unfolding*), met name het verzekeren van de eindigheid ervan. In hoofdstuk 3 stellen we vooreerst een algemeen kader voor waarin eindigende methodes voor het ontvouwen van positieve (*definite*) logische programma's en vragen geformuleerd worden. Vervolgens behandelen we automatische partiële deductie met dergelijk eindig ontvouwen in hoofdstuk 4. Hoofdstuk 5, op zijn beurt, stelt diverse verfijningen voor betreffende automatisch eindig ontvouwen. Tenslotte is een kort algemeen besluit begrepen in hoofdstuk 6.

2 Herbrand semantiek van meta-programma's

Talrijke vorsers hebben meta-programmatie in logisch programmeren bestudeerd; zie bijvoorbeeld [18], [99], [161], [64], [159], [77], [158], [28], [164], [29]. Ondermeer de verslagen van de tweejaarlijkse gespecialiseerde META conferenties bevatten verder interessant materiaal ([1], [24] en [130]).

Heel wat toepassingen zijn gebaseerd op meta-programma's waarbij objectveranderlijken op het meta-niveau worden voorgesteld door veranderlijken. Dergelijke meta-vertolkers kunnen gebruik maken van de ingebouwde unificatie en resolutie. Dit leidt tot eenvoudige en relatief efficiënte programma's; de welbekende "vanille" meta-vertolker is een standaard voorbeeld en vormt de kern van talrijke nuttige varianten. Een alternatieve mogelijkheid behelst het gebruik van constanten (of, meer algemeen, volledig bepaalde (*ground*) termen) ter representatie van object-veranderlijken. Deze laatste strategie werd bijvoorbeeld weerhouden als basis voor meta-programmatie in Gödel ([80]).

In dit hoofdstuk onderzoeken we de semantische grondslagen van vanille-achtige meta-programma's. Daartoe benutten we het gebruikelijke kader met typeloze Herbrand interpretaties en modellen. Vooraleer we resultaten op dit vlak kunnen voorstellen, dienen we echter eerst twee, bij de verdere behandeling cruciale, begrippen in te voeren.

2.1 Twee inleidende begrippen

Taalonafhankelijkheid

In het algemeen hangt het perfecte Herbrand model van een normaal (*normal*) gelaagd (*stratified*) programma af van de taal waarin we het programma beschouwen en het model bouwen. We zullen een programma taalonafhankelijk noemen als zulks niet het geval is.

Een logische taal wordt bepaald door verzamelingen $\mathcal{R}$, $\mathcal{F}$ en $\mathcal{C}$, met respectievelijk symbolen voor predikaten, functies en constanten. Indien P een programma is en $\mathcal{R}_P$, $\mathcal{F}_P$ en $\mathcal{C}_P$ de verzamelingen van symbolen die in het programma voorkomen, dan noemen we $\mathcal{L}_P$, aldus bepaald[1], P's inherente taal (*underlying language*).

We definiëren:

Definitie 2.1 Zij P een normaal programma met inherente taal $\mathcal{L}_P$. Een taal $\mathcal{L}'$, bepaald door $\mathcal{R}'$, $\mathcal{F}'$ en $\mathcal{C}'$, noemen we een *uitbreiding* van $\mathcal{L}_P$ asa $\mathcal{R}_P \subseteq \mathcal{R}'$, $\mathcal{F}_P \subseteq \mathcal{F}'$ en $\mathcal{C}_P \subseteq \mathcal{C}' \neq \emptyset$.

Definitie 2.2 Zij P een normaal programma met inherente taal $\mathcal{L}_P$. Een Herbrand interpretatie van P in een taal $\mathcal{L}'$, uitbreiding van $\mathcal{L}_P$, is een *$\mathcal{L}'$-Herbrand interpretatie* van P.

Definitie 2.3 We noemen een gelaagd programma P met inherente taal $\mathcal{L}_P$ *taalonafhankelijk* asa het perfecte $\mathcal{L}'$-Herbrand model van P gelijk is aan het perfecte $\mathcal{L}_P$-Herbrand model van P voor elke uitbreiding $\mathcal{L}'$ van $\mathcal{L}_P$.

Taalonafhankelijkheid is een veralgemening van het welbekende "beperkt bereik" (*range restriction*) concept.

Definitie 2.4 Een regel in een programma P heeft een *beperkt bereik* asa elke veranderlijke in de regel voorkomt in een (positief) atoom in het lichaam van de regel. Een programma P heeft een *beperkt bereik* asa het enkel regels met een beperkt bereik bevat.

Stelling 2.5 Zij P een gelaagd programma. Indien P een beperkt bereik heeft, dan is P taalonafhankelijk.

[1] Als $\mathcal{C}_P = \emptyset$ dan nemen we een willekeurige constante $*$ in $\mathcal{L}_P$.

Tenslotte kunnen we opmerken dat, aangezien voor positieve programma's het perfecte en het (unieke) kleinste (*least*) Herbrand model samenvallen, de bovenstaande definities en resultaten rechtstreeks toepasbaar zijn in een dergelijke, meer beperkte context.

Zwak gelaagde programma's en zwak perfecte modellen

Verderop wensen we meta-programma's te bestuderen die overeenkomen met gelaagde object-programma's. Nu blijken zulke meta-programma's zelf niet gelaagd te zijn. En hun semantiek kan dus niet worden beschreven met behulp van een perfect Herbrand model. De meer algemene notie van zwakke gelaagdheid ([138]) biedt echter soelaas. In dit beperkte bestek is een gedetailleerde opbouw van de relevante begrippen helaas niet mogelijk. We beperken ons tot een informele beschrijving die voldoende zou moeten zijn als achtergrond bij de resultaten in de rest van dit hoofdstuk.

De kerngedachte bij zwakke gelaagdheid is dat recursie via negatie voldoende beperkt is om, ondanks de afwezigheid van gelaagdheid, toch de berekening van één welbepaald "intuïtief juist" (tweewaardig) Herbrand model toe te laten. Om te beslissen of een gegeven normaal programma P zwak gelaagd is en, in dat geval, zijn zwak perfecte Herbrand model te berekenen, kan men ruwweg als volgt te werk gaan. Beschouw alle volledig bepaalde instanties (*ground instances*) van regels in P. Neem een voldoend grote verzameling A van volledig bepaalde atomen die niet negatief van andere atomen afhangen (direct dan wel indirect). Als er zo geen atomen zijn, dan is het programma *niet zwak gelaagd* en de constructie faalt. In het andere geval vormen de regelinstanties met een hoofding in A een positief logisch programma Pg en A dient zo gekozen dat alle atomen die voorkomen in dit programma in A zitten (in die zin moet A "voldoend groot" wezen). Bereken het kleinste Herbrand model H_{Pg} van Pg. Bekijk vervolgens de overgebleven regelinstanties (degene die niet in Pg terechtgekomen zijn). Schrap regelinstanties met in het lichaam een doel (*goal*) waarvan het atoom in A zit, maar dat niet voldaan is volgens H_{Pg}. Verwijder uit de resterende regelinstanties alle doelen met een atoom in A. Herhaal de hele constructie op de nieuwe, kleinere verzameling volledig bepaalde regels. Indien tenslotte geen regels meer overblijven, slaagt de constructie: alle ogenschijnlijke lussen via negatie konden dynamisch worden verwijderd. P heet *zwak gelaagd* en de unie van de berekende kleinste Herbrand modellen het *zwak perfecte* Herbrand model van P.

Voorbeeld 2.6 Het volgende programma is niet gelaagd en ook niet locaal gelaagd, maar wel zwak gelaagd:

$p(1,2) \leftarrow$
$q(X) \leftarrow p(X,Y), \mathit{not}\ q(Y)$

2.2 Vanille en andere smaken

Nu kunnen we de eigenlijke studie van meta-programma's en hun semantiek aanvatten.

Vanille

We definiëren vanille meta-programma's zoals in [158] en [82].

Definitie 2.7 Het volgende normale programma M noemen we *vanille meta-vertolker*:

$solve(empty) \leftarrow$
$solve(X\&Y) \leftarrow solve(X), solve(Y)$
$solve(\neg X) \leftarrow not\ solve(X)$
$solve(X) \leftarrow clause(X, Y), solve(Y)$

Definitie 2.8 Zij P een normaal programma. Dan weze M_P, het *vanille meta-programma verbonden met* P, het normale programma gevormd door M samen met een feit

$clause(A, \ldots \&B\& \ldots \&\neg C\& \ldots) \leftarrow$

voor elke regel $A \leftarrow \ldots, B, \ldots, notC, \ldots$ in P en een feit

$clause(A, empty) \leftarrow$

voor elk feit $A \leftarrow$ in P.

Een eerste belangrijk resultaat volgt.

Stelling 2.9 Zij P een gelaagd normaal programma. Dan is M_P zwak gelaagd.

We voeren enige notatie in:

- U_P: het Herbrand universum van een programma P
- ${U_P}^n = U_P \times \ldots \times U_P$ (n keer)
- p/r: een predikaatsymbool met ariteit r in $\mathcal{R}_P$

Nu kunnen we de centrale stelling van dit hoofdstuk formuleren.

Stelling 2.10 Zij P een taalonafhankelijk gelaagd programma en M_P het verbonden vanille meta-programma. Laat H_P het perfecte Herbrand model van P voorstellen en H_{M_P} het zwak perfecte Herbrand model van M_P. Dan geldt het volgende voor elke $p/r \in \mathcal{R}_P$:

$\forall \bar{t} \in {U_{M_P}}^r : \ solve(p(\bar{t})) \in H_{M_P} \Longleftrightarrow \bar{t} \in {U_P}^r \ \& \ p(\bar{t}) \in H_P$

Voor taalonafhankelijke programma's bevat het zwak perfecte model van het meta-programma dus zinnige informatie. Een eenvoudig voorbeeld toont aan dat taalonafhankelijkheid cruciaal is.

Voorbeeld 2.11
Stel dat P bestaat uit de feiten $p(X) \leftarrow$ en $q(a) \leftarrow$.
Dan verkrijgen we

$$H_P = \{p(a), q(a)\}$$

en nochtans

$$solve(p(empty)), solve(p(q(a)), \ldots \in H_{M_P}$$

Uitbreidingen

De vanille meta-vertolker is een boeiend studie-object vanuit theoretisch oogpunt omdat hij de essentie van talrijke meta-programma's weergeeft. Maar meer praktisch belang hebben verwante programma's met meer argumenten in *solve* en/of bijkomende doelen (betreffende predikaten die afzonderlijk gedefiniëerd zijn) in de lichamen van de regels. Dergelijke meta-programma's zullen we *uitgebreid* noemen. Betreffende de semantiek van dergelijke gevallen hebben we eveneens enkele algemene resultaten.

We beperken ons in eerste instantie tot *positieve* object-programma's en meta-programma's zonder een regel voor negatie (hierna aangeduid door de "d" annotatie).

Stelling 2.12 Zij P een taalonafhankelijk positief programma en E_P een uitgebreid d-meta-programma verbonden met P. Laat H_P en H_{E_P} hun respectievelijke kleinste Herbrand modellen voorstellen. Dan geldt het volgende voor elke $p/r \in \mathcal{R}_P$:

$$\forall \bar{t} \in {U_{E_P}}^r : (\exists \bar{s} \in {U_{E_P}}^n : solve(p(\bar{t}), \bar{s}) \in H_{E_P}) \Longrightarrow \bar{t} \in {U_P}^r \ \& \ p(\bar{t}) \in H_P$$

Voor specifieke uitgebreide meta-programma's is natuurlijk een equivalentie mogelijk in stelling 2.12. Het bewijsboom bouwende programma in [159] verschaft een voorbeeld.

Laten we vervolgens *gelaagde* object-programma's en hun normale uitgebreide meta-programma's beschouwen. Ten eerste blijken dergelijke meta-programma's weer zwak gelaagd als hun eventuele extra predikaten gedefiniëerd zijn via een gelaagd programma. En dan verkrijgen we:

Stelling 2.13 Zij P een gelaagd programma *met beperkt bereik* en E_P een uitgebreid meta-programma verbonden met P. Laat H_{E_P} het zwak perfecte Herbrand model van E_P voorstellen. Dan geldt het volgende voor elke $p/r \in \mathcal{R}_P$:

$$\forall \bar{t} \in {U_{E_P}}^r : (\exists \bar{s} \in {U_{E_P}}^n : solve(p(\bar{t}), \bar{s}) \in H_{E_P}) \Longrightarrow \bar{t} \in {U_P}^r$$

Het blijkt in het algemeen niet mogelijk $p(\bar{t}) \in H_P$ op te nemen in het rechterlid. In het licht van bepaalde toepassingen lijkt zulks gepast. Maar bovendien geldt zelfs het beperkte resultaat in stelling 2.13 niet voor sommige object-programma's die wel taalonafhankelijk zijn, maar geen beperkt bereik hebben.

2.3 Versmelting

Het is mogelijk een verdere stap te zetten: object- en meta-niveau kunnen samengenomen worden in één enkel programma. Deze praktijk noemen we *versmelting* en de resulterende programma's heten *versmolten.*

In dergelijke gevallen figureren dezelfde symbolen nu eens als predikaatsymbolen, dan weer als functoren. Dit is ongebruikelijk zowel in logica, als in de meer beperkte context van logisch programmeren. Men kan echter verifiëren dat geen noemenswaardige semantische problemen opduiken. In het bijzonder blijft de resulterende logica vrij van paradoxen.

Als we op deze wijze de betekenis van symbolen overladen, kunnen we in een eerste stap P en M_P tekstueel samenvoegen. De voor de hand liggende resultaten zijn makkelijk bewijsbaar. En programma's met *neerwaartse reflectie* passen in dit conceptuele kader.

Boeiender vanuit theoretisch oogpunt is het toevoegen van meta-feiten voor *solve* (en *clause*) zelf.

Definitie 2.14 Zij P een normaal programma. Dan weze $M^2{}_P$, het *vanille meta2-programma verbonden met P*, het programma M (zie definitie 2.7) samen met de volgende regel:

$$clause(clause(X,Y), empty) \leftarrow clause(X,Y) \qquad (*)$$

en een feit

$$clause(A, \ldots \& B \& \ldots \& \neg C \& \ldots) \leftarrow$$

voor elke regel $A \leftarrow \ldots, B, \ldots, notC, \ldots$ in P of M en een feit

$$clause(A, empty) \leftarrow$$

voor elk feit $A \leftarrow$ in P of M.

Bemerk dat regel $(*)$ alle gevallen van "feiten over feiten" dekt.

We verkrijgen:

Stelling 2.15 Zij P een gelaagd programma. Dan is $M^2{}_P$ zwak gelaagd.

Stelling 2.16 Zij P een taalonafhankelijk gelaagd programma en $M^2{}_P$ het verbonden vanille meta2-programma. Laat H_P het perfecte Herbrand model van P voorstellen en $H_{M^2{}_P}$ het zwak perfecte Herbrand model van $M^2{}_P$. Dan is waar:

$$\forall t \in U_{M^2{}_P} : \ solve(solve(t)) \in H_{M^2{}_P} \iff solve(t) \in H_{M^2{}_P}$$

Bovendien geldt voor elke $p/r \in \mathcal{R}_P$:

$$\forall \bar{t} \in U_{M^2{}_P}{}^r : \ solve(p(\bar{t})) \in H_{M^2{}_P} \iff \bar{t} \in U_P{}^r \ \& \ p(\bar{t}) \in H_P$$

Deze resultaten kunnen een basis zijn voor de studie van programma's met *opwaartse reflectie*, waar er niet langer een wezenlijk onderscheid gehandhaafd wordt tussen object- en meta-programma.

2.4 S-semantiek

De voorgaande secties behandelen meta-programma's in het kader van de standaard Herbrand semantiek. Taalonafhankelijkheid blijkt daarbij een belangrijke rol te spelen. Nu kan men zich afvragen waarom deze voorwaarde nauwelijks of niet opduikt in praktische toepassingen. Welnu, Herbrand modellen bevatten volledig bepaalde atomen, terwijl programma's dikwijls niet volledig bepaalde (*non-ground*) antwoordsubstituties opleveren. Deze laatste zijn zinvol bij vele meta-vertolkers, ook als het object-programma niet taalonafhankelijk is.

In [55] en [56] wordt voor positieve logische programma's een semantiek voorgesteld die hun operationele gedrag getrouwer weerspiegelt, de zogenaamde *S-semantiek*. In essentie worden nog steeds Herbrand interpretaties en modellen gebruikt, maar deze kunnen willekeurige atomen in de gegeven taal (en niet enkel volledig bepaalde) als elementen bevatten. Men toont aan dat elk positief programma één enkel kleinste S-Herbrand model heeft, berekenbaar als vast punt van een karakteristieke operator. Bovendien beschrijft dit model exact de antwoordsubstituties voor vragen met enkel veranderlijken als argumenten.

Wanneer men nu de S-semantiek van *vanille* meta-programma's onderzoekt, blijkt dat, voor positieve programma's, stelling 2.10 kan worden veralgemeend naar niet taalonafhankelijke.

Stelling 2.17 Zij P een positief programma en M_P het verbonden vanille d-meta-programma. Laat ${H^S}_P$ het kleinste S-Herbrand model van P voorstellen en ${H^S}_{M_P}$ het kleinste S-Herbrand model van M_P. Dan geldt het volgende voor elke $p/r \in \mathcal{R}_P$:

$$\forall \bar{t} \in {{U^S}_{M_P}}^r : solve(p(\bar{t})) \in {H^S}_{M_P} \Longleftrightarrow \bar{t} \in {{U^S}_P}^r \,\&\, p(\bar{t}) \in {H^S}_P$$

Bij het beschouwen van *uitgebreide* meta-programma's duikt taalonafhankelijkheid echter wel weer op. Verder onderzoek kan uitwijzen welke beperkingen op de aard der uitbreiding hieraan verhelpen. (Het al eerder vernoemde bewijsboom bouwende programma stelt bijvoorbeeld geen probleem.)

2.5 Besluit

In dit hoofdstuk hebben we de Herbrand semantiek van een belangrijke klasse meta-programma's onderzocht. Taalonafhankelijkheid en zwakke gelaagdheid bleken beide een sleutelrol te spelen bij zulke onderneming. Via een eenvoudige overladingstechniek was het ook mogelijk bepaalde vormen van versmelting te beschouwen.

Een verdere studie van wezenlijk meer complexe object- en meta-programma's (bijvoorbeeld met een volledige behandeling van kwantoren en/of meer exotische vormen van versmelting) vergt wellicht een meer gespecialiseerd semantisch apparaat.

3 Een raamwerk voor eindig ontvouwen

Zoals reeds aangekondigd in hoofdstuk 1, verleggen we vanaf nu onze aandacht naar meer operationele beslommeringen. Met name partiële deductie en in de eerste plaats ontvouwen, een belangrijk onderdeel daarvan, worden bestudeerd. Globale partiële deductie komt aan bod in het volgende hoofdstuk. In het huidige stellen we eerst een algemeen raamwerk voor betreffende het bouwen van eindige SLD-bomen bij het ontvouwen van logische programma's. Tenslotte dient aangestipt dat zowel dit hoofdstuk als de volgende expliciet enkel *positieve* programma's behandelen.

De structuur van het onderhavige hoofdstuk is als volgt. Vooreerst worden eindige SLD-bomen statisch gekarakteriseerd in sectie 3.1. Vervolgens leiden we een operationeel bruikbaar concept af in sectie 3.2 en een meer verfijnde variant in sectie 3.3. Tevens formuleren we algoritmen voor eindig ontvouwen. Sectie 3.4, tenslotte, bereidt de weg voor volledige automatisering, een onderwerp dat aan bod komt in sectie 4.1 en verder uitgediept wodt in hoofdstuk 5.

3.1 Welgegronde en gedeeld gegronde SLD$^-$-bomen

Welgegronde verzamelingen en bomen

Naast succesvolle en falende takken laten we verder in SLD-bomen ook *onvolledige* takken toe waarbij het blad een willekeurige (conjunctie van) doel(en) kan bevatten. Een boom met één of meer onvolledige takken heet *onvolledig*, zonder is hij *volledig*. Indien een programma P, een doel A en een computatieregel R gegeven zijn, dan bestaat er slechts één volledige (eventueel oneindige) SLD-boom τ_0 voor $P \cup \{\leftarrow A\}$ via R. Eindige SLD-bomen voor $P \cup \{\leftarrow A\}$ via R zijn deelbomen (met dezelfde wortel) van τ_0. Bij gelegenheid zullen we veronderstellen dat alle knopen in τ_0, en dus ook in elke deelboom, een uniek nummer bezitten. En we zullen een dergelijke knoop (een (conjunctie van) doel(en) dus) dan ook soms aanduiden met behulp van een koppel (G, i), waarbij i het natuurlijk getal voorstelt dat als nummer werd toegekend. Voor een SLD-boom τ, zal G_τ de aldus verkregen verzameling koppels aanduiden. Tenslotte ordenen we G_τ via een orderelatie $>_\tau$ die de voorouder-afstammeling (*ancestor-descendent*) relatie tussen knopen in τ weerspiegelt.

We definiëren nu formeel nog enkele benodigde concepten.

Definitie 3.1 Zij P een programma en $\leftarrow A$ een doel. Dan noemen we om het even welke deelboom van een SLD-boom τ voor $P \cup \{\leftarrow A\}$ met dezelfde wortel als τ een *SLD$^-$-boom* (voor $P \cup \{\leftarrow A\}$).

Voor elke SLD$^-$-boom τ is er een *unieke kleinste omvattende SLD-boom*, genoteerd als τ^+.

Definitie 3.2 Zij $V, >$ een strikt (eventueel partiëel) geordende verzameling. We noemen $V, >$ *welgegrond* als er geen oneindige serie elementen $e_1, e_2, \ldots$ in V bestaat waarvoor geldt $e_i > e_{i+1}$ voor alle $i \geq 1$.

Definitie 3.3 Zij $V, >$ een strikt (eventueel partiëel) geordende verzameling. Een *welgegronde maat op* $V, >_V$ is een monotone functie van $V, >_V$ naar een welgegronde verzameling $W, >_W$.

Definitie 3.4 Een SLD$^-$-boom τ is *welgegrond* als er een welgegronde maat bestaat op $G_\tau, >_\tau$.

Dan kunnen we nu eindige bomen kenschetsen als volgt:

Stelling 3.5 Een SLD$^-$-boom is eindig asa hij welgegrond is.

Nu blijkt echter deze beschrijving niet voldoende soepel. We voeren daarom een tweede, makkelijker bruikbaar concept in.

Gedeeld gegronde bomen

Definitie 3.6 Een SLD$^-$-boom τ is *gedeeld gegrond* indien

1. er een eindig aantal verzamelingen $C_0, \ldots, C_N$ bestaan zodanig dat
   $G_\tau = \bigcup_{i \leq N} C_i$
2. er voor elke $i = 1, \ldots, N$ een welgegronde maat bestaat
   $f_i : C_i, >_\tau \rightarrow W_i, >_i$
3. voor elke $(G, k) \in C_0$ en elke tak D in τ die (G, k) bevat:
   - ofwel D eindig is
   - ofwel D een afstammeling (G', j) van (G, k) bevat zodat $(G', j) \in C_i$ voor een $i > 0$.

Stelling 3.7 Een SLD$^-$-boom is eindig asa hij gedeeld gegrond is.

Bemerk dat op C_0 geen maat gedefiniëerd is. Voorwaarde 3 zorgt ervoor dat zulks veilig kan.

3.2 Ontvouwen met eindige pregronden

Laten we nu een eerste aanpak voor ontvouwen voorstellen. In de rest van dit hoofdstuk verwijst P naar een positief programma, $\leftarrow A$ naar een positief doel (met één enkel atoom), $\mathcal{L}$ naar de taal waarin P en A gesteld zijn, R naar een computatieregel voor $P \cup \{\leftarrow A\}$ en τ_0 naar de volledige SLD-boom voor $P \cup \{\leftarrow A\}$ via R.

Definitie 3.8 Een koppel $((C_0, \ldots, C_N), (f_1, \ldots, f_N))$ is een *eindige pregrond* voor τ_0 indien

1. elke C_i, $i \leq N$, bestaat uit koppels (G, k), zodanig dat G een doel in $\mathcal{L}$ is, $k \in \mathbb{N}$ en $G_{\tau_0} \subseteq \bigcup_{i \leq N} C_i$
2. voor elke $i = 1, \ldots, N$, f_i een functie is, $f_i : C_i \rightarrow W_i, >_i$, die C_i afbeeldt op een welgegronde verzameling $W_i, >_i$
3. voor elke $(G, k) \in C_0$ en elke tak D in τ die (G, k) bevat:
   - ofwel D eindig is
   - ofwel D een afstammeling (G', j) van (G, k) bevat zodat $(G', j) \in C_i$ voor een $i > 0$.

Stel nu dat een dergelijke eindige pregrond gegeven is. Dan kunnen we een eindige deelboom τ van τ_0 konstrueren door ervoor te zorgen dat de gegeven functies f_i welgegrond zijn op τ. Het is precies dat wat het volgende algoritme doet als $((C_0, \ldots, C_N), (f_1, \ldots, f_N))$ de gegeven pregrond is.

Algoritme 3.9

```
Initialisatie
  $\tau := \{\{(\leftarrow A, 1)\}\}$ {* 1 tak met 1 knoop *}
  $Gedaan := \emptyset$
Zolang er een tak $D$ in $\tau$ bestaat zodanig dat $D \notin Gedaan$ doe
  Zij $(G, i)$ het blad van $D$
  Laat $Afstam(G, i)$ alle directe $>_{\tau_0}$-afstammelingen van $(G, i)$ bevatten
  Laat $Afname(G, i)$ alle $(G', j) \in Afstam(G, i)$ bevatten zodaning dat
    voor alle $k > 0$ zodanig dat $(G', j) \in C_k$,
    voor alle $(G'', j') \in D \cap C_k$:
      $f_k(G'', j') >_k f_k(G', j)$
  Indien $Afname(G, i) = \emptyset$
    Dan voeg $D$ toe aan $Gedaan$
  Anders {* $\tau$ wordt verder uitgebreid *}
    Vervang $\tau$ door $\tau \setminus D \cup \{D \cup \{(G', j)\} | (G', j) \in Afname(G, i)\}$
Einde
```

Tenslotte verkrijgt men een eindige SLD-boom voor $P \cup \{\leftarrow A\}$ via R door τ te vervolledigen tot τ^+.

De volgende stelling toont aan dat algoritme 3.9 altijd eindigt. Ze legt ook het verband met sectie 3.1.

Stelling 3.10 Algoritme 3.9 eindigt. De verkregen eindige SLD$^-$-boom τ is gedeeld gegrond met betrekking tot de verzamelingen $C_0 \cap \tau, \ldots, C_N \cap \tau$ en de welgegronde maten $f_1, \ldots, f_N$ beperkt tot deze verzamelingen.

3.3 Ontvouwen met hiërarchische pregronden

Voor een aantal gevallen geeft algoritme 3.9 goede resultaten. Maar in het algemeen schiet het tekort inzake de *behandeling van recursie*. Een meer verfijnde aanpak vereist het opsplitsen van (een oneindige) τ_0 in een *oneindig aantal deelverzamelingen*, en wel op een manier die de dieper liggende structuur van de aanwezige recursie weerspiegelt.

We voeren enige verdere terminologie en notatie in.

- Voor een doel (G, i) stelt $R(G, i)$ het (door R) geselecteerde atoom voor.
- Zij (G, i) en (G', j) doelen in een SLD$^-$-boom τ, met $(G', j) >_\tau (G, i)$. Dan noemen we (G', j) een *eigenlijke voorouder* van (G, i) indien $R(G, i)$ afstamt van $R(G', j)$. We noteren: $(G', j) >_{pr} (G, i)$.
- Een serie doelen $(G_{i_1}, i_1) >_{pr} (G_{i_2}, i_2) >_{pr} \ldots$ in τ is een *eigenlijke vketting* indien voor alle m, $R(G_{i_m}, i_m)$ in (G_{i_m}, i_m) een directe voorouder is van $R(G_{i_{m+1}}, i_{m+1})$ in $(G_{i_{m+1}}, i_{m+1})$.

Wat we nu willen is doelen enkel vergelijken met (relevante) $>_{pr}$-voorouders. Dit ligt ten grondslag aan de omschrijving van de C_i-verzamelingen in de volgende definitie. De R_j-partitie bouwt de extra "relevantie" notie in. (In het bijzonder zullen we enkel doelen waarbij het geselecteerde atoom hetzelfde predikaatsymbool bevat, met elkaar vergelijken.)

Definitie 3.11 Een koppel $((C_0, C_1, C_2, \ldots), (f_1, f_2, \ldots))$ is een *hiërarchische pregrond* voor τ_0 indien:

1. er een eindige partitie $R_0, \ldots, R_N$ van $R_{\tau_0} = \{R(G, i) | (G, i) \in \tau_0\}$ bestaat zodanig dat:
   - $\forall (G, i) \in \tau_0 \setminus C_0$:
     $C_i = \{(G, i)\} \cup$
     $\{(G', j) \in \tau_0 | \exists k : 1 \leq k \leq N$, zodanig dat $R(G, i), R(G', j) \in R_k$ en $(G, i) >_{pr} (G', j)$ of $(G', j) >_{pr} (G, i)\}$
   - $C_0 = \{(G, i) \in \tau_0 | R(G, i) \in R_0\} \cup \{(\Box, i) \in \tau_0\}$
2. $f_1, f_2, \ldots$ functies zijn die respectievelijk $C_1, C_2, \ldots$ afbeelden op één van een eindig aantal welgegronde verzamelingen $W_1, >_1, \ldots, W_N, >_N$ zodanig dat f_i C_i afbeeldt op W_k indien de geselecteerde atomen van de doelen in C_i behoren tot R_k. Bovendien moet voor alle $i, j > 0 : f_i|_{C_i \cap C_j} = f_j|_{C_i \cap C_j}$.
3. C_0 bevat geen oneindige eigenlijke vketting.

Het verband tussen wat voorafging en het gebruik van een hiërarchische pregrond als basis voor eindig ontvouwen, is vervat in de volgende stelling.

Stelling 3.12 Zij $((C_0, C_1, C_2, \ldots), (f_1, f_2, \ldots))$ een hiërarchische pregrond voor τ_0 en τ een SLD$^-$-deelboom van τ_0. Indien elke f_i een welgegronde maat op $C_i \cap \tau, >_\tau$ is, dan zijn er tussen $C_1, C_2, \ldots$ een eindig aantal verzamelingen $C_{i_1}, \ldots, C_{i_M}$, zodanig dat τ gedeeld gegrond is met betrekking tot het koppel $((C_0 \cap \tau, C_{i_1} \cap \tau, \ldots, C_{i_M} \cap \tau), (f_{i_1}, \ldots, f_{i_M}))$.

$((C_0, C_1, C_2, \ldots), (f_1, f_2, \ldots))$, een koppel potentiëel oneindige reeksen, is voor praktische doeleinden natuurlijk nauwelijks geschikt. We kunnen echter een hiërarchische pregrond ondubbelzinnig (en bruikbaar) beschrijven met behulp van een eindige partitie $R_0, \ldots, R_N$ van R_{τ_0} en functies $F_1, \ldots, F_N$ zodanig dat

$$F_k : \{(G,i) \in \tau_0 | R(G,i) \in R_k\} \rightarrow W_k, >_k$$

waarbij $W_k, >_k, 1 \leq k \leq N$ welgegronde verzamelingen zijn.

Als we nu nog een laatste vereist begrip invoeren, kunnen we een verbeterd algoritme formuleren. We zeggen dat een doel (G', j) een doel (G, i) ***dekt*** als $(G', j) >_{pr} (G, i)$ en $R(G', j)$, $R(G, i)$ tot dezelfde R_k-klasse behoren. De meest nabije voorouder die een doel dekt, heet zijn ***directe dekkende voorouder.***

Algoritme 3.13 vooronderstelt een gegeven computatieregel R en een gegeven koppel $((R_0, R_1, \ldots, R_N), (F_1, \ldots, F_N))$.

Algoritme 3.13

```
Initialisatie
  $\tau := \{\{(\leftarrow A, 1)\}\}$
  $Pr := \emptyset$ {* in $Pr$ construeren we de $>_{pr}$-relatie *}
  $Gedaan := \emptyset$
Zolang er een tak $D$ in $\tau$ bestaat zodanig dat $D \notin Gedaan$ doe
  Zij $(G, i)$ het blad van $D$
  Laat $Afstam(G, i)$ alle directe $>_{\tau_0}$-afstammelingen van $(G, i)$ bevatten
  Laat $Afname(G, i)$ alle $(G', j) \in Afstam(G, i)$ bevatten zodanig dat
    Indien $(G'', k)$ de directe dekkende voorouder van $(G', j)$ is
          en $R(G', j), R(G'', k) \in R_n (1 \leq n \leq N)$
      Dan $F_n(G'', k) > F_n(G', j)$
  Indien $Afname(G, i) = \emptyset$
    Dan voeg $D$ toe aan $Gedaan$
  Anders {* $\tau$ wordt verder uitgebreid *}
    Vervang $\tau$ door $\tau \setminus D \cup \{D \cup \{(G', j)\} | (G', j) \in Afname(G, i)\}$
    Breid overeenkomstig de $Pr$-relatie uit
Einde
```

Natuurlijk moet ook nu weer de verkregen SLD$^-$-boom τ vervolledigd worden tot τ^+, zijn kleinste omvattende SLD-boom.

Tenslotte geldt volgende stelling:

Stelling 3.14 Algoritme 3.13 eindigt. Er bestaan een eindig aantal verzamelingen $C_0, \ldots, C_M$ en functies $f_1, \ldots, f_M$, zodanig dat de resulterende eindige SLD$^-$-boom τ gedeeld gegrond is met betrekking tot $C_0 \cap \tau, \ldots, C_M \cap \tau$ en de welgegronde maten $f_1, \ldots, f_M$ beperkt tot deze verzamelingen.

3.4 Naar automatisering

Het aanwenden van algoritme 3.13 vereist concrete keuzen voor R, $R_0, R_1, \ldots, R_N$ en $F_1, \ldots, F_N$. Dit onderwerp komt verderop aan bod, in sectie 4.1 en hoofdstuk 5, bij het voorstellen van volledig automatische algoritmen voor ontvouwen. In dat kader blijkt een variant van algoritme 3.13 echter beter bruikbaar als uitvalsbasis. Deze variant meet doelen, niet wanneer ze toegelaten worden in de SLD-boom in opbouw, maar nadien, bij eventueel verder ontvouwen.

Algoritme 3.15

```
Initialisatie
  $\tau := \{\{(\leftarrow A, 1)\}\}$
  $Pr := \emptyset$
  $Gedaan := \emptyset$
Zolang er een tak $D$ in $\tau$ bestaat zodanig dat $D \notin Gedaan$ doe
  Zij $(G, i)$ het blad van $D$
  Laat $Afstam(G, i)$ alle directe $>_{\tau_0}$-afstammelingen van $(G, i)$ bevatten
  Indien $Afstam(G, i) = \emptyset$ {* $(G, i)$ is een succes- of een faalknoop *}
    Dan voeg $D$ toe aan $Gedaan$
  Anders indien er een directe dekkende voorouder $(G', j)$ van $(G, i)$ is
                met $R(G', j), R(G, i) \in R_n$
                zodanig dat niet$(F_n(G', j) >_n F_n(G, i))$
    Dan voeg $D$ toe aan $Gedaan$
  Anders
    Vervang $\tau$ door $\tau \setminus D \cup \{D \cup \{(G'', k)\} | (G'', k) \in Afstam(G, i)\}$
    Breid overeenkomstig de $Pr$-relatie uit
Einde
```

3.5 Besluit

In dit hoofdstuk hebben we een aantal concepten ingevoerd die eindigheid van SLD-bomen op een interessante wijze karakteriseren. Ze zijn namelijk geschikt als basis voor vrij algemene en elegante algoritmen. Deze bouwen eindige SLD-bomen zonder gebruik te maken van adhocmaatregelen.

Drie verschillende semi-automatische algoritmen werden geformuleerd. Het tweede biedt een meer verfijnde behandeling van recursie en werd daarom uitverkoren om in een wat aangepaste vorm (het derde algoritme) als vertrekpunt te fungeren voor volledige automatisering.

4 Correcte en volledige partiële deductie

In dit hoofdstuk bestuderen we partiële deductie voor positieve programma's en vragen. In een eerste stap leiden we een concreet, volledig automatisch algoritme af voor het eindig ontvouwen van dergelijke programma's en vragen. In sectie 4.2 benutten we dit vervolgens als bouwsteen in een aantoonbaar correcte en volledige methode voor partiële deductie. Ook hier weer formuleren we een geheel automatisch algoritme dat zeker altijd eindigt. Tenslotte hebben we verschillende globale methodes voor partiële deductie vergeleken aan de hand van een vijftal testprogramma's uit [104]. In sectie 4.3 formuleren we kort enkele tentatieve conclusies naar aanleiding van deze beperkte proefondervindelijke studie.

4.1 Automatisch eindig ontvouwen

We dienen dus in de eerste plaats algoritme 3.15 compleet te automatiseren. Keuzen voor $R_0, R_1, \ldots, R_N$ zijn snel gemaakt: *één klasse per recursief predikaat*, R_0 voor de niet recursieve. De vereiste *maten* laten we de complexiteit van argumenten opmeten.

Definitie 4.1 Zij *Term* de verzameling van termen in $\mathcal{L}$, de beschouwde taal. We definiëren de *functornorm* als de functie $|.| : Term \rightarrow I\!N$:

Indien $t = f(t_1, \ldots, t_n), n > 0$
dan $|t| = 1 + |t_1| + \cdots + |t_n|$
anders $|t| = 0$

De functornorm telt dus het aantal functoren in een gegeven term.

Definitie 4.2 Zij p een predikaatsymbool met ariteit n en $S = \{a_1, \ldots, a_m\}, 1 \leq a_k \leq n, 1 \leq k \leq m$ een verzameling argumentposities voor p. We definiëren de *functormaat* met betrekking tot p en S als de functie

$|.|_{p,S} : \{A|A \text{ is een atoom met predikaatsymbool } p\} \rightarrow I\!N$:
$|p(t_1, \ldots, t_n)|_{p,S} = |t_{a_1}| + \cdots + |t_{a_m}|$

Functormaten reiken geschikte keuzemogelijkheden aan voor $F_1, \ldots, F_N$. Het is echter niet zonder meer duidelijk welke specifieke S voor een bepaald predikaat in een gegeven probleem optimaal ontvouwen toelaat. Het onderstaande algoritme vertrekt daarom met maximale verzamelingen voor elk recursief predikaat. Deze initiële keuzen worden dynamisch verfijnd: posities met groeiende argumenten worden zo nodig verwijderd. Ook het selecteren van atomen ter ontvouwing (met andere woorden het vastleggen van R) geschiedt tijdens de uitvoering van het algoritme.

Al deze ingrediënten tezamen maken het mogelijk algoritme 4.3 te formuleren en de eindigheid ervan te bewijzen.

Algoritme 4.3

```
Invoer
  een positief programma $P$
  een positief doel $\leftarrow A$
Uitvoer
  een eindige SLD-boom $\tau$ voor $P \cup \{\leftarrow A\}$
Initialisatie
  $\tau := \{(\leftarrow A, 1)\}$
  $Pr := \emptyset$
  $Gedaan := \emptyset$
  $Gefaald := \emptyset$
  Voor elk recursief predikaat $p/n$ in $P$: $S_p := \{1, \ldots, n\}$
Zolang er een tak $D$ in $\tau$ bestaat zodanig dat $D \notin Gedaan$ doe
  Zij $(G, i)$ het blad van $D$
  Indien $(G, i) = (\Box, i)$
  Dan {* $(G, i)$ is a succesknoop *}
    Voeg $D$ toe aan $Gedaan$
  Anders
    {* Tracht $R(G, i)$ te bepalen*}
    Selecteer het meest linkse atoom $p(t_1, \ldots, t_n)$ in $G$ zodanig dat
    één van de volgende (elkaar uitsluitende) voorwaarden vervuld is:
      • $(G, i)$ heeft geen directe dekkende voorouder
      • $(G', j)$ is de directe dekkende voorouder van $(G, i)$
        en $|R(G', j)|_{p,S_p} > |p(t_1, \ldots, t_n)|_{p,S_p}$
      • $(G', j)$ is de directe dekkende voorouder van $(G, i)$
        en $|R(G', j)|_{p,S_p} \leq |p(t_1, \ldots, t_n)|_{p,S_p}$
        en $|R(G', j)|_{p,{S_p}^{nieuw}} > |p(t_1, \ldots, t_n)|_{p,{S_p}^{nieuw}}$ waarbij
          ${S_p}^{nieuw} = S_p \setminus \{a_k \in S_p \,||p(t_1, \ldots, t_n)|_{p,\{a_k\}} > |R(G', j)|_{p,\{a_k\}}\} \neq \emptyset$
        en $\tau$ blijft gedeeld gegrond met betrekking tot
          $((R_0, R_1, \ldots, R_N), (|\cdot|_{p_1,S_{p_1}}, \ldots, |\cdot|_{p,{S_p}^{nieuw}}, \ldots, |\cdot|_{p_N,S_{p_N}}))$
    Indien een dergelijk atoom $p(t_1, \ldots, t_n)$ niet kan gevonden worden
    Dan
      Voeg $D$ toe aan $Gedaan$
    Anders
      $R(G, i) := p(t_1, \ldots, t_n)$
      Indien $R(G, i)$ geselecteerd werd via de derde voorwaarde hierboven
        Dan $S_p := {S_p}^{nieuw}$
      Laat $Afstam(G, i)$ alle directe afstammelingen van $(G, i)$ bevatten
      Indien $Afstam(G, i) = \emptyset$
```

```
        Dan {* $(G, i)$ is een faalknoop *}
          Voeg $D$ toe aan $Gedaan$ en $Gefaald$
        Anders
          {* Breid de tak uit *}
          Breid $D$ in $\tau$ uit met de elementen van $Afstam(G, i)$
          Pas $Pr$ aan
Einde
```

Stelling 4.4 Algoritme 4.3 eindigt. Indien de invoer bestaat uit een positief programma P en een positief doel $\leftarrow A$, dan is de uitvoer τ een eindige (eventueel onvolledige) SLD-boom voor $P \cup \{\leftarrow A\}$.

4.2 Een algoritme voor partiële deductie

In deze sectie nemen we aan dat de lezer vertrouwd is met [114], een belangrijk artikel waarin theoretische grondslagen voor partiële deductie in logisch programmeren gelegd werden. De belangrijkste stelling is de volgende:

Stelling 4.5 Zij P een positief logisch programma, G een positief doel, $\mathbf{A}$ een eindige, onafhankelijke verzameling atomen, en P' a partiële deductie van P met betrekking tot $\mathbf{A}$ zodanig dat $P' \cup \{G\}$ $\mathbf{A}$-gedekt is. Dan gelden:

- $P' \cup \{G\}$ heeft een SLD-refutatie met berekend antwoord θ asa zulks het geval is voor $P \cup \{G\}$.
- $P' \cup \{G\}$ heeft een eindig falende SLD-boom asa zulks het geval is voor $P \cup \{G\}$.

Met andere woorden, partiële deductie dient erover te waken dat geen twee atomen in de resulterende verzameling $\mathbf{A}$ een gemeenschappelijke instantie bezitten (de onafhankelijkheidsvoorwaarde) en dat alle relevante atomen in $P' \cup \{G\}$ instanties zijn van atomen in $\mathbf{A}$ (de dekkingsvoorwaarde). In dat geval wordt namelijk G op dezelfde wijze (maar hopelijk wel efficiënter) beantwoord in het oorspronkelijke en in het gespecialiseerde programma: partiële deductie is correct en volledig.

We formuleren nu een algoritme dat partiële deductie uitvoert. Het maakt gebruik van twee begrippen die een korte uitleg behoeven. Een *msv* van een aantal atomen is een *meest specifieke veralgemening* (*most specific generalisation*): het minst algemene atoom waarvan alle gegeven atomen instanties zijn. Elk stel atomen met hetzelfde predikaatsymbool bezit een *msv*, enig op het hernoemen van veranderlijken na. Een *pp'-hernoeming* van een programma is een equivalent programma, waar het predikaatsymbool p in lichaamsdoelen werd vervangen door een vers symbool p' en regels voor p' werden gekopiëerd van die voor p.

Algoritme 4.6

```
Invoer
  een positief programma $P$
  een positief doel $\leftarrow A = \leftarrow p(t_1, \ldots, t_n)$
  een vers predikaatsymbool $p'$ met dezelfde ariteit als $p$
Uitvoer
  een verzameling atomen $\mathbf{A}$
  een partiële deductie $P_r{}'$ van $P_r$, $P$'s $pp'$-hernoeming, met betrekking tot $\mathbf{A}$
Initialisatie
  $P_r :=$ de $pp'$-hernoeming van $P$
  $\mathbf{A} := \{A\}$ en label $A$ niet gemerkt
Zolang er een niet gemerkt atoom $B$ is in $\mathbf{A}$ doe
  Pas algoritme 4.3 toe met $P_r$ en $\leftarrow B$ als invoer
  Zij $\tau_B$ de resulterende SLD-boom
  Stel uit $\tau_B$ $P_{rB}$ samen, een partiële deductie voor $B$ in $P_r$
  Label $B$ gemerkt
  Zij $\mathbf{A}_B$ de verzameling atomen in de lichamen van regels in $P_{rB}$
  Voor elk predikaatsymbool $q$ dat voorkomt in een atoom in $\mathbf{A}_B$
    Zij $msv_q$ een msv van alle atomen in $\mathbf{A}$ en $\mathbf{A}_B$ met predikaatsymbool $q$
    Indien er in $\mathbf{A}$ een atoom bestaat met predikaatsymbool $q$
                  en minder algemeen dan $msv_q$,
      Dan verwijder dit atoom uit $\mathbf{A}$
    Indien er nu in $\mathbf{A}$ geen atoom is met predikaatsymbool $q$
      Dan voeg $msv_q$ toe aan $\mathbf{A}$ and label het niet gemerkt
  Einde
Einde
Stel tenslotte de partiële deductie $P_r{}'$ van $P_r$ met betrekking tot $\mathbf{A}$ samen
uit de partiële deducties voor de elementen van $\mathbf{A}$ in $P_r$.
```

Algoritme 4.6 bezit volgende formele eigenschappen:

Stelling 4.7 Algoritme 4.6 eindigt.

Stelling 4.8 Zij P een positief programma, $\leftarrow p(t_1, \ldots, t_n)$ een doel en p' een predikaatsymbool, invoer voor algoritme 4.6. Zij $\mathbf{A}$ de verzameling atomen en $P_r{}'$ het programma door algoritme 4.6 geproduceerd als uitvoer. Dan gelden:

- $\mathbf{A}$ is onafhankelijk.
- Voor elk doel $G = \leftarrow A_1, \ldots, A_m$ dat bestaat uit instanties van atomen in $\mathbf{A}$, is $P_r{}' \cup \{G\}$ $\mathbf{A}$-gedekt.

De voorwaarden voor het toepassen van stelling 4.5 zijn dus vervuld.

Gevolg 4.9 Zij P een positief programma, $\leftarrow p(t_1, \ldots, t_n)$ een doel en p' een predikaatsymbool, invoer voor algoritme 4.6. Zij $\mathbf{A}$ de (eindige) verzameling atomen en $P_r{}'$ het programma door algoritme 4.6 geproduceerd als uitvoer. Zij $G = \leftarrow A_1, \ldots, A_m$ een doel dat bestaat uit instanties van atomen in $\mathbf{A}$. Dan gelden:

- $P' \cup \{G\}$ heeft een SLD-refutatie met berekend antwoord θ asa zulks het geval is voor $P \cup \{G\}$.
- $P' \cup \{G\}$ heeft een eindig falende SLD-boom asa zulks het geval is voor $P \cup \{G\}$.

Stelling 4.10 Zij P een positief programma en $\leftarrow A$ een atomair doel, gebruikt als invoer voor algoritme 4.6. Zij $\mathbf{A}$ de verzameling atomen geproduceerd door algoritme 4.6. Dan $A \in \mathbf{A}$.

In het bijzonder geldt gevolg 4.9 dus voor instanties van A.

4.3 Experimenten

Horváth ([83]) heeft een systeem geïmplementeerd dat toelaat diverse methoden voor partiële deductie met elkaar te vergelijken. De twee centrale parameters daarbij zijn het beheersen van ontvouwen en de strategie gevolgd bij de opbouw van de verzameling $\mathbf{A}$. We hebben een aantal experimenten uitgevoerd op vijf testprogramma's genomen uit [104]. Telkens werden partiële deducties berekend volgens verschillende recepten. Vervolgens werd de efficiëntie van de verkregen programma's vergeleken. Met het nodige voorbehoud kunnen aan de resultaten enkele opmerkingen vastgeknoopt worden.

Ten eerste lijkt ontvouwen met maten en hiërarchische pregronden een solide basis te verschaffen voor partiële deductie. Men verkrijgt eindige SLD-bomen die de inherente eigenschappen van het gegeven probleem goed weerspiegelen. Een alternatieve techniek die ontvouwt indien niet eerder een variant van het geselecteerde atoom ontvouwd werd, lijkt dikwijls te diepe bomen te bouwen. Ook het concept van een dekkende voorouder blijkt van primordiaal belang.

Algoritme 4.3 zelf blijkt echter niet in staat altijd behoorlijk te functioneren. Een combinatie met "niet variant" ontvouwen blijkt een aanzienlijke verbetering. Complexe meta-programma's worden eveneens niet behoorlijk behandeld. Deze en andere beperkingen van algoritme 4.3, en vooral technieken om ze te overwinnen, genieten onze anndacht in het volgende hoofdstuk.

Enkel deterministische oproepen ontvouwen is, althans in zijn meest elementaire vorm, te beperkt. Een "kijk dieper en zie of er slechts één tak slaagt" verfijning lijkt noodzakelijk. In het algemeen is het niet duidelijk wanneer bepaalde stukken van een eindige SLD-boom, verkregen door veilig ontvouwen, best alsnog weggeknipt worden. Dit lijkt een boeiend onderwerp voor verder onderzoek.

Algoritme 4.6 gebruikt een zeer eenvoudige methode om de grootte van A te beheersen: één enkel atoom per predikaatsymbool wordt toegelaten. Op die manier is, bij veilig ontvouwen, eindigheid verzekerd. Bovendien zijn ook de praktische resultaten interessant wanneer er krachtig (redelijk diep) ontvouwd wordt. Toch bleek duidelijk dat een soepeler beheer van A gewenst is. Methoden die meer dan één atoom per predikaatsymbool toelaten en eventueel dynamisch nieuwe predikaten invoeren, leverden dikwijls goede resultaten. Het is echter nog niet duidelijk hoe A in het algemeen voldoende precies en steeds eindig kan gehouden worden. Misschien bieden ook hier maatfuncties een uitkomst.

4.4 Besluit

In dit hoofdstuk hebben we, binnen het raamwerk opgebouwd in hoofdstuk 3, een eerste volledig automatisch algoritme voor eindig ontvouwen van positieve programma's en vragen voorgesteld. Dit maakte het mogelijk een altijd eindigende methode voor partiële deductie te formuleren. Bovendien zijn correctheid en volledigheid in de zin van [114] vervuld. Tenslotte hebben we enkele opmerkingen geformuleerd naar aanleiding van een beperkte proefondervindelijke studie.

Ontvouwen met maten zoals voorgesteld in hoofdstuk 3 blijkt een veelbelovende aanpak, ook in de praktijk. Nochtans vragen verscheidene onderwerpen verdere studie, betreffende het beheersen van automatisch eindig ontvouwen zowel als het geleiden van het partiële deductieproces in zijn geheel. Deze laatste blijven hier verder onbehandeld, maar verfijningen betreffende ontvouwen komen aan bod in het volgende hoofdstuk.

5 Meer over eindig ontvouwen

Tenslotte bekijken we automatisch eindig ontvouwen van naderbij. Vooral het construeren en verfijnen van maatfuncties geniet onze aandacht. Algoritme 4.3 gebruikt functornormen en de daarop gebaseerde maten zoals geïntroduceerd in respectievelijk definities 4.1 en 4.2. De eerste zullen we ook hierna aanhouden als basis voor onze beschouwingen, maar het louter handelen met deelverzamelingen van argumentposities zoals in definitie 4.2 blijkt in het algemeen te beperkt.

In dit hoofdstuk bestuderen we enkele meer gevorderde ontvouwingstechnieken. We voorzien de mogelijkheid lexicografische prioriteiten tussen verschillende deelverzamelingen van argumentposities aan te brengen in sectie 5.1. In diezelfde sectie stippen we een optimalisering aan die ook kan toegepast worden op algoritme 4.3. Daarna, in sectie 5.2, bespreken we mogelijkheden en moeilijkheden bij ontvouwen waarbij niet enkel de maat van het in een doel geselecteerde atoom beschouwd wordt, maar ook (sommige) andere atomen in hetzelfde doel bijdragen tot de beslissing over al dan niet verdergaan. Dit alles leidt tot een beter begrip van meer fundamentele aspecten in de zoektocht naar optimale maten. Enkele algemene inzichten in dit verband komen aan bod in sectie 5.3. Vervolgens stelt sectie 5.4 een praktisch belangrijke uitbreiding van de basismethoden voor waarbij ontvouwen met maten en met "niet variant" tests worden in elkaar gepast tot één globale aanpak. Sectie 5.5, tenslotte, vermeldt de mogelijkheid ontvouwen niet enkel te gronden op het gedrag van hele argumenten, maar ook afzondelijke delen van dergelijke argumenten hiervoor in aanmerking te nemen. Aldus zetten we een eerste stap naar automatisch en krachtig ontvouwen van meta-vertolkers.

5.1 Lexicografische prioriteiten

Krachtiger maten

Ter herinnering: $\mathcal{P}(V)$ is de verzameling van deelverzamelingen (machtsverzameling) van V en V^n het n-voudig Cartesisch produkt van V. We definiëren:

Definitie 5.1 Zij V een verzameling en $S_1, \ldots, S_k$ k wederzijds disjuncte, niet ledige deelverzamelingen van V, samen een partitie van V. Dan noemen we het k-tal $(S_1, \ldots, S_k) \in \mathcal{D}(V)^k$ een *geordende k-partitie* van V.

Wij zullen het verder vooral hebben over geordende partities van de verzameling van argumentposities van een predikaat p, en spreken dan kortweg over een met *p verbonden* geordende partitie.

Definitie 5.2 Zij p een predikaat met ariteit n en een verbonden geordende k-partitie $O = (\{i_{11}, \ldots, i_{1j}\}, \ldots, \{i_{k1}, \ldots, i_{kl}\})$. We definiëren $|.|_{p,O} : \{A|A$ is een atoom met predikaatsymbool $p\} \rightarrow I\!N^k$ als volgt:

$$|p(t_1, \ldots, t_n)|_{p,O} = (|t_{i_{11}}| + \cdots + |t_{i_{1j}}|, \ldots, |t_{i_{k1}}| + \cdots + |t_{i_{kl}}|)$$

waarbij $|.|$ de functornorm is zoals in definitie 4.1.

Als we $I\!N^k$ ordenen via $\succ_k$, de voor de hand liggende lexicografische uitbreiding voor k-tallen van $>$ op $I\!N$, kunnen we dergelijke functies als maat aanwenden.

Stelling 5.3 Zij O een geordende k-partitie verbonden met een predikaat p. Zij τ een SLD-boom en S_τ een deelverzameling van G_τ zodanig dat alle doelen in S_τ een geselecteerd atoom met predikaat p bezitten. Definiëer F_p als volgt:

$$F_p : S_\tau, >_\tau \rightarrow I\!N^k, \succ_k : (G, i) \in S_\tau \mapsto |R(G, i)|_{p,O}$$

Dan is F_p een welgegronde maat op $S_\tau, >_\tau$ asa F_p monotoon is.

Automatisering

Het is nu mogelijk algoritme 4.3 te herformuleren, deze keer gebruikmakend van $|.|_{p,\mathcal{O}}$- in plaats van $|.|_{p,S}$-functies. Daartoe dient een nieuw recept bedacht voor het automatisch zoeken naar optimale maten of, met andere woorden, optimale geordende partities. We zullen hier noch de formele opbouw van de vereiste concepten, noch het resulterende algoritme reproduceren. Wel trachten we de *maatverfijningsstrategie* in woorden te beschrijven.

Initiëel nemen we voor elk recursief predikaat p in het gegeven programma een 1-partitie, alle argumentposities van p in één enkele verzameling bevattend. Neem nu aan dat G een doel is waarin we een atoom ter ontvouwing wensen te selecteren. Veronderstel bovendien dat G's maat niet daalt ten opzichte van de directe dekkende voorouder van G als we een atoom $A = p(t_1, \ldots, t_n)$ verkiezen. Dan kunnen we A desalniettemin selecteren indien de mogelijkheid zich voordoet $\mathcal{O}_p$ op gepaste wijze te verfijnen. (Voorheen werden eenvoudig èèn of meer argumentposities verwijderd indien de aldus verkregen kleinere S_p leidde tot dalende maten.) Welnu, $\mathcal{O}_p$ kan nuttig verfijnd worden als een geschikte component zodanig kan *opgesplitst worden in twee nieuwe componenten* dat de eerste van deze twee (de lexicaal prioritaire) nadien overeenstemt met een *dalend* gewicht (de in definitie 5.2 vermelde som van zijn $|t_{ij}|$'s moet kleiner zijn in G dan in diens directe dekkende voorouder). Geen "stijgende" component mag de gesplitste voorafgaan; zulks zou vanzelfsprekend het gewenste effect tenietdoen. Tenslotte is het voordelig een component te splitsen met zo laag mogelijke prioriteit. Een dergelijke keuze houdt de globale gewichtsafname van doelen onder de nieuwe maat en de daarmee gepaard gaande vermindering van het totale ontvouwingspotentiëel namelijk zo gering mogelijk.

Bemerk dat voor elk predikaat p, net zoals S_p in algoritme 4.3, $\mathcal{O}_p$ slechts een eindig aantal keren verfijnd kan worden. Het resulterende algoritme eindigt daarom weer voor alle mogelijke (positieve) programma's en doelen en levert een eindige SLD-boom af. Deze laatste is in een aantal gevallen op gepaste wijze groter dan de door algoritme 4.3 onder gelijke omstandigheden geproduceerde en (wellicht) nooit kleiner.

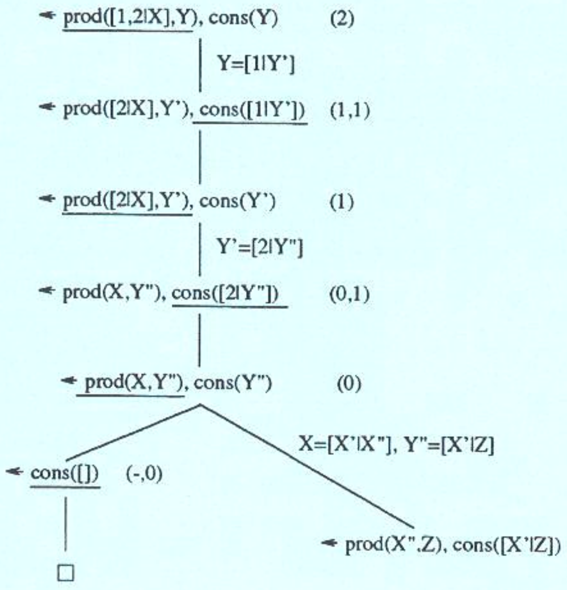
Een meer efficiënte variant

Tenslotte stippen we aan dat het niet strikt noodzakelijk is de in opbouw zijnde SLD-boom globaal gedeeld gegrond te houden. In het bijzonder kunnen maatfuncties verfijnd worden zonder te testen of de tot dan reeds gebouwde deelboom gedeeld gegrond blijft.

In een aantal (eerder zeldzame) gevallen leidt zulks tot dieper ontvouwen. Belangrijker is dat het potentiëel kostbare hertesten van hele bomen uitgeschakeld wordt. Ontvouwen vergt nu nog enkel gewichtsvergelijkingen tussen een doel en zijn directe dekkende voorouder. Dit is mogelijk zonder enig zoeken in de SLD-boom. De rekenkost verbonden aan (bijvoorbeeld) een aldus gewijzigde versie van algoritme 4.3 gedraagt zich derhalve *lineair* in functie van de grootte van de geconstrueerde SLD-boom.

5.2 De context beschouwen

Een verdere stap wordt nu denkbaar.


← prod([1,2|X],Y), cons(Y) (2)
Y=[1|Y']
← prod([2|X],Y'), cons([1|Y']) (1,1)
← prod([2|X],Y'), cons(Y') (1)
Y'=[2|Y'']
← prod(X,Y''), cons([2|Y'']) (0,1)
← prod(X,Y''), cons(Y'') (0)
X=[X'|X''], Y''=[X'|Z]
← cons([]) (-,0)
← prod(X'',Z), cons([X'|Z])

Figuur 1: Niet geselecteerde atomen verrekenen bij het ontvouwen.

Voorbeeld 5.4 Beschouw het volgende schematische programma:

$produce([], []) \leftarrow$

$produce([X|Xs], [X|Ys]) \leftarrow produce(Xs, Ys)$

$consume([]) \leftarrow$

$consume([X|Xs]) \leftarrow consume(Xs)$

en het doel:

$\leftarrow produce([1, 2|X], Y), consume(Y)$

We passen nu algoritme 3.15 toe. Daarbij leggen we een wisselwerkende (*coroutining*) computatieregel op en kiezen we het koppel $((R_0, R_1, R_2), (F_1, F_2))$ als volgt:

- $R_0 = \emptyset$
- $R_1 = \{R(G, i) \text{ met } \mathit{produce}\}$
- $R_2 = \{R(G, i) \text{ met } \mathit{consume}\}$
- $F_1 = |\cdot|_{produce,(\{1,2\})}$
- $F_2 = (|\cdot|_{produce,\{1,2\}}, |\cdot|_{consume,\{1\}})$

De resulterende SLD-boom is afgebeeld in figuur 1. Geselecteerde atomen zijn onderlijnd en knopen zijn gemerkt met hun gewicht volgens F_1 of F_2.
Bemerk dat de tweede *consume*-ontvouwing niet toegelaten zou zijn onder een maatfunctie die enkel rekening houdt met het argument van *consume*.

Cruciaal in voorbeeld 5.4 is de mogelijkheid informatie in aanmerking te nemen die buiten het geselecteerde atoom in de ruimere context van het hele doel te vinden valt.

Wisselwerking behandelen

We veralgemenen definitie 5.1:

Definitie 5.5 Zij P be een programma en p een (recursief) predikaatsymbool in P. Een *context beschouwende geordende k-partitie (cbo-k-partitie) vebonden met p in P* is een k-tal $O = (\{i_{11}, \ldots, i_{1j}\}, \ldots, \{i_{k1}, \ldots, i_{kl}\})$ dat voldoet aan de volgende voorwaarden:

1. Er zijn *twee soorten componenten* in O. Sommige, die we *p-componenten* zullen heten, bestaan uit argumentposities van p. De andere bevatten argumentposities van recursieve predikaatsymbolen in P, en zullen bij gelegenheid *niet-p-componenten* genoemd worden.
2. De p-componenten vormen samen een *geordende partitie van p's vezameling argumentposities.*
3. Argumentposities van recursieve predikaten in P (met inbegrip van p) kunnen in ten hoogste één niet-p-component voorkomen.

Als we nu $\mathbb{N}$ uitbreiden tot $\mathbb{N}_b$ door toevoeging van een bijkomend "kleinste" element $\perp$, opslorpend voor $+$, en stellen dat $max(\emptyset) = \perp$, dan kunnen we op zinvolle wijze definiëren:

Definitie 5.6 Zij G een doel bestaand uit een aantal atomen waarvan er één is geselecteerd voor verder ontvouwen, p een n-air predikaat en $1 \leq i \leq n$. Dan:

$$M(G, p, i) = max(\{|t_i| \,|\, t_i \text{ is de term die voorkomt als } i\text{-de argument in een } \textit{niet geselecteerd} \text{ atoom } p(t_1, \ldots, t_n) \text{ in } G\})$$

Definitie 5.7 Zij P een programma en p een n-air (recursief) predikaatsymbool dat voorkomt in P. Zij O een cbo-k-partitie verbonden met p in P. Dan definiëren we $||.||_{p,O} : \{G|G$ is een doel in P's inherente taal waarvan het geselecteerde atoom predikaatsymbool p heeft $\} \rightarrow {\mathbb{N}_b}^k, G \mapsto (v_1, \ldots, v_k)$ als volgt:

- Indien $O[r] = \{i_{r1}, \ldots, i_{rs}\}$ een *p-component* is en $p(t_1, \ldots, t_n)$ is G's geselecteerde atoom, dan $v_r = |t_{i_{r1}}| + \cdots + |t_{i_{rs}}|$.
- Indien $O[r] = \{i_{r1,p_{r1}}, \ldots, i_{rj,p_{rj}}\}$ (de p_{rl}-annotaties verwijzen naar recursieve predikaatsymbolen in P), dan $v_r = M(G, p_{r1}, i_{r1}) + \cdots + M(G, p_{rj}, i_{rj})$.

Dergelijke $||.||_{p,O}$-functies leveren de benodigde maten voor het soort ontvouwen in voorbeeld 5.4. Bemerk dat $\perp$ de eventuele afwezigheid in het gemeten doel van één of meer argumentposities in de gegeven cbo-partitie opvangt.

Automatisering verloopt weer min of meer volgens de in sectie 5.1 geschetste lijnen, maar is vanzelfsprekend een stuk complexer. Het resulterende algoritme aanvaardt voorkeursrichtlijnen in verband met het selecteren van atomen. Zo kan bijvoorbeeld wisselwerking tussen twee predikaten (zie voorbeeld 5.4) door de gebruiker opgelegd worden.

Achterwaartse specialisaties

Ook zonder exotische computatieregels kan het gebruik van cbo-partities vruchten afwerpen. Inderdaad, ook achterwaarts (tegen de zoekrichting van de computatieregel in) informatie doorgeven kan nu beter behandeld worden. We bekijken weer een schematisch voorbeeld.

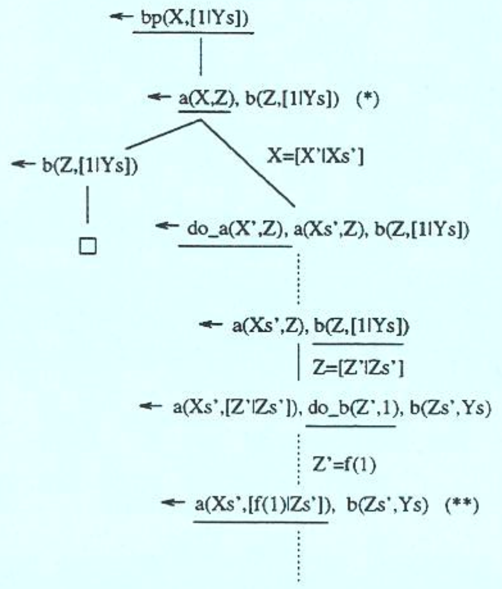
Voorbeeld 5.8

$bp(X, Y) \leftarrow a(X, Z), b(Z, Y)$
$a([], Y) \leftarrow$
$a([X|Xs], Y) \leftarrow do\_a(X, Y), a(Xs, Y)$
$b([], []) \leftarrow$
$b([X|Xs], [Y|Ys]) \leftarrow do\_b(X, Y), b(Xs, Ys)$

(De definities van *do_a* en *do_b* zijn hier verder niet van belang. We nemen enkel aan dat we dergelijke atomen volledig kunnen behandelen tijdens het ontvouwen.)
We wensen het volgende doel te ontvouwen:

$\leftarrow bp(X, [1|Ys])$

Een gedeelte van een nu mogelijke SLD-boom vindt men in figuur 2.
Het a-atoom ontvouwen in (**) kan als gewichtsprioriteit wordt verleend aan b's tweede argument: $||(*)||_{a,(\{2_b\},\{1,2\})} = (1, 0)$ terwijl $||(**)||_{a,(\{2_b\},\{1,2\})} = (0, 1)$.


← bp(X,[1|Ys])
← a(X,Z), b(Z,[1|Ys]) (*)
← b(Z,[1|Ys])
X=[X'|Xs']
← do_a(X',Z), a(Xs',Z), b(Z,[1|Ys])
← a(Xs',Z), b(Z,[1|Ys])
Z=[Z'|Zs']
← a(Xs',[Z'|Zs']), do_b(Z',1), b(Zs',Ys)
Z'=f(1)
← a(Xs',[f(1)|Zs']), b(Zs',Ys) (**)

Figuur 2: Achterwaartse specialisaties behandelen.

Atomen enkel vergelijken met echte voorouders

Het tweede deel van definitie 5.7 maakt geen onderscheid tussen atomen in verschillende ketens van voorouders en afstammelingen. Het blijkt dat zulks soms aanleiding geeft tot minder dan optimaal ontvouwen. Men kan cbo-partities en de erop gebaseerde maten natuurlijk aanpassen: vergelijk (maximale) argumentgewichten van contextatomen in een doel met de argumentgewichten van het overeenkomstige atoom in de relevante voorouder dat op dezelfde tak van de bewijsboom te vinden is. Het zonder verdere beperkingen doorvoeren van deze wijziging maakt echter *oneindig* ontvouwen mogelijk.

5.3 Automatisch maten verfijnen

Een zeer belangrijke component van automatische ontvouwingsalgoritmen vormen technieken om maten dynamisch te verfijnen. We hebben een eerste eenvoudige, concrete strategie ingebouwd in algoritme 4.3. En in sectie 5.1 hebben we kort besproken hoe de aldaar voorgestelde maten kunnen verfijnd worden. In deze sectie bekijken we het genoemde aspect van nabij en formuleren we een soort mal voor concrete algoritmen.

Definitie 5.9 Zij P een positief programma, dan noemen we $Atom_P$ de vezameling van atomen die kunnen gevormd worden in de taal inherent aan P.

Definitie 5.10 Zij P een positief programma met inherente taal $\mathcal{L}_P$. Veronderstel dat $R_0, R_1, \ldots, R_N$ een partitie is van $Atom_P$. Zij $W, >_W$ een welgegronde verzameling. Dan noemen we een functie

$F : \{G | G \text{ is een positief doel in } \mathcal{L}_P \text{ met geselecteerd atoom} \in R_k\} \rightarrow W, >_W$

een ***(R_k, P)-toepasbare maat met doelverzameling W***.

Definitie 5.11 Een verzameling $\{F | F \text{ is een } (R_k, P)\text{-toepasbare maat}\}$ wordt een ***(R_k, P)-toepasbare maatruimte*** genoemd.

Als P (of $\mathcal{L}_P$) duidelijk zijn uit de context zullen we een (R_k, P)-toepasbare maatruimte noteren als $\mathcal{F}_k$. Tenslotte wensen we op $\mathcal{F}_k$ een *(partiële) orderelatie*, genoteerd als $\gg_k$.

We kunnen nu een generisch ontvouwingsalgoritme formuleren.

Algoritme 5.12

```
Invoer
  een positief programma $P$
  een positief doel $\leftarrow A$
Uitvoer
  een SLD-boom $\tau$ voor $P \cup \{\leftarrow A\}$
Initialisatie
  $\tau$ wordt geïnitialiseerd als een SLD-boom met 1 tak,
    bestaande uit het doel $\leftarrow A$, zonder geselecteerd atoom.
  Initiële maten $F_1, \ldots, F_N$ worden gekozen.
Zolang er een niet beëindigde tak $D$ in $\tau$ bestaat doe
  Indien $D$ succesvol is, Dan beëindig $D$
  Anders indien $D$'s blad geen selecteerbaar atoom bevat, Dan beëindig $D$
  Anders
    Selecteer een selecteerbaar atoom
    Indien geen afleidingsstappen mogelijk zijn, Dan beëindig en faal $D$
    Anders breid $D$ uit
  Waarbij een atoom $p(t_1, \ldots, t_n) \in R_k$ in een doel $G$ selecteerbaar is
  indien één van de volgende (elkaar uitsluitende) voorwaarden vervuld is,
  indien het werkelijk geselecteerd wordt:
    • $G$ heeft geen directe dekkende voorouder
    • $G'$ is de directe dekkende voorouder van $G$ en
      $F_k(G') >_k F_k(G)$
    • $G'$ is de directe dekkende voorouder van $G$ en
      niet$(F_k(G') >_k F_k(G))$ en
      $\exists F \in \mathcal{F}_k$:   (1) $F_k \gg_k F$
                      (2) $F(G') >_k F(G)$
  Indien een atoom $p(t_1, \ldots, t_n)$ geselecteerd werd via de derde voorwaarde,
    Dan vervang $F_k$, in het stel actueel gebruikte maten,
    door een $F'_k$ die voldoet aan (1) en (2).
Einde
```

Een werkbare specialisatie van algoritme 5.12 vereist concrete keuzes voor de computatieregel R (gedeeltelijk), de partitie $R_0, R_1, \ldots, R_N$ (met een veilige R_0), $\mathcal{F}_1, \gg_1, \ldots, \mathcal{F}_N, \gg_N$ en initiële waarden voor $F_1, \ldots, F_N$.

De volgende stelling is waar:

Stelling 5.13 Een concrete specialisatie van algoritme 5.12 eindigt voor een positief programma P en doel $\leftarrow A$, een eindige SLD-boom τ voor $P \cup \{\leftarrow A\}$ producerend, indien $\forall k, 1 \leq k \leq N : \mathcal{F}_k, \gg_k$ een welgegronde verzameling is.

De tot hiertoe beschouwde concrete maatruimten zijn allemaal geordend via de respektievelijke verfijningsnoties. Ze zijn bovendien *eindig* en derhalve triviaal welgegrond, behalve in één geval: bij het rechtstreekse gebruik van cbo-partities van het gevorderde type dat kort beschreven werd onder het laatste punt van de vorige sectie.

5.4 Een gecombineerde aanpak

Alle tot hiertoe behandelde concrete methoden voor ontvouwen maken gebruik van definitie 4.1. Op die manier peilen de geconstrueerde maatfuncties structurele complexiteit. Een redelijke keuze in vele gevallen, maar duidelijk ongeschikt voor het behandelen van programma's waar de manipulatie van constanten een grote rol speelt.

Het is misschien mogelijk om zinvolle maten te bedenken die niet enkel voor structuur gevoelig zijn. In deze sectie volgen we echter een andere weg. Inderdaad, een bekende (maar niet geheel veilige) heuristiek bepaalt dat ontvouwen mag indien het beschouwde atoom geen variant is van een al eerder in dezelfde afleiding (*derivation*) geselecteerd atoom. Nu blijkt een combinatie mogelijk van deze regel met diverse methoden voor ontvouwen steunend op structuurmaten. Zoals in sectie 4.3 reeds aangestipt werd, leverde een dergelijke gecombineerde aanpak trouwens goede resultaten op in een beperkte proefondervindelijke studie.

Het raamwerk vertimmeren

We willen algoritme 3.15 aanpassen als volgt: ontvouwen is toegelaten, ook wanneer de maat gelijk blijft, op voorwaarde dat de "niet variant" test positief uitvalt.

Daartoe voeren we vooreerst (informeel) volgende noties in:

- Het deel van een doel G dat beschouwd wordt bij het berekenen van G's gewicht onder een maatfunctie F noemen we het *F-gemeten deel* van G.
- Indien G en G' twee doelen zijn zodanig dat hun *F-gemeten delen varianten zijn van elkaar*, dan noteren we dat als $G \sim_F G'$.

De ***hoofdlus*** in algoritme 3.15 wordt dan:

Algoritme 5.14

```
Zolang er een tak $D$ in $\tau$ bestaat zodanig dat $D \notin Gedaan$ doe
  Zij $(G, i)$ het blad van $D$
  Laat $Afstam(G, i)$ alle directe $>_{\tau_0}$-afstammelingen van $(G, i)$ bevatten
  Indien $Afstam(G, i) = \emptyset$
    Dan voeg $D$ toe aan $Gedaan$
  Anders indien er een directe dekkende voorouder $(G', j)$ van $(G, i)$ is
                met $R(G', j), R(G, i) \in R_n$
                zodanig dat geen van de volgende gevallen geldt:
                  1) $F_n(G', j) >_n F_n(G, i)$
                  2) $F_n(G', j) \equiv F_n(G, i) \wedge$
                     $\neg\exists(G'', k) \in D : (G'', k)$ dekt $(G, i) \wedge$
                                           $F_n(G'', k) \equiv F_n(G, i) \wedge$
                                           $(G'', k) \sim_{F_n} (G, i)$
    Dan voeg $D$ toe aan $Gedaan$
  Anders
    Vervang $\tau$ door $\tau \setminus D \cup \{D \cup \{(G^*, l)\} | (G^*, l) \in Afstam(G, i)\}$
Einde
```

Een belangrijke vraag is nu natuurlijk of het aldus gewijzigde basisalgoritme nog steeds eindig ontvouwen garandeert. Het blijkt dat de aard van de gebruikte maten in deze kwestie een cruciale rol speelt. Noem een maat F ***eindig metend*** voor een taal $\mathcal{L}$ indien slechts een *eindig* aantal doelen in $\mathcal{L}$ met een niet variant F-gemeten deel hetzelfde gewicht hebben onder F. Dan geldt volgende stelling:

Stelling 5.15 Algoritme 5.14 eindigt voor een positief programma P en doel $\leftarrow A$, gebruik makend van een vooropgestelde computatieregel R en een gegeven koppel $((R_0, R_1, \ldots, R_N), (F_1, \ldots, F_N))$, indien $F_1, \ldots, F_N$ eindig meten voor $\mathcal{L}_P$.

Maatfuncties gebaseerd op verzamelingen (definitie 4.2) of op partities (definitie 5.2) zijn eindig metend. Maatfuncties die gebruik maken van cbo-partities (definitie 5.7) daarentegen in het algemeen niet.

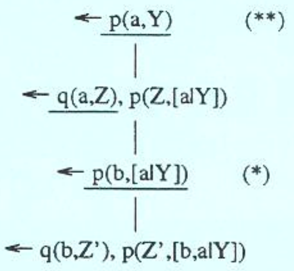
Automatisering

Volledig automatische algoritmen kunnen redelijk gemakkelijk aangepast worden; details laten we hier achterwege. Eén merkwaardig punt is echter een nadere beschouwing waard.

Voorbeeld 5.16

$p(X, Y) \leftarrow q(X, Z), p(Z, [X|Y])$
$q(a, b) \leftarrow$

Enkele afleidingsstappen voor $\leftarrow p(a, Y)$ ziet men in figuur 3.



Figuur 3: Gecombineerd ontvouwen.

In (∗) zouden we graag de aangegeven ontvouwing uitvoeren. Zulks is mogelijk via gecombineerd ontvouwen en de maat $|.|_{p,\{1\}}$. Om deze functie af te leiden uit de initiële $|.|_{p,\{1,2\}}$ moeten we echter *verfijnen ook toelaten in gevallen waar de nieuwe maat niet kleiner wordt* (maar ook niet groter) ten opzichte van de directe dekkende voorouder (hier (∗∗)). Dergelijke maatregel kan tot gevolg hebben dat gecombineerd ontvouwen met maatfuncties gebaseerd op verzamelingen dieper doorgaat dan bij het gebruik van maatfuncties gebaseerd op partities. Natuurlijk kan men een tussenvorm beschouwen en een deelverzameling van de verzameling argumentposities in een partitie opdelen.

5.5 Subtermen

Tenslotte hebben we ook ontvouwen verkend duidelijk buiten het in sectie 5.3 geschetste kader. We hebben namelijk een formeel apparaat ontwikkeld dat toelaat maatfuncties te definiëren die in plaats van hele argumenten eventueel enkel delen van dergelijke argumenten wegen. Automatisch dergelijke maten gebruiken, met de nodige mogelijkheden tot verfijning, houdt echter het dynamisch aanpassen van de $R_0, \ldots, R_N$ partitie (zie definitie 3.11) in: nieuwe klassen met een fijnere structuur worden gecreëerd waar nodig.

Meer onderzoek is vereist om de praktische waarde van aldus verkregen ontvouwingsalgoritmen in te schatten. Bovendien stelt in het bijzonder de behandeling van meta-vertolkers nog onvolledig opgeloste problemen. Enkele verdere uitbreidingen van de bestaande aanpak lijken echter veelbelovend.

5.6 Besluit

In het huidige hoofdstuk werd een verdere studie van verschillende automatische methoden voor eindig ontvouwen besproken. We hebben diverse, meer gevorderde maatfuncties voorgesteld, werkend met lexicale prioriteiten tussen veschillende componenten, en laten zien hoe ontvouwen erdoor verbeterd wordt. In het bijzonder hebben we bestudeerd hoe contextinformatie uit het beschouwde doel kan in aanmerking genomen worden. We hebben kort besproken welke aspecten betrokken zijn bij het automatisch zoeken naar optimale maatfuncties in deze omstandigheden. In sectie 5.3 hebben we trouwens expliciet aandacht besteed aan onderliggende wetmatigheden in dit verband. We hebben aangegeven hoe ontvouwen met maatfuncties kan gecombineerd worden met de welbekende "geen variant" regel. Een praktisch vruchtbaar samengaan, zo blijkt. Tenslotte werd zeer kort de mogelijkheid vermeld de structuur van deeltermen apart te beschouwen. Technieken van deze laatste soort vormen wellicht een goede basis voor het volstrekt automatisch effectief ontvouwen van meta-vertolkers.

6 Besluit

In dit proefschrift hebben we de semantiek van meta-programma's en het beheersen van partiële deductie in logisch programmeren bestudeerd. Beide onderwerpen zijn verwant via hun belang voor het efficiënte gebruik van praktische meta-vertolkers.

Eerst kwam de Herbrand semantiek van "vanille-achtige" meta-programma's aan bod. Verbonden met een gelaagd object-programma blijken dergelijke meta-programma's zelf *zwak gelaagd*. Hiermee wordt een interessant toepassingsgebied voor dat laatste concept geïdentificeerd. Zwakke gelaagdheid lijkt ons trouwens, in veel grotere mate dan lokale gelaagdheid, een krachtige en natuurlijke uitbreiding van gelaagdheid.

Verder hebben we het concept *taalonafhankelijkheid* ingevoerd. We hebben aangetoond dat het een sleutelrol speelt in de semantiek van de beschouwde categorie meta-programma's en in veel gevallen een mooie overeenstemming garandeert tussen object- en meta-model.

Tenslotte stippen we nog eens aan dat de beschreven aanpak de mogelijkheid biedt een aantal *versmolten* programma's zinvol te behandelen. Het is echter duidelijk dat verder werk wacht, ook hier.

Een tweede deel van het onderzoek betreft het beheersen van partiële deductie en ontvouwen. Het in hoofdstuk 3 beschreven *raamwerk voor eindig ontvouwen* vormt een eerste bijdrage in die context. In het bijzonder blijkt de notie van een *hiërarchische pregrond* een zeer geschikte basis voor automatische ontvouwings-algoritmes.

Een eerste dergelijk algoritme hebben we aangewend als bouwsteen voor *correcte, volledige en immer eindigende partiële deductie* (van positieve logische programma's). Zowel met de beschreven als met een stel aanverwante methoden werden *experimenten* uitgevoerd. Enkele voorlopige gevolgtrekkingen zijn opgenomen in sectie 4.3.

Hoofdstuk 5, tenslotte, beschrijft verschillende *verdere concrete manieren om te ontvouwen* met hiërarchische pregronden en maatfuncties. Onder andere methoden die ook informatie buiten het geselecteerde atoom verrekenen via lexicale prioriteiten, een combinatie met de bekende heuristiek het herhaaldelijk ontvouwen van varianten te vermijden, en een verkennende studie in verband met het beschouwen van deeltermen werden voorgesteld. Aldus kwamen zowel algemeenheid als praktische bruikbaarheid van het in hoofdstuk 3 opgestelde raamwerk scherper in beeld.

Er zijn echter ook inherente beperkingen. Met name de afwezigheid van enige substantiële *programma-analyse* vooraf doet zich soms als een tekort gevoelen. Verder onderzoek moet uitmaken wat hier mogelijk en wenselijk is. Ook en vooral het gedeeltelijk herdenken en verder uitwerken van methoden geschikt voor een doeltreffende behandeling van *meta-vertolkers* lijkt boeiend en belangrijk.

Das ist dem Menschen erlaubt und gegeben, daß er sich
der Wirklichkeit bediene zur Anschauung der Wahrheit,
und es ist das Wort "Poesie", welches die Sprache
fur diese Gegebenheit und Erlaubnis geprägt hat.

Thomas Mann, Die vertauschten Köpfe