Advanced Techniques for Logic Program Specialisation

Michael Leuschel

Department of Computer Science, K.U.Leuven

Abstract

Program specialisation, also called *partial evaluation* or *partial deduction*, is an automatic technique for program optimisation. The central idea is to specialise a given source program for a particular application domain. Program specialisation encompasses traditional compiler optimisation techniques, but uses more aggressive transformations, yielding both much greater speedups and more difficulty in controlling the transformation process.

Because of their clear (and often simple) semantical foundations, declarative languages offer significant advantages for the design of semantics based program analysers, transformers and optimisers. This thesis exploits these advantages in the context of logic programming and develops advanced techniques for program specialisation, striving to produce tangible practical benefits within the larger objective of turning declarative languages and program specialisation into valuable tools for constructing reliable, maintainable *and* efficient programs.

This thesis contains contributions within that context around several themes. New, powerful methods for the control problem within partial deduction, based on characteristic trees, are developed. A method for automatic compiler generation, which does not have to resort to self-applicable specialisers, is presented. A practical and very successful application in the area of integrity checking in deductive databases is worked out. Finally, an integration of "unfold/fold" transformations with partial deduction, as well as a further integration with abstract interpretation, are developed.

Acknowledgements

I would like to direct my first words of gratitude towards my supervisor, Professor Danny De Schreye. Even when submerged in administrative work — trying to find money for us, younger researchers — he somehow always found the time to thoroughly read, comment and improve upon my drafts. Flaws in the presentation were unavoidably detected and his talent for presenting complex ideas in a clear and comprehensible manner is hard to equal. He also helped me to get in touch with the research community, and provided the necessary focus for my research.

I would also like to thank Professor Van Assche for accepting to be my second supervisor, as well as him and the other members of the thesis committee, Professors Maurice Bruynooghe, Karel De Vlaminck, John Gallagher and Neil Jones, for reading my thesis and providing me with very valuable comments.

I am also indebted to all the colleagues at the Department of Computer Science at the K.U. Leuven. The research environment proved to be very stimulating and it was a big help to have so many good researchers in logic programming close by: for any particular sub-field, a local expert was always on duty. Among these colleages, Bern Martens had maybe the biggest impact on my thesis: I co-authored several papers with him and he should be considered a co-supervisor. He always had an ear for my latest ideas or comments and the ensuing discussions often lead to new research ideas. It was always a pleasure to work with him and his ability to forge phrases in English is unequalled among the people I worked with. Maurice Bruynooghe, apart from sharing with me his expertise in abstract interpretation, often pointed me to very relevant references. His comments were often to the point, helping me to improve the technical content of my papers as well as providing me with new research ideas. Bart Demoen helped me uncountable times, notably with efficiency and implementation issues of logic programming. He even went so far as to add some custom features to Prolog by BIM for me. Marc Denecker also often greatly helped me with semantical issues related to model theory and the treatment of negation. The following is a (probably non-exhaustive) list of other persons from our department to whom I am grateful for their help: Eddy Bevers, Dirk Dussart, Gerda Janssens, Anne Mulkers, Jacques Riche, Kostis Sagonas, Kristof Van Belleghem, Henk Vandecasteele and Wim Vanhoof.

I would also like to express my appreciation towards Jesper Jørgensen, Morten Heine Sørensen and André De Waal, with whom I co-authored some of the papers which are at the basis of my thesis. It was a pleasure

to collaborate with Jesper Jørgensen and his expertise in functional programming, partial evaluation and supercompilation was extremely helpful and stimulating. With Morten Heine Sørensen I managed to write a paper, after some initial discussions, strictly by e-mail collaboration. It was a very pleasurable and fruitful experience.

On the more personal side, I am greatly indebted to my wife Béatrice who helped me attain the perseverance and determination to do the research which led to this thesis. Also, without the support of my parents, doing the research would have been much more difficult and I might have chosen a more commercially oriented career. I also thank my grandparents which, in contrast to myself, always believed in my ability to become a "Dr.".

Finally I want to thank all the persons with whom I had discussions over the years and which helped me in accumulating the necessary knowledge to write this thesis. In particular, in addition to all the persons already mentioned above, I would like to thank the participants of the Compulog II project. The Compulog II meeting in 1994 in Cyprus immediately got me into contact with a large part of the European logic programming research community, which, in retrospect, helped me a lot in my subsequent research efforts. To name but a few additional persons with whom I had stimulating discussions, sometimes at the source of precious improvements in the thesis: Krzysztof Apt, Hendrik Decker, Stefan Decker, Włodek Drabent, Robert Glück, Corin Gurr, Fergus Henderson, Manuel Hermenegildo, Jan Hric, Robert Kowalski, Laura Lafave, John Lloyd, Torben Mogensen, Ulrich Neumerkel, Alberto Pettorossi, Maurizio Proietti, Dan Sahlin, Zoltan Somogyi and Valentin Turchin.

Michael Leuschel
Leuven, May 1996

Financial support for my work has been kindly provided by:

- the Katholieke Universiteit Leuven,
- ESPRIT Basic Research Action COMPULOG II and
- the GOA "Non-Standard Applications of Abstract Interpretation" (Belgium).

Brevity is the soul of wit;

William Shakespeare in *Hamlet, 2,2*

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Logic programming and program specialisation

> Mathematical logic, then, is a branch of mathematics which has much the same relation to the analysis and criticism of thought as geometry does to the science of space.
>
> *Haskell B. Curry* in [58]

Declarative programming languages

Declarative programming languages, are high-level programming languages in which one only has to state *what* is to be computed and not necessarily *how* it is to be computed.

Logic programming and *functional programming* are two prominent members of this class of programming languages. While functional programming is based on the λ-calculus, logic programming has its roots in first-order logic and automated theorem proving. Both approaches share the view that a program is a *theory* and execution consists in performing *deduction* from that theory (sometimes complemented by abduction or induction).

Logic programming

Logic programming grew out of the insight that a subset of first-order logic, based on *Horn clauses*, has an efficient operational reading and can thus be used as the basis of a programming language.

Take the following Horn clause, part of a logic program computing the derivative of a function:

$$\forall(\mathit{derivative}(F+G, F'+G') \leftarrow \mathit{derivative}(F, F') \wedge \mathit{derivative}(G, G'))$$

This Horn clause has a clear logical semantics, whose validity can be checked *independently* of the rest of the logic program. Furthermore a *natural language translation* of this clause can be produced in a straightforward manner, making the program more accessible to people less versed with the (black) art of computer programming:

> If F' is the derivative of F and G' is the derivative of G then $F'+G'$ is the derivative of $F+G$.

Finally, the clause has an operational reading as well, allowing for an efficient execution mechanism:

> To calculate $\mathit{derivative}(F+G, F'+G')$ one should first calculate $\mathit{derivative}(F, F')$ and then calculate $\mathit{derivative}(G, G')$

Using logic as the basis of a programming language also means that a uniform language can be used to express and reason about programs, specifications, databases, queries and integrity constraints.

On the more practical side, logic programming languages allow *nondeterminism*, making them especially well-suited for applications like parsing. They also provide for *automatic memory management*, thus avoiding a major source of errors in other programming languages. Another advantage of logic programming languages is that they can compute with *partially specified data* and that the input/output relation is not fixed beforehand. For instance, the above mentioned program can not only be used to compute the derivative of e.g. $x*x$ via the query $\leftarrow \mathit{derivative}(x*x, F)$ it can also be used to calculate its integral via the query $\leftarrow \mathit{derivative}(G, x*x)$. Finally, although early logic programming languages have been renowned for their lack of efficiency, the implementations have grown ever more efficient, recent efforts reaching or even surpassing the speeds of imperative languages for some applications.

Program specialisation

Program specialisation, also called *partial evaluation* or *partial deduction*, is an automatic technique for program optimisation. The central idea is to specialise a given source program for a particular application domain. This is (mostly) done by a *well-automated* application of parts of the Burstall and Darlington unfold/fold transformation framework. Program specialisation encompasses traditional compiler optimisation techniques, such as

constant folding and *in-lining*, but uses more aggressive transformations, yielding both (much) greater speedups and more difficulty in controlling the transformation process. It is thus similar in concept to, but in several ways stronger than highly optimising compilers.

Program specialisation can be used to speed up existing programs for certain application domains, sometimes achieving speedups of several orders of magnitude. It however also allows the user to conceive more generally applicable programs using a more secure, readable and maintainable style. The program specialiser then takes care of transforming this general purpose, readable, but inefficient program into an efficient one (e.g. incorporating Knuth-Morris-Pratt like optimisations or taking advantage of the hidden implementation parts of modules).

Program specialisation has proven to be useful in many application areas. For instance, a lot of programs exhibit interpretive behaviour, the overhead of which can be removed by program specialisation. The specialisation process can in such circumstances be seen as compilation from the higher level language down to the language in which the interpreter is written. Furthermore, given that a specialiser is *self-applicable*, i.e. is able to effectively specialise itself, one can obtain compilers as well as compiler generators automatically from an interpreter by the so called *Futamura projections*.

Apart from that, whenever part of the input changes more slowly than the remaining input, program specialisation can prove to be very valuable. One such often mentioned application is ray tracing, where a ray tracer can be specialised for a particular scene with an unknown viewpoint. Other successful applications of program specialisation range from neural network training, spreadsheet computations over to scientific computing. Simpler forms of program specialisation have also found their way into compilers and program analysers. For instance, the Mercury compiler can often get rid of the overhead of higher-order programming via partial evaluation while a lot of abstract interpretation systems use a specialisation technique called abstract compilation.

1.2 Overview of the thesis

Aim of the thesis

Because of their clear (and often simple) semantical foundations, declarative languages offer significant advantages for the design of semantics based program analysers, transformers and optimisers.

First, because there *exists* a *clear* and *simple* semantical foundation,

techniques for program specialisation *can* be proven correct in a formal way. Furthermore, program specialisation does not have to preserve every execution aspect of the source program, as long as the declarative semantics is respected. This permits much more powerful optimisations, impossible to obtain when the specialiser has to preserve every operational aspect of the source program.

We[1] will try to exploit these advantages in the context of logic programming and develop *automatic methods* for program specialisation, striving to produce tangible practical benefits within the larger objective of turning declarative languages and program specialisation into valuable tools for constructing reliable, maintainable *and* efficient programs.

Roadmap

This thesis consists of 6 major parts. Figure 1.1 contains a graphical representation of the chapters. Arrows indicate dependencies between chapters; topological sorting can be used to determine an admissible reading sequence.

Structure of the parts

- **Part I** contains background material about first-order logic, logic programming and program specialisation. A brief summary of other mathematical notations and conventions can be found in Appendix A. Chapter 2 starts out from the roots of logic programming in first-order logic and automated theorem proving and presents the syntax, semantics and proof theory of logic programs. In Chapter 3 the general idea of program specialisation, based on Kleene's S-M-N theorem, is introduced. A particular technique for specialising logic programs, called *partial deduction*, is then formalised. The theoretical underpinnings of this approach, based on the correctness results by Lloyd and Shepherdson [185], are exhibited. We also elaborate on the control issues of partial deduction and define the *control of polyvariance problem*. The latter is basically related to how many specialised versions should be generated for a particular predicate.
- **Part II** of the thesis attends to a method for precise and fine-grained control of polyvariance, whose termination only depends on the termination of the unfolding component. Chapter 4 first shows that a

[1] The word "we" should not be interpreted as to some royal ambitions of mine. In fact, as most chapters of this thesis are adapted from papers which other persons have co-authored, it just avoids confusing switches of narrative.

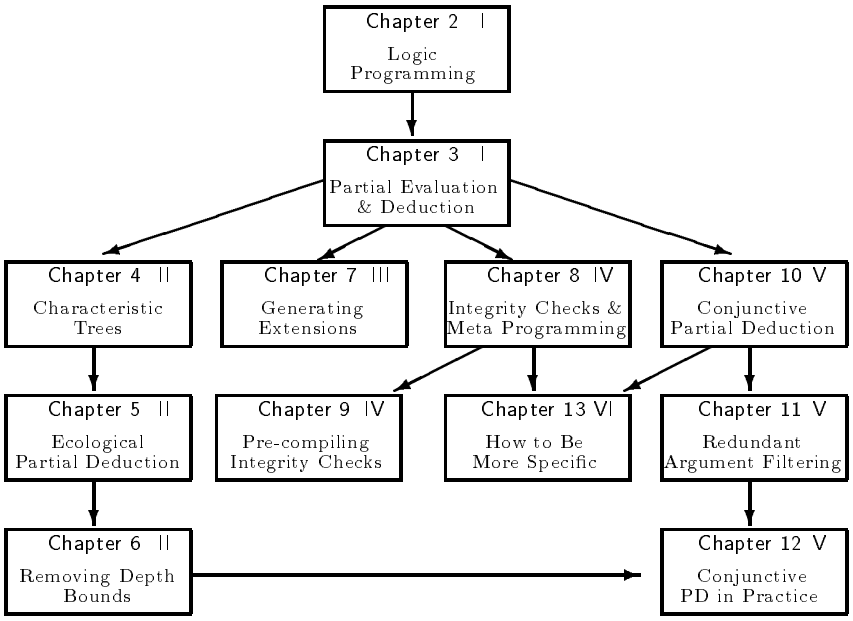


Figure 1.1: Overview of the chapters

good control of polyvariance should not (solely) rely on syntactic information. *Characteristic trees* and *paths*, which are abstractions of the computational behaviour of goals introduced by Gallagher and Bruynooghe [100, 97], are shown to be much more viable in that context. The chapter however also illustrates shortcomings of the existing approaches using characteristic trees, notably their incapacity to preserve them upon generalisation. This can lead to severe specialisation losses, sometimes combined with non-termination of the partial deduction process. Chapter 5 attends to solving this problem. To that end an extension of the Lloyd and Shepherdson framework, called *ecological partial deduction*, is developed which allows the preservation of characteristic trees upon generalisation in a straightforward manner. Chapter 6 solves another important problem with the existing approach based on characteristic trees: an *ad-hoc depth bound* has to be imposed on characteristic trees to ensure termination of the specialisation process. A full-fledged algorithm for partial deduction, which does not require this depth bound, is developed in the chapter.

Finally, an implementation of this algorithm is described and used for extensive experiments, showing its improved performance over existing specialisation systems.

- **Part III**, consisting just of Chapter 7, concentrates on issues related to self-application. In particular it shows how one can get the benefits of self-application *without* self-application. Self-application has not been as much in focus in partial deduction as in partial evaluation of functional and imperative languages, and the attempts to self-apply partial deduction systems have not been altogether that successful. To overcome this predicament, the chapter adapts the so called "cogen approach" for logic programming languages. The central idea is to write a compiler generator instead of a self-applicable specialiser. It is demonstrated that using the cogen approach one gets very efficient compiler generators which generate very efficient generating extensions which in turn yield (for some examples at least) very good and non-trivial specialisation.
- **Part IV** elaborates on a particular (and novel) application of program specialisation to integrity checking in more detail. Integrity constraints are useful for the specification of deductive databases, as well as for inductive and abductive logic programs. Verifying integrity constraints upon updates is a major efficiency bottleneck and specialised methods have been developed to speedup this task. They can however still incur a considerable overhead and the main idea in this part of the thesis is to further optimise these methods in a principled way using the techniques of meta-programming and program specialisation. Chapter 8 first presents some background in deductive databases, focussing on the problem of integrity checking. It also discusses issues in meta-programming, notably elaborating on the differences between the ground, non-ground and mixed representations. In Chapter 9, a meta-interpreter for integrity checking in hierarchical databases is presented, which is then partially evaluated for certain transaction patterns. Extensive experiments are conducted, exhibiting considerable speedups over existing integrity checking techniques. The good results hinge on the fact that a lot of the integrity checking can already be performed given an update pattern without knowing the actual, concrete update.
- **Part V** develops a major improvement of partial deduction. In fact, partial deduction within the Lloyd and Shepherdson framework, as well as within the extension presented in Part II, cannot achieve certain optimisations which are possible within the full framework of unfold/fold transformations. In this part of the thesis we therefore endeavour to combine the advantages of partial deduction in terms of

complexity and control with the power of unfold/fold transformations. In Chapter 10 we present the formal framework of *conjunctive partial deduction*, extending the Lloyd and Shepherdson framework by specialising entire conjunctions instead of simply individual atoms. Chapter 11 presents a complementary technique detecting and removing redundant arguments. Together these techniques are able to accommodate optimisations like tupling and deforestation. In Chapter 12 these new techniques are put on trial on an extensive set of pure Prolog programs, illustrating the increased potential of conjunctive partial deduction but also highlighting some remaining control problems.

- **Part VI**, consisting just of Chapter 13, presents a further extension of partial deduction, obtained by integrating abstract interpretation techniques. Chapter 13 starts out by presenting some remaining limitations of standard and conjunctive partial deduction in terms of inferring success-information. These shortcomings are remedied by combining conjunctive partial deduction with an abstract interpretation technique known as more specific program construction. The practical impact of this approach is illustrated on some applications, notably specialising meta-programs written in the ground representation and extending the approach of Part IV to recursive databases.

Origin of the chapters

Earlier versions of Chapters 5 and 6 appeared in [168] and [178] respectively. [179] combines these two papers and was written concurrently with Part II of the thesis. Parts of [172] have also been incorporated into Chapters 3–5. Chapter 7 is adapted from [142]. Chapters 8 and 9 are revised and extended versions of [173] [2], [174] and incorporate some parts from [177]. Chapters 10, 11 and 12 have been adapted from [175], [182] and [143] respectively. Parts of these chapters have been incorporated into [62]. Finally, Chapter 13 is adapted from [181].

[2] "Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission."

Part I

Technical Background

Chapter 2

Logic and Logic Programming

In this chapter we summarise some essential background in first-order logic and logic programming, required for the proper comprehension of this thesis. The exposition is mainly inspired by [7] and [184] and in general adheres to the same terminology. The reader is referred to these works for a more detailed presentation, comprising motivations, examples and proofs. Some other good introductions to logic programming can also be found in [218], [85, 8] and [202], while a good introduction to first-order logic and automated theorem proving can be found in [94].

2.1 First-order logic and syntax of logic programs

We start with a brief presentation of first-order logic.

Definition 2.1.1 (alphabet) An *alphabet* consists of the following classes of symbols:

1. *variables*;
2. *function symbols*;
3. *predicate symbols*;
4. *connectives*, which are $\neg$ negation, $\wedge$ conjunction, $\vee$ disjunction, $\leftarrow$ implication, and $\leftrightarrow$ equivalence;

5. *quantifiers*, which are the existential quantifier $\exists$ and the universal quantifier $\forall$;

6. *punctuation symbols*, which are "(", ")" and ",".

Function and predicate symbols have an associated *arity*, a natural number indicating how many arguments they take in the definitions following below. *Constants* are function symbols of arity 0, while *propositions* are predicate symbols of arity 0.

The classes 4 to 6 are the same for all alphabets. In the remainder of this thesis we suppose the set of variables is countably infinite. In addition, alphabets with a finite set of function and predicate symbols will simply be called *finite*. An *infinite* alphabet is one in which the number of function and/or predicate symbols is not finite but countably infinite.

We will try to adhere as much as possible to the following syntactical conventions throughout the thesis:

- Variables will be denoted by upper-case letters like X, Y, Z, usually taken from the later part of the (Latin) alphabet.
- Constants will be denoted by lower-case letters like a, b, c, usually taken from the beginning of the (Latin) alphabet.
- The other function symbols will be denoted by lower-case letters like f, g, h.
- Predicate symbols will be denoted by lower-case letters like p, q, r.

Definition 2.1.2 (terms, atoms) The set of *terms* (over some given alphabet) is inductively defined as follows:

- a variable is a term
- a constant is a term and
- a function symbol f of arity $n > 0$ applied to a sequence $t_1, \ldots, t_n$ of n terms, denoted by $f(t_1, \ldots, t_n)$, is also a term.

The set of *atoms* (over some given alphabet) is defined in the following way:

- a proposition is an atom and
- a predicate symbol p of arity $n > 0$ applied to a sequence $t_1, \ldots, t_n$ of n terms, denoted by $p(t_1, \ldots, t_n)$, is an atom.

We will also allow the notations $f(t_1, \ldots, t_n)$ and $p(t_1, \ldots, t_n)$ in case $n = 0$. $f(t_1, \ldots, t_n)$ then simply represents the term f and $p(t_1, \ldots, t_n)$ represents the atom p. For terms representing lists we will use the usual Prolog [80, 262, 53] notation: e.g. [] denotes the empty list, $[H|T]$ denotes a non-empty list with first element H and tail T.

Definition 2.1.3 (formula) A *(well-formed) formula* (over some given alphabet) is inductively defined as follows:

- An atom is a formula.
- If F and G are formulas then so are $(\neg F)$, $(F \vee G)$, $(F \wedge G)$, $(F \leftarrow G)$, $(F \leftrightarrow G)$.
- If X is a variable and F is a formula then $(\forall X F)$ and $(\exists X F)$ are also formulas.

To avoid formulas cluttered with the punctuation symbols we give the connectives and quantifiers the following precedence, from highest to lowest:

$$1.\ \neg, \forall, \exists, \qquad 2.\ \vee, \qquad 3.\ \wedge, \qquad 4.\ \leftarrow, \leftrightarrow.$$

For instance, we will write $\forall X(p(X) \leftarrow \neg q(X) \wedge r(X))$ instead of the less readable $(\forall X(p(X) \leftarrow ((\neg q(X)) \wedge r(X))))$.

The set of all formulas constructed using a given alphabet A is called the *first-order language* given by A.

First-order logic assigns meanings to formulas in the form of *interpretations* over some domain D:

- Each function symbol of arity n is assigned an n-ary function $D^n \mapsto D$. This part, along with the choice of the domain D, is referred to as a *pre-interpretation*.
- Each predicate symbol of arity n is assigned an n-ary relation, i.e. a subset of D^n (or equivalently an n-ary function $D^n \mapsto \{\mathbf{true}, \mathbf{false}\}$).
- Each formula is given a truth value, **true** or **false**, depending on the truth values of the sub-formulas. (For more details see e.g. [94] or [184]).

A *model* of a formula is simply an interpretation in which the formula has the value **true** assigned to it. Similarly, a model of a set S of formulas is an interpretation which is a model for all $F \in S$.

For example, let I be an interpretation whose domain D is the set of natural numbers $\mathbb{N}$ and which maps the constant a to 1, the constant b to 2 and the unary predicate p to the unary relation $\{(1)\}$. Then the truth value of $p(a)$ under I is **true** and the truth value of $p(b)$ under I is **false**. So I is a model of $p(a)$ but not of $p(b)$. I is also a model of $\exists X p(X)$ but not of $\forall X p(X)$.

We say that two formulas are *logically equivalent* iff they have the same models. A formula F is said to be a *logical consequence* of a set of formulas

S, denoted by $S \models F$, iff F is assigned the truth value **true** in all models of S. A set of formulas S is said to be *inconsistent* iff it has no model. It can be easily shown that $S \models F$ holds iff $S \cup \{\neg F\}$ is inconsistent. This observation lies at the basis of what is called a proof by *refutation*: to show that F is a logical consequence of S we show that $S \cup \{\neg F\}$ leads to inconsistency.

From now on we will also use **true** (resp. **false**) to denote some arbitrary formula which is assigned the truth value **true** (resp. **false**) in every interpretation. If there exists a proposition p in the underlying alphabet then **true** could e.g. stand for $p \vee \neg p$ and **false** could stand for $p \wedge \neg p$.[1] We also introduce the following shorthands for formulas:

- if F is a formula, then $(F \leftarrow)$ denotes the formula $(F \leftarrow \mathbf{true})$ and $(\leftarrow F)$ denotes the formula $(\mathbf{false} \leftarrow F)$.
- $(\leftarrow)$ denotes the formula $(\mathbf{false} \leftarrow \mathbf{true})$.

In the following we define some other frequently occurring kinds of formulas.

Definition 2.1.4 (literal) If A is an atom then the formulas A and $\neg A$ are called *literals*. Furthermore, A is called a *positive* literal and $\neg A$ a *negative* literal.

Definition 2.1.5 (conjunction, disjunction) Let $A_1, \ldots, A_n$ be literals, where $n > 0$. Then $A_1 \wedge \ldots \wedge A_n$ is a *conjunction* and $A_1 \vee \ldots \vee A_n$ is a *disjunction*.

Usually we will assume $\wedge$ (respectively $\vee$) to be associative, in the sense that we do not distinguish between the logically equivalent, but syntactically different, formulas $p \wedge (q \wedge r)$ and $(p \wedge q) \wedge r$.

Definition 2.1.6 (scope) Given a formula $(\forall X F)$ (resp. $(\exists X F)$) the *scope* of $\forall X$ (resp. $\exists X$) is F. A *bound occurrence* of a variable X inside a formula F is any occurrence immediately following a quantifier or an occurrence within the scope of a quantifier $\forall X$ or $\exists X$. Any other occurrence of X inside F is said to be *free*.

Definition 2.1.7 (universal and existential closure) Given a formula F, the *universal closure* of F, denoted by $\forall(F)$, is a formula of the form $(\forall X_1 \ldots (\forall X_m F) \ldots)$ where $X_1, \ldots, X_m$ are all the variables having a free occurrence inside F (in some arbitrary order). Similarly the *existential closure* of F, denoted by $\exists(F)$, is the formula $(\exists X_1 \ldots (\exists X_m F) \ldots)$.

[1] In some texts on logic (e.g. [94]) **true** and **false** are simply added to the alphabet and treated in a special manner by interpretations. The only difference is then that **true** and **false** can be considered as atoms, which can be convenient in some places.

The following class of formulas plays a central role in logic programming.

Definition 2.1.8 (clause) A *clause* is a formula of the form $\forall(H_1 \vee \ldots \vee H_m \leftarrow B_1 \wedge \ldots \wedge B_n)$, where $m \geq 0, n \geq 0$ and $H_1, \ldots, H_m, B_1, \ldots, B_n$ are all literals. $H_1 \vee \ldots \vee H_m$ is called the *head* of the clause and $B_1 \wedge \ldots \wedge B_n$ is called the *body*.
A *(normal) program clause* is a clause where $m = 1$ and H_1 is an atom. A *definite program clause* is a normal program clause in which $B_1, \ldots, B_n$ are atoms. A *fact* is a program clause with $n = 0$. A *query* or *goal* is a clause with $m = 0$ and $n > 0$. A *definite goal* is a goal in which $B_1, \ldots, B_n$ are atoms. The *empty clause* is a clause with $n = m = 0$. As we have seen earlier, this corresponds to the formula **false** $\leftarrow$ **true**, i.e. a contradiction. We also use $\square$ to denote the empty clause.

In logic programming notation one usually omits the universal quantifiers encapsulating the clause and one also often uses the comma (',') instead of the conjunction in the body, e.g. one writes $p(s(X)) \leftarrow q(X), p(X)$ instead of $\forall X(p(f(X)) \leftarrow (q(X) \wedge p(X)))$. We will adhere to this convention.

Definition 2.1.9 (program) A *(normal) program* is a set of program clauses. A *definite program* is a set of definite program clauses.

In order to be able to express a given program P in a first-order language L given by some alphabet A, the alphabet A must of course contain the function and predicate symbols occurring within P. The alphabet might however contain additional function and predicate symbols which do not occur inside the program. We therefore denote the underlying first-order language of a given program P by $\mathcal{L}_P$ and the underlying alphabet by $\mathcal{A}_P$. For technical reasons related to definitions below, we suppose that there is at least one constant symbol in $\mathcal{A}_P$.

2.2 Semantics of logic programs

Given that a program P is just a set of formulas, which happen to be clauses, the logical meaning of P might simply be seen as all the formulas F for which $P \models F$. For normal programs this approach will turn out to be insufficient, but for definite programs it provides a good starting point.

2.2.1 Definite programs

To determine whether a formula F is a logical consequence of another formula G, we have to examine whether F is true in *all* models of G. One big

advantage of clauses is that it is sufficient to look just at certain canonical models, called the Herbrand models.

In the following we will define these canonical models. Any term, atom, literal, clause will be called *ground* iff it contains no variables.

Definition 2.2.1 Let P be a program written in the underlying first-order language $\mathcal{L}_P$ given by the alphabet $\mathcal{A}_P$. Then the *Herbrand universe* $\mathcal{U}_P$ is the set of all ground terms over $\mathcal{A}_P$.[2] The *Herbrand base* $\mathcal{B}_P$ is the set of all ground atoms in $\mathcal{L}_P$.

A *Herbrand interpretation* is simply an interpretation whose domain is the Herbrand universe $\mathcal{U}_P$ and which maps every term to itself. A *Herbrand model* of a set of formulas S is an Herbrand interpretation which is a model of S.

The interest of Herbrand models for logic programs derives from the following proposition (the proposition does *not* hold for arbitrary formulas).

Proposition 2.2.2 A set of clauses has a model iff it has a Herbrand model.

This means that a formula F which is true in all Herbrand models of a set of clauses C is a logical consequence of C. Indeed if F is true in all Herbrand models then $\neg F$ is false in all Herbrand models and therefore, by Proposition 2.2.2, $C \cup \{\neg F\}$ is inconsistent and $C \models F$.

Note that a Herbrand interpretation or model can be identified with a subset H of the Herbrand base $\mathcal{B}_P$ (i.e. $H \in 2^{\mathcal{B}_P}$): the interpretation of $p(d_1, \ldots, d_n)$ is **true** iff $p(d_1, \ldots, d_n) \in H$ and the interpretation of $p(d_1, \ldots, d_n)$ is **false** iff $p(d_1, \ldots, d_n) \notin H$. This means that we can use the standard set order on Herbrand models and define minimal Herbrand models as follows.

Definition 2.2.3 A Herbrand model $H \subseteq \mathcal{B}_P$ for a given program P is a *minimal Herbrand model* iff there exists no $H' \subset H$ which is also a Herbrand model of P.

For definite programs there exists a *unique* minimal Herbrand model, called the *least Herbrand model*, denoted by $\mathcal{H}_P$. Indeed it can be easily shown that the intersection of two Herbrand models for a definite program P is still a Herbrand model of P. Furthermore, the entire Herbrand base $\mathcal{B}_P$ is always a model for a definite program and one can thus obtain the least Herbrand model by taking the intersection of all Herbrand models.

[2] It is here that the requirement that $\mathcal{A}_P$ contains at least one constant symbol comes into play. It ensures that the Herbrand universe is never empty.

The least Herbrand model $\mathcal{H}_P$ can be seen as capturing the *intended meaning* of a given definite program P as it is sufficient to infer all the logical consequences of P. Indeed, a formula which is true in the least Herbrand model $\mathcal{H}_P$ is true in all Herbrand models and is therefore a logical consequence of the program.

Example 2.2.4 Take for instance the following program P:

$int(0) \leftarrow$
$int(s(X)) \leftarrow int(X)$

Then the least Herbrand model of P is $\mathcal{H}_P = \{int(0), int(s(0)), \ldots\}$ and indeed $P \models int(0)$, $P \models int(s(0))$, ... But also note that for definite programs the entire Herbrand base $\mathcal{B}_P$ is also a model. Given a suitable alphabet $\mathcal{A}_P$, we might have $\mathcal{B}_P = \{int(a), int(0), int(s(a)), int(s(0)), \ldots\}$. This means that the atom $int(a)$ is consistent with the program P (i.e. $P \not\models \neg int(a)$), but is not implied either (i.e. $P \not\models int(a)$).

It is here that logic programming goes beyond "classical" first-order logic. In logic programming one (usually) assumes that the program gives a *complete* description of the intended interpretation, i.e. anything which cannot be inferred from the program is assumed to be false. For example, one would say that $\neg int(a)$ is a consequence of the above program P because $int(a) \notin \mathcal{H}_P$. This means that, from a logic programming perspective, the above program captures exactly the natural numbers, something which is impossible to accomplish within first-order logic (for a formal proof see e.g. Corollary 4.10.1 in [78]).

A possible inference scheme, capturing this aspect of logic programming, was introduced in [238] and is referred to as the *closed world assumption* (CWA). The CWA cannot be expressed in first-order logic (a second-order logic axiom has to be used to that effect, see e.g. the approach adopted in [183]). Note that using the CWA leads to *non-monotonic* inferences, because the addition of new information can remove certain, previously valid, consequences. For instance, by adding the clause $int(a) \leftarrow$ to the above program the literal $\neg int(a)$ is no longer a consequence of the logic program.

2.2.2 Fixpoint characterisation of $\mathcal{H}_P$

We now present a more constructive characterisation of the least Herbrand model, as well as the associated set of consequences, using fixpoint concepts. We first need the following definitions:

Definition 2.2.5 (substitution) A *substitution* θ is a finite set of the form $\theta = \{X_1/t_1, \ldots, X_n/t_n\}$ where $X_1, \ldots, X_n$ are distinct variables and

$t_1, \ldots, t_n$ are terms such that $t_i \neq X_i$. Each element X_i/t_i of θ is called a *binding*.

Alternate definitions of substitutions exist in the literature (e.g. in [89, 84], see also the discussion in [149]), but the above is the most common one in the logic programming context.

We also define an *expression* to be either a term, an atom, a literal, a conjunction, a disjunction or a program clause.

Definition 2.2.6 (instance) Let $\theta = \{X_1/t_1, \ldots, X_n/t_n\}$ be a substitution and E an expression. Then the *instance* of E by θ, denoted by $E\theta$, is the expression obtained by simultaneously replacing each occurrence of a variable X_i in E by the term t.

We can now define the following operator mapping Herbrand interpretations to Herbrand interpretations.

Definition 2.2.7 (T_P) Let P be a program. We then define the *(ground) immediate consequence operator* T_P $2^{\mathcal{B}_P} \mapsto 2^{\mathcal{B}_P}$ by:

$$T_P(I) = \{A \in \mathcal{B}_P \mid A \leftarrow A_1, \ldots A_n \text{ is a ground instance of a clause in } P \text{ and } \{A_1, \ldots, A_n\} \subseteq I\}$$

Every pre-fixpoint I of T_P, i.e. $T_P(I) \subseteq I$, corresponds to a Herbrand model of P and vice versa. This means that to study the Herbrand models of P one can also investigate the pre-fixpoints of the operator T_P. For definite programs, one can prove that T_P is a continuous mapping and that it has a least fixpoint $lfp(T_P)$ which is also its least pre-fixpoint.

The following definition will provide a way to calculate the least fixpoint:

Definition 2.2.8 Let T be a mapping $2^D \mapsto 2^D$. We then define $T \uparrow 0 = \emptyset$ and $T \uparrow i + 1 = T(T \uparrow i)$. We also define $T \uparrow \infty$ to stand for $\bigcup_{i<\infty} T \uparrow i$.

The following theorem from [277] links the least Herbrand model with the least fixpoint of T_P and provides a way of constructing it.

Theorem 2.2.9 (Fixpoint characterisation of the least Herbrand model) Let P be a definite program. Then $\mathcal{H}_P = lfp(T_P) = T_P \uparrow \infty$.

2.2.3 Normal programs

We have already touched upon the CWA. Given a formula F, this rule amounts to inferring that $\neg F$ is a logical consequence of a program P if F is not a logical consequence of P. In the context of normal programs the

situation is complicated by the fact that negations can occur in the bodies of clauses and therefore the truth of $\neg F$ can propagate further and may be used to infer positive formulas as well. This entails that a normal logic program does not necessarily have a unique minimal Herbrand model. To give a meaning to normal logic programs a multitude of semantics have been developed. We cannot delve into the details of these semantics and have to refer the interested reader to e.g. [9].

We will just present a few details of the completion semantics, which is the approach closest to first-order logic.

Definition 2.2.10 Given a program P the *definition* of a predicate p is the set of clauses in P whose head has p as its predicate symbol.

To make the link with first-order logic, Clark developed the concept of *completion* in [52]. The basic idea is to replace every definition of some predicate p by one *if-and-only-if* formula, called the *completed definition of* p. The resulting formulas are then combined with what is called *Clark's equality theory* or simply CET, which forces equality $=$ to be interpreted as the identity relation on the Herbrand universe $\mathcal{U}_P$. The resulting program is called the *completion* of the original program P and is denoted by $comp(P)$.

For instance, the completed definition of the predicate int from the program P of Example 2.2.4 is:

$$\forall X(int(X) \leftrightarrow X = 0 \vee \exists Z(X = s(Z) \wedge int(Z)))$$

CET for the same program will contain such formulas as:

$$\forall X \neg(s(X) = 0)$$
$$\forall X \forall Y(X = Y \rightarrow s(X) = s(Y))$$

The completion semantics has its shortcomings (e.g. the program P from Example 2.2.4 no longer captures *only* the natural numbers: although $comp(P) \models \neg int(a)$ whereas $P \not\models \neg int(a)$, $comp(P)$ also has models in which the interpretation of int is not isomorphic to the natural numbers) but has the advantage of being rather straightforward and can be seen as the theoretical basis for "negation as failure", which we will present in the next section.

2.3 Proof theory of logic programs

We start with some additional useful terminology related to substitutions. If $E\theta = F$ then E is said to be *more general* than F. If E is more general than F and F is more general than E then E and F are called *variants* (of each other). If $E\theta$ is a variant of E then θ is called a *renaming substitution for* E. Because a substitution is a *set* of bindings we will denote, in contrast

to e.g. [184], the *empty* or *identity substitution* by $\emptyset$ and not by the empty sequence ϵ. Substitutions can also be applied to sets of expressions by defining $\{E_1, \ldots, E_n\}\theta = \{E_1\theta, \ldots, E_n\theta\}$.

Substitutions can also be composed in the following way:

Definition 2.3.1 (composition of substitutions) Let $\theta = \{X_1/s_1, \ldots, X_n/s_n\}$ and $\sigma = \{Y_1/t_1, \ldots, Y_k/t_k\}$ be substitutions. Then the *composition* of θ and σ, denoted by $\theta\sigma$, is defined to be the substitution $\{X_i/s_i\sigma \mid 1 \leq i \leq n \wedge s_i\sigma \neq X_i\} \cup \{Y_i/t_i \mid 1 \leq i \leq k \wedge Y_i \notin \{X_1, \ldots, X_n\}\}$.[3]

When viewing substitutions as functions from expressions to expressions, then the above definition behaves just like ordinary function composition, i.e. $E(\theta\sigma) = (E\theta)\sigma$. We also have that (for proofs see [184]) the identity substitution acts as a left and right identity for composition, i.e. $\theta\emptyset = \emptyset\theta = \theta$, and that composition is associative, i.e. $(\theta\sigma)\gamma = \theta(\sigma\gamma)$.

We call a substitution θ *idempotent* iff $\theta\theta = \theta$. We also define the following notations: the set of variables occurring inside an expression E is denoted by $vars(E)$, the *domain* of a substitution θ is defined as $dom(\theta) = \{X \mid X/t \in \theta\}$ and the *range* of θ is defined as $ran(\theta) = \{Y \mid X/t \in \theta \wedge Y \in vars(t)\}$. Finally, we also define $vars(\theta) = dom(\theta) \cup ran(\theta)$ as well as the restriction $\theta|_{\mathcal{V}}$ of a substitution θ to a set of variables $\mathcal{V}$ by $\theta|_{\mathcal{V}} = \{X/t \mid X/t \in \theta \wedge X \in \mathcal{V}\}$.

The following concept will form the link between the model-theoretic semantics and the procedural semantics of logic programs.

Definition 2.3.2 (answer) Let P be a program and $G = \leftarrow L_1, \ldots, L_n$ a goal. An *answer* for $P \cup \{G\}$ is a substitution θ such that $dom(\theta) \subseteq vars(G)$.

2.3.1 Definite programs

We first define correct answers in the context of definite programs and goals.

Definition 2.3.3 (correct answer) Let P be a definite program and $G = \leftarrow A_1, \ldots, A_n$ a definite goal. An answer θ for $P \cup \{G\}$ is called a *correct answer* for $P \cup \{G\}$ iff $P \models \forall((A_1 \wedge \ldots \wedge A_n)\theta)$.

Take for instance the program $P = \{p(a) \leftarrow\}$ and the goal $G = \leftarrow p(X)$. Then $\{X/a\}$ is a correct answer for $P \cup \{G\}$ while $\{X/c\}$ and $\emptyset$ are not.

We now present a way to calculate correct answers based on the concepts of resolution and unification.

[3] This definition deviates slightly from the one in [7, 184, 218]. Indeed, taking the definition in [7, 184, 218] literally we would have that $\{X/a\}\{X/a\} = \emptyset$ and not the desired $\{X/a\}$ (the definition in [7, 184, 218] says to *delete* any binding Y_i/t_i with $Y_i \in \{X_1, \ldots, X_n\}$ from a *set* of bindings). Our definition does not share this problem.

Definition 2.3.4 (mgu) Let S be a finite set of expressions. A substitution θ is called a *unifier* of S iff the set $S\theta$ is a singleton. θ is called *relevant* iff its variables $vars(\theta)$ all occur in S. θ is called a *most general unifier* or *mgu* iff for each unifier σ of S there exists a substitution γ such that $\sigma = \theta\gamma$.

The concept of unification dates back to [119] and has been rediscovered in [240]. A survey on unification, also treating other application domains, can be found in [147].

If a unifier for a finite set S of expressions exists then there exists an idempotent and relevant most general unifier which is unique modulo variable renaming (see [7, 184]). Unifiability of a set of expressions is decidable and there are efficient algorithms for calculating an idempotent and relevant *mgu*. See for instance the unification algorithms in [7, 184] or the more complicated but linear ones in [193, 221]. From now on we denote, for a unifiable set S of expressions, by $mgu(S)$ an idempotent and relevant unifier of S. If we just want to unify two terms t_1, t_2 then we will also sometimes write $mgu(t_1, t_2)$ instead of $mgu(\{t_1, t_2\})$.

We define the *most general instance*, of a finite set S to be the only element of $S\theta$ where $\theta = mgu(S)$. The opposite of the most general instance is the *most specific generalisation* of a finite set of expressions S, also denoted by $msg(S)$, which is the most specific expression M such that all expressions in S are instances of M. Algorithms for calculating the *msg* exist [160], and this process is also referred to as *anti-unification* or *least general generalisation*.

We can now define *SLD-resolution*, which is based on the resolution principle [240] and which is a special case of SL-resolution [154]. Its use for a programming language was first described in [153] and the name SLD (which stands for Selection rule-driven Linear resolution for Definite clauses), was coined in [12]. For more details about the history see e.g. [7, 184].

Definition 2.3.5 (SLD-derivation step) Let $G = \leftarrow L_1, \ldots, L_m, \ldots, L_k$ be a goal and $C = A \leftarrow B_1, \ldots, B_n$ a program clause such that $k \geq 1$ and $n \geq 0$. Then G' *is derived from* G *and* C *using* θ *(and* L_m*)* iff the following conditions hold:

1. L_m is an atom, called the *selected* atom (at position m), in G.
2. θ is a relevant and idempotent *mgu* of L_m and A.
3. G' is the goal $\leftarrow (L_1, \ldots, L_{m-1}, B_1, \ldots, B_n, L_{m+1}, \ldots, L_k)\theta$.

G' is also called a *resolvent* of G and C.

In the following we define the concept of a complete SLD-derivation (we will define incomplete ones later on).

Definition 2.3.6 (complete SLD-derivation) Let P be a normal program and G a normal goal. A *complete SLD$^+$-derivation* of $P \cup \{G\}$ is a tuple $(\mathcal{G}, \mathcal{L}, \mathcal{C}, \mathcal{S})$ consisting of a sequence of goals $\mathcal{G} = \langle G_0, G_1, \ldots \rangle$, a sequence $\mathcal{L} = \langle L_0, L_1 \ldots \rangle$ of selected literals,[4] a sequence $\mathcal{C} = \langle C_1, C_2, \ldots \rangle$ of variants of program clauses of P and a sequence $\mathcal{S} = \langle \theta_1, \theta_2, \ldots \rangle$ of *mgu*'s such that:

- for $i > 0$, $vars(C_i) \cap vars(G_0) = \emptyset$;
- for $i > j$, $vars(C_i) \cap vars(C_j) = \emptyset$;
- for $i \geq 0$, L_i is a positive literal in G_i and G_{i+1} is derived from G_i and C_{i+1} using θ_{i+1} and L_i;
- the sequences $\mathcal{G}, \mathcal{C}, \mathcal{S}$ are maximal given $\mathcal{L}$.

A *complete SLD-derivation* is just a complete SLD$^+$ derivation of a definite program and goal.

The process of producing variants of program clauses of P which do not share any variable with the derivation sequence so far is called *standardising apart*. Some care has to be taken to avoid variable clashes and the ensuing technical problems; see the discussions in [149] or [84].

We now come back to the idea of a proof by refutation and its relation to SLD-resolution. In a proof by refutation one adds the negation of what is to be proven and then tries to arrive at inconsistency. The former corresponds to adding a goal $G = \leftarrow A_1, \ldots, A_n$ to a program P and the latter corresponds to searching for an SLD-derivation of $P \cup \{G\}$ which leads to $\square$. This justifies the following definition.

Definition 2.3.7 (SLD-refutation) An *SLD-refutation* of $P \cup \{G\}$ is a finite complete SLD-derivation of $P \cup \{G\}$ which has the empty clause $\square$ as the last goal of the derivation.

In addition to refutations there are (only) two other kinds of complete derivations:

- Finite derivations which do not have the empty clause as the last goal. These derivations will be called *(finitely) failed*.
- Infinite derivations. These will be called *infinitely failed*.

We can now define computed answers, which correspond to the output calculated by a logic program.

[4] Again we slightly deviate from [7, 184]: the inclusion of $\mathcal{L}$ avoids some minor technical problems wrt the maximality condition.

Definition 2.3.8 (computed answer) Let P be a definite program, G a definite goal and D a SLD-refutation for $P \cup \{G\}$ with the sequence $\langle \theta_1, \ldots, \theta_n \rangle$ of *mgu*'s. The substitution $(\theta_1 \ldots \theta_n)|_{vars(G)}$ is then called a *computed answer* for $P \cup \{G\}$ (via D).

If θ is a computed (respectively correct) answer for $P \cup \{G\}$ then $G\theta$ is called a *computed* (respectively *correct*) *instance* for $P \cup \{G\}$.

Theorem 2.3.9 (soundness of SLD) Let P be a definite program and G a definite goal. Every computed answer for $P \cup \{G\}$ is a correct answer for $P \cup \{G\}$.

Theorem 2.3.10 (completeness of SLD) Let P be a definite program and G a definite goal. For every correct answer σ for $P \cup \{G\}$ there exists a computed answer θ for $P \cup \{G\}$ and a substitution γ such that $G\sigma = G\theta\gamma$.

A proof of the previous theorem can be found in [7].[5]
We will now examine systematic ways to search for SLD-refutations.

Definition 2.3.11 (complete SLD-tree) A *complete SLD-tree* for $P \cup \{G\}$ is a labelled tree satisfying the following:

1. Each node of the tree is labelled with a definite goal along with an indication of the selected atom
2. The root node is labelled with G.
3. Let $\leftarrow A_1, \ldots, A_m, \ldots, A_k$ be the label of a node in the tree and suppose that A_m is the selected atom. Then for each clause $A \leftarrow B_1, \ldots, B_q$ in P such that A_m and A are unifiable the node has one child labelled with

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

   where θ is an idempotent and relevant *mgu* of A_m and A.
4. Nodes labelled with the empty goal have no children.

[5] The corresponding theorem in [184] states that $\sigma = \theta\gamma$, which is known to be false. Indeed, take for example the program $P = \{p(f(X,Y)) \leftarrow\}$ and the goal $G = \leftarrow p(Z)$. Then $\{Z/f(a,a)\}$ is a correct answer (because $p(f(a,a))$ is a consequence of P), but

- there is no computed answer $\{X/f(a,a)\}$
- for any computed answer $\{Z/f(X',Y')\}$ (where either X' or Y' must be different from Z; these are the only computed answers) composing it with $\{X'/a, Y'/a\}$ will give $\{Z/f(a,a), X'/a, Y'/a\}$ (or $\{Z/f(a,a), Y'/a\}$ if $X' = Z$ or $\{Z/f(a,a), X'/a\}$ if $Y' = Z$) which is different from $\{X/f(a)\}$.

To every branch of a complete SLD-tree corresponds a complete SLD-derivation. The choice of the selected atom is performed by what is called a *selection rule*. Maybe the most well known selection rule is the *left-to-right selection rule* of *Prolog* [80, 262, 53], which always selects the leftmost literal in a goal. The complete SLD-derivations and SLD-trees constructed via this selection rule are called *LD-derivations* and *LD-trees*.

Usually one confounds goals and nodes (e.g. in [7, 184, 218]) although this is strictly speaking not correct because the same goal can occur several times inside the same SLD-tree.

We will often use a graphical representation of SLD-trees in which the selected atoms are identified by underlining. For instance, Figure 2.1 contains a graphical representation of a complete SLD-tree for $P \cup \{\leftarrow int(s(0))\}$, where P is the program of Example 2.2.4.

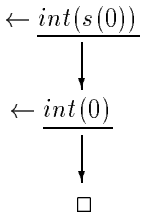


Figure 2.1: Complete SLD-tree for Example 2.2.4

2.3.2 Normal programs

We now define correct answers in the context of normal goals and the completion of normal programs.

Definition 2.3.12 (correct answer) Let P be a definite program and $G = \leftarrow A_1, \ldots, A_n$ a definite goal. A substitution θ is called a *correct answer* for $comp(P) \cup \{G\}$ iff $comp(P) \models \forall((A_1 \wedge \ldots \wedge A_n)\theta)$.

In [184] it is shown that this concept generalises the earlier concept of a correct answer in the definite case.

Finding an efficient proof procedure for normal programs is much less obvious than in the definite case. The most commonly used procedure is the so called *SLDNF-procedure*. It is an extension of SLD-resolution which also allows the selection of ground negative literals. Basically a selected ground negative literal $\neg A$ succeeds (without computed answer) if $\leftarrow A$ fails *finitely*. Similarly a selected ground negative literal fails if there exists

a refutation for $\leftarrow A$. This implements what is called the "negation as failure" (NAF) rule, a less powerful but more tractable inference mechanism than the CWA.

To define SLDNF-derivations we use the approach presented in [184] based on ranks, where the rank indicates the maximal nesting of sub-derivations and sub-trees created by negative calls. Note that the definition of [184] exhibits some technical problems, in the sense that some problematic goals do not have an associated SLDNF-derivation (failed or otherwise, see [194, 10, 9]). The definition is however sufficient for our purposes, especially since most correctness results for partial deduction (e.g. [184]), to be introduced in the next chapter, use this definition anyway.

Note that in Definition 2.3.6 we already introduced SLD^+-derivations for normal programs and goals which do not allow the selection of negative literals. We also call an SLD^+-refutation an *SLDNF-refutation of rank 0.*

The notions of SLDNF-refutations of rank k and finitely failed SLDNF-trees of rank k, as well as the notions of SLDNF-derivation and SLDNF-tree, depend on each other. We start by defining SLDNF-trees.

Definition 2.3.13 (SLDNF-tree) Let P be a normal program and G a normal goal. A *pseudo SLDNF-tree* for $P \cup \{G\}$ is a labelled tree satisfying the following:

1. Each node of the tree is labelled with a goal along with an indication of the selected atom
2. The root node is labelled with G.
3. Let $\leftarrow L_1, \ldots, L_m, \ldots, L_k$ be the label of a node in the tree and suppose that L_m is the selected literal which is an atom. Then for each clause $A \leftarrow B_1, \ldots, B_q$ in P such that L_m and A are unifiable the node has one child labelled with

   $$\leftarrow (L_1, \ldots, L_{m-1}, B_1, \ldots, B_q, L_{m+1}, \ldots, L_k)\theta,$$

   where θ is an idempotent and relevant *mgu* of L_m and A.
4. Let $\leftarrow L_1, \ldots, L_m, \ldots, L_k$ be the label of a node in the tree and suppose that $L_m = \neg A_m$ is the selected literal. Then A_m is *ground* and the node is *either* a leaf *or* has a child labelled with

   $$\leftarrow L_1, \ldots, L_{m-1}, L_{m+1}, \ldots, L_k.$$
5. Nodes labelled with the empty goal have no children.

A finitely *failed SLDNF-tree of rank 0* for $P \cup \{G\}$ is a finite pseudo SLDNF-tree which satisfies:

1. Only positive literals are selected.

2. Each leaf is labelled with a non-empty goal $\leftarrow L_1, \ldots, L_m, \ldots, L_k$ in which the selected literal L_m is an atom (which thus unifies with no clause head).

A *finitely failed SLDNF-tree of rank* $k+1$ for $P \cup \{G\}$ is a finite pseudo SLDNF-tree for $P \cup \{G\}$ which satisfies:

1. For each non-leaf node labelled with $\leftarrow L_1, \ldots, L_m, \ldots, L_{k'}$ in which the selected literal $L_m = \neg A_m$ is negative there is a finitely failed SLNDF-tree of rank k for $P \cup \{\leftarrow A_m\}$.

2. Each leaf is labelled with a non-empty goal $\leftarrow L_1, \ldots, L_m, \ldots, L_{k'}$ in which the selected literal L_m is either
   - an atom (which thus unifies with no clause head) or
   - a negative literal $L_m = \neg A_m$ and there is an SLDNF-refutation of rank k of $P \cup \{\leftarrow A_m\}$.

A *finitely failed SLDNF-tree* is a finitely failed SLDNF-tree of rank k for some k.
A *complete SLDNF-tree* for $P \cup \{G\}$ is a finite pseudo SLDNF-tree for $P \cup \{G\}$ which satisfies:

1. For each non-leaf node labelled with $\leftarrow L_1, \ldots, L_m, \ldots, L_k$ in which the selected literal $L_m = \neg A_m$ is negative there is a finitely failed SLNDF-tree for $P \cup \{\leftarrow A_m\}$.

2. If a leaf is labelled with a non-empty goal $\leftarrow L_1, \ldots, L_m, \ldots, L_k$ in which the selected literal $L_m = \neg A_m$ is negative then there is an SLDNF-refutation of $P \cup \{\leftarrow A_m\}$.

Definition 2.3.14 (SLDNF-derivation) Let P be a normal program and G a normal goal. A *pseudo SLDNF-derivation* of $P \cup \{G\}$ is a tuple $(\mathcal{G}, \mathcal{L}, \mathcal{C}, \mathcal{S})$ consisting of a sequence of goals $\mathcal{G} = \langle G_0, G_1, \ldots \rangle$, a sequence $\mathcal{L} = \langle L_0, L_1, \ldots \rangle$ of selected literals, a sequence $\mathcal{C} = \langle C_1, C_2, \ldots \rangle$ of variants of program clauses of P or ground negative literals and a sequence $\mathcal{S} = \langle \theta_1, \theta_2, \ldots \rangle$ of substitutions such that:
- for $i > 0$, $vars(C_i) \cap vars(G_0) = \emptyset$;
- for $i > j$, $vars(C_i) \cap vars(C_j) = \emptyset$;
- for $i \geq 0$, L_i is a literal in G_i and either
  1. G_{i+1} is derived from G_i and C_{i+1} using θ_{i+1} and L_i or
  2. $G_i = \leftarrow L'_1, \ldots, L'_m, \ldots, L'_k$ and the selected literal $L_i = L'_m = \neg A_m$ is ground. In this case, either G_i is the last goal or $\theta_{i+1} = \emptyset$ (the identity substitution), $C_{i+1} = \neg A_m$ and

$$G_{i+1} = \leftarrow L'_1, \ldots, L'_{m-1}, L'_{m+1}, \ldots, L'_k.$$

- the sequences $\mathcal{G}, \mathcal{C}, \mathcal{S}$ are maximal given $\mathcal{L}$.

An *SLDNF-refutation of rank $k+1$* of $P \cup \{G\}$ is a finite pseudo SLDNF-derivation of $P \cup \{G\}$ ending with the empty goal $\square$ and such that for every selected ground negative literal $L_m = \neg A_m$ there exists a finitely failed SLNDF-tree of rank k for $P \cup \{\leftarrow A_m\}$.
An *SLDNF-refutation* is simply a SLDNF-refutation of rank k for some k and a *complete SLDNF-derivation* is a pseudo SLDNF-derivation such that for every selected ground negative literal $L_m = \neg A_m$ in some G_i either

- G_i is the last goal and there exists an SLDNF-refutation of $P \cup \{\leftarrow A_m\}$ or
- G_i is not the last goal and there exists a finitely failed SLNDF-tree for $P \cup \{\leftarrow A_m\}$.

The following theorem establishes soundness of SLDNF and is due to Clark [52].

Theorem 2.3.15 (Soundness of Negation as Failure and SLDNF) Let P be a normal program and $\leftarrow Q$ a normal goal.

- If $P \cup \{\leftarrow Q\}$ has a finitely failed SLDNF-tree then $comp(P) \models \neg Q$.
- Every computed answer for $P \cup \{\leftarrow Q\}$ is a correct answer for $comp(P) \cup \{\leftarrow Q\}$.

Unfortunately SLDNF-resolution is in general not complete, even wrt the completion semantics, mainly (but not only) due to *floundering*, i.e. computation reaches a state in which only non-ground negative literals exist. See also [253] for some limitations of SLDNF. Some completeness results for some specific settings have been developed in e.g. [45, 44, 265, 87].

To remedy the incompleteness of SLDNF, several extensions have been proposed. Let us briefly mention some of them.

First, a straightforward extension is described in [184] (on page 94; already mentioned in [52] and called SLDNFE in [9]). The idea is to allow the (tentative) selection of non-ground negative literals $\neg A$: if a refutation of $P \cup \{\leftarrow A\}$ with the *empty* computed answer substitution can be found then we declare failure of $\neg A$ and if a finitely failed tree can be constructed for $P \cup \{\leftarrow A\}$ we declare success. In the other cases $\neg A$ cannot be selected.

Another extension is the so called SLS procedure [236]. However, its purpose is mainly theoretical, as it requires the detection of infinitely failed branches (and treats them like the finitely failed ones in SLDNF).

The approach of *constructive negation* overcomes some of the incompleteness problems of SLDNF [48, 49, 86, 252, 267, 266] and can be useful inside partial deduction [115]. The main idea is to allow the selection of

non-ground negative literals, replacing them by disequality constraints. For instance, given $P = \{p(a) \leftarrow\}$ the negative literal $\neg p(X)$ could be replaced by $\neg(X = a)$. Another related approach is presented in [105].

2.3.3 Programs with built-ins

Most practical logic programs make (heavy) usage of built-ins. Although a lot of these built-ins, like e.g. `assert/1` and `retract/1`, are extra-logical and ruin the declarative nature of the underlying program, a reasonable number of them can actually be seen as syntactic sugar. Take for example the following program which uses the Prolog [80, 262, 53] built-ins $= ../2$ and $call/1$.

$map(P, [], []) \leftarrow$
$map(P, [X|T], [P_X|P_T]) \leftarrow C = ..[P, X, P_X], call(C), map(P, T, P_T)$
$inv(0, 1) \leftarrow$
$inv(1, 0) \leftarrow$

For this program the query $\leftarrow map(inv, [0, 1, 0], R)$ will succeed with the computed answer $\{R/[1, 0, 1]\}$. Given that query, the Prolog program can be seen as a pure definite logic program by simply adding the following definitions (where we use the prefix notation for the predicate $= ../2$):

$= ..(inv(X, Y), [inv, X, Y]) \leftarrow$
$call(inv(X, Y)) \leftarrow inv(X, Y)$

The so obtained pure logic program will succeed for $\leftarrow map(inv, [0, 1, 0], R)$ with the same computed answer $\{R/[1, 0, 1]\}$.

This means that some predicates like $map/3$, which are usually taken to be higher-order, can simply be mapped to pure definite (first-order) logic programs ([283, 212]). Some built-ins, like for instance $is/2$, have to be defined by infinite relations. Usually this poses no problems as long as, when selecting such a built-in, only a finite number of cases apply (Prolog will report a run-time error if more than one case applies while the programming language Gödel [123] will delay the selection until only one case applies).

In the remainder of this thesis we will usually restrict our attention to those built-ins that can be given a logical meaning by such a mapping.

Chapter 3

Partial Evaluation and Partial Deduction

3.1 Partial evaluation

In contrast to ordinary (full) evaluation, a *partial evaluator* is given a program P along with only *part* of its input, called the *static input*. The remaining part of the input, called the *dynamic input*, will only be known at some later point in time. Given the static input S, the partial evaluator then produces a *specialised* version P_S of P which, when given the dynamic input D, produces the same output as the original program P. This process is illustrated in Figure 3.1. The program P_S is also called the *residual program*.

The theoretical feasibility of this process, in the context of recursive functions, has already been established by Kleene [146] and is known as Kleene's S-M-N theorem. However, while Kleene was concerned with theoretical issues of computability and his construction yields specialised programs which are slower than the original, the goal of partial evaluation is to exploit the static input in order to derive more efficient programs.

To obtain the specialised program P_S, a partial evaluator performs a mixture of evaluation, i.e. it executes those parts of P which only depend on the static input S, and of code generation for those parts of P which require the dynamic input D. This process has therefore also been called *mixed computation* in [90]. Also, it is precisely this approach which distinguishes partial evaluation from other program specialisation approaches.

Because part of the computation has already been performed beforehand by the partial evaluator, the hope that we obtain a more efficient program

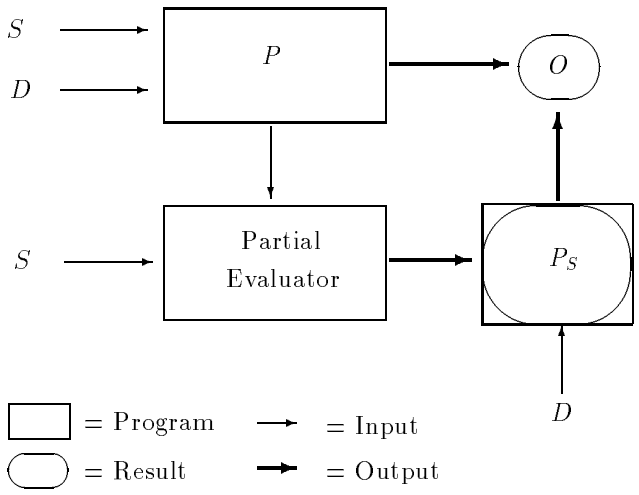


Figure 3.1: Partial evaluation of programs with static and dynamic input

P_S seems justified. The simple example in Figure 3.2 illustrates this point: the control of the loop in P is fully determined by the static input $e = 3$ and was executed beforehand by the partial evaluator, resulting in a more efficient specialised program P_e.

Partial evaluation has been applied to a lot of programming languages and paradigms: functional programming (e.g. [138]), logic programming (e.g. [98, 152, 222]), functional logic programming (e.g. [2]) term rewrite systems (e.g. [22, 23], [204]) and imperative programming (e.g. [5, 3]). A general introduction to partial evaluation can also be found in [136]. An important concern in partial evaluation has also been the issue of *self-application*, i.e. to try to write partial evaluators which are able to specialise themselves. We will return to this issue in Chapter 7.

In the context of logic programming, full input to a program P consists of a goal G and evaluation corresponds to constructing a complete SLDNF-tree for $P \cup \{G\}$. For partial evaluation, the static input then takes the form of a *partially instantiated* goal G'. In contrast to other programming languages and paradigms, one can still execute P for G' and (try to) construct a SLDNF-tree for $P \cup \{G'\}$. So, at first sight, it seems that partial evaluation for logic programs is almost trivial and just corresponds to ordinary evaluation.

However, because G' is not yet fully instantiated, the SLDNF-tree for $P \cup \{G'\}$ is usually infinite and ordinary evaluation will not terminate. A

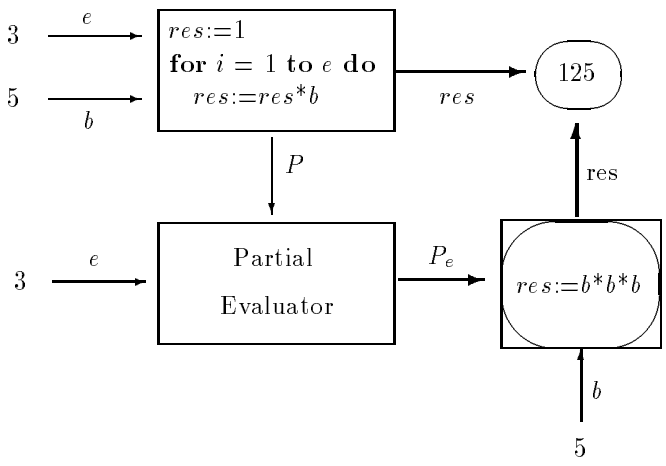


Figure 3.2: Partial evaluation of a simple imperative program

more refined approach to partial evaluation of logic programs is therefore required. A technique which solves this problem is known under the name of *partial deduction*. Its general idea is to construct a finite number of finite trees which "cover" the possibly infinite SLDNF-tree for $P \cup \{G'\}$. We will present the essentials of this technique in the next section.

The term "partial deduction" has been introduced by Komorowski (see [152]) to replace the term of partial evaluation in the context of pure logic programs. We will adhere to this terminology because the word "deduction" places emphasis on the purely logical nature of the source programs. Also, while partial evaluation of e.g. functional programs evaluates only those expressions which depend exclusively on the static input, in logic programming one can, as we have seen above, in principle also evaluate expressions which depend on the unknown dynamic input. This puts partial deduction much closer to techniques such as *supercompilation* [273, 274, 258, 114] and *unfold/fold* program transformations [43, 222], and therefore using a different denomination seems justified. We will return to the relation of partial deduction to these and other techniques in Chapters 6 and 10 (see also [113, 135, 259]). Finally, note that program specialisation in general is not limited to just evaluating expressions, whether they depend on the static input or not. A striking illustration of this statement features in Chapter 13.

3.2 Partial deduction

In this section we present the technique of partial deduction, which originates from [150, 151]. Other introductions to partial deduction can be found in [152, 98, 63].

In order to avoid constructing infinite SLDNF-trees for partially instantiated goals, the technique of *partial deduction* is based on constructing finite, but possibly *incomplete* SLDNF-trees. The derivation steps in these SLDNF-trees correspond to the computation steps which have already been performed by the *partial deducer* and the clauses of the specialised program are then extracted from these trees by constructing one specialised clause per branch.

In this section we will formalise this technique and present conditions which will ensure correctness of the so obtained specialised programs.

Definition 3.2.1 (SLDNF-derivation) A SLDNF-derivation is defined like a complete SLDNF-derivation but may, in addition to leading to success or failure, also lead to a last goal where no literal has been selected for a further derivation step. Derivations of the latter kind will be called *incomplete*.

An SLDNF-derivation can thus be either failed, incomplete, successful or infinite. Now, an incomplete SLDNF-tree is obtained in much in the same way.

Definition 3.2.2 An SLDNF-tree is defined like a complete SLDNF-tree but may, in addition to success and failure leaves, also contain leaves where no literal has been selected for a further derivation step. Leaves of the latter kind are called *dangling* ([199]) and SLDNF-trees containing dangling leaves are called *incomplete*. Also, an SLDNF-tree is called *trivial* iff its root is a dangling leaf, and *non-trivial* otherwise.

The process of selecting a literal inside a dangling leaf of an incomplete SLDNF-tree and adding all the resolvents as children is called *unfolding*. An SLDNF-tree for $P \cup \{G\}$ can thus be obtained from a trivial SLDNF-tree for $P \cup \{G\}$ by performing a sequence of unfolding steps. We will return to this issue in Section 3.3.3.

Note that every branch of an SLDNF-tree has an associated (possibly incomplete) SLDNF-derivation. We also extend the notion of a *computed answer substitution (c.a.s.)* to finite incomplete SLDNF-derivations (it is just the composition of the *mgu*'s restricted to the variables of the top-level goal). Also, a *resolvent* of a finite (possibly incomplete) SLDNF-derivation

is just the last goal of the derivation. Finally, if $\langle G_0, \ldots, G_n \rangle$ is the sequence of goals of a finite SLDNF-derivation, we say D has *length* n.

We will now examine how specialised clauses can be extracted from SLDNF-derivations and trees. The following definition associates a first-order formula with a finite SLDNF-derivation.

Definition 3.2.3 Let P be a program, $\leftarrow Q$ a goal and D a finite SLDNF-derivation of $P \cup \{\leftarrow Q\}$ with computed answer θ and resolvent $\leftarrow B$. Then the formula $Q\theta \leftarrow B$ is called the *resultant* of D.

This concept can be extended to SLDNF-trees in the following way:

Definition 3.2.4 Let P be a program, G a goal and let τ be a finite SLDNF-tree for $P \cup \{G\}$. Let $D_1, \ldots, D_n$ be the non-failing SLDNF-derivations associated with the branches of τ. Then the set of resultants $resultants(\tau)$ is the union of the resultants of the non-failing SLDNF-derivations $D_1, \ldots, D_n$ associated with the branches of τ. We also define the set of leaves, $leaves(\tau)$, to be the atoms occurring in the resolvents of $D_1, \ldots, D_n$.

Example 3.2.5 Let P be the following program:

$member(X, [X \mid T]) \leftarrow$
$member(X, [Y \mid T]) \leftarrow member(X, T)$
$inboth(X, L1, L2) \leftarrow member(X, L1), member(X, L2)$

The tree in Figure 3.3 represents a finite incomplete SLD-tree τ for $P \cup \{\leftarrow inboth(X, [a], L)\}$. This tree has just one non-failing branch and the set of resultants $resultants(\tau)$ contains the single clause:

$$inboth(a, [a], L) \leftarrow member(a, L)$$

Note that the complete SLD-tree for $P \cup \{\leftarrow inboth(X, [a], L)\}$ is infinite.

If the goal in the root of a finite SLDNF-tree is atomic then the resultants associated with the tree are all clauses. We can thus formalise partial deduction in the following way.

Definition 3.2.6 (partial deduction) Let P be a normal program and A an atom. Let τ be a finite non-trivial SLDNF-tree for $P \cup \{\leftarrow A\}$. Then the set of clauses $resultants(\tau)$ is called a *partial deduction of* A *in* P.
If $\mathcal{A}$ is a finite set of atoms, then a *partial deduction of* $\mathcal{A}$ *in* P is the union of one partial deduction for each element of $\mathcal{A}$.
A *partial deduction of* P *wrt* $\mathcal{A}$ is a normal program obtained from P by replacing the set of clauses in P, whose head contains one of the predicate symbols appearing in $\mathcal{A}$ (called the partially deduced predicates), with a partial deduction of $\mathcal{A}$ in P.

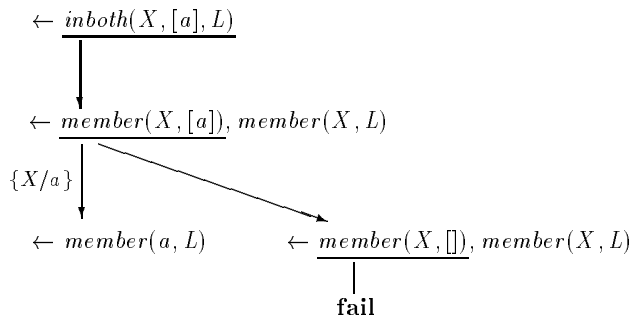


Figure 3.3: Incomplete SLD-tree for Example 3.2.5

Example 3.2.7 Let us return to the program P of Example 3.2.5. Based on the incomplete SLDNF-tree in Figure 3.3, we can construct the following partial deduction of P wrt $\mathcal{A} = \{inboth(X, [a], L)\}$:

$$member(X, [X \mid T]) \leftarrow$$
$$member(X, [Y \mid T]) \leftarrow member(X, T)$$
$$inboth(a, [a], L) \leftarrow member(a, L)$$

Note that if τ is a trivial SLDNF-tree for $P \cup \{\leftarrow A\}$ then $resultants(\tau)$ consists of the problematic clause $A \leftarrow A$ and the specialised program contains a loop. That is why trivial trees are not allowed in Definition 3.2.6. This is however not a sufficient condition for correctness of the specialised programs. In [185], Lloyd and Shepherdson presented and proved a fundamental correctness theorem for partial deduction. The two (additional) basic requirements for correctness of a partial deduction of P wrt $\mathcal{A}$ are the *independence* and *closedness* conditions. The independence condition guarantees that the specialised program does not produce additional answers and the closedness condition guarantees that all calls, which might occur during the execution of the specialised program, are covered by some definition. Below we summarise the correctness result of [185].

Definition 3.2.8 (closedness, independence) Let S be a set of first order formulas and $\mathcal{A}$ a finite set of atoms. Then S is $\mathcal{A}$-*closed* iff each atom in S, containing a predicate symbol occurring in an atom in $\mathcal{A}$, is an instance of an atom in $\mathcal{A}$. Furthermore we say that $\mathcal{A}$ is *independent* iff no pair of atoms in $\mathcal{A}$ have a common instance.

Note that two atoms which cannot be unified may still have a common instance (i.e. unify after renaming apart). For example, $p(X)$ and $p(f(X))$ are not unifiable but have e.g. the common instance $p(f(X))$.

Theorem 3.2.9 (correctness of partial deduction [185]) Let P be a normal program, G a normal goal, $\mathcal{A}$ a finite, independent set of atoms, and P' a partial deduction of P wrt $\mathcal{A}$ such that $P' \cup \{G\}$ is $\mathcal{A}$-closed. Then the following hold:

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

For instance, the partial deduction of P wrt $\mathcal{A} = \{inboth(X, [a], L)\}$ in Example 3.2.7 satisfies the conditions of Theorem 3.2.9 for the goals $\leftarrow inboth(X, [a], [b, a])$ and $\leftarrow inboth(X, [a], L)$ but not for the goal $\leftarrow inboth(X, [b], [b, a])$.

Note that the original unspecialised program P is also a partial deduction wrt $\mathcal{A} = \{member(X, L), inboth(X, L1, L2)\}$ which furthermore satisfies the correctness conditions of Theorem 3.2.9 for any goal G. In other words, neither Definition 3.2.6 nor the conditions of Theorem 3.2.9 ensure that any specialisation has actually been performed. Nor do they give any indication on how to construct a suitable set $\mathcal{A}$ and a suitable partial deduction wrt $\mathcal{A}$ satisfying the correctness criteria for a given goal G of interest. These are all considerations generally delegated to the *control* of partial deduction, which we discuss in the next section.

[18] also proposes an extension of Theorem 3.2.9 which uses a notion of coveredness instead of closedness. The basic idea is to restrict the attention to those parts of the specialised program P' which can be reached from G. The formalisation is as follows:

Definition 3.2.10 Let P be a set of clauses. The *predicate dependency graph* of P is a directed graph

- whose nodes are the predicate symbols in the alphabet $\mathcal{A}_P$ and
- which contains an arc from p to q iff there exists a clause in P in which p occurs as a predicate symbol in the head and q as a predicate symbol in the body.

Definition 3.2.11 Let P be a program and G a goal. We say that G *depends* upon a predicate p in $\mathcal{A}_P$ iff there exists a path from a predicate symbol occurring in G to p in the predicate dependency graph of P.
We denote by $P \downarrow_G$ the definitions in P of those predicates in $\mathcal{A}_P$ upon which G depends.

Let $\mathcal{A}$ be a finite set of atoms. We say that $P \cup \{G\}$ is $\mathcal{A}$-*covered* iff $P \downarrow_G \cup \{G\}$ is $\mathcal{A}$-closed.

By replacing the condition in Theorem 3.2.9 that "$P' \cup \{G\}$ is $\mathcal{A}$-closed" by the more general "$P' \cup \{G\}$ is $\mathcal{A}$-covered", we still have a valid theorem (see [18]).

Example 3.2.12 Let us again return to the program P of Example 3.2.5. By building a complete SLD-tree for $P \cup \{\leftarrow member(X, [a])\}$, we get the following partial deduction P' of P wrt $\mathcal{A} = \{member(X, [a])\}$:

$$member(a, [a]) \leftarrow$$
$$inboth(X, L1, L2) \leftarrow member(X, L1), member(X, L2)$$

Unfortunately, Theorem 3.2.9 cannot be applied for $G = \leftarrow member(X, [a])$ because $P' \cup \{G\}$ is not $\mathcal{A}$-closed (due to the body of the second clause of P'). However, $P' \cup \{G\}$ is $\mathcal{A}$-covered, because $P'\downarrow_G$ just consists of the first clause of P'. Therefore correctness of P' wrt G can be established by the above extension of Theorem 3.2.9.

3.3 Control of partial deduction

In partial deduction one usually distinguishes two levels of control [98, 201]:

- the *global control*, in which one chooses the set $\mathcal{A}$, i.e. one decides *which* atoms will be partially deduced, and
- the *local control*, in which one constructs the finite (possibly incomplete) SLDNF-trees for each individual atom in $\mathcal{A}$ and thus determines *what* the definitions for the partially deduced atoms look like.

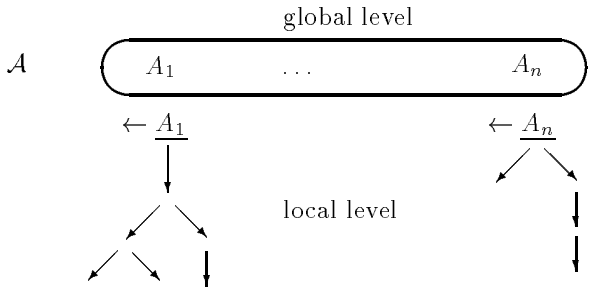


Figure 3.4: Global and local level of control

Below we examine how these two levels of control interact.

3.3.1 Correctness, termination and precision

When controlling partial deduction the three following, often conflicting, aspects have to be reconciled:

1. *Correctness*, i.e. ensuring that Theorem 3.2.9 or its extension can be applied. This can be divided into a local condition, requiring the construction of non-trivial trees, and into a global one related to the independence and coveredness (or closedness) conditions.

2. *Termination*. This aspect can also be divided into a local and a global one. First, the problem of keeping each SLDNF-tree finite is referred to as the *local* termination problem. Secondly keeping the set $\mathcal{A}$ finite is referred to as the *global* termination problem.

3. *Precision*. For precision of the specialisation we can again discern two aspects. One which we might call *local* precision and which is related to the unfolding rule and to the fact that (potential for) specialisation can be lost if we stop unfolding an atom in $\mathcal{A}$ prematurely. Indeed, when we stop the unfolding process at a given goal Q, then all the atoms in Q are treated separately (partial deductions are defined for sets of *atoms* and not for sets of *goals*; see however Chapters 10–13). For instance, if we stop the unfolding process in Example 3.2.5 for $G = \leftarrow inboth(X, [a, b, c], [c, d, e])$ at the goal $G' = \leftarrow member(X, [a, b, c]), member(X, [c, d, e])$, partial deduction will not be able to infer that the only possible answer for G' and G is $\{X/c\}$.

   The second aspect could be called the *global* precision and is related to the structure of $\mathcal{A}$. In general having a more precise and fine grained set $\mathcal{A}$ (with more *instantiated* atoms) will lead to better specialisation. For instance, given the set $\mathcal{A} = \{member(a, [a, b]), member(c, [d])\}$, partial deduction can perform much more specialisation (i.e. detecting that the goal $\leftarrow member(a, [a, b])$ always succeeds exactly once and that $\leftarrow member(c, [d])$ fails) than given the less instantiated set $\mathcal{A}' = \{member(X, [Y \mid T])\}$.

A good partial deduction algorithm will ensure correctness and termination while minimising the precision loss of point 3. Let us now examine more closely how those three conflicting aspects can be reconciled.

3.3.2 Independence and renaming

On the side of correctness there are two ways to ensure the independence condition. One is to apply a generalisation operator like the *msg* on all the

atoms which are not independent (first proposed in [18]). Applying this technique e.g. on the dependent set $\mathcal{A} = \{member(a, L), member(X, [b])\}$ yields the independent set $\{member(X, L)\}$. This approach also alleviates to some extent the global termination problem. However, it also diminishes the global precision and, as can be guessed from the above example, can seriously diminish the potential for specialisation.

This loss of precision can be completely avoided by using a *renaming* transformation to ensure independence. Renaming will map dependent atoms to new predicate symbols and thus generate an independent set without precision loss. For instance, the dependent set $\mathcal{A}$ above can be transformed into the independent set $\mathcal{A}' = \{member(a, L), member'(X, [b])\}$. The renaming transformation then has to map the atoms inside the residual program P' and the partial deduction goal G to the correct versions of $\mathcal{A}'$ (e.g. it has to rename the goal $G = \leftarrow member(a, [a, c]), member(b, [b])$ into $\leftarrow member(a, [a, c]), member'(b, [b])$). Renaming can often be combined with argument filtering to improve the efficiency of the specialised program. The basic idea is to filter out constants and functors and only keep the variables as arguments. For instance, instead of renaming $\mathcal{A}$ into $\mathcal{A}'$, $\mathcal{A}$ can be directly renamed into $\{mem_a(L), mem_b(X)\}$ and G into $\leftarrow mem_a([a, c]), mem_b(b)$. Further details about filtering can be found in e.g. [100] or [17]. See also [232], where filtering can be obtained automatically when using folding. Filtering has also been referred to as "pushing down meta-arguments" in [261] or "PDMA" in [220]. In functional programming the term of "arity raising" has also been used.

Renaming and filtering are used in a lot of practical approaches (e.g. [97, 98, 100, 173, 167, 168]) and adapted correctness results can be found in [17]. We will return to filtering in Section 5.1 of Chapter 5 and will prove some correctness results in Section 5.2.

3.3.3 Local termination and unfolding rules

The local control component is usually encapsulated in what is called an unfolding rule, defined as follows.

Definition 3.3.1 (unfolding rule) An *unfolding rule* U is a function which, given a program P and a goal G, returns a finite and possibly incomplete SLDNF-tree for $P \cup \{G\}$.

In addition to local correctness, termination and precision, the requirements on unfolding rules also include avoiding search space explosion as well as work duplication. Approaches to the local control have been based on one or more of the following elements:

- *determinacy* [100, 98, 97]
  Only (except once) select atoms that match a single clause head. The strategy can be refined with a so-called "look-ahead" to detect failure at a deeper level. Methods solely based on this heuristic, apart from not guaranteeing termination, tend not to worsen a program, but are often somewhat too conservative.
- *well-founded orders* [37, 200, 199, 196]
  Imposing some (essentially) well-founded order on selected atoms guarantees termination, but, on its own, can lead to overly eager unfolding.
- *homeomorphic embedding* [258, 178]
  Instead of well-founded ones, *well-quasi orders* can be used [21, 245]. Homeomorphic embedding on selected atoms has recently gained popularity as the basis for such an order.

We will examine the above concepts in somewhat more detail. First the notion of determinate unfolding can be defined as follows.

Definition 3.3.2 (determinate unfolding) A tree is *(purely) determinate* if each node of the tree has at most 1 child. An unfolding rule is *purely determinate without lookahead* if for every program P and every goal G it returns a determinate SLDNF-tree. An unfolding rule is *purely determinate (with lookahead)* if for every program P and every goal G it returns a SLDNF-tree τ such that the subtree τ^- of τ, obtained by removing the failed branches, is determinate.

Usually the above definitions of determinate unfolding rules are extended to allow one non-determinate unfolding step, ensuring that non-trivial trees can be constructed. Depending on the definition, this non-determinate step may either occur only at the root (e.g. in [97]), anywhere in the tree or only at the bottom (i.e. its resolvents must be leaves, as e.g. in [100, 172]). These three additional forms of determinate trees, which we will call *shower*, *fork* and *beam* determinate trees respectively, are illustrated in Figure 3.5.

Determinate unfolding has been proposed as a way to ensure that partial deduction will never duplicate computations in the residual program [100, 97, 98]. Indeed, in the context of the left-to-right selection rule of Prolog, the following fairly simple example shows that non-leftmost, non-determinate unfolding may duplicate (large amounts of) work in the transformation result. The one non-determinate unfolding step performed by a shower, fork or beam determinate unfolding rule, is therefore generally supposed to mimic the runtime selection rule.

Example 3.3.3 Let us return to the program P of Example 3.2.5:

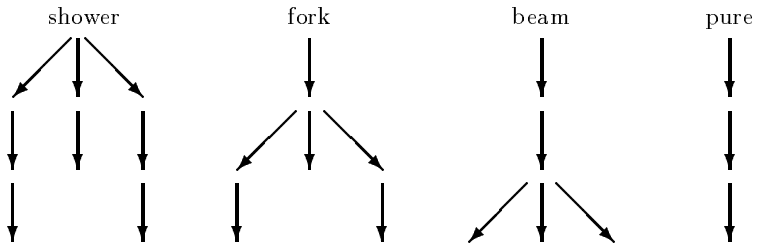


Figure 3.5: Four forms of determinate trees

$member(X, [X \mid T]) \leftarrow$
$member(X, [Y \mid T]) \leftarrow member(X, T)$
$inboth(X, L1, L2) \leftarrow member(X, L1), member(X, L2)$

Let $\mathcal{A} = \{inboth(a, L1, [X, Y])\}$. By performing the non-leftmost non-determinate unfolding in Figure 3.6, we obtain the following partial deduction P' of P wrt $\mathcal{A}$:

$member(X, [X \mid T]) \leftarrow$
$member(X, [Y \mid T]) \leftarrow member(X, T)$
$inboth(a, L1, [a, Y]) \leftarrow member(a, L1)$
$inboth(a, L1, [X, a]) \leftarrow member(a, L1)$

Let us examine the run-time goal $G = \leftarrow inboth(a, [z, y, \ldots, a], [X, Y])$, for which $P' \cup \{G\}$ is $\mathcal{A}$-covered. Using the Prolog left-to-right computation rule the expensive sub-goal $\leftarrow member(a, [z, y, \ldots, a])$ is only evaluated once in the original program P, while it is executed twice in the specialised program P'.

Restricting ourselves to determinate unfolding ensures that such bad cases of deterioration do not occur. It also ensures that the order of solutions, e.g. under Prolog execution, is not altered and that termination is preserved (termination might however be improved, as e.g. $\leftarrow loop, fail$ can be transformed into $\leftarrow fail$; for further details related to the preservation of termination we refer to e.g. [230, 27, 30]). Leftmost, non-determinate unfolding, usually allowed to compensate for the all too cautious nature of purely determinate unfolding, avoids the more drastic deterioration pitfalls in the context of e.g. Prolog, but can still lead to multiplying unifications.

Example 3.3.4 Let us adapt Example 3.3.3 by using the following set $\mathcal{A} = \{inboth(X, [Y], [V, W])\}$. We can fully unfold $\leftarrow inboth(X, [Y], [V, W])$ and we then obtain the following partial deduction P' of P wrt $\mathcal{A}$:

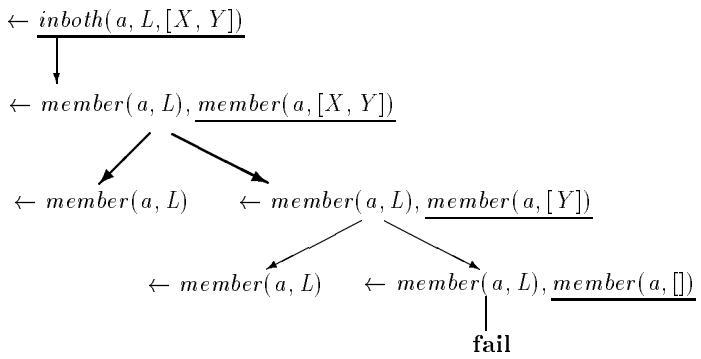


Figure 3.6: Non-leftmost non-determinate unfolding for Example 3.3.3

$member(X, [X \mid T]) \leftarrow$
$member(X, [Y \mid T]) \leftarrow member(X, T)$
$inboth(\underline{X}, [\underline{X}], [X, W]) \leftarrow$
$inboth(\underline{X}, [\underline{X}], [V, X]) \leftarrow$

No goal has been duplicated by the leftmost non-determinate unfolding, but the unification $X = Y$ for $\leftarrow inboth(X, [Y], [V, W])$ has potentially been duplicated. E.g., when executing the runtime goal $\leftarrow inboth(t_x, [t_y], [t_v, t_w])$ in P' the terms t_x and t_y will be unified when resolving with the third clause of P' and then unified again when resolving with the fourth clause of P'.[1] In the original program P this unification will only be performed once, namely when resolving with the first clause defining *member*. For run-time goals where t_x and t_y are very complicated structures this might actually result in P' being slower than the original P. However, as unifications are generally much less expensive than executing entire goals, this problem is (usually) less of an issue.

In practical implementations one has also to take care of such issues as the clause indexing performed by the compiler as well as how terms are created (i.e. avoid duplication of term construction operations). Again for these issues, determinate unfolding has proven to be a generally safe, albeit sometimes too conservative, approach. Fully adequate solutions to these, more implementation oriented, aspects are still topics of ongoing research.

Let us return to the aspect of local termination. Restricting oneself to determinate unfolding in itself does not guarantee termination, as there can

[1] A very smart compiler might detect this and produce more efficient code which does not re-execute unifications. It is promised that future versions of Mercury [257] will do this.

be infinitely failing determinate computations. In (strict) functional programs such a condition is equivalent to an error in the original program. In logic programming the situation is somewhat different: a goal can infinitely fail (in a deterministic way) at partial deduction time but still finitely fail at run time (see also the examples in Chapter 13). In applications like theorem proving, even infinite failures at run-time do not necessarily indicate an error: they might simply be due to unprovable statements. This is why, contrary to maybe functional programming, additional measures on top of determinacy should be adopted to ensure local termination.

One, albeit ad-hoc, way to solve this local termination problem is to simply impose an arbitrary depth bound. Such a depth bound is of course not motivated by any property, structural or otherwise, of the program or goal under consideration. The depth bound will therefore lead either to too little or too much unfolding in a lot of interesting cases.

As already mentioned, more refined approaches to ensure termination of unfolding exist. The methods in [37, 200, 199, 196] are based on well-founded orders, inspired by their usefulness in the context of static termination analysis (see e.g. [83, 61]). These techniques ensure termination, while at the same time allowing unfolding related to the structural aspect of the program and goal to be partially deduced, e.g. permitting the consumption of static input within the atoms of $\mathcal{A}$.

Formally, well-founded sets and orders are defined as follows:

Definition 3.3.5 (s-poset) A *strict partial order* on a set S is an anti-reflexive, anti-symmetric and transitive binary relation on $S \times S$. A couple $S, >_S$ consisting of a set S and a strict partial order $>_S$ on S is called an *s-poset* or partially strictly ordered set.

Definition 3.3.6 (wfo) An s-poset $S, >_S$ is called *well-founded* iff there is no infinite sequence of elements $s_1, s_2, \ldots$ in S such that $s_i > s_{i+1}$, for all $i \geq 1$. The order $>_S$ is also called a *well-founded order (wfo)* on S.

To ensure local termination, one has to find a sensible well-founded order on atoms and then only allow SLDNF-trees in which the sequence of selected atoms is strictly decreasing wrt the well-founded order. If an atom that we want to select is not strictly smaller than its ancestors, we either have to select another atom or stop unfolding altogether.

Example 3.3.7 Let us return to the *member* program P of Example 3.2.5. A simple well-founded order on atoms of the form $member(t_1, t_2)$ might be based on comparing the list length of the second argument.
The list length $list\_length(t)$ of a term t is defined to be:

- $1 + list\_length(t')$ if $t = [h|t']$ and
- 0 otherwise.

We then define the wfo on atoms by $member(t_1, t_2) > member(s_1, s_2)$ iff $list\_length(t_2) > list\_length(s_2)$.

Based on that wfo, the goal $\leftarrow member(X, [a, b|T])$ can be unfolded into $\leftarrow member(X, [b|T])$ and further into $\leftarrow member(X, T)$ because the list length of the second argument strictly decreases at each step. However, $\leftarrow member(X, T)$ cannot be further unfolded into $\leftarrow member(X, T')$ because the list length does not strictly decrease.

Much more elaborate well-founded orders, which are e.g. continuously refined in the unfolding process, exist and we refer the reader to [37, 200, 199, 196] for further details. These works also present a further refinement which, instead of requiring a decrease with every ancestor, only requires a decrease wrt the *covering ancestors*, i.e. one only compares with the ancestor atoms from which the current atom descends (via resolution).

Let us now turn our attention to approaches based on well-quasi orders, which are formally defined as follows.

Definition 3.3.8 (quasi order) A *quasi order* on a set S is a reflexive and transitive binary relation on $S \times S$. A couple $S, \geq_S$ consisting of a set S and a quasi order $\geq_S$ on S is called a *quasi ordered set*.

Henceforth, we will use symbols like $<, >$ (possibly annotated by some subscript) to refer to strict partial orders and $\leq, \geq$ to refer to quasi orders. We will use either "directionality" as is convenient in the context.

Definition 3.3.9 (wqo) A quasi ordered set $V, \leq_V$ is called *well-quasi-ordered (wqo)* iff for any infinite sequence of elements $e_1, e_2, \ldots$ in V there are $i < j$ such that $e_i \leq_V e_j$. We also say that $\leq_V$ is a *well-quasi order (wqo)* on V.

One problematic aspect for the approach based on well-founded orders, is the satisfactory *automatic* unfolding of meta-interpreters. This issue remains largely unsolved, although some initial efforts can be found in [195, 196]. In that context an approach based on well-quasi orders seems to be more flexible. Indeed, while an approach based on wfo requires a strict decrease at every unfolding step, an approach based on wqo can allow incomparable steps as well. This e.g. allows a wqo to have no a priori fixed weight or order attached to functors and arguments.

An interesting wqo is the *homeomorphic embedding* relation of [82]. We will later use it in several experiments, superimposed on e.g. determinate

unfolding to ensure local termination. We will also use it in the context of global control later in Chapter 6.

The homeomorphic embedding relation is very generous and will for example allow to unfold from $p([],[a])$ to $p([a],[])$ but also the other way around. This illustrates the flexibility of using well-quasi orders compared to well-founded ones, as there exists *no* wfo which will allow both these unfoldings. It however also illustrates why, when using a wqo, one has to compare with every predecessor. Otherwise one will get infinite derivations of the form $p([a],[]) \rightarrow p([],[a]) \rightarrow p([a],[]) \rightarrow \ldots$. When using a wfo one has to compare only to the closest predecessor [199], because of the transitivity of the order and the strict decrease enforced at each step. However, wfo are usually extended to incorporate variant checking (see e.g. [196, 199]) and therefore require inspecting every predecessor anyway (though only when there is no strict weight decrease).

3.3.4 Control of polyvariance

If we use renaming to ensure independence and (for the moment) suppose that the local termination and precision problems have been solved by the approaches presented above, we are still left with the problem of ensuring *closedness* and *global termination* while minimising the *global precision loss*. We will call this combination of problems the *control of polyvariance problem* as it is very closely related to how many different specialised versions of some given predicate should be put into $\mathcal{A}$.[2] It is this important problem we address in Part II of this thesis.

Let us examine how the 3 subproblems of the control of polyvariance problem interact.

- *Coveredness vs. Global Termination*
  Coveredness (or respectively closedness) can be simply ensured by repeatedly adding the uncovered (i.e not satisfying Definition 3.2.11 or Definition 3.2.8 respectively) atoms to $\mathcal{A}$ and unfolding them. Unfortunately this process generally leads to non-termination, even when using the *msg* to ensure independence. For instance, the "reverse with accumulating parameter" program (see Example 4.3.2 below or e.g. [196, 200]) exposes this non-terminating behaviour.
- *Global Termination vs. Global Precision*
  To ensure finiteness of $\mathcal{A}$ we can repeatedly apply an "abstraction" operator which generates a set of more general atoms. Unfortunately this induces a loss of global precision.

By using the two ideas above to (try to) ensure coveredness and global

[2] A method is called *monovariant* if it allows only one specialised version per predicate.

termination, we can formulate a generic partial deduction algorithm. First, the concept of abstraction has to be formally defined.

Definition 3.3.10 (abstraction) Let $\mathcal{A}$ and $\mathcal{A}'$ be sets of atoms. Then $\mathcal{A}'$ is an *abstraction* of $\mathcal{A}$ iff every atom in $\mathcal{A}$ is an instance of an atom in $\mathcal{A}'$. An *abstraction operator* is an operator which maps every finite set of atoms to a finite abstraction of it.

The above definition guarantees that any set of clauses covered by $\mathcal{A}$ is also covered by $\mathcal{A}'$. Note that sometimes an abstraction operator is also referred to as a *generalisation operator*.

The following generic scheme, based on a similar one in [97, 98], describes the basic layout of practically all algorithms for controlling partial deduction.

Algorithm 3.3.11 (standard partial deduction)
Input: A program P and a goal G
Output: A specialised program P'
Initialise: $i = 0$, $\mathcal{A}_0 = \{A \mid A \text{ is an atom in } G\}$
repeat
for each $A_k \in \mathcal{A}_i$ **do**
compute a finite SLDNF-tree τ_k for $P \cup \{\leftarrow A_k\}$ by
applying an unfolding rule U;
let $\mathcal{A}'_i := \mathcal{A}_i \cup \{B_l | B_l \in leaves(\tau_k)$ for some tree τ_k, such that B_l is not an instance[3] of any $A_j \in \mathcal{A}_i\}$;
let $\mathcal{A}_{i+1} := abstract(\mathcal{A}'_i)$; where *abstract* is an abstraction operator
let $i := i + 1$;
until $\mathcal{A}_{i+1} = \mathcal{A}_i$
Apply a renaming transformation to $\mathcal{A}_i$ to ensure independence;
Construct P' by taking resultants.

In itself the use of an abstraction operator does not yet guarantee global termination. But, if the above algorithm terminates then coveredness is ensured, i.e. $P' \cup \{G\}$ is $\mathcal{A}_i$-covered (modulo renaming). With this observation we can reformulate the *control of polyvariance problem* as one of finding an *abstraction operator which maximises specialisation while ensuring termination.*

A very simple abstraction operator which ensures termination can be obtained by imposing a finite maximum number of atoms in $\mathcal{A}_i$ and using the *msg* to stick to that maximum. For example, in [200] one atom per predicate is enforced by using the *msg*. However, using the *msg* in this

[3] One can also use the variant test to make the algorithm more precise.

way can induce an even bigger *loss of precision* (compared to using the *msg* to ensure independence), because it will now also be applied on *independent* atoms. For instance, calculating the *msg* for the set of atoms $\{solve(p(a)), solve(q(f(b)))\}$ yields the atom $solve(X)$ and all potential for specialisation is probably lost.

In [200] this problem has been remedied to some extent by using a static pre-processing renaming phase (as defined in [18]) which will generate one extra renamed version for the top-level atom to be specialised. However, this technique only works well if all relevant input can be consumed in one local unfolding of this top-most atom. Apart from the fact that this huge local unfolding is not always a good idea from a point of view of efficiency (e.g. it can slow down the program as illustrated by the Examples 3.3.3 and 3.3.4), in a lot of cases this simply cannot be accomplished (for instance if partial input is not consumed but carried along, like the representation of an object-program inside a meta-interpreter).

The basic goal pursued in Part II of this thesis is to define a flexible abstraction operator which does not exhibit this dramatic loss of precision and provides a fine-grained control of polyvariance, while still guaranteeing termination of the partial deduction process.

Part II

On-line Control of Partial Deduction: Controlling Polyvariance

Chapter 4

Characteristic Trees

4.1 Structure and abstraction

In the previous chapter we have presented the generic partial deduction Algorithm 3.3.11. This algorithm is parametrised by an unfolding rule for the local control and by an abstraction operator for the control of polyvariance. The abstraction operator examines a set of atoms and then decides which of the atoms should be abstracted and which ones should be left unmodified.

An abstraction operator like the *msg* is just based on the *syntactic structure* of the atoms to be specialised. This is generally not such a good idea. Indeed, two atoms can be unfolded and specialised in a very similar way in the context of one program P_1, while in the context of another program P_2 their specialisation behaviour is drastically different. The syntactic structure of the two atoms is of course unaffected by the particular context and a operator like the *msg* will perform exactly the same abstraction within P_1 and P_2, although vastly different generalisations might be called for.

A better candidate for an abstraction might be to examine the finite, possibly incomplete SLDNF-trees generated for these atoms. These trees capture (to some depth) how the atoms behave computationally in the context of the respective programs. They also capture (part of) the specialisation that has been performed on these atoms. An abstraction operator which takes these trees into account will notice their similar behaviour in the context of P_1 and their dissimilar behaviour within P_2, and can therefore take appropriate actions in the form of different generalisations. The following example illustrates these points.

Example 4.1.1 Let P be the *append* program:

$$(1) \quad append([\,], Z, Z) \leftarrow$$
$$(2) \quad append([H|X], Y, [H|Z]) \leftarrow append(X, Y, Z)$$

Note that we have added clause numbers, which we will henceforth take the liberty to incorporate into illustrations of SLD-trees in order to clarify which clauses have been resolved with. To avoid cluttering the figures we will also sometimes drop the substitutions in such figures.

Let $\mathcal{A} = \{B, C\}$ be a dependent set of atoms, where $B = append([a], X, Y)$ and $C = append(X, [a], Y)$. Typically a partial deducer will unfold the two atoms of $\mathcal{A}$ in the way depicted in Figure 4.1, returning the finite SLD-trees τ_B and τ_C. These two trees, as well as the associated resultants, have a very different structure. The atom $append([a], X, Y)$ has been fully unfolded and we obtain for $resultants(\tau_B)$ the single fact:

$$append([a], X, [a|X]) \leftarrow$$

while for $append(X, [a], Y)$ we obtain for $resultants(\tau_C)$ the following set of clauses:

$$append([\,], [a], [a]) \leftarrow$$
$$append([H|X], [a], [H|Z]) \leftarrow append(X, [a], Z)$$

So, in this case, it is vital to keep separate specialised versions for B and C and not abstract them by e.g. their *msg*.
However, it is very easy to come up with another context in which the specialisation behaviour of B and C are almost indiscernible. Take for instance the following program P^* in which $append^*$ no longer appends two lists but finds common elements at common positions:

$$(1^*) \quad append^*([X\,|\,T_X], [X\,|\,T_Y], [X]) \leftarrow$$
$$(2^*) \quad append^*([X\,|\,T_X], [Y\,|\,T_Y], E) \leftarrow append^*(T_X, T_Y, E)$$

The associated finite SLD-trees τ_B^* and τ_C^*, depicted in Figure 4.2, are now almost fully identical. In that case, it is not useful to keep different specialised versions for B and C because the following single set of specialised clauses could be used for B and C without specialisation loss:

$$append^*([a\,|\,T_1], [a\,|\,T_2], [a]) \leftarrow$$

This illustrates that the syntactic structures of B and C alone provide insufficient information for a satisfactory control of polyvariance and that a refined abstraction operator should also take the associated SLDNF-trees into consideration.

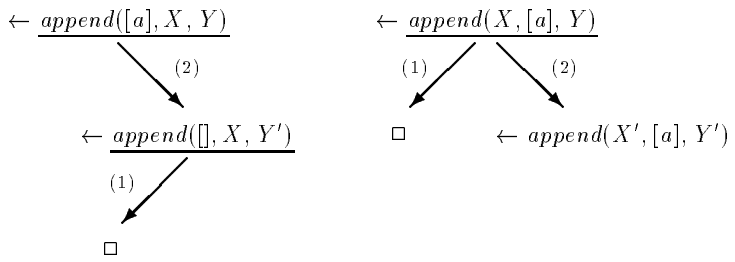


Figure 4.1: SLD-trees τ_B and τ_C for Example 4.1.1

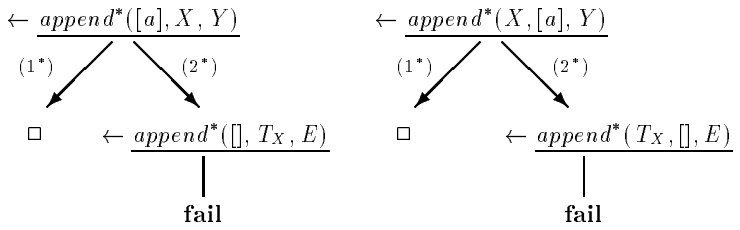


Figure 4.2: SLD-trees τ_B^* and τ_C^* for Example 4.1.1

4.2 Characteristic paths and trees

In the previous section we have illustrated the interest of examining the (possibly incomplete) SLDNF-trees generated for the atoms to be partially deduced and e.g. only abstract two (or more) atoms if their associated trees are "similar enough". A crucial question is of course which part of these SLDNF-trees should be taken into account to decide upon similarity. If two atoms are abstracted only if their associated trees are identical, this amounts to performing no abstraction at all. So an abstraction operator should focus on the "essential" structure of an SLDNF-tree and for instance disregard the particular substitutions and goals within the tree. The following two definitions, adapted from [97], do just that: they characterise the essential structure of SLDNF-derivations and trees.

Definition 4.2.1 (characteristic path) Let G_0 be a goal and let P be a normal program whose clauses are numbered. Let $G_0, \ldots, G_n$ be the goals

of a finite, possibly incomplete SLDNF-derivation D of $P \cup \{G_0\}$. The *characteristic path* of the derivation D is the sequence $\langle l_0 \circ c_0, \ldots, l_{n-1} \circ c_{n-1} \rangle$, where l_i is the position of the selected literal in G_i, and c_i is defined as:

- if the selected literal is an atom, then c_i is the number of the clause chosen to resolve with G_i.
- if the selected literal is $\neg p(\bar{t})$, then c_i is the predicate p.

The set containing the characteristic paths of all finite SLDNF-derivations of $P \cup \{G_0\}$ will be denoted by $chpaths(P, G_0)$.

For example, the characteristic path of the derivation associated with the only branch of the SLD-tree τ_B in Figure 4.1 is $\langle 1 \circ 2, 1 \circ 1 \rangle$.

Recall that an SLDNF-derivation D can be either failed, incomplete, successful or infinite. As we will see below, characteristic paths will only be used to characterise *finite* and *non-failing* derivations of *atomic* goals, corresponding to the atoms to be partially deduced. Still, one might wonder why a characteristic path does not contain information on whether the associated derivation is successful or incomplete. The following proposition gives an answer to that question.

Proposition 4.2.2 Let P be a normal program and let G_1, G_2 be two goals with the same number of literals. Let D_1, D_2 be two non-failed, finite derivations of $P \cup \{G_1\}$ and $P \cup \{G_2\}$ respectively. Also let D_1 and D_2 have the same characteristic path δ. Then
(1) D_1 is successful iff D_2 is and
(2) D_1 is incomplete iff D_2 is.

Proof As D_1 and D_2 can only be successful or incomplete, points (1) and (2) are equivalent and it is sufficient to prove point (1). Also, as D_1 and D_2 have the same characteristic path they must have the same length (i.e. same number of derivation steps) and we will prove the lemma by induction on the length of D_1 and D_2.
Induction Hypothesis: Proposition 4.2.2 holds for derivations D_1, D_2 with length $\leq n$.
Base Case: D_1, D_2 have the length 0.
This means that G_1 is the final goal of D_1 and G_2 the final goal of D_2. As G_1 and G_2 have the same number of literals it is impossible to have that $G_1 = \square$ while $G_2 \neq \square$ or $G_1 \neq \square$ while $G_2 = \square$.
Induction Step: D_1, D_2 have length $n+1$.
Let $R_0, \ldots, R_{n+1}$ be the sequence of goals of D_1 (with $R_0 = G_1$) and let $Q_0, \ldots, Q_{n+1}$ be the sequence of goals of D_2 (with $Q_0 = G_2$). Let D_1' be the suffix of D_1 whose sequence of goals is $R_1, \ldots, R_{n+1}$. Similarly, let

D_2' be the suffix of D_2 whose sequence of goals is $Q_1, \ldots, Q_{n+1}$. Let $\delta = \langle l_0 \circ c_0, \ldots, l_n \circ c_n \rangle$ be the characteristic path of D_1 and D_2. There are two possibilities for $l_0 \circ c_0$, corresponding to whether a positive or negative literal has been selected. If a negative literal has been selected then (for both R_0 and Q_0) one literal has been removed and R_1 and Q_1 have the same number of literals. Similarly if a positive literal has been selected then trivially R_1 and Q_1 have the same number of literals (because the same clause c_1 in the same program P has been used). In both cases R_1 and Q_1 have the same number of literals and we can therefore apply the induction hypothesis on D_1' and D_2' to prove that D_1' is successful iff D_2' is. Finally, because D_1 (respectively D_2) is successful iff D_1' (respectively D_2') is, the induction step holds. □

The information whether a finite, non-failing derivation of an atomic goal is incomplete or successful is thus already implicitly present in the characteristic path and no further precision would be gained by adding it.

Also, once the top-level goal is known, the characteristic path is sufficient to reconstruct all the intermediate goals as well as the final one. So, one could actually replace the predicate p in point 2 of Definition 4.2.1 and use a unique symbol to signal the selection of a negative literal.

Now that we have characterised derivations, we can capture the computational behaviour of goals by characterising the derivations of their associated finite SLDNF-trees.

Definition 4.2.3 (characteristic tree) Let $\leftarrow Q$ be a goal, P a normal program and τ_Q a finite SLDNF-tree for $P \cup \{\leftarrow Q\}$. Then the *characteristic tree* τ *of* τ_Q is the set containing the characteristic paths of the non-failing SLDNF-derivations associated with the branches of τ_Q. τ is called *a characteristic tree* iff it is the characteristic tree of some finite SLDNF-tree for some program and goal.
Let U be an unfolding rule such that $U(P, \leftarrow Q) = \tau_Q$. Then τ is also called *the characteristic tree of* Q *in* P *under* U, and will be denoted by $chtree(\leftarrow Q, P, U)$. When P and U are clear from the context we will simply talk about the characteristic tree of Q. We also say that τ is *a characteristic tree of* Q *(in* P*)* iff for some unfolding rule U we have $chtree(\leftarrow Q, P, U) = \tau$.

Note that the characteristic path of an empty derivation is the empty path $\langle\rangle$, and the characteristic tree of a trivial SLDNF-tree is $\{\langle\rangle\}$, while the characteristic tree of a finitely failed SLDNF-tree is $\emptyset$.

Although a characteristic tree only contains a collection of characteristic paths, the actual tree structure is not lost and can be reconstructed

without ambiguity. The "glue" is provided by the clause numbers inside the characteristic paths (branching in the tree is indicated by differing clause numbers).

Example 4.2.4 The characteristic trees of the finite SLD-trees τ_B and τ_C in Figure 4.1 are $\{\langle 1 \circ 2, 1 \circ 1\rangle\}$ and $\{\langle 1 \circ 1\rangle, \langle 1 \circ 2\rangle\}$ respectively.
The characteristic trees of the finite SLD-trees τ_B^* and τ_C^* in Figure 4.2 are both $\{\langle 1 \circ 1^*\rangle\}$.

The following observations reveal the interest of characteristic trees in the context of partial deduction. Indeed, the characteristic tree of an atom A explicitly or implicitly captures the following important aspects of specialisation:

- the branches that have been pruned through the unfolding process (namely those that are absent from the characteristic tree). For instance, by inspecting the characteristic trees of τ_B and τ_C from Examples 4.1.1 and 4.2.4, we can see that two branches have been pruned for the atom B (thereby removing recursion) whereas no pruning could be performed for C.

- how deep $\leftarrow A$ has been unfolded and which literals and clauses have been resolved with each other in that process. This captures the computation steps that have already been performed at partial deduction time.

- the number of clauses in the resultants of A (namely one per characteristic path) and also (implicitly) which predicates are called in the bodies of the resultants. As we will see later, this means that a single predicate definition can (in principle) be used for two atoms which have the same characteristic tree.

In other words, a characteristic tree captures all the relevant aspects of specialisation, attained by the local control for a particular atom. A specialisation aspect that does not materialise within a characteristic tree is how the atoms in the leaves of the associated SLDNF-tree are further specialised, i.e. the global control and precision are not captured. However, the deeper an atom gets unfolded, the more information is carried by the associated characteristic tree. If the unfolding rule is e.g. based on well-founded or well-quasi orders, it will (ideally) unfold until all partial input has been consumed. In that case it is more likely that two atoms with the same characteristic tree will also have similar call patterns in the leaves (e.g. atoms containing no more partial input) and thus lead to similar global specialisation.

Also note that a characteristic trees only contains paths for the non-failing branches and therefore do not capture *how* exactly some branches were pruned. The following example illustrates why this is adequate and even beneficial.

Example 4.2.5 Let P be the following program:

(1) $member(X, [X|T]) \leftarrow$
(2) $member(X, [Y|T]) \leftarrow member(X, T)$

Let $A = member(a, [a, b])$ and $B = member(a, [a])$. Suppose that A and B are unfolded as depicted in Figure 4.3. Then both these atoms have the same characteristic tree $\tau = \{\langle 1 \circ 1 \rangle\}$ although the associated SLDNF-trees differ by the structure of their failing branches. However, this is of no relevance, because the failing branches do not materialise within the resultants (i.e. the specialised code generated for the atoms) and furthermore the single resultant $member(a, [a|T]) \leftarrow$ could be used for both A and B without loosing any specialisation.

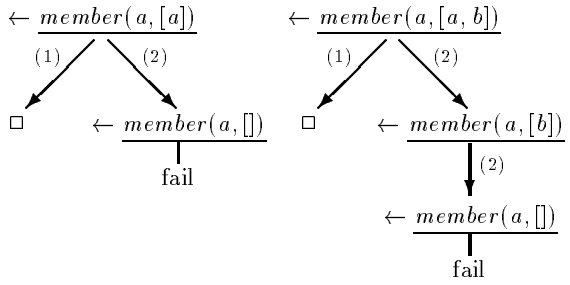


Figure 4.3: SLD-trees for Example 4.2.5

In summary, characteristic trees seem to be an almost ideal vehicle for a refined control of polyvariance, a fact we will try to exploit in the following section.

4.3 An abstraction operator using characteristic trees

The following abstraction operator represents a first attempt at using characteristic trees for the control of polyvariance. Basically it classifies atoms

by their associated characteristic tree. Generalisation, in this case the *msg*, is then only applied on those atoms which have the same characteristic tree.

Definition 4.3.1 ($chabs_{P,U}$) Let P be a normal program, U an unfolding rule and $\mathcal{A}$ a set of atoms. For every characteristic tree τ, let $\mathcal{A}_\tau$ be defined as $\mathcal{A}_\tau = \{A \mid A \in \mathcal{A} \wedge chtree(\leftarrow A, P, U) = \tau\}$.
The abstraction operator $chabs_{P,U}$ is then defined as:

$$chabs_{P,U}(\mathcal{A}) = \{msg(\mathcal{A}_\tau) \mid \tau \text{ is a characteristic tree}\}$$

The following example illustrates the above definition.

Example 4.3.2 Let P be the program reversing a list using an accumulating parameter:

(1) $rev([], Acc, Acc) \leftarrow$
(2) $rev([H|T], Acc, Res) \leftarrow rev(T, [H|Acc], Res)$

We will use $chabs_{P,U}$ together with an unfolding rule U based on well-founded orders inside the generic Algorithm 3.3.11 for partial deduction. When starting out with $\mathcal{A}_0 = \{rev([a|B], [], R)\}$ the following steps are performed by Algorithm 3.3.11:

- the only atom in $\mathcal{A}_0$ is unfolded (see Figure 4.4) and the atoms in the leaves are added, yielding: $\mathcal{A}'_0 = \{rev([a|B], [], R), rev(B, [a], R)\}$.
- the abstraction operator $chabs_{P,U}$ is applied:
  $\mathcal{A}_1 = chabs_{P,U}(\mathcal{A}'_0) = \{rev([a|B], [], R), rev(B, [a], R)\}$ (because the atoms in $\mathcal{A}'_0$ have different characteristic trees).
- the atoms in $\mathcal{A}_1$ are unfolded (see Figure 4.4) and the atoms in the leaves are added, yielding:
  $\mathcal{A}'_1 = \{rev([a|B], [], R), rev(B, [a], R), rev(T, [H, a], R)\}$.
- the abstraction operator $chabs_{P,U}$ is applied:
  $\mathcal{A}_2 = chabs_{P,U}(\mathcal{A}'_1) = \{rev([a|B], [], R), rev(T, [A|B], R)\}$ (because the atoms $rev(B, [a], R)$ and $rev(T, [H, a], R)$ have the same characteristic tree, see Figure 4.4).
- the atoms in $\mathcal{A}_2$ are unfolded and the leaf atoms added:
  $\mathcal{A}'_2 = \{rev([a|B], [], R), rev(T, [A|B], R), rev(T', [H', A|B], R)\}$.
- the abstraction operator is applied: $\mathcal{A}_3 = chabs_{P,U}(\mathcal{A}'_2) = \mathcal{A}_2$ and we have reached a fixpoint and thus obtain the following partial deduction satisfying the coveredness condition (and which is also independent without renaming):

$$rev([a|B], [], R) \leftarrow rev(B, [a], R)$$

$$rev([], [A|B], [A|B]) \leftarrow$$
$$rev([H|T], [A|B], Res) \leftarrow rev(T, [H, A|B], Res)$$

Because of the selective application of the *msg*, no loss of precision has been incurred by $chabs_{P,U}$, i.e. the pruning and pre-computation for e.g. the atom $rev([a|B], [], R)$ has been preserved. An abstraction operator allowing just one version per predicate would have lost this local specialisation, while a method with unlimited polyvariance (also called dynamic renaming, in e.g. [17]) does not terminate.

For this example, $chabs_{P,U}$ provides a terminating and fine grained control of polyvariance, conferring just as many polyvariant versions as necessary. The abstraction operator $chabs_{P,U}$ is thus much more flexible than e.g. the static pre-processing renaming of [18, 200]).

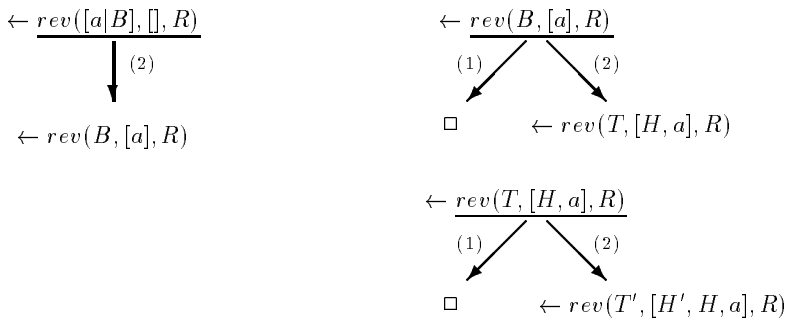


Figure 4.4: SLD-trees for Example 4.3.2

The above example is thus very encouraging and one might hope that $chabs_{P,U}$ always preserves the characteristic trees upon generalisation and that it already provides a refined solution for the control of polyvariance problem. Unfortunately, although for a lot of practical cases $chabs_{P,U}$ performs quite well, it does not always preserve the characteristic trees, entailing a sometimes quite severe loss of precision and specialisation. Let us examine an example:

Example 4.3.3 Let P be the program:

(1) $p(X) \leftarrow$
(2) $p(c) \leftarrow$

Let $\mathcal{A} = \{p(a), p(b)\}$. Independently of the unfolding rule, $p(a)$ and $p(b)$ have the same characteristic tree $\tau = \{\langle 1 \circ 1 \rangle\}$. Thus $chabs_{P,U}(S) = \{p(X)\}$

and the generalisation $p(X)$ has unfortunately the characteristic tree $\tau' = \{\langle 1 \circ 1\rangle, \langle 1 \circ 2\rangle\}$ and the pruning that was possible for the atoms $p(a)$ and $p(b)$ has been lost. More importantly, there exists *no* atom, more general than $p(a)$ and $p(b)$, which has τ as its characteristic tree.

The problem in the above example is that, through generalisation, a new non-failing derivation has been added, thereby modifying the characteristic tree. Another problem can occur when negative literals are selected by the unfolding rule.

Example 4.3.4 Let us examine the following program P:

(1) $p(X) \leftarrow \neg q(X)$
(2) $q(f(X)) \leftarrow$

For this program and using e.g. a determinate unfolding rule, $p(a)$ and $p(b)$ have the same characteristic tree $\{\langle 1 \circ 1, 1 \circ q\rangle\}$. The abstraction operator $chabs_{P,U}$ will therefore produce $\{p(X)\}$ as a generalisation of $\{p(a), p(b)\}$. Again however, $p(X)$ has a different characteristic tree, namely $\{\langle 1 \circ 1\rangle\}$, because the non-ground literal $\neg q(X)$ cannot be selected in the resolvent of $\leftarrow p(X)$. The problem is that, by generalisation, a previously selectable ground negative literal in a resolvent can become non-ground and thus no longer selectable by SLDNF.

These losses of precision can have some regrettable consequences in practice:

- important opportunities for specialisation can be lost and
- termination of Algorithm 3.3.11 can be undermined.

Let us illustrate the possible precision losses through two simple, but more realistic examples.

Example 4.3.5 Let P be the following program, checking two lists for equality.

(1) $eqlist([], []) \leftarrow$
(2) $eqlist([H|X], [H|Y]) \leftarrow eqlist(X, Y)$

Given any determinate unfolding rule, the atoms $A = eqlist([1, 2], L)$ and $B = eqlist(L, [3, 4])$ are unfolded as shown in Figure 4.5 and have the same characteristic tree $\tau = \{\langle 1 \circ 2, 1 \circ 2, 1 \circ 1\rangle\}$. Unfortunately the abstraction operator $chabs_{P,U}$ is unable to preserve τ. Indeed, $chabs_{P,U}(\{A, B\}) = \{eqlist(X, Y)\}$ whose characteristic tree is $\{\langle 1 \circ 1\rangle, \langle 1 \circ 2\rangle\}$ and the precomputation and pruning performed on A and B has been lost.

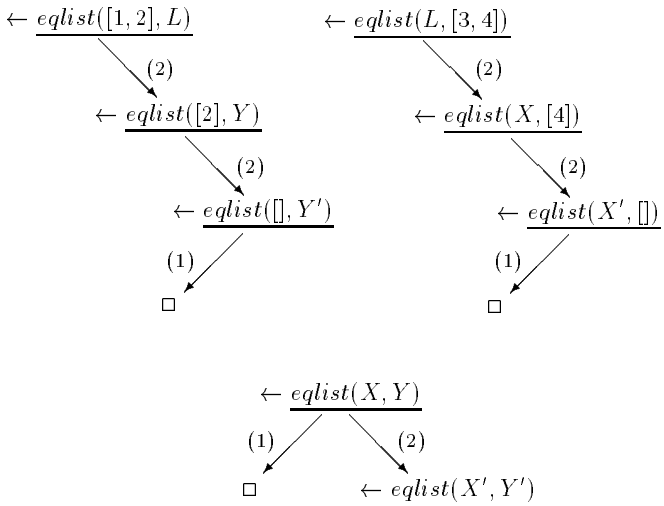


Figure 4.5: SLD-trees for Example 4.3.5

The previous example is taken from [97], whose abstraction mechanism can solve the example. The following two examples can however not be solved by [97].

Example 4.3.6 Let us return to Example 4.2.5 and specialise the *member* program for $A = member(a, [a, b])$ and $B = member(a, [a])$. First we put the atoms into $\mathcal{A}_0$: $\mathcal{A}_0 = \{member(a, [a, b]), member(a, [a])\}$. For both atoms the associated characteristic tree is $\tau = \{\langle 1 \circ 1 \rangle\}$, given e.g. a determinate unfolding rule with lookahead (see Figure 4.3). Thus $chabs_{P,U}$, as well as the method of [97], abstracts the two atoms and produces the generalised set $\mathcal{A}_1 = \{member(a, [a|T])\}$. Unfortunately, the generalised atom $member(a, [a|T])$ has a different characteristic tree τ', independently of the particular unfolding rule. For a determinate unfolding rule with lookahead we obtain $\tau' = \{\langle 1 \circ 1 \rangle, \langle 1 \circ 2 \rangle\}$.
This loss of precision leads to sub-optimal specialised programs. At the next step of the algorithm the atom $member(a, T)$ will be added to $\mathcal{A}_1$. This atom also has the characteristic tree τ' under U. Hence the final set $\mathcal{A}_2$ equals $\{member(a, L)\}$ (containing the *msg* of $member(a, [a|T])$ and $member(a, T)\}$), and we obtain the following specialisation, sub-optimal for $\leftarrow member(a, [a, b]), member(a, [a])$:

(1') $member(a, [a|T]) \leftarrow$
(2') $member(a, [X|T]) \leftarrow member(a, T)$

So, although partial deduction was able to figure out that $member(a, [a, b])$ as well as $member(a, [a])$ have only one non-failing resolvent, this information has been lost due to an imprecision of the abstraction operator, thereby leading to a sub-optimal residual program in which the determinacy is not explicit (and redundant computation steps occur at run-time). Note that a "perfect" program for $\leftarrow member(a, [a, b]), member(a, [a])$ would just consist of (a filtered version of) clause (1').

Let us discuss the termination aspects next. One might hope that $chabs_{P,U}$ ensures termination of the partial deduction Algorithm 3.3.11 if the number of characteristic trees is finite (which can be ensured by using a depth-bound for characteristic trees[1]).

Actually if the characteristic trees are preserved, then the abstraction operator $chabs_{P,U}$ does ensure termination of the partial deduction Algorithm 3.3.11.[2]

However, if characteristic trees are not preserved by the abstraction operator, then termination is no longer guaranteed (even assuming a finite number of characteristic trees). The following example illustrates this.

Example 4.3.7 Let P be the following program:

(1) $p(X, Y) \leftarrow p(Y, X), p(a, X)$
(2) $p(c, c) \leftarrow$

Also let U be a very simple unfolding rule with a depth bound of 1. If we perform Algorithm 3.3.11 with $chabs_{P,U}$ as abstraction operator, starting from the set of atoms $\mathcal{A}_0 = \{p(X, Y)\}$, we obtain the following:

1. we unfold the atoms in $\mathcal{A}_0$ (see Figure 4.6) which gives us the leaves $\{p(Y, X), p(a, X)\}$ and thus $\mathcal{A}'_0 = \{p(X, Y), p(a, X)\}$.

2. $\mathcal{A}_1 = chabs_{P,U}(\{p(X, Y), p(a, X)\}) = \{p(X, Y), p(a, X)\}$
(because the atom $p(a, X)$ has a different characteristic tree than $p(X, Y)$, see Figure 4.6).

3. we unfold the atoms in $\mathcal{A}_1$ (see Figure 4.6) giving us the leaves:
$\{p(Y, X), p(a, X), p(X, a), p(a, a)\}$ and thus
$\mathcal{A}'_1 = \{p(X, Y), p(a, X), p(X, a), p(a, a)\}$.

[1] The unfolding rule can still unfold as deeply as it wants to. See the discussion after Definition 5.1.8 in Chapter 5.

[2] The proof of Theorem 5.3.9 (for a more refined partial deduction algorithm) can actually be easily adapted to demonstrate this.

4. we apply $chabs_{P,U}$ to $\mathcal{A}'_1$, giving $\mathcal{A}_2 = \{p(X,Y)\}$ (because $p(a,X)$, $p(X,a)$ and $p(a,a)$ have the same characteristic tree $\{\langle 1 \circ 1 \rangle\}$, see Figure 4.6) tree but their *msg* is (a variant of) $p(X,Y)$. So $\mathcal{A}_2 = \mathcal{A}_0$ and we have a loop.

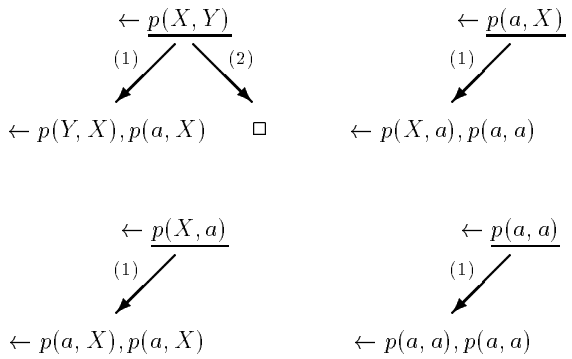


Figure 4.6: SLD-trees for Example 4.3.7

The above example heavily exploits the non-monotonic nature of Algorithm 3.3.11. Indeed, termination of partial deduction based on $chabs_{P,U}$, given a finite number of characteristic trees, could be ensured by making Algorithm 3.3.11 monotonic, i.e. replacing $\mathcal{A}_{i+1} := abstract(\mathcal{A}'_i)$ by $\mathcal{A}_{i+1} := A_i \cup abstract(\mathcal{A}'_i)$. From a practical point of view, this solution is however not very satisfactory as it might unnecessarily increase the polyvariance, possibly leading to a code explosion of the specialised program as well as to an increase in the transformation complexity. The former can be solved by a post-processing phase removing unnecessary polyvariance. However, by using an altogether more precise abstraction operator, preserving characteristic trees, these two problem will disappear automatically. We will then obtain an abstraction operator for partial deduction with optimal local precision (in the sense that all the specialisation achieved by the local control component is preserved by the abstraction) and guaranteeing termination. This quest is pursued in Chapters 5 and 6.

4.4 Characteristic trees in the literature

Characteristic trees have been introduced, in the context of definite programs and determinate unfolding rules without lookahead, by Gallagher

and Bruynooghe in [100] and were later refined by Gallagher in [97], leading to the definitions that we have presented in this chapter. Both [100] and [97] use a refined version of the abstraction operator $chabs_{P,U}$ and [97] uses a partial deduction algorithm very similar to Algorithm 3.3.11.

Unfortunately, although the abstraction operators in [97, 100] are more sophisticated than $chabs_{P,U}$ and can e.g. handle the *eqlist* Example 4.3.5, they share the same problems. Indeed, for e.g. the Examples 4.3.3 and 4.3.7, the abstraction operators of [97, 100] behave exactly like $chabs_{P,U}$ and the examples can thus easily be adapted (to account for some slight differences in the unfolding rule) to form counterexamples not only for the precision claim of [100] but also for the termination claims of both [100] and [97]. In Appendix B we actually adapt Example 4.3.3 to form a counterexample to the Lemma 4.11 in [100], which asserts that the abstraction operator of [100] preserves a structure quite similar to characteristic trees in the context of definite programs and beam determinate unfolding rules *without* lookahead (cf. Definition 3.3.2). Some additional comments can be found in [171], where the counterexample to Lemma 4.11 of [100] was first presented, as well as in its reworked version [172].

Example 4.3.6, based upon the *member* program, forms another, more natural counterexample to [97]. However, it is not a counterexample to [100], as it is crucial in that example to use a determinate unfolding rule *with* lookahead (otherwise A and B have a different characteristic tree). Thus one might wonder if it is possible to come up with "natural" examples, using definite programs and unfolding rules *without* lookahead, which cannot be solved by the techniques in [97] or [100]. (The simple programs of Example 4.3.3, Appendix B and Example 4.3.7 might be called somewhat unnatural, Example 4.3.4 uses negation and the *eqlist* Example 4.3.5 can be solved by [97, 100].) The following shows that such natural examples do indeed exist and are not too difficult to come by.

Example 4.4.1 Let P be again the *member* program from Example 4.2.5 and let $A = member(a, [b, c|T])$, $B = member(a, [c, d|T])$. We will use a beam determinate unfolding rule without lookahead. In that case the atoms A and B have the same characteristic tree $\tau = \{\langle 1 \circ 2, 1 \circ 2, 1 \circ 1\rangle, \langle 1 \circ 2, 1 \circ 2, 1 \circ 2\rangle\}$ (see Figure 4.7). However, $chabs_{P,U}(\{A, B\}) = \{member(a, [X, Y|T])\}$ and, as can be seen in Figure 4.7, the characteristic tree of the generalisation is unfortunately $\{\langle 1 \circ 1\rangle, \langle 1 \circ 2\rangle\}$. The precomputation and pruning that was possible for A and B has again not been preserved by $chabs_{P,U}$. Applying e.g. Algorithm 3.3.11, we obtain at the next iteration the set $chabs_{P,U}(\{member(a, [X, Y|T]), member(a, [Y|T])\}) = \{member(a, [Y|T])\}$ and then the final set $chabs_{P,U}(\{member(a, [Y|T]), member(a, T)\}) = \{member(a, T)\}$. We thus obtain the following subop-

timal, unpruned program P', performing redundant computations for both A and B:

(1') $member(a, [a|T]) \leftarrow$
(2') $member(a, [Y|T]) \leftarrow member(a, T)$

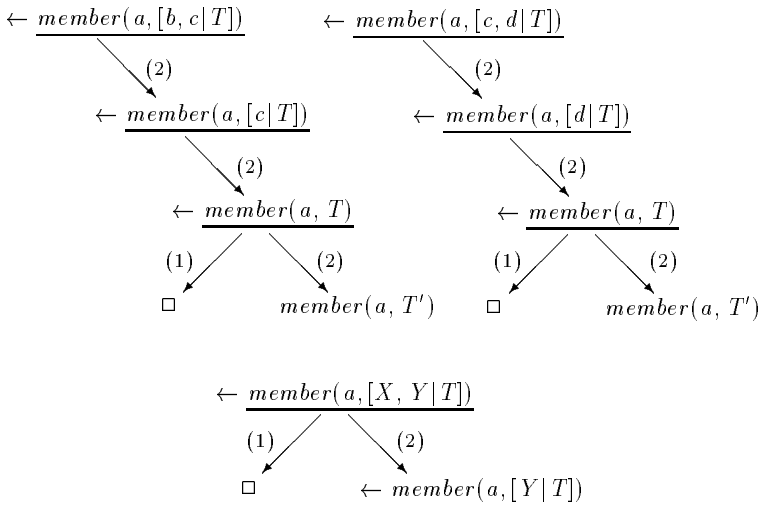


Figure 4.7: SLD-trees for Example 4.4.1

4.5 Extensions of characteristic trees

Less precision

Sometimes different characteristic trees can represent the same local specialisation behaviour. Indeed, the characteristic tree also encodes the particular order in which literals are selected and thus does not take advantage of the independence of the computation rule. This is not a problem for precision, but can lead to the production of more specialised versions than really necessary, as the following example demonstrates.

Example 4.5.1 Let P be this program:

(1) $p(X, Y) \leftarrow q(X), q(Y)$
(2) $q(a) \leftarrow$

Let $A = p(a, X)$ and $B = p(X, a)$. Then an unfolding rule U which tries to unfold more instantiated atoms first will give the following results:

- $chtree(\leftarrow A, P, U) = \{\langle 1 \circ 1, 1 \circ 2, 1 \circ 2 \rangle\}$
- $chtree(\leftarrow B, P, U) = \{\langle 1 \circ 1, 2 \circ 2, 1 \circ 2 \rangle\}$

So although the resultants for A and B are identical their characteristic trees are different.

A quite simple solution to this problem exists:[3] after having performed the local unfolding we just have to *normalise* the characteristic trees by imposing a fixed (e.g. left-to-right) computation rule and delaying the selection of all negative literals to the end. The results and discussions of this and the following chapters remain valid independently of whether this normalisation is applied or not. In Example 4.5.1 both A and B then have the same normalised characteristic tree $\{\langle 1 \circ 1, 1 \circ 2, 1 \circ 2 \rangle\}$.

A similar effect can be obtained, in the context of definite programs, via the *trace terms* of [103].

More precision

Supposing that we have an abstraction operator which preserves the (normalised) characteristic trees, we will in fact get the *minimal* amount of polyvariance which avoids any loss of local specialisation: a different characteristic tree now unavoidably leads to different resultants and thus specialisation. In most cases this will be the ideal amount of polyvariance, (because too much polyvariance makes the specialised program larger and often slower) and no further local specialisation can be obtained by introducing more polyvariance.

For some particular applications however, a need for further polyvariance arises. This is for instance the case for some examples in [66, 68, 201], where the specialised program is not directly executed but first submitted to an abstract interpretation phase (in order to detect redundant clauses) which is monovariant and cannot generate further polyvariance by itself. In that case further polyvariance can be generated during the specialisation by adorning the characteristic trees with some extra information. For instance, in [66] the characteristic trees are extended to include a depth-k abstraction of the selected literals and the abstraction operator will only abstract atoms with the same characteristic tree *and* the same depth-k abstraction.

[3] Thanks to Maurice Bruynooghe for pointing this out.

Chapter 5

Ecological Partial Deduction

5.1 Partial deduction based on characteristic atoms

In the previous chapter we have dwelled upon the appeal of characteristic trees for controlling polyvariance, but we have also highlighted the difficulty of preserving characteristic trees in the abstraction process as well as the ensuing problems concerning termination and precision. In this chapter, we present a solution to this entanglement. Its basic idea is to simply *impose* characteristic trees on the generalised atoms. This amounts to associating characteristic trees with the atoms to be specialised, allowing the preservation of characteristic trees in a straightforward way and circumventing the need for intricate generalisations.

5.1.1 Characteristic atoms

We first introduce the crucial notion of a *characteristic atom*, which associates a characteristic tree with an atom.

Definition 5.1.1 (characteristic atom) A *characteristic atom* is a couple (A, τ) consisting of an atom A and a characteristic tree τ.

Note that τ is not required to be a characteristic tree of A in the context of the particular program P under consideration.

Example 5.1.2 Let $\tau = \{\langle 1 \circ 1 \rangle\}$ be a characteristic tree. Then both $CA_1 = (member(a, [a, b]), \tau)$ and $CA_2 = (member(a, [a]), \tau)$ are characteristic atoms. $CA_3 = (member(a, [a|T]), \tau)$ is also a characteristic atom, but e.g. in the context of the *member* program P from Example 4.2.5, τ is *not* a characteristic tree of its atom component $member(a, [a|T])$ (cf. Example 4.3.6). Intuitively, such a situation corresponds to *imposing* the characteristic tree τ on the atom $member(a, [a|T])$. Indeed, as we will see later, CA_3 can be seen as a *precise* generalisation (in P) of the atoms $member(a, [a, b])$ and $member(a, [a])$, solving the problem of Example 4.3.6.

A characteristic atom will be used to represent a possibly infinite set of atoms, called its concretisations. This is nothing really new: in "standard" partial deduction, as defined in Chapter 3, an atom A also represents a possibly infinite set of concrete atoms, namely its instances. The characteristic tree component of a characteristic atom will just act as a constraint on the instances, i.e. keeping only those instances which have a particular characteristic tree. This is captured by the following definition.

Definition 5.1.3 (concretisation) Let (A, τ_A) be a characteristic atom and P a program. An atom B is a *precise concretisation* of (A, τ_A) (in P)[1] iff B is an instance of A and, for some unfolding rule U, $chtree(\leftarrow B, P, U) = \tau_A$. An atom is a *concretisation* of (A, τ_A) (in P) iff it is an instance of a precise concretisation of (A, τ_A) (in P).
We denote the set of concretisations of (A, τ_A) in P by $\gamma_P(A, \tau_A)$.

A characteristic atom with a non-empty set of concretisations in P will be called *well-formed* in P or a *P-characteristic atom*. We will from now on usually restrict our attention to P-characteristic atoms. In particular, the partial deduction algorithm presented later on in Section 5.3 will only bring forth P-characteristic atoms, where P is the original program to be specialised.

Example 5.1.4 Take the characteristic atom $CA_3 = (member(a, [a|T]), \tau)$ with $\tau = \{\langle 1 \circ 1 \rangle\}$ from Example 5.1.2 and take the *member* program P from Example 4.2.5. The atoms $member(a, [a])$ and $member(a, [a, b])$ are precise concretisations of CA_3 in P (see Figure 4.3). Also, neither $member(a, [a|T])$ nor $member(a, [a, a])$ are concretisations of CA_3 in P. Finally, observe that CA_3 is a P-characteristic atom while for instance $(member(a, [a|T]), \{\langle 2 \circ 5 \rangle\}$ or even $(member(a, [a|T]), \{\langle 1 \circ 2 \rangle\}$ are not.

Example 5.1.5 Let P be the following simple program:

[1] If P is clear from the context, we will not explicitly mention it.

(1) $p(a, Y) \leftarrow$
(2) $p(X, Y) \leftarrow \neg q(Y), q(X)$
(3) $q(b) \leftarrow$

Let $\tau = \{\langle 1 \circ 1\rangle, \langle 1 \circ 2, 1 \circ q, 1 \circ 3\rangle\}$ and $CA = (p(X, Y), \tau)$. Then $p(X, a)$ and $p(X, c)$ are precise concretisations of CA in P (see Figure 5.1) and neither $p(b, b)$, $p(X, b)$, $p(a, Y)$ nor $p(X, Y)$ are concretisations of CA in P. Also $p(b, a)$ and $p(a, a)$ are both concretisations of CA in P, but they are not precise concretisations. Observe that the selection of the negative literal $\neg q(Y)$, corresponding to $1 \circ q$ in τ, is unsafe for $\leftarrow p(X, Y)$, but that for any concretisation of CA in P the corresponding derivation step is safe and succeeds (i.e. the negated atom is ground and fails finitely, see e.g. the SLDNF-tree for $P \cup \{\leftarrow p(X, a)\}$ in Figure 5.1).

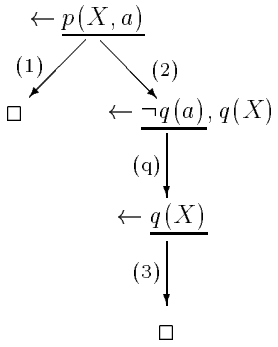


Figure 5.1: SLDNF-tree for Example 5.1.5

Note that by definition, the set of concretisations associated with a characteristic atom is *downwards closed* (or *closed under substitution*).[2] This observation justifies the following definition.

Definition 5.1.6 (unconstrained characteristic atom) We will call a P-characteristic atom (A, τ_A) which has A as one of its concretisations (in P) *unconstrained* (in P).

The concretisations of an unconstrained characteristic atom (A, τ_A) are identical to the instances of the ordinary atom A (because the concretisations are downwards closed and $\gamma_P(A, \tau)$ contains no atom strictly more

[2] In other words a characteristic atom can (almost) be seen as a *type* in the sense of [11] (where a type is defined to be a decidable set of terms closed under substitution).

general than A). In still other words, characteristic atoms along with Definition 5.1.3 provide a proper generalisation of the way atoms are used in the standard partial deduction approach.

5.1.2 Generating resultants

We will now address the generation of resultants for characteristic atoms. The simplest approach is just to unfold the atom A of a characteristic atom (A, τ) precisely as indicated by τ. As illustrated in Example 5.1.5 above, this might lead to the selection of non-ground negative literals. This will not turn out to be a problem however, because, as we will prove later, the corresponding derivations for any concretisation of (A, τ) will be safe and the selected negative literals will succeed.

We need the following definition in order to formalise the resultants associated with a characteristic atom.

Definition 5.1.7 (generalised SLDNF-derivation)
A *generalised SLDNF-derivation* is either composed of SLDNF-derivation steps or of derivation steps in which a non-ground negative literal is selected and removed. Generalised SLDNF-derivations will be called *unsafe* if they contain steps of the latter kind and *safe* if they do not.

Most of the definitions for ordinary SLDNF-derivations, like the associated characteristic path and resultant, carry over to generalised derivations. Observe that a generalised SLDNF-derivation can be seen as a pseudo SLDNF-derivation (cf. Definition 2.3.14) which is allowed to be incomplete.

We first define a set of possibly unsafe generalised SLDNF-derivations associated with a characteristic atom:

Definition 5.1.8 ($D_P(A, \tau)$) Let P be a program and (A, τ) a P-characteristic atom. If $\tau \neq \{\langle\rangle\}$ then $D_P(A, \tau)$ is the set of all generalised SLDNF-derivations of $P \cup \{\leftarrow A\}$ such that their characteristic paths are in τ. If $\tau = \{\langle\rangle\}$ then $D_P(A, \tau)$ is the set of all non-failing SLD-derivations of $P \cup \{\leftarrow A\}$ of length 1.[3]

Note that the derivations in $D_P(A, \tau)$ are necessarily finite and non-failing (because (A, τ) is a P-characteristic atom, see also Lemma 5.2.12).

We will call a P-characteristic atom (A, τ) *safe* (in P) iff all derivations in $D_P(A, \tau)$ are safe. Note that an unconstrained characteristic atom in P is also inevitably safe in P (because A must be a precise concretisation of

[3] Just like in Definition 3.2.6 of standard partial deduction, we want to construct only non-trivial SLDNF-trees for $P \cup \{\leftarrow A\}$ to avoid the problematic resultant $A \leftarrow A$.

(A, τ) we have that τ is a characteristic tree of A and thus all derivations in $D_P(A, \tau)$ are ordinary SLDNF-derivations and therefore safe).

Based on the definition of $D_P(A, \tau)$, we can now define the resultants, and hence the partial deduction, associated with characteristic atoms:

Definition 5.1.9 (partial deduction of (A, τ)) Let P be a program and (A, τ) a P-characteristic atom. Let $\{D_1, \ldots, D_n\}$ be the generalised SLDNF-derivations in $D_P(A, \tau)$ and let $\leftarrow G_1, \ldots, \leftarrow G_n$ be the goals in the leaves of these derivations. Let $\theta_1, \ldots, \theta_n$ be the computed answers of the derivations from $\leftarrow A$ to $\leftarrow G_1, \ldots, \leftarrow G_n$ respectively. Then the set of resultants $\{A\theta_1 \leftarrow G_1, \ldots, A\theta_n \leftarrow G_n\}$ is called the *partial deduction of* (A, τ) *in* P.
Every atom occurring in some of the G_i will be called a *leaf atom (in P)* of (A, τ). We will denote the set of such leaf atoms by $leaves_P(A, \tau)$.

Example 5.1.10 The partial deduction of $(member(a, [a|T]), \{\langle 1 \circ 1 \rangle\})$ in the program P of Example 4.2.5 is $\{member(a, [a|T]) \leftarrow\}$. Note that it is different from any set of resultants that can be obtained for the ordinary atom $member(a, [a|T])$. However, as we will prove below, the partial deduction is correct for any concretisation of $(member(a, [a|T]), \{\langle 1 \circ 1 \rangle\})$.

Example 5.1.11 The partial deduction P' of $(p(X, Y), \tau)$ with $\tau = \{\langle 1 \circ 1 \rangle, \langle 1 \circ 2, 1 \circ q, 1 \circ 3 \rangle\}$ of Example 5.1.5 is:

(1') $p(a, Y) \leftarrow$
(2') $p(b, Y) \leftarrow$

Note that using P' instead of P is correct for e.g. the concretisations $p(b, a)$ or $p(X, a)$ of $(p(X, Y), \tau)$ but not for $p(b, b)$ or $p(X, b)$ which are not concretisations of $(p(X, Y), \tau)$.

We can now generate partial deductions not for sets of atoms, but for sets of *characteristic* atoms. As such, the same atom A might occur in several characteristic atoms but with different associated characteristic trees. This means that renaming, as a way to ensure independence, becomes even more compelling than in the standard partial deduction setting (cf. Section 3.3.2).

In addition to renaming, we will also incorporate argument filtering, leading to the following definition.

Definition 5.1.12 (atomic renaming, renaming function) An *atomic renaming* α for a set $\tilde{\mathcal{A}}$ of characteristic atoms is a mapping from $\tilde{\mathcal{A}}$ to atoms such that

- for each $(A, \tau) \in \tilde{\mathcal{A}}$, $vars(\alpha((A, \tau))) = vars(A)$;

- for $CA, CA' \in \tilde{\mathcal{A}}$, such that $CA \neq CA'$, the predicate symbols of $\alpha(CA)$ and $\alpha(CA')$ are distinct (but not necessarily fresh, in the sense that they can occur in $\tilde{\mathcal{A}}$).

Let P be a program. A *renaming function* ρ_α for $\tilde{\mathcal{A}}$ in P based on α is a mapping from atoms to atoms such that:

$\rho_\alpha(A) = \alpha((A', \tau'))\theta$ for some $(A', \tau') \in \tilde{\mathcal{A}}$ with $A = A'\theta$ and $A \in \gamma_P(A', \tau')$.

We leave $\rho_\alpha(A)$ undefined if A is not a concretisation in P of an element in $\tilde{\mathcal{A}}$. A renaming function ρ_α can also be applied to a first-order formula, by applying it individually to each atom of the formula.

Note that, if the sets of concretisations of two or more elements in $\tilde{\mathcal{A}}$ overlap, then ρ_α must make a choice for the atoms in the intersection and several renaming functions based on the same α exist.

Definition 5.1.13 (partial deduction wrt $\tilde{\mathcal{A}}$) Let P be a program, $\tilde{\mathcal{A}} = \{(A_1, \tau_1), \ldots, (A_n, \tau_n)\}$ a finite set of P-characteristic atoms and ρ_α a renaming function for $\tilde{\mathcal{A}}$ in P based on the atomic renaming α. For each $i \in \{1, \ldots, n\}$, let R_i be the partial deduction of (A_i, τ_i) in P. Then the program $\{\alpha((A_i, \tau_i))\theta \leftarrow \rho_\alpha(Bdy) \mid A_i\theta \leftarrow Bdy \in R_i \wedge 1 \leq i \leq n \wedge \rho_\alpha(Bdy) \text{ is defined}\}$ is called the *partial deduction of P wrt $\tilde{\mathcal{A}}$ and ρ_α*.

Example 5.1.14 Let P be the following program:

(1) $member(X, [X \mid T]) \leftarrow$
(2) $member(X, [Y \mid T]) \leftarrow member(X, T)$
(3) $t \leftarrow member(a, [a]), member(a, [a, b])$

Let $\tau = \{\langle 1 \circ 1 \rangle\}$, $\tau' = \{\langle 1 \circ 3 \rangle\}$ and let $\tilde{\mathcal{A}} = \{(member(a, [a \mid T]), \tau), (t, \tau')\}$. Also let $\alpha((member(a, [a \mid T]), \tau)) = m_1(T)$ and $\alpha((t, \tau')) = t$. Because the concretisations in P of the elements in $\tilde{\mathcal{A}}$ are disjoint there exists only one renaming function ρ_α based on α.
Notably $\rho_\alpha(\leftarrow member(a, [a]), member(a, [a, b])) = \leftarrow m_1([]), m_1([b])$ because both atoms are concretisations of $(member(a, [a \mid T]), \tau)$. Therefore the partial deduction of P wrt $\tilde{\mathcal{A}}$ and ρ_α is:[4]

(1') $m_1(X) \leftarrow$
(2') $t \leftarrow m_1([]), m_1([b])$

Note that in Definition 5.1.13 the original program P is completely "thrown away". This is actually what a lot of practical partial evaluators for functional or logic programming languages do, but is dissimilar to the

[4] The FAR filtering algorithm of Chapter 11 can be used to further improve the specialised program by removing the redundant argument of m_1.

Lloyd and Shepherdson framework [185] (cf. Definition 3.2.6). However, there is no fundamental difference between these two approaches: keeping part of the original program can be simulated in our approach by using unconstrained characteristic atoms of the form $(A, \{\langle\rangle\})$ combined with a renaming α such that $\alpha((A, \{\langle\rangle\})) = A$.

5.2 Correctness results

Let us first rephrase the coveredness condition of Chapter 3 in the context of characteristic atoms. This definition will ensure that the renamings, applied for instance in Definition 5.1.13, are always defined.

Definition 5.2.1 (P-covered) Let P be a program and $\tilde{\mathcal{A}}$ a set of characteristic atoms. Then $\tilde{\mathcal{A}}$ is called *P-covered* iff for every characteristic atom in $\tilde{\mathcal{A}}$, each of its leaf atoms (in P) is a concretisation in P of a characteristic atom in $\tilde{\mathcal{A}}$.
Also, a goal G is *P-covered by $\tilde{\mathcal{A}}$* iff every atom A occurring in G is a concretisation in P of a characteristic atom in $\tilde{\mathcal{A}}$.

Note that we use the term coveredness instead of closedness here because, as mentioned above, the original program is (usually) thrown away, i.e. our approach already focusses its attention on only part of the program and is thus closer to coveredness than closedness. Also, the move from closedness to coveredness was necessary in the Lloyd and Shepherdson framework [185] (c.f. Example 3.2.12) whenever unreachable atoms in the original program were not instances of a partially deduced atom. Indeed, because the original program could not be thrown away, there was no way to apply the standard correctness Theorem 3.2.9 in such situations. This problem does not arise in the context of Definition 5.1.13, because we can freely decide which parts of the original program we carry over and which ones we throw away.

The main correctness result for partial deduction with characteristic atoms is as follows:

Theorem 5.2.2 Let P be a normal program, G a goal, $\tilde{\mathcal{A}}$ any finite set of P-characteristic atoms and P' the partial deduction of P wrt $\tilde{\mathcal{A}}$ and some ρ_α. If $\tilde{\mathcal{A}}$ is P-covered and if G is P-covered by $\tilde{\mathcal{A}}$ then the following hold:

1. $P' \cup \{\rho_\alpha(G)\}$ has an SLDNF-refutation with computed answer θ iff $P \cup \{G\}$ does.

2. $P' \cup \{\rho_\alpha(G)\}$ has a finitely failed SLDNF-tree iff $P \cup \{G\}$ does.

We will prove this theorem in three successive stages.

1. First, in Subsection 5.2.1, we will restrict ourselves to unconstrained characteristic atoms. This will allow us to reuse the correctness results for standard partial deduction with renaming in a rather straightforward manner.

2. In Subsection 5.2.2 we will then move on to safe characteristic atoms. Their partial deductions can basically be obtained from partial deductions for unconstrained characteristic atoms by removing certain clauses. We will show that these clauses can be safely removed without affecting computed answers or finite failure.

3. Finally, in Subsection 5.2.3 we will allow any characteristic atom. Basically, the associated partial deductions can be obtained from partial deductions for safe characteristic atoms by removing negative literals from the clauses. We will establish correctness by showing that, for all concrete executions, these negative literals will be ground and succeed.

The reader not interested in the details of the correctness proof can immediately continue with Subsection 5.3 starting on page 86.

5.2.1 Correctness for unconstrained characteristic atoms

If $\tilde{\mathcal{A}}$ is a set of unconstrained characteristic atoms, a partial deduction wrt $\tilde{\mathcal{A}}$ can be seen as a standard partial deduction with renaming. We will make use of this observation to prove the following theorem.

Theorem 5.2.3 Let P' be a partial deduction of P wrt $\tilde{\mathcal{A}}$ and ρ_α such that $\tilde{\mathcal{A}}$ is a finite set of *unconstrained* P-characteristic atoms and such that $\tilde{\mathcal{A}}$ is P-covered and G is P-covered by $\tilde{\mathcal{A}}$. Then:

1. $P' \cup \{\rho_\alpha(G)\}$ has an SLDNF-refutation with computed answer θ iff $P \cup \{G\}$ does.

2. $P' \cup \{\rho_\alpha(G)\}$ has a finitely failed SLDNF-tree iff $P \cup \{G\}$ does.

Proof First note that the P-coveredness conditions on $\tilde{\mathcal{A}}$ and G ensure that the renamings performed to obtain P' (according to Definition 5.1.13), as well as the renaming $\rho_\alpha(G)$, are defined (because all the atoms in G, as well as all the leaf atoms of $\tilde{\mathcal{A}}$, are concretisations of elements in $\tilde{\mathcal{A}}$). The result then follows in a rather straightforward manner from the Theorems

3.5 and 4.11 in [17]. In [17] the renaming has been split into 2 phases: one which does just the renaming to ensure independence (called partial deduction with dynamic renaming; correctness of this phase is proven in Theorem 3.5 of [17]) and one which does the filtering (called post-processing renaming; the correctness of this phase is proven in Theorem 4.11 of [17]).

To apply these results we simply have to notice that:

- P' corresponds to partial deduction with dynamic renaming and post-processing renaming for the multiset of atoms $\mathcal{A} = \{A \mid (A, \tau) \in \tilde{\mathcal{A}}\}$ (indeed the same atom could in principle occur in several characteristic atoms; this is not a problem however, as the results in [17] carry over to multisets of atoms — alternatively one could add an extra unused argument to P', $\rho_\alpha(G)$ and $\mathcal{A}$ and then place different variables in that new position to transform the multiset $\mathcal{A}$ into an ordinary set).

- $P' \cup \{\rho_\alpha(G)\}$ is $\mathcal{A}$-covered (cf. Definition 3.2.11 in Chapter 3) because $\tilde{\mathcal{A}}$ is P-covered and G is P-covered by $\tilde{\mathcal{A}}$ (and because the original program P is unreachable in the predicate dependency graph from within P' or within $\rho_\alpha(G)$).

Three minor technical issues have to be addressed in order to reuse the theorems from [17]:

- Theorem 3.5 of [17] requires that no renaming be performed on G, i.e. $\rho_\alpha(G)$ must be equal to G. However, without loss of generality, we can assume that the top-level query is the unrenamed atom $new(X_1, \ldots, X_k)$, where new is a fresh predicate symbol and where $vars(G) = \{X_1, \ldots, X_k\}$. We then just have to add the clause $new(X_1, \ldots, X_k) \leftarrow Q$, where $G =\!\leftarrow Q$, to the initial program. Trivially the query $\leftarrow new(X_1, \ldots, X_k)$ and G are equivalent wrt c.a.s. and finite failure (see also Lemma 2.2 in [99]).

- Theorem 4.11 of [17] requires that G contains no variables or predicates in $\mathcal{A}$. The requirement about the variables is not necessary in our case because we do not base our renaming on the *mgu*. The requirement about the predicates is another way of ensuring that $\rho_\alpha(G)$ must be equal to G, which can be circumvented in a similar way as for the point above.

- Theorems 3.5 and 4.11 of [17] require that the predicates of the renaming do not occur in the original P. Our Definition 5.1.12 does not require this. This is of no importance as the original program is always "completely thrown away" in our approach. We can still apply

these theorems by using an intermediate renaming ρ' which satisfies the requirements of Theorems 3.5 and 4.11 of [17] and then applying an additional one step post-processing renaming ρ'', with $\rho_\alpha = \rho'\rho''$, along with an extra application of Theorem 4.11 of [17].

□

5.2.2 Correctness for safe characteristic atoms

We first need the following adaptation of Lemma 4.12 from [185] (we just formalise the use in [185] of "corresponding SLDNF-derivation" in terms of characteristic paths).

Lemma 5.2.4 Let R be the resultant of a finite (possibly incomplete) SLDNF-derivation of $P \cup \{\leftarrow A\}$ whose characteristic path is δ. If $\leftarrow A\theta$ resolves with R giving the resultant R_A then there exists a finite (possibly incomplete) SLDNF-derivation of $P \cup \{\leftarrow A\theta\}$ whose characteristic path is δ and whose resultant is R_A.

The following lemma is based upon Lemma 5.2.4 and will prove useful later on. It establishes a link between SLD$^+$-derivations in the unrenamed specialised program and the original one.

Lemma 5.2.5 Let P be a program, G a goal, $\tilde{\mathcal{A}}$ a set of *safe* characteristic atoms and P'' the union of partial deductions $R_{(A,\tau)}$ (in P), one for every element (A,τ) of $\tilde{\mathcal{A}}$. Let D be a finite SLD$^+$-derivation of $P'' \cup \{G\}$ with computed answer θ and resultant R and such that every selected atom A', which is resolved with a clause in $R_{(A,\tau)}$, is a concretisation of (A,τ). Then there exists a finite SLD$^+$-derivation of $P \cup \{G\}$ with computed answer θ and resultant R.

Proof We do the proof by induction on the length of the SLD$^+$-derivation D of $P'' \cup \{G\}$.
Induction Hypothesis: Lemma 5.2.5 holds for SLD$^+$-derivations of $P'' \cup \{G\}$ with length $\leq k$.
Base Case: Let D have length 0. Then, trivially, the empty derivation of $P \cup \{G\}$ has the same resultant $G \leftarrow G$ and the same computed answer $\emptyset$.
Induction Step: Let D have length $k+1$. Let D''_k be the SLD$^+$-derivation of $P'' \cup \{G\}$ consisting of the first k steps of D. Let θ_k be the computed answer of D''_k and R_k its resultant. We can then apply the induction hypothesis to conclude that there is a SLD$^+$-derivation D_k of $P \cup \{G\}$ with the same resultant and computed answer. This also means that the resolvent — which is just the body of the resultant — of D''_k and D_k are the same.

We denote this resolvent by RG. Let A' be the atom selected at the last step of D in the resolvent RG. Let $C \in P''$ be the clause with which A' is resolved. We know that C is the resultant of a finite SLDNF-derivation of an atom $P \cup \{A\}$, where $(A, \tau) \in \tilde{\mathcal{A}}$, because $\tilde{\mathcal{A}}$ contains only safe characteristic atoms. We also know, by assumption, that A' is a concretisation of (A, τ) and therefore an instance of A. We can thus apply Lemma 5.2.4 for the last derivation step of D to conclude that we can extend D_k in a similar way, obtaining the same resolvent (the resolvent is just the body of the resultant), the same overall computed answer θ (if the head of two resultants for the same goal RG are identical then so are the c.a.s., and by composing with θ_k we obtain the same overall computed answer) and thus also the same overall resultant (because, conversely, if the c.a.s. and resolvent, for a derivation starting from the same goal G, are the same then so are the resultants). □

We will now extend Lemmas 5.2.4 and 5.2.5 and establish a more precise link between derivations in the renamed (standard) partial deduction and derivations in the original program. For that, the following concept will prove to be useful.

Definition 5.2.6 (admissible renaming) Let F and F' be first-order formulas. Let P be a program and α an atomic renaming for a set $\tilde{\mathcal{A}}$ of characteristic atoms. We call F' an *admissible renaming of* F *under* α *in* P iff there exists some renaming function ρ'_α for $\tilde{\mathcal{A}}$ in P based on α such that $F' = \rho'_\alpha(F)$.

Example 5.2.7 Let P be the *member* program (from Example 4.2.5) and let $\tilde{\mathcal{A}} = \{CA_1, CA_2\}$ with $CA_1 = (member(a, L), \{\langle 1 \circ 1 \rangle\})$ and $CA_2 = (member(a, L), \{\langle 1 \circ 1 \rangle, \langle 1 \circ 2 \rangle\})$. Let α be an atomic renaming such that $\alpha(CA_1) = mem_1(L)$ and $\alpha(CA_2) = mem_2(L)$. Then both $mem_1([a])$ and $mem_2([a])$ are admissible renamings of $member(a, [a])$ under α in P. Now $mem_2([a, a])$ is an admissible renaming of $member(a, [a, a])$ under α in P, while $mem_1([a, a])$ is not (because $member(a, [a, a])$ is not a concretisation of CA_1). Also, $member(b, [a])$ has no admissible renamings under α in P.

The following lemma establishes a link between SLD$^+$-derivation steps in the (renamed) specialised program and the unrenamed specialised program.

Lemma 5.2.8 Let P' be a partial deduction of P wrt $\tilde{\mathcal{A}}$ and ρ_α such that $\tilde{\mathcal{A}}$ is a finite set of *safe* P-characteristic atoms and such that $\tilde{\mathcal{A}}$ is P-covered and G is P-covered by $\tilde{\mathcal{A}}$.
Let $C' = \alpha((A, \tau))\theta \leftarrow \rho_\alpha(Body)$, with $(A, \tau) \in \tilde{\mathcal{A}}$, be a clause in P'

and let $C = A\theta \leftarrow Body$ be the unrenamed version of C'. Let G' be an admissible renaming of G under α in P and let γ be a substitution (which e.g. standardises C' apart wrt G'). If RG' is derived from G' and $C'\gamma$ using θ' then there exists a goal RG such that:

1. RG is derived from G and $C\gamma$ using θ' and
2. RG' is an admissible renaming of RG under α in P.

Proof Let L' be the selected atom in G' and let L be the corresponding atom in G. Because G' is an admissible renaming of G under α in P, we know that for some renaming function ρ'_α, we have $\rho'_\alpha(G) = G'$ and thus also $\rho'_\alpha(L) = L'$. Furthermore, as L' unifies with $\alpha((A,\tau))\theta\gamma$, we know that $L \in \gamma_P(A,\tau)$ and thus also $L = A\sigma$ and $L' = \alpha((A,\tau))\sigma$ for some substitution σ.
Now, as of Definition 2.3.5, we know that θ' is an idempotent and relevant *mgu* of $L' = \alpha((A,\tau))\sigma$ and $\alpha((A,\tau))\theta\gamma$. Because $vars(A) = vars(\alpha(A,\tau))$ (cf. Definition 5.1.12), we have that θ' is also an idempotent and relevant *mgu* of $L = A\sigma$ and $A\theta\gamma$. Hence, by selecting L in G for resolution with $C\gamma$, we obtain a goal RG which is derived from G and $C\gamma$ using the same θ'.
Finally, RG' is an admissible renaming of RG under α in P because:

- all atoms in $Body$ are P-covered by $\tilde{\mathcal{A}}$ (because $\tilde{\mathcal{A}}$ is P-covered) and are therefore, by Definition 5.1.12 of a renaming, admissible renamings under α in P.
- the resolvent RG' will contain, in addition to instances of atoms in G', instances of the atoms in $Body$, all of which are still P-covered by $\tilde{\mathcal{A}}$, because the set of concretisations of characteristic atoms is downwards closed. RG' is therefore still an admissible renaming of RG under α in P.

□

Observe that if the substitution γ in the above lemma standardises C' apart wrt G', then it also standardises C apart wrt G (because, by Definition 5.1.12, $vars(G) = vars(G')$ as well as $vars(C) = vars(C')$).

Usually one will call the specialised program with one specific renaming and not just with an admissible one. So one might wonder why we only prove in Lemma 5.2.8 above that RG' is an admissible renaming of RG under α in P and not that $RG' = \rho_\alpha(RG)$. The reason is that in the course of performing resolution steps, atoms might become more instantiated and applying the renaming function ρ_α on the more instantiated atom might result

in a different renaming. Take for example the set $\tilde{\mathcal{A}} = \{(p(X), \tau), (p(a), \tau')\}$ of unconstrained characteristic atoms, the goal $G = \leftarrow p(X), p(X)$ and take α such that:

- $\alpha((p(X), \tau)) = p'(X)$ and
- $\alpha((p(a), \tau')) = p_a$.

Then $\rho_\alpha(G) = \leftarrow p'(X), p'(X)$. Also assume that $\rho_\alpha(p(a)) = p_a$. Now suppose that the clause $C' = p'(a) \leftarrow$ is in the partial deduction P' wrt an original P and the set $\tilde{\mathcal{A}}$. The clause $C = p(a) \leftarrow$ is then the unrenamed version of C'. Then $RG' = \leftarrow p'(a)$ is derived from $\rho_\alpha(G)$ and C' using $\{X/a\}$. Similarly, $RG = \leftarrow p(a)$ is derived from G and C using $\{X/a\}$ (no matter which literal we select). Now $\rho_\alpha(\leftarrow p(a)) = \leftarrow p_a \not= \leftarrow p'(a)$, i.e. $RG' \not= \rho_\alpha(RG)$! However, $\leftarrow p'(a)$ is still an admissible renaming of $\leftarrow p(a)$ under α in P and Lemma 5.2.8 holds.

We now combine Lemmas 5.2.5 and 5.2.8 to establish a link between entire SLDNF-derivations in the renamed specialised program and original one. However, for the time being, we have restricted ourselves to unconstrained characteristic atoms in order to establish the result. An illustration of the following lemma can be found in Figure 5.2.

Lemma 5.2.9 Let P' be a partial deduction of P wrt $\tilde{\mathcal{A}}$ and ρ_α such that $\tilde{\mathcal{A}}$ is a finite set of *unconstrained* P-characteristic atoms and such that $\tilde{\mathcal{A}}$ is P-covered and G is P-covered by $\tilde{\mathcal{A}}$.
Let G' be an admissible renaming of G under α in P. Let D' be a finite SLDNF-derivation of $P' \cup \{G'\}$ leading to the resolvent RG' via the c.a.s. θ. Then there exists a finite SLDNF-derivation of $P \cup \{G\}$ leading to a resolvent RG via c.a.s. θ such that RG' is an admissible renaming of RG under α in P and such that RG is P-covered by $\tilde{\mathcal{A}}$.

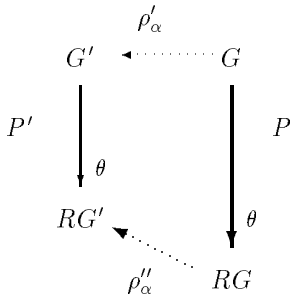


Figure 5.2: Illustrating Lemma 5.2.9

Proof First note that, if RG' is an admissible renaming of RG, RG must by definition be P-covered by $\tilde{\mathcal{A}}$.
Also note that not all resolvents of $P \cup \{G\}$ are P-covered by $\tilde{\mathcal{A}}$ — there can be some intermediate goals which have no correspondent goal in P' (because entire derivation sequences can be compressed into one resultant of P'). So it is only at some specific points that a resolvent of $P \cup \{G\}$ has a counterpart in $P' \cup \{G'\}$. We will however show in the following that, as asserted by the lemma, every derivation of $P' \cup \{G'\}$ has a counterpart in $P \cup \{G\}$.
We first do the proof for SLD^+-derivations, i.e. SLDNF-derivations in which no negative literals are selected. Let P'' be the unrenamed version of P' (i.e. the union of the partial deductions of the elements of $\tilde{\mathcal{A}}$). We can now inductively apply Lemma 5.2.8 on every step of the SLD^+-derivation D' of $P' \cup \{G'\}$ and thus obtain a SLD^+-derivation of $P'' \cup \{G\}$ with the same computed answer θ and with a resolvent RG, such that RG' is an admissible renaming of RG. Furthermore, the so obtained derivation of $P'' \cup \{G\}$ will satisfy the conditions of Lemma 5.2.5 (because every intermediate goal of this derivation can be renamed into an intermediate goal of D' and is thus P-covered by $\tilde{\mathcal{A}}$ in P, i.e. every selected atom is a concretisation of an element of $\tilde{\mathcal{A}}$). We can thus apply Lemma 5.2.5 to conclude that an SLD^+-derivation of $P \cup \{G\}$ with the same c.a.s. θ and the same resolvent RG (which is just the body of the resultant) — of which RG' is an admissible renaming — exists.
So the lemma holds for SLDNF-derivations in which no negative literals are selected.
Let us now allow the selection of a negative literal in the SLD^+-derivation D' (of $P' \cup \{G'\}$ leading to the resolvent RG'). In that case we can apply the just established result for the derivation leading up to the goal NG', in which the selected negative literal is selected and succeeds (because D' is not a failed derivation as it leads to a goal RG'). Then, because $NG' = \rho'_\alpha(NG)$ for some P-covered goal NG and some renaming function ρ'_α, and because $\tilde{\mathcal{A}}$ contains only unconstrained characteristic atoms, we can apply Theorem 5.2.3, to deduce that the negative literal must also succeed in the original program (with the same computed answer, namely the identity substitution $\emptyset$). The next resolvents, $\widetilde{NG}$ and $\widetilde{NG}'$, are obtained from NG and NG' respectively, by simply removing a negative literal at the same position. Therefore, $\widetilde{NG}'$ is still an admissible renaming of $\widetilde{NG}$. We can thus re-apply the above result for SLD^+-derivations until the end of the derivation D'. Finally, we can repeat this same reasoning inductively for any number of selected negative literals. □

In the proof of Lemma 5.2.9, we used the fact that $\tilde{\mathcal{A}}$ contained only unconstrained characteristic atoms (to show that the behaviour of the selected negative literals was preserved). We will now move to safe characteristic atoms and show that their partial deductions can be obtained from partial deductions of unconstrained characteristic atoms by removing certain clauses. This means that Lemma 5.2.9 can actually be used to show soundness of SLD$^+$-derivations for partial deductions of safe characteristic atoms. To be able to show completeness, as well as allowing the selection of negative literals, we then show that these additional clauses can be removed without affecting the finite failure behaviour.

The following lemmas will prove useful later on.

Lemma 5.2.10 Let τ be a characteristic tree. Let $\delta_1 \in \tau$ and $\delta_2 \in \tau$. If δ_2 is a prefix of δ_1 then $\delta_1 = \delta_2$.

Proof The property follows immediately from the definitions of SLDNF-trees and characteristic trees. □

Lemma 5.2.11 Let A and B be atoms and τ_A a characteristic tree of A in the program P. If B is an instance of A then there exists a characteristic tree τ_B of B in P such that $\tau_B \subseteq \tau_A$.

Proof By Definition 4.2.3, for some unfolding rule U we have that $\tau_A = chtree(\leftarrow A, P, U)$. Because B is more instantiated than A, all resolution steps for $\leftarrow A$ are either also possible for $\leftarrow B$ or they fail. Therefore, for some unfolding rule U', we have that $\tau_B = chtree(\leftarrow B, P, U') \subseteq \tau_A$. □

Lemma 5.2.12 Let τ be a characteristic tree, P a program and (A, τ) a P-characteristic atom. If (A, τ) is safe then there exists an unfolding rule such that $\tau \subseteq chtree(\leftarrow A, P, U)$.

Proof As (A, τ) is a P-characteristic atom, we must have by definition at least one precise concretisation A' whose characteristic path is τ in P. As all the derivations in $D_P(A, \tau)$ are safe, we can unfold $\leftarrow A$ in a similar way as $\leftarrow A'$. This will result in a characteristic tree τ' which contains all the paths in τ as well as possibly some additional paths (which failed for $\leftarrow A'$). □

The above Lemma 5.2.12 (also) establishes that the partial deduction $P_{(A,\tau)}$ of a safe characteristic atom (A, τ) is a subset of a partial deduction P_A of the ordinary atom A. The Lemma 5.2.14 below shows that it is correct to remove the resultants $P_A \setminus P_{(A,\tau)}$ for the concretisations of (A, τ). In the proof of this lemma we need to combine characteristic paths. A

characteristic path being a sequence, we can simply concatenate two characteristic paths δ_1, δ_2. For this we will use the standard notation $\delta_1\delta_2$ from formal language theory (cf. Appendix A.2 or [1, 128]).

We also need the following lemma from [185], where it is Lemma 4.10.

Lemma 5.2.13 (persistence of failure) Let P be a normal program and G a normal goal. If $P \cup \{G\}$ has a finitely failed SLDNF-tree of height h and there is an SLDNF-derivation from G to G_1, then $P \cup \{G_1\}$ has a finitely failed SLDNF-tree of height $\leq h$.

Lemma 5.2.14 Let P be a program and (A, τ_A), (A, τ_A^*) be safe P-characteristic atoms with $\tau_A \subseteq \tau_A^*$. Let $C \in resultants(\tau_A^*) \backslash resultants(\tau_A)$ and let $A' \in \gamma_P(A, \tau_A)$.
If $\leftarrow A'$ resolves with C to G' then G' fails finitely in P.

Proof Let δ_C be the characteristic path associated in Definition 5.1.8 (of $D_P(A, \tau)$) with C, i.e. C is the resultant of the generalised SLDNF-derivation for $P \cup \{\leftarrow A\}$ whose characteristic path is δ_C. Because (A, τ_A^*) is safe, C is even the resultant of an SLDNF-derivation (and not of an unsafe generalised one). We can therefore apply Lemma 5.2.4 to deduce that:

(1) there exists a finite SLDNF-derivation of $P \cup \{\leftarrow A'\}$ whose characteristic path is δ_C and whose resolvent is G'.

Because $C \in resultants(\tau_A^*) \setminus resultants(\tau_A)$ we know that $\delta_C \in \tau_A^*$ and $\delta_C \notin \tau_A$. Furthermore $A' \in \gamma_P(A, \tau_A)$ implies by Lemma 5.2.11 that, for some unfolding rule U, $\hat{\tau} = chtree(\leftarrow A', P, U) \subseteq \tau_A$.
If $\hat{\tau} = \emptyset$ then $\leftarrow A'$ fails finitely. Therefore, because a finite SLDNF-derivation from $\leftarrow A'$ to G' exists, we can deduce by persistence of failure (Lemma 5.2.13), that G' must also fail finitely.
If $\hat{\tau} \neq \emptyset$ then the largest prefix δ'_C of δ_C, such that for some $\hat{\gamma}$ we have $\delta'_C\hat{\gamma} \in \hat{\tau}$, must exist (the smallest one being $\langle\rangle$). By Lemma 5.2.10 we know that no proper prefix of δ_C can be in τ_A^* (because $\delta_C \in \tau_A^*$), and therefore neither in τ_A nor $\hat{\tau}$ (because $\hat{\tau} \subseteq \tau_A \subseteq \tau_A^*$). This means that $\hat{\gamma}$ must be non-empty. We also know, again by Lemma 5.2.10, that δ'_C is a proper prefix of δ_C (because $\delta_C \in \tau_A^*$ and $\delta'_C\hat{\gamma} \in \tau_A^*$), i.e. $\delta_C = \delta'_C\langle l \circ m\rangle\delta''_C$. We can also see that $\hat{\gamma} \neq \langle l \circ m\rangle\delta''_C$ because $\delta'_C\hat{\gamma} \in \hat{\tau}$ while $\delta_C \notin \hat{\tau}$. This means that there is branching immediately after δ'_C (otherwise δ'_C is not the largest prefix of δ_C such that an extension of it is in $\hat{\tau}$). We even know that in $\delta_C = \delta'_C\langle l \circ m\rangle\delta''_C$ the selected literal at position l is a positive literal (the selection of a negative literal can never lead to branching), that m is therefore a clause number and also that $\hat{\gamma} = \langle l \circ \hat{m}\rangle\hat{\gamma}'$ with $\hat{m} \neq m$. The situation is summarised in Figure 5.3.

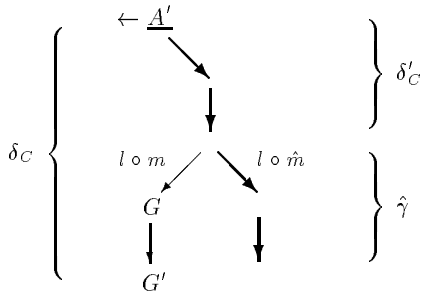


Figure 5.3: Illustrating the proof of Lemma 5.2.14

Let G be the goal obtained from the finite SLDNF-derivation of $P \cup \{\leftarrow A'\}$ whose characteristic path is $\delta'_C\langle l \circ m\rangle$ (this must exist because an SLDNF-derivation of $P \cup \{\leftarrow A'\}$ with characteristic path δ_C exists by point (1) above). In order to arrive at the characteristic tree $\hat{\tau}$ for $\leftarrow A'$ the unfolding rule U also had to reach the goal G, because $\hat{\tau}$ contains the characteristic path $\delta'_C\langle l \circ \hat{m}\rangle\hat{\gamma}'$ and G is "reached" via $\delta'_C\langle l \circ m\rangle$, a step which cannot be avoided by U if it wants to get as far as $\delta'_C\langle l \circ \hat{m}\rangle$. Furthermore, as neither $\delta'_C\langle l \circ m\rangle$ nor any extension of it are in $\hat{\tau}$ (by definition of δ'_C) this means that $\leftarrow G$ finitely fails.
Finally, as δ_C is an extension of $\delta'_C\langle l \circ m\rangle$, we know that a finite (possibly empty) SLDNF-derivation from G to G' exists and therefore, by persistence of failure (Lemma 5.2.13), G' must also fail finitely. □

We now present a correctness theorem for safe characteristic atoms.

Theorem 5.2.15 Let P be a normal program, G a goal, $\tilde{\mathcal{A}}$ a finite set of safe P-characteristic atoms and P' the partial deduction of P wrt $\tilde{\mathcal{A}}$ and some ρ_α. If $\tilde{\mathcal{A}}$ is P-covered and if G is P-covered by $\tilde{\mathcal{A}}$ then the following hold:

1. $P' \cup \{\rho_\alpha(G)\}$ has an SLDNF-refutation with computed answer θ iff $P \cup \{G\}$ does.

2. $P' \cup \{\rho_\alpha(G)\}$ has a finitely failed SLDNF-tree iff $P \cup \{G\}$ does.

Proof The basic idea of the proof is as follows.

1. First, we transform the characteristic atoms in $\tilde{\mathcal{A}}$ so as to make them unconstrained (which is possible by Lemma 5.2.12). For the so obtained partial deduction P'' we can reuse Theorem 5.2.3 to prove that P'' is totally correct.

2. By construction, P'' will be a superset of P', i.e. $P'' = P' \cup P_{New}$, and we will show, mainly using Lemma 5.2.14, that the clauses P_{New} can be safely removed without affecting the total correctness.

The details of the proof are elaborated in the following.

1. In order to make a characteristic atom (A, τ) in $\tilde{\mathcal{A}}$ unconstrained we have to add some characteristic paths to τ. We have that, for every $(A, \tau) \in \tilde{\mathcal{A}}$, the set of derivations $D_P(A, \tau)$ is safe. By Lemma 5.2.12 we then know that for some unfolding rule U: $\tau \subseteq chtree(\leftarrow A, P, U)$. Let $\tau' = chtree(\leftarrow A, P, U) \setminus \tau$ and let R_{New} be the partial deduction of (A, τ') (i.e. the unrenamed clauses that have to be added to the partial deduction of (A, τ) in order to arrive at a standard partial deduction of the unconstrained characteristic atom $(A, chtree(\leftarrow A, P, U))$). We denote by $New_{(A,\tau)}$ the following set of clauses $\{\alpha((A, \tau))\theta \leftarrow \rho_\alpha(Bdy) \mid A\theta \leftarrow Bdy \in R_{New}\}$. By adding for each $(A, \tau) \in \tilde{\mathcal{A}}$ the clauses $New_{(A,\tau)}$ to P' we obtain a partial deduction of P wrt an unconstrained set $\tilde{\mathcal{A}}'$[5] and the renaming function ρ_α (where we extend α so that $\alpha(A, \tau \cup \tau') = \alpha(A, \tau)$). Note that, every concretisation of (A, τ) is also a concretisation of $(A, \tau \cup \tau')$ (because $(A, \tau \cup \tau')$ is unconstrained, and thus *any* instance of A is a concretisation). Unfortunately, although G remains P-covered by $\tilde{\mathcal{A}}'$, $\tilde{\mathcal{A}}'$ is not necessarily P-covered any longer. The reason is that new uncovered leaf atoms can arise inside $New_{(A,\tau)}$. Let UC be these uncovered atoms. To arrive at a P-covered partial deduction we simply have to add, for every predicate symbol p of arity n occurring in UC, the characteristic atom $(p(X_1, \ldots, X_n), \langle\rangle)$ to $\tilde{\mathcal{A}}'$, where $X_1, \ldots, X_n$ are distinct variables. This will give us the new set $\tilde{\mathcal{A}}'' \supseteq \tilde{\mathcal{A}}'$ (and we extend α and ρ_α in an arbitrary manner for the elements in $\tilde{\mathcal{A}}'' \setminus \tilde{\mathcal{A}}'$). Let P'' be the partial deduction of P wrt $\tilde{\mathcal{A}}''$ and ρ_α. Now $\tilde{\mathcal{A}}''$ is trivially P-covered and we can apply the correctness Theorem 5.2.3 to deduce that the computations of $P'' \cup \{\rho_\alpha(G)\}$ are totally correct wrt the computations in $P \cup \{G\}$.

2. Note that, by construction, we have that $P' \subseteq P''$. We will now show that by removing the clauses $P_{New} = P'' \setminus P'$ we do not lose any computed answer nor do we remove any infinite failure, i.e.:

[5] As in the proof of Theorem 5.2.3, $\tilde{\mathcal{A}}'$ might actually be a multiset. However, this poses no problems, as all results so far also hold for multisets of characteristic atoms. Alternatively, one could add an extra unused argument to all predicates and ensure that all elements in $\tilde{\mathcal{A}}$ have a different variable in that position, thus guaranteeing that $\tilde{\mathcal{A}}'$ is an ordinary set.

- $P'' \cup \{\rho_\alpha(G)\}$ has an SLDNF-refutation with computed answer θ iff $P' \cup \{\rho_\alpha(G)\}$ does.
- $P'' \cup \{\rho_\alpha(G)\}$ has a finitely failed SLDNF-tree iff $P' \cup \{\rho_\alpha(G)\}$ does.

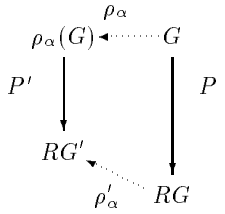
Combined with point 1., this is sufficient to establish that P' is also totally correct wrt P. We do the proof by induction of the rank of the SLDNF-derivations and trees.

Induction Hypothesis:

- if $P'' \cup \{\rho_\alpha(G)\}$ has an SLDNF-refutation of rank k with computed answer θ then $P' \cup \{\rho_\alpha(G)\}$ has an SLDNF-refutation with computed answer θ.
- if $P' \cup \{\rho_\alpha(G)\}$ has an SLDNF-refutation of rank k with computed answer θ then $P'' \cup \{\rho_\alpha(G)\}$ has an SLDNF-refutation (of rank k) with computed answer θ.
- if $P'' \cup \{\rho_\alpha(G)\}$ has a finitely failed SLDNF-tree of rank k then $P' \cup \{\rho_\alpha(G)\}$ has a finitely failed SLDNF-tree (of rank k).
- if $P' \cup \{\rho_\alpha(G)\}$ has a finitely failed SLDNF-tree of rank k then $P'' \cup \{\rho_\alpha(G)\}$ has a finitely failed SLDNF-tree.

Base Case: Because $P' \subseteq P''$, we have that whenever $P'' \cup \{\rho_\alpha(G)\}$ has a finitely failed SLD$^+$-tree so does $P' \cup \{\rho_\alpha(G)\}$, and whenever $P' \cup \{\rho_\alpha(G)\}$ has a SLD$^+$-refutation with computed answer θ so does $P'' \cup \{\rho_\alpha(G)\}$. We now show that every SLD$^+$-derivation of $P'' \cup \{\rho_\alpha(G)\}$, which uses at least a clause in $P'' \setminus P'$, fails finitely. This will ensure that $P'' \cup \{\rho_\alpha(G)\}$ cannot have any additional computed answer and that it fails finitely iff $P' \cup \{\rho_\alpha(G)\}$ does.

Let D be an SLD$^+$-derivation of $P'' \cup \{\rho_\alpha(G)\}$ which uses at least one clause in $P'' \setminus P'$. Let D' be the largest prefix SLD$^+$-derivation of D such that D' uses only clauses within P'. Let RG' be the last goal of D'. We can apply Lemma 5.2.9 to deduce that there exists an SLD$^+$-derivation of $P \cup \{G\}$ leading to RG such that RG' is an admissible renaming of RG under α in P and such that RG is P-covered by $\tilde{\mathcal{A}}''$. Let $RG' = \rho'_\alpha(RG)$ and let $\rho'_\alpha(p(\bar{t}))$ be the literal selected in RG' by D (i.e. the next step after performing D').



Because RG' is an admissible renaming of RG, we have $p(\bar{t}) \in \gamma_P(A, \tau)$ where $\rho'_\alpha(p(\bar{t})) = \alpha((A, \tau))\sigma'$ for some σ', ρ'_α and (A, τ). Furthermore, we

now that $p(\bar{t}) \in \gamma_P(A, \tau')$ for some $(A, \tau') \in \tilde{\mathcal{A}}$ with $\tau' \subseteq \tau$, because the elements in $\tilde{\mathcal{A}}'' \setminus \tilde{\mathcal{A}}'$ cannot be reached from the clauses in P'. We can therefore apply Lemma 5.2.14 to deduce that $\leftarrow p(\bar{t})$, and therefore also RG, fails finitely in P. We can now apply the correctness Theorem 5.2.3 for the goal $\leftarrow p(\bar{t})$ (it is possible to apply this theorem because $\leftarrow p(\bar{t})$ is P-covered by $\tilde{\mathcal{A}}$) to deduce that $\leftarrow \rho_\alpha(p(\bar{t}))$ also fails finitely in P''.
Induction Step: Let us now prove the hypothesis for SLDNF-derivations and trees of rank $k+1$. Because $P' \subseteq P''$, and by the induction hypothesis (because all sub-derivations and sub-trees for negative literals have rank $\leq k$), we have that whenever $P'' \cup \{\rho_\alpha(G)\}$ has a finitely failed SLDNF-tree of rank $k+1$ so does $P' \cup \{\rho_\alpha(G)\}$, and whenever $P' \cup \{\rho_\alpha(G)\}$ has a SLDNF-refutation or rank $k+1$ with computed answer θ so does $P'' \cup \{\rho_\alpha(G)\}$. Similar to the base case, if we show that every SLDNF-derivation D of rank $k+1$ of $P'' \cup \{\rho_\alpha(G)\}$ which uses at least a clause in $P'' \setminus P'$ fails finitely, then $P'' \cup \{\rho_\alpha(G)\}$ cannot have any additional computed answer (over P) and it fails finitely iff $P' \cup \{\rho_\alpha(G)\}$ does. This can be done in exactly the same manner as for the base case, because Lemmas 5.2.9, 5.2.14 and Theorem 5.2.3 hold for SLDNF-derivations and trees of any rank. □

5.2.3 Correctness for unrestricted characteristic atoms

We are finally in the position to prove the general correctness Theorem 5.2.2 for partial deductions of safe and unsafe P-characteristic atoms.
Proof of Theorem 5.2.2 For every $(A, \tau) \in \tilde{\mathcal{A}}$ let us remove the derivation steps from τ which correspond to the selection of a non-ground negative literal in $D_P(A, \tau)$, resulting in a modified characteristic tree τ'. Note that, trivially, any concretisation of (A, τ) is also a concretisation of (A, τ') (if we can build and SLDNF-tree for $P \cup \{\leftarrow A\theta\}$ with characteristic tree τ we can also construct an SLDNF-tree with characteristic tree τ' by simply not selecting the offending negative literals). By substituting (A, τ') for (A, τ) in $\tilde{\mathcal{A}}$ we obtain a set $\tilde{\mathcal{A}}'$ of safe characteristic atoms.[6] Every clause in the partial deduction wrt $\tilde{\mathcal{A}}'$ and ρ_α can be obtained from a clause of P' by adding the negative literals which are no longer selected. So, just like in the proof of Theorem 5.2.15, we might have to add characteristic atoms to $\tilde{\mathcal{A}}'$ (to cover all these negative literals), in order to arrive at a P-covered set, giving the new set of characteristic atoms $\tilde{\mathcal{A}}'' \supseteq \tilde{\mathcal{A}}'$. As in the proof of

[6] As in the proofs of Theorems 5.2.3 and Theorem 5.2.15, $\tilde{\mathcal{A}}'$ might actually be a multiset. Again, this poses no problems, as all results so far also hold for multisets of characteristic atoms. Alternatively, one could add an extra unused argument to all predicates and ensure that all elements in $\tilde{\mathcal{A}}$ have a different variable in that position, thus guaranteeing that $\tilde{\mathcal{A}}'$ is an ordinary set.

Theorem 5.2.15 we also extend α so that $\alpha(A,\tau') = \alpha(A,\tau)$ (and we extend α and ρ_α in an arbitrary manner for the elements in $\tilde{\mathcal{A}}'' \setminus \tilde{\mathcal{A}}'$).
Let P'' be the partial deduction of P wrt $\tilde{\mathcal{A}}''$ and ρ_α. We can apply Theorem 5.2.15 to deduce that the computations of $P'' \cup \{\leftarrow \rho_\alpha(G)\}$ are totally correct wrt $P \cup \{\leftarrow G\}$.
We will now establish total correctness of P' (wrt P) by proving that:

- $P'' \cup \{\rho_\alpha(G)\}$ has an SLDNF-refutation with computed answer θ iff $P' \cup \{\rho_\alpha(G)\}$ does.
- $P'' \cup \{\rho_\alpha(G)\}$ has a finitely failed SLDNF-tree iff $P' \cup \{\rho_\alpha(G)\}$ does.

We will do this proof by induction on the rank of the SLDNF-refutations and trees.
Induction Hypothesis:

- if $P'' \cup \{\rho_\alpha(G)\}$ has an SLDNF-refutation of rank k with computed answer θ then $P' \cup \{\rho_\alpha(G)\}$ has an SLDNF-refutation (of rank k) with computed answer θ.
- if $P' \cup \{\rho_\alpha(G)\}$ has an SLDNF-refutation of rank k with computed answer θ then $P'' \cup \{\rho_\alpha(G)\}$ has an SLDNF-refutation with computed answer θ.
- if $P'' \cup \{\rho_\alpha(G)\}$ has a finitely failed SLDNF-tree of rank k then $P' \cup \{\rho_\alpha(G)\}$ has a finitely failed SLDNF-tree.
- if $P' \cup \{\rho_\alpha(G)\}$ has a finitely failed SLDNF-tree of rank k then $P'' \cup \{\rho_\alpha(G)\}$ has a finitely failed SLDNF-tree (of rank k).

Base Case: For every SLD$^+$-derivation in P' there exists a corresponding SLD$^+$-derivation in P'', with however additional negative literals in the resolvent. Thus, every computed answer of $P'' \cup \{\rho_\alpha(G)\}$ (via an SLD$^+$-refutation) is also a computed answer of $P' \cup \{\rho_\alpha(G)\}$ (via an SLD$^+$-refutation). Also, if $P' \cup \{\rho_\alpha(G)\}$ has a finitely failed SLD$^+$-tree then $P'' \cup \{\rho_\alpha(G)\}$ also has a finitely failed SLD$^+$-tree. We will now show that these additional negative literals always succeed. Thus, if for every derivation in P'' we impose the (fair) condition that the additional negative literals are immediately selected, we can establish a one to one correspondence between derivations in P'' and P'. Also, the clauses that had to be added for coveredness are not reachable within P' and can therefore also be removed. So by showing that the additional negative literals always succeed, we establish the base case.
Let G'_1 be a goal which is an admissible renaming (under α in P) of a goal G_1 for P (i.e. $G'_1 = \rho'_\alpha(G_1)$ for some ρ'_α — only such goals can occur for derivations inside P') which resolves with a clause C_δ (constructed for the characteristic path δ) in P' and with selected literal $\rho'_\alpha(p(\bar{t}))$ leading to a resolvent RG'. Then G'_1 also resolves with a clause C' in P'', under the same selected literal, leading to a resolvent RG'' which has the same atoms as RG' plus possibly some additional negative literals N. Because

G'_1 is an admissible renaming of G_1 we know that $p(\bar{t}) \in \gamma_P(A, \tau)$ where $\rho'_\alpha(p(\bar{t})) = \alpha((A, \tau))\sigma'$ for some σ'. Because $p(\bar{t})$ is a concretisation of an element in $\tilde{\mathcal{A}}$ we know that it must be the instance of a precise concretisation A. For this concretisation A there exists a characteristic tree which contains the characteristic path δ and for which the negative literals corresponding to N are ground and succeed. Therefore, because $p(\bar{t})$ is an instance of A, the literals in N are identical to the ones selected for $P \cup \{\leftarrow A\}$ and are thus also ground and succeed. We can thus immediately (by Theorem 5.2.15) select them in P'' and can thus construct a corresponding derivation in P'.
Induction Step: The induction step is very similar to the base case and can basically be obtained by replacing SLD$^+$ by SLDNF of rank $k+1$. □

5.3 A set based algorithm and its termination

In this section, we present a first, simple (set based) algorithm for partial deduction through characteristic atoms. We will prove correctness as well as termination of this algorithm, given certain conditions.

We first define an abstraction operator which, by definition, preserves the characteristic trees.

Definition 5.3.1 ($chmsg(.), \tilde{\mathcal{A}}|_\tau$) Let $\tilde{\mathcal{A}}$ be a set of characteristic atoms. Also let, for every characteristic tree τ, $\tilde{\mathcal{A}}|_\tau$ be defined as $\tilde{\mathcal{A}}|_\tau = \{A \mid (A, \tau) \in \tilde{\mathcal{A}}\}$. The operator $chmsg()$ is defined as:

$$chmsg(\tilde{\mathcal{A}}) = \{(msg(\tilde{\mathcal{A}}|_\tau), \tau) \mid \tau \text{ is a characteristic tree } \}.$$

In other words, only one characteristic atom per characteristic tree is allowed in the resulting abstraction. For example, given the set $\tilde{\mathcal{A}} = \{(p(a), \{\langle 1 \circ 1\rangle\}), (p(b), \{\langle 1 \circ 1\rangle\})\}$, we obtain the abstraction $chmsg(\tilde{\mathcal{A}}) = \{(p(X), \{\langle 1 \circ 1\rangle\})\}$.

Definition 5.3.2 ($chatom$) Let A be an ordinary atom, U an unfolding rule and P a program. We then define:

$$chatom(A, P, U) = (A, \tau), \text{ where } \tau = chtree(\leftarrow A, P, U).$$

We extend $chatom$ to sets of atoms:

$$chatoms(\tilde{\mathcal{A}}, P, U) = \{chatom(A, P, U) \mid A \in \tilde{\mathcal{A}}\}.$$

Note that A is a precise concretisation of $chatom(A, P, U)$.

The following algorithm for partial deduction with characteristic atoms is parametrised by an unfolding rule U, thus leaving the particulars of local control unspecified. Recall that $leaves_P(A, \tau_A)$ represents the leaf atoms of (A, τ_A) (see Definition 5.1.9).

Algorithm 5.3.3

Input
a program P and a goal G
Output
a specialised program P'
Initialisation
$k := 0$; $\tilde{\mathcal{A}}_0 := chatoms(\{A \mid A \text{ is an atom in } G\}, P, U)$;
repeat
$\tilde{\mathcal{L}}_k := chatoms(\{leaves_P(A, \tau_A) \mid (A, \tau_A) \in \tilde{\mathcal{A}}_k\}, P, U)$;
$\tilde{\mathcal{A}}_{k+1} := chmsg(\tilde{\mathcal{A}}_k \cup \tilde{\mathcal{L}}_k)$;
$k := k + 1$;
until $\tilde{\mathcal{A}}_k = \tilde{\mathcal{A}}_{k+1}$ (modulo variable renaming)
$\tilde{\mathcal{A}} := \tilde{\mathcal{A}}_k$;
$P' :=$ a partial deduction of P wrt $\tilde{\mathcal{A}}$ and some renaming function ρ_α;

Let us illustrate the operation of Algorithm 5.3.3 on Example 4.2.5.

Example 5.3.4 Let $G = \leftarrow member(a, [a]), member(a, [a, b])$ and P be the program from Example 4.2.5. Also let $chtree(\leftarrow member(a, [a]), P, U) = chtree(\leftarrow member(a, [a, b]), P, U) = \{\langle 1 \circ 1 \rangle\} = \tau$ (see Figure 4.3 for the corresponding SLD-trees). The algorithm operates as follows:

1. $\tilde{\mathcal{A}}_0 = \{(member(a, [a]), \tau), (member(a, [a, b]), \tau)\}$
2. $leaves_P(member(a, [a]), \tau) = leaves_P(member(a, [a, b]), \tau) = \emptyset$, $\tilde{\mathcal{A}}_1 = chmsg(\tilde{\mathcal{A}}_0) = \{(member(a, [a|T]), \tau)\}$
3. $leaves_P(member(a, [a|T]), \tau) = \emptyset$, $\tilde{\mathcal{A}}_2 = chmsg(\tilde{\mathcal{A}}_1) = \tilde{\mathcal{A}}_1$ and we have reached the fixpoint $\tilde{\mathcal{A}}$.

A partial deduction P' wrt $\tilde{\mathcal{A}}$ and ρ_α with $\alpha((member(a, [a|T]), \tau)) = m_1(T)$ is:

$m_1(X) \leftarrow$

$\tilde{\mathcal{A}}$ is P-covered and every atom in G is a concretisation of a characteristic atom in $\tilde{\mathcal{A}}$. Hence Theorem 5.2.2 can be applied: we obtain the renamed goal $G' = \rho_\alpha(G) = \leftarrow m_1([]), m_1([b])$ and $P' \cup \{G'\}$ yields the correct result.

In the remainder of this subsection, we formally prove the correctness of Algorithm 5.3.3, as well as its termination under a certain condition.

As already defined in Section 2, an *expression* is either a term, an atom, a literal, a conjunction, a disjunction or a program clause. Expressions are

constructed using the language $\mathcal{L}_P$ which we implicitly assume underlying the program P under consideration. Remember that $\mathcal{L}_P$ may contain additional symbols not occurring in P, but that, unless explicitly stated otherwise, $\mathcal{L}_P$ contains only finitely many constant, function and predicate symbols. To simplify the presentation we also assume that, when talking about expressions, predicate symbols and connectives are treated like functors which cannot be confounded with the original functors and constants (e.g. $\wedge$ and $\leftarrow$ are binary functors distinct from the other binary functors).

The following well-founded measure function is taken from [100] (also in the extended version of [201]):

Definition 5.3.5 ($s(.)$, $h(.)$) Let $Expr$ denote the sets of expressions. We define the function $s : Expr \rightarrow \mathbb{N}$ counting symbols by:

- $s(t) = 1 + s(t_1) + \ldots + s(t_n)$ if $t = f(t_1, \ldots, t_n)$, $n > 0$
- $s(t) = 1$ otherwise

Let the number of distinct variables in an expression t be $v(t)$. We now define the function $h : Expr \rightarrow \mathbb{N}$ by $h(t) = s(t) - v(t)$.

The well-founded measure function h has the property that $h(t) \geq 0$ for any expression t and $h(t) > 0$ for any non-variable expression t. The following important lemma is proven for $h(.)$ in [99] (see also [201]).

Lemma 5.3.6 If A and B are expressions such that B is strictly more general than A, then $h(A) > h(B)$.

It follows that, for every expression A, there are no infinite chains of strictly more general expressions.

Definition 5.3.7 (weight vector) Let $\tilde{\mathcal{A}}$ be a set of characteristic atoms and let $T = \langle \tau_1, \ldots, \tau_n \rangle$ be a finite vector of characteristic trees. We then define the *weight vector* of $\tilde{\mathcal{A}}$ wrt T by $hvec_T(\tilde{\mathcal{A}}) = \langle w_1, \ldots, w_n \rangle$ where

- $w_i = \infty$ if $\tilde{\mathcal{A}}|_{\tau_i} = \emptyset$
- $w_i = \sum_{A \in \tilde{\mathcal{A}}|_{\tau_i}} h(A)$ if $\tilde{\mathcal{A}}|_{\tau_i} \neq \emptyset$

The set of weight vectors is partially ordered by the usual order relation among vectors:
$\langle w_1, \ldots, w_n \rangle \leq \langle v_1, \ldots, v_n \rangle$ iff $w_1 \leq v_1, \ldots, w_n \leq v_n$. Also, given a quasi order $\leq_S$ on a set S, we from now on assume that the associated equivalence relation $\equiv_S$ and the associated strict partial order $>_S$ are implicitly defined in the following way:

- $s_1 \equiv_S s_2$ iff $s_1 \leq s_2 \wedge s_2 \leq s_1$ and $s_1 <_S s_2$ iff $s_1 \leq s_2 \wedge s_2 \not\leq s_1$.

The set of weight vectors is well-founded (no infinitely decreasing sequences exist) because the weights of the atoms are well-founded.

Proposition 5.3.8 Let P be a normal program, U an unfolding rule and let $T = \langle \tau_1, \ldots, \tau_n \rangle$ be a finite vector of characteristic trees. For every pair of finite sets of characteristic atoms $\tilde{\mathcal{A}}$ and $\tilde{\mathcal{B}}$, such that the characteristic trees of their elements are in T, we have that one of the following holds:

- $chmsg(\tilde{\mathcal{A}} \cup \tilde{\mathcal{B}}) = \tilde{\mathcal{A}}$ (up to variable renaming) or
- $hvec_T(chmsg(\tilde{\mathcal{A}} \cup \tilde{\mathcal{B}})) < hvec_T(\tilde{\mathcal{A}})$.

Proof Let $hvec_T(\tilde{\mathcal{A}}) = \langle w_1, \ldots, w_n \rangle$ and let $hvec_T(chmsg(\tilde{\mathcal{A}} \cup \tilde{\mathcal{B}})) = \langle v_1, \ldots, v_n \rangle$. Then for every $\tau_i \in T$ we have two possible cases:

- $\{msg(\tilde{\mathcal{A}}|_{\tau_i} \cup \tilde{\mathcal{B}}|_{\tau_i})\} = \tilde{\mathcal{A}}|_{\tau_i}$ (up to variable renaming). In this case the abstraction operator performs no modification for τ_i and $v_i = w_i$.
- $\{msg(\tilde{\mathcal{A}}|_{\tau_i} \cup \tilde{\mathcal{B}}|_{\tau_i})\} = \{M\} \neq \tilde{\mathcal{A}}|_{\tau_i}$ (up to variable renaming). In this case $(M, \tau_i) \in chmsg(\tilde{\mathcal{A}} \cup \tilde{\mathcal{B}})$, $v_i = h(M)$ and there are three possibilities:
  - $\tilde{\mathcal{A}}|_{\tau_i} = \emptyset$. In this case $v_i < w_i = \infty$.
  - $\tilde{\mathcal{A}}|_{\tau_i} = \{A\}$ for some atom A. In this case M is strictly more general than A (by definition of msg because $M \neq A$ up to variable renaming) and hence $v_i < w_i$.
  - $\#(\tilde{\mathcal{A}}|_{\tau_i}) > 1$. In this case M is more general (but not necessarily strictly more general) than any atom in $\tilde{\mathcal{A}}|_{\tau_i}$ and $v_i < w_i$ because at least one atom is removed by the abstraction.

We have that $\forall i \in \{1, \ldots, n\} : v_i \leq w_i$ and either the abstraction operator performs no modification (and $\vec{v} = \vec{w}$) or the well-founded measure $hvec_T$ strictly decreases. □

Theorem 5.3.9 If Algorithm 5.3.3 generates a finite number of distinct characteristic trees then it terminates and produces a partial deduction satisfying the requirements of Theorem 5.2.2 for any goal G' whose atoms are instances of atoms in G.

Proof Reaching the fixpoint guarantees that all predicates in the bodies of resultants are precise concretisations of at least one characteristic atom in $\tilde{\mathcal{A}}_k$, i.e. $\tilde{\mathcal{A}}_k$ is P-covered. Furthermore $chmsg()$ always generates more general characteristic atoms (even in the sense that any precise concretisation of an atom in $\tilde{\mathcal{A}}_i$ is a precise concretisation of an atom in $\tilde{\mathcal{A}}_{i+1}$ — this follows immediately from Definitions 5.1.3 and 5.3.1). Hence, because any instance of an atom in the goal G is a precise concretisation of a characteristic atom in $\tilde{\mathcal{A}}_0$, the conditions of Theorem 5.2.2 are satisfied for goals G' whose atoms are instances of atoms in G, i.e. G' is P-covered by $\tilde{\mathcal{A}}_k$. Finally, termination is guaranteed by Proposition 5.3.8, given that the number of distinct characteristic trees is finite. □

The method for partial deduction as described in this section, using the framework of Section 5.1, has been called *ecological* partial deduction

in [168] because it guarantees the preservation of characteristic trees. A prototype partial deduction system, using Algorithm 5.3.3, has been implemented and experiments with it can be found in [168] and [203]. We will however further refine the algorithm in the next chapter and present extensive benchmarks in Section 6.4.2.

5.4 Some further discussions

5.4.1 Increasing precision

Let us return Definition 5.1.8 of D_P. For a given characteristic atom (A, τ), $D_P(A, \tau)$ uses the characteristic tree τ to determine exactly how $\leftarrow A$ should be unfolded (to generate the resultants). This is not always totally satisfactory. For instance, it might be interesting to use a very deep and precise tree τ, in order to arrive at a precise control of polyvariance, but only use a shallow, determinate unfolding for the resultants to guarantee efficiency of the specialised program.

Example 5.4.1 Let P be the Program from Examples 3.2.5 and 3.3.3:

$$member(X, [X \mid T]) \leftarrow$$
$$member(X, [Y \mid T]) \leftarrow member(X, T)$$
$$inboth(X, L1, L2) \leftarrow member(X, L1), member(X, L2)$$

Let $A = inboth(a, L1, [a, a])$ and $B = inboth(b, L1, L2)$. If we use a (shower) determinate unfolding rule U then $chtree(\leftarrow A, P, U) = chtree(\leftarrow B, P, U) = \{\langle 1 \circ 3 \rangle\}$. If we start Algorithm 5.3.3 with $G = \leftarrow A, B$ then A and B are abstracted by $(inboth(X, L1, L2), \tau)$, resulting in a loss of specialisation. Note that even if we allow left-most non-determinate unfolding the characteristic trees of A and B will remain identical. Only if we use a more aggressive unfolding rule U', which performs non-leftmost non-determinate unfolding, will the difference between $\leftarrow A$ and $\leftarrow B$ be spotted, thus preventing the unnecessary abstraction. However, as shown in Example 3.3.3, such an unfolding will yield resultants in which (possibly expensive) work gets duplicated.

In such a case it might be interesting to base the control of polyvariance on U' but construct the specialised program using U. Fortunately, Definition 5.1.8 can be quite easily adapted to account for this desire. More formally, any unfolding whose associated characteristic tree τ' satisfies $\gamma_P(A, \tau') \supseteq \gamma_P(A, \tau)$ can be used in Definition 5.1.8 and the correctness theorems still hold.[7] This refinement might be especially interesting

[7] Note that, by adapting the definition of D_P, we also implicitly adapt the definition of the leaf atoms $leaves_P(A, \tau)$ and therefore Algorithm 5.3.3 still ensures coveredness

for very conservative unfolding rules, e.g. based solely on determinacy.

The ECCE partial deduction system, which we are going to present in the next chapter, actually incorporates this refinement (which we will not use in the experiments; it will however turn out to be useful for some examples in Chapter 12).

5.4.2 An alternative constraint-based approach

In this subsection we summarise an alternative approach to preserving characteristic trees upon generalisation. We will only mention the essentials; the full details can be found in the revised version of [172].

In standard partial deduction an atom $A \in \mathcal{A}$ represents a possibly infinite set of concretisations, namely *all* its instances. As shown in Chapter 4, this makes it impossible to preserve characteristic trees upon generalisation. In this chapter we have solved this problem by introducing the framework of ecological partial deduction. By using characteristic atoms instead of plain atoms, together with a more refined notion of concretisation, we were able to exclude certain instances from the concretisations (namely those which do not have the desired characteristic tree). [172] presents a generalisation of this idea, which, in the context of definite programs, allows to restrict the set of potential concretisations by using *constraints*. Based upon the so obtained framework of *constrained partial deduction*, [172] also presents a concrete technique which ensures the preservation of characteristic trees for certain unfolding rules. The benefits of constrained partial deduction however surpass that context, and it can for instance also be used to "drive negative information" (using the terminology of supercompilation [273, 275]).

Constraint logic programming

We briefly recall some basic terminology from *constraint logic programming (CLP)* [133]. In CLP the predicate symbols are partitioned into two disjoint sets Π_c (the predicate symbols to be used for constraints, notably including "=") and Π_b (the predicate symbols for user-defined predicates). A *constraint* is a first-order formula whose predicate symbols are all contained in Π_c. In the context of CLP one often uses the connective " $\square$ " in the place of "$\wedge$" to separate constraints from "ordinary" formulas. For instance, a *CLP-goal* is denoted by $\leftarrow c \square B_1, \ldots, B_n$, where c is a constraint and $B_1, \ldots, B_n$ are ordinary atoms. The semantics of constraints is given by a *structure* $\mathcal{D}$, consisting of a domain D and an assignment of functions and relations on D to function symbols and to predicate symbols in Π_c.

and correctness.

Given a constraint c, $\mathcal{D} \models c$ then denotes the fact that c is true under the interpretation provided by $\mathcal{D}$.

[172] defines a counterpart to SLD-derivations for CLP-goals. In the given context of partial deduction, the initial and final programs are ordinary logic programs (ordinary logic programs can be seen as CLP-programs using equality constraints over the structure of feature terms $\mathcal{FT}$, see [189]). In order for the constraint manipulations to be sound wrt the initial logic program, we have to ensure that equality in the constraint domain is a safe approximation of equality as used for ordinary logic programs. In other words, if there is no SLD-refutation for $P \cup \{\leftarrow Q\}$ then there should be no CLP-refutation for any $P \cup \{\leftarrow c \ \square \ Q\}$ either. This property is ensured by adapting the definition of CLP-derivations from [92] such as to make substitutions explicit. This also makes it possible in [172] to define computed answers and CLP-resultants for CLP-derivations.

Constrained partial deduction

The basic idea is instead of producing a partial deduction for a set of atoms to produce it for a set of *constrained atoms*. A constrained atom is a formula of the form $c \ \square \ A$, where A is an ordinary atom and c a constraint. The set of concretisations of a constrained atom $c \ \square \ A$ are then all the atoms $A\theta$ such that $c\theta$ holds, i.e. $\mathcal{D} \models \forall(c\theta)$. This set of concretisations (called valid instances in [172]) is downwards-closed.

A partial deduction of $c \ \square \ A$ in P is then obtained by taking the CLP-resultants of a finite, non-trivial and possibly incomplete CLP-tree for $P \cup \{\leftarrow c \ \square \ A\}$. This partial deduction still contains constraints, which are then removed by a renaming function, defined in a very similar manner to Definition 5.1.12 (i.e. based upon an atomic renaming α which maps constrained atoms to ordinary atoms). Given a coveredness condition, all the necessary constraint processing has been performed at partial deduction time and correctness is formally established in [172]

Pruning constraints

As we have shown on multiple occasions in Chapter 4, when taking the *msg* of two atoms A and B with the same characteristic tree τ, we do not necessarily obtain an atom C which has the same characteristic tree. Now, instead of C, the technique in [172] generates $c \ \square \ C$ as the generalisation of A and B, where the constraint c is designed in such a way as to prune the possible computations of C into the right shape, namely τ. Indeed, all the derivations that were possible for A and B are also possible for C (because we only consider definite programs and goals) and c only has to ensure that

the additional matches wrt τ are pruned off at some point (cf. Figure 5.4).

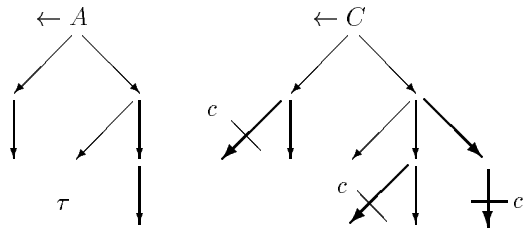


Figure 5.4: Pruning Constraints

To be able to calculate a *finite* constraint, covering *all* instances of C which have τ as their characteristic tree, [172] is restricted to certain unfolding rules (basically beam determinate unfolding rules without lookahead; cf. Definition 3.3.2). The *pruning constraints* are then formulated using Clark's equality theory). More precisely, [172] uses the structure $\mathcal{D} = \mathcal{FT}$ consisting of CET over the domain of finite trees (with infinitely many functors of every arity). It is basically the same structure as the one used for CLP($\mathcal{FT}$) (see e.g. [256, 254, 255]) as well as in constructive negation techniques (e.g. [48, 49, 86, 252, 267, 266]).

Let us revisit Example 4.3.3 and illustrate how characteristic trees can be preserved using constrained partial deduction.

Example 5.4.2 Let us return to the program P from Example 4.3.3:

(1) $p(X) \leftarrow$
(2) $p(c) \leftarrow$

We can abstract the atoms $p(a)$ and $p(b)$ of Example 4.3.3 (as well as the equivalent constrained atoms $true \ \square \ p(a)$ and $true \ \square \ p(b)$) by the more general constrained atom $\neg(X = c) \ \square \ p(X)$, having the same characteristic tree $\tau = \{\langle 1 \circ 1 \rangle\}$. The additional match with clause (2) is pruned for $\neg(X = c) \ \square \ p(X)$, because $\neg(c = c)$ is unsatisfiable in CET.

Let us also briefly discuss some further applications of constrained partial deduction, beyond preserving characteristic trees. For example, a constraint structure over integers or reals could handle Prolog built-ins like $<, >, \leq, \geq$ in a much more sophisticated manner than ordinary partial evaluators. Also, one can provide a refined treatment of the \== Prolog built-in using the $\mathcal{FT}$ structure. The following example illustrates this,

where a form of "driving of negative information" (using the terminology of supercompilation [273, 275]) is achieved by constrained partial deduction.

Example 5.4.3 Take the following adaptation of the *member* program which only succeeds once.

(1) $member(X, [X \mid T]) \leftarrow$
(2) $member(X, [Y \mid T]) \leftarrow X \backslash == Y, member(X, T)$

Let us start specialisation with the goal $\leftarrow member(X, [a, Y, a])$. Using a determinate unfolding rule (even with a lookahead) we would unfold this goal once and get the resolvent $\leftarrow X \backslash == a, member(X, [Y, a])$ in the second branch. Ordinary partial deduction would ignore $X \backslash == a$ and unfold $member(X, [Y, a])$, thus producing an extra superfluous resultant with the impossible (given the context) computed answer $\{X/a\}$. In the constrained partial deduction setting, we can incorporate $X \backslash == a$ as a constraint and unfold $\neg(X = a) \square member(X, [Y, a])$ instead of just $member(X, [Y, a])$ and thereby prune the superfluous resultant. (We will see a technique which can achieve a similar effect in Part V of the thesis).

Some discussions

Compared to ecological partial deduction, the method of [172] using pruning constraints is more complex and restricted to determinate unfolding rules *without lookahead* as well as to *definite* programs. Within that context however, it enjoys a better overall precision than ecological partial deduction. In fact, the characteristic tree τ inside a characteristic atom (A, τ) can be seen as an implicit representation of constraints on A. In ecological partial deduction, these constraints are used only locally to ensure preservation of the characteristic tree and are not propagated towards other characteristic atoms. The pruning constraints in [172] are propagated and thus used globally. Future work is needed to determine whether this has any real influence in practice.

5.4.3 Conclusion

We have presented a new framework and a new algorithm for partial deduction. The framework and the algorithm can handle *normal* logic programs and place *no* restrictions on the unfolding rule. We provided general correctness results for the framework as well as correctness and termination proofs for the algorithm. Also, the abstraction operator of the algorithm preserves the characteristic trees of the atoms to be specialised and ensures termination (when the number of distinct characteristic trees is bounded) while providing a fine grained control of polyvariance.

Chapter 6

Removing Depth Bounds by Adding Global Trees

6.1 The depth bound problem

The algorithm for ecological partial deduction presented in Section 5.3 only terminates when imposing a depth bound on characteristic trees. In this section we present some natural examples which show that this leads to undesired results in cases where the depth bound is actually required. (These examples can also be adapted to prove a similar point about neighbourhoods in the context of supercompilation of functional programs. We will return to the relation of neighbourhoods to characteristic trees in Section 6.4.4.)

When, for the given program, query and unfolding rule, the method of Section 5.3 generates a *finite number of different characteristic trees*, its global control regime guarantees termination and correctness of the specialised program as well as "optimal" polyvariance: *For every predicate, exactly one specialised version is produced for each of its different associated characteristic trees.* Now, Algorithm 5.3.3 of the previous chapter, as well as all earlier approaches based on characteristic trees ([100, 97, 172]), achieves the mentioned finiteness condition at the cost of imposing an ad hoc (typically very large) depth bound on characteristic trees. However, for a fairly *large class of realistic programs* (and unfolding rules), *the number of different characteristic trees generated, is infinite.* In those cases, the underlying depth bound will have to ensure termination, meanwhile propagating its ugly, ad hoc nature into the resulting specialised program.

We illustrate this problem through some examples, setting out with a

slightly artificial, but very simple one.

Example 6.1.1 The following is the well known reverse with accumulating parameter where a list type check on the accumulator has been added.

(1) $rev([], Acc, Acc) \leftarrow$
(2) $rev([H|T], Acc, Res) \leftarrow ls(Acc), rev(T, [H|Acc], Res)$
(3) $ls([]) \leftarrow$
(4) $ls([H|T]) \leftarrow ls(T)$

As can be noticed in Figure 6.1, (determinate [100, 97, 172] and well-founded [37, 200, 199], among others) unfolding produces an infinite number of different characteristic atoms, all with a different characteristic tree. Imposing a depth bound of say 100, we obtain termination, but 100 different characteristic trees (and instantiations of the accumulator) arise and the algorithm produces 100 different versions of rev: one for each characteristic tree. The specialised program thus looks like:

(1') $rev([], [], []) \leftarrow$
(2') $rev([H|T], [], Res) \leftarrow rev_2(T, [H], Res)$
(3') $rev_2([], [A], [A]) \leftarrow$
(4') $rev_2([H|T], [A], Res) \leftarrow rev_3(T, [H, A], Res)$
$\vdots$
(197') $rev_{99}([], [A_1, \ldots, A_{98}], [A_1, \ldots, A_{98}]) \leftarrow$
(198') $rev_{99}([H|T], [A_1, \ldots, A_{98}], Res) \leftarrow$
$rev_{100}(T, [H, A_1, \ldots, A_{98}], Res)$
(199') $rev_{100}([], [A_1, \ldots, A_{99}|AT], [A_1, \ldots, A_{99}|AT]) \leftarrow$
(200') $rev_{100}([H|T], [A_1, \ldots, A_{99}|AT], Res) \leftarrow$
$ls(AT), rev_{100}(T, [H, A_1, \ldots, A_{99}|AT], Res)$
(201') $ls([]) \leftarrow$
(202') $ls([H|T]) \leftarrow ls(T)$

This program is certainly far from optimal and clearly exhibits the ad hoc nature of the depth bound.

Situations like the above typically arise when an accumulating parameter influences the computation, because then the growing of the accumulator causes a corresponding growing of the characteristic trees. To be fair, it must be admitted that with most simple programs, this is not the case. For instance, in the standard reverse with accumulating parameter, the accumulator is only copied in the end, but never influences the computation. As illustrated by Example 6.1.1 above, this state of affairs will often already be changed when one adds type checking in the style of [101] to even the simplest logic programs.

Among larger and more sophisticated programs, cases like the above become more and more frequent, even in the absence of type checking. For

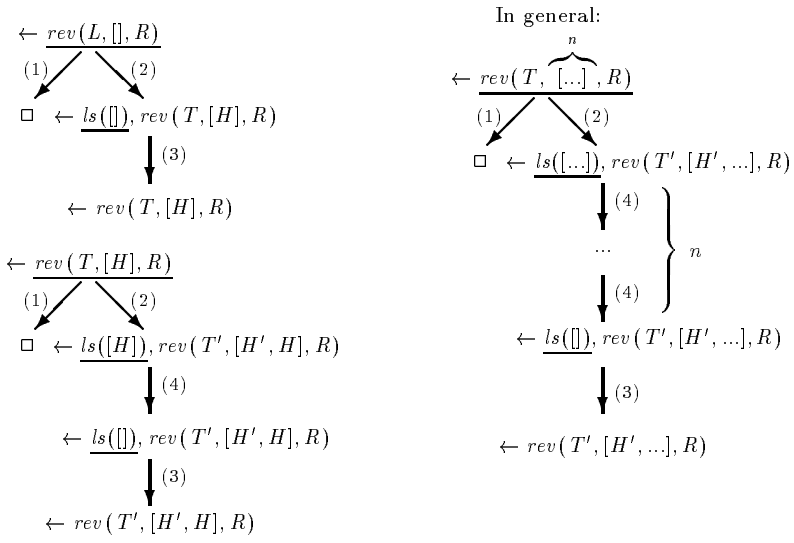


Figure 6.1: SLD-trees for Example 6.1.1

instance, in an explicit unification algorithm, one accumulating parameter is the substitution built so far. It heavily influences the computation because new bindings have to be added and checked for compatibility with the current substitution. Another example is the "mixed" meta-interpreter of [122, 173] (part of it is depicted in Figure 6.2; see also Chapter 8, notably Section 8.4.2) for the ground representation in which the goals are "lifted" to the non-ground representation for resolution. To perform the lifting, an accumulating parameter is used to keep track of the variables that have already been encountered. This accumulator influences the computation: Upon encountering a new variable, the program inspects the accumulator.

Example 6.1.2 Let $A = l\_mng(Lg, Ln, [sub(N, X)], S)$ and P be the program of Figure 6.2 in which the predicate l_mng transforms a list of ground terms into a list of non-ground terms. As can be seen in Figure 6.3, unfolding A (e.g. using well-founded measures), the atom

$$l\_mng(Tg,\ Tn, [sub(N, X), sub(J, Hn)], S)$$

is added at the global control level (this situation arose in an actual experiment). Notice that the third argument has grown, i.e. we have an accumu-

Program:

(1) $make\_non\_ground(GrTerm, NgTerm) \leftarrow$
$mng(GrTerm, NgTerm, [], Sub)$

(2) $mng(var(N), X, [], [sub(N, X)]) \leftarrow$

(3) $mng(var(N), X, [sub(N, X) | T], [sub(N, X) | T]) \leftarrow$

(4) $mng(var(N), X, [sub(M, Y) | T], [sub(M, Y) | T1]) \leftarrow$
$not(N = M),\ mng(var(N), X, T, T1)$

(5) $mng(struct(F, GrArgs), struct(F, NgArgs), InSub, OutSub) \leftarrow$
$l\_mng(GrArgs, NgArgs, InSub, OutSub)$

(6) $l\_mng([], [], Sub, Sub) \leftarrow$

(7) $l\_mng([GrH | GrT], [NgH | NgT], InSub, OutSub) \leftarrow$
$mng(GrH, NgH, InSub, InSub1),$
$l\_mng(GrT, NgT, InSub1, OutSub)$

Example query:

$\leftarrow make\_non\_ground(struct(f, [var(1), var(2), var(1)]), F)$
$\rightsquigarrow$ c.a.s. $\{F / struct(f, [Z, V, Z])\}$

Figure 6.2: Lifting the ground representation

lator. When in turn unfolding $l\_mng(Tg, Tn, [sub(N, X), sub(J, Hn)], S)$, we will obtain a deeper characteristic tree (because *mng* traverses the third argument and thus needs one more step to reach the end) with

$$l\_mng(Tg', Tn', [sub(N, X), sub(J, Hn), sub(J', Hn')], S)$$

as one of its leaves. An infinite sequence of ever growing characteristic trees results and again, as in Example 6.1.1, we obtain non-termination without a depth bound, and very unsatisfactory ad hoc specialisations with it.

Summarising, computations influenced by one or more growing data structures are by no means rare and will, very often, lead to ad hoc behaviour of partial deduction, where the global control is founded on characteristic trees with a depth bound. In the next section, we show how this annoying depth bound can be removed without endangering termination.

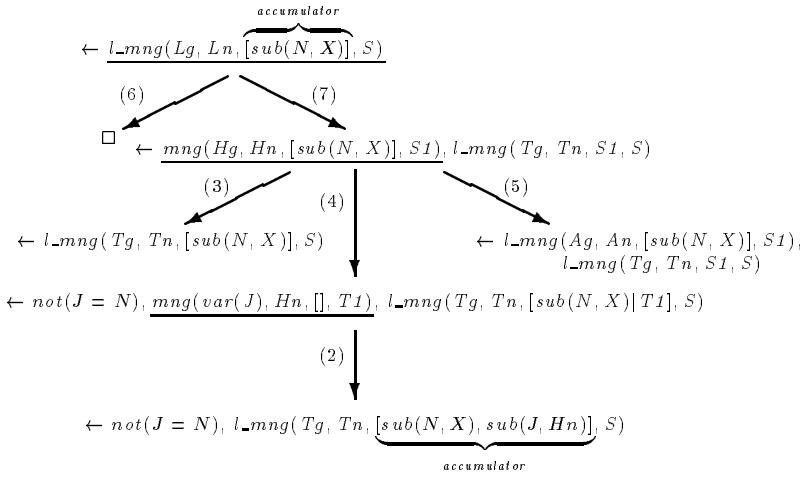


Figure 6.3: Accumulator growth in Example 6.1.2

6.2 Partial deduction using global trees

6.2.1 Introduction

A general framework for global control, not relying on any depth bounds, is proposed in [201]. Marked trees (m-trees) are introduced to register descendency relationships among atoms at the global level. These trees are subdivided into classes of nodes and associated measure functions map nodes to well-founded sets. The overall tree is then kept finite through ensuring monotonicity of the measure functions and termination of the algorithm follows, provided the abstraction operator is similarly well-founded. It is to this framework that we now turn for inspiration on how to solve the depth bound problem uncovered for characteristic trees in Section 6.1.

First, we have chosen to use the term "global tree" rather than "marked tree" in the present chapter, because it better indicates its functionality. Moreover, global trees rely on a well-quasi-order (or a well-quasi relation) between nodes, rather than a well-founded one, to ensure their finiteness. Apart from that, in essence, their structure is similar: They register which atoms derive from which at the global control level. The initial part of such a tree, showing the descendency relationship between the atom in the root and those in the dangling leaves of the SLDNF-tree in Figure 6.3, is depicted in Figure 6.4.[1]

[1] Observe that the global tree in Figure 6.3 also contains two nodes which are variants of their parent node. Usually, atoms which are variants of one of their ancestor nodes, will

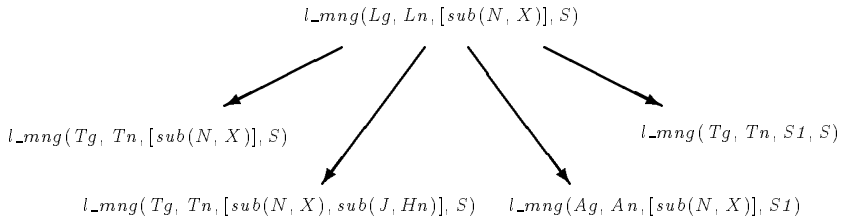


Figure 6.4: Initial section of a global tree for Example 6.1.2 and the unfolding of Figure 6.3

Now, the basic idea will be to have just a single class covering the whole global tree structure and to watch over the evolution of characteristic trees associated to atoms along its branches. Obviously, just measuring the depth of characteristic trees would be far too crude: Global branches would be cut off prematurely and entirely unrelated atoms could be mopped together through generalisation, resulting in unacceptable specialisation losses. No, as can be seen in Figure 6.1, we need a more refined measure which would somehow spot when a characteristic tree (piecemeal) "contains" characteristic trees appearing earlier in the same branch of the global tree. If such a situation arises — as it indeed does in Example 6.1.1 — it seems reasonable to stop expanding the global tree, generalise the offending atoms and produce a specialised procedure for the generalisation instead.

However, a closer look at the following variation of Example 6.1.2 shows that also this approach would sometimes overgeneralise and consequently fall short of providing sufficiently detailed polyvariance.

Example 6.2.1 Reconsider the program in Figure 6.2, and suppose that determinate unfolding is used for the local control.
The atom $A = mng(G, struct(cl, [struct(f, [X, Y])|B]), [], S)$ will now be the starting point for partial deduction (also this situation arose in an actual experiment). When unfolding A (see Figure 6.5), we obtain an SLD-tree containing the atom $mng(H, struct(f, [X, Y]), [], S1)$ in one of its leaves. If we subsequently determinately unfold the latter atom, we obtain a tree that is "larger" than its predecessor, also in the more refined sense. Potential non-termination would therefore be detected and a generalisation operator executed. However, the atoms in the leaves of the second tree are more general than those already met, and simply continuing partial deduction

not be explicitly added to the global tree, as they do not give rise to further specialisation.

without generalisation will lead to natural termination without any depth bound intervention.

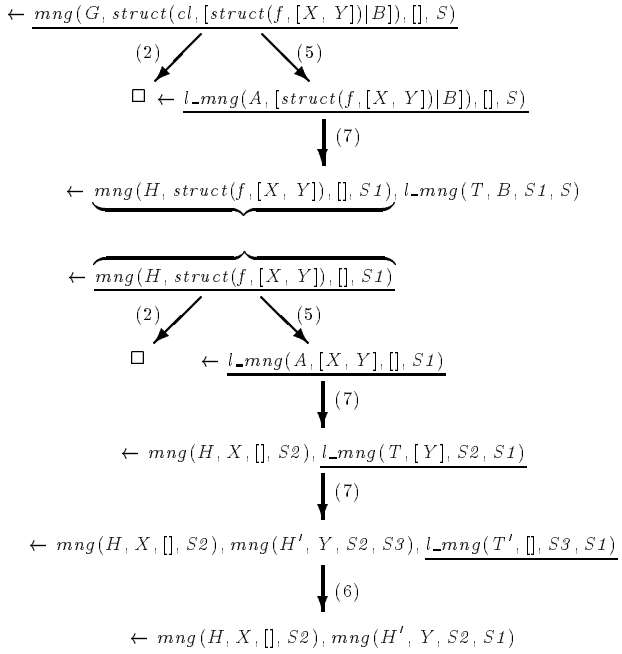


Figure 6.5: SLD-trees for Example 6.2.1

Example 6.2.1 demonstrates that only measuring growth of characteristic trees, even in a refined way, does not always lead to satisfactory specialisation. In fact, whenever the unfolding rule does not unfold "as deeply" as would be possible using a refined termination relation (for whatever reason, e.g. efficiency of the specialised program or because the unfolding rule is not refined enough), then a growing characteristic tree might simply be caused by splitting the "maximally deep tree" (i.e. the one constructed using a refined termination relation) in such a way that the second part "contains" the first part. Indeed, in Example 6.2.1, an unfolding rule based on well-founded measures could have continued unfolding more deeply for the first atom, thus avoiding the fake growing problem in this case.

Luckily, the same example also suggests a solution to this problem: Rather than measuring and comparing characteristic trees, we will *scrutinise entire characteristic atoms, comparing both the syntactic content of the ordinary atoms they contain and the associated characteristic trees.* Ac-

cordingly, the global tree nodes will not be labelled by plain atoms as in [201], but by entire characteristic atoms. A growing of a characteristic tree not coupled with a growing of the syntactic structure then indicates a fake growing caused by a conservative unfolding rule.

The rest of this section, then, contains the formal elaboration of this new approach. In Subsection 6.2.2, we first extend the familiar generalisation notion defined on atoms to characteristic atoms, and subsequently proceed to introduce the precise comparison operation to be used on the latter. Some important properties connecting both operations are also stated and proved. Next, Subsection 6.2.3 introduces global trees and a characterisation of their finiteness. Finally, our refined algorithm for partial deduction is presented and proved correct and terminating in Subsection 6.2.4.

6.2.2 More on characteristic atoms

Generalising characteristic atoms

In this subsection we extend the notions of variants, instances and generalisations, familiar for ordinary atoms, to characteristic trees and atoms.

For ordinary atoms, $A_1 \preceq A_2$ will denote that A_1 is more general than A_2. We will now define a similar notion for characteristic atoms along with an operator to compute the most specific generalisation. In a first attempt one might use the concretisation function to that end, i.e. one could stipulate that (A, τ_A) is more general than (B, τ_B) iff $\gamma_P(A, \tau_A) \supseteq \gamma_P(B, \tau_B)$. The problem with this definition, from a practical point of view, is that this notion is undecidable in general and a most specific generalisation is uncomputable. For instance, $\gamma_P(A, \tau) = \gamma_P(A, \tau \cup \{\delta\})$ holds iff for every instance of A, the last goal associated with δ fails finitely. Deciding this is equivalent to the halting problem. We will therefore present a safe but computable approximation of the above notion, for which a most specific generalisation can be easily computed and which has some nice properties in the context of a partial deduction algorithm (see e.g. Definition 6.2.27 below).

We first define an ordering on characteristic trees. In that context, the following notation will prove to be useful:

$$\mathit{prefix}(\tau) = \{\delta \mid \exists \gamma \text{ such that } \delta\gamma \in \tau\}.$$

Definition 6.2.2 ($\preceq_\tau$) Let τ_1, τ_2 be characteristic trees. We say that τ_1 is *more general* than τ_2, and denote this by $\tau_1 \preceq_\tau \tau_2$, iff

1. $\delta \in \tau_1 \Rightarrow \delta \in \mathit{prefix}(\tau_2)$ and
2. $\delta' \in \tau_2 \Rightarrow \exists \delta \in \mathit{prefix}(\{\delta'\})$ such that $\delta \in \tau_1$.

Note that $\preceq_\tau$ is a quasi order on the set of characteristic trees and that $\tau_1 \preceq_\tau \tau_2$ is equivalent to saying that τ_2 can be obtained from τ_1 by attaching sub-trees to the leaves of τ_1. Remember that, given a quasi order $\preceq$, we also use the associated equivalence relation $\equiv$ and strict partial order $\prec$ as defined in Section 5.3.

Example 6.2.3 Given $\tau_1 = \{\langle 1 \circ 3\rangle\}$, $\tau_2 = \{\langle 1 \circ 3, 2 \circ 4\rangle\}$ and $\tau_3 = \{\langle 1 \circ 3\rangle, \langle 1 \circ 4\rangle\}$ we have that $\tau_1 \preceq_\tau \tau_2$ and even $\tau_1 \prec_\tau \tau_2$ but not that $\tau_1 \preceq_\tau \tau_3$ nor $\tau_2 \preceq_\tau \tau_3$. We also have that $\{\langle\rangle\} \prec_\tau \tau_1$ but not that $\emptyset \preceq_\tau \tau_1$. In fact, $\{\langle\rangle\} \preceq_\tau \tau$ holds for any $\tau \neq \emptyset$, while $\emptyset \preceq_\tau \tau$ only holds for $\tau = \emptyset$. Also $\tau \preceq_\tau \{\langle\rangle\}$ only holds for $\tau = \{\langle\rangle\}$ and $\tau \preceq_\tau \emptyset$ only holds for $\tau = \emptyset$.

The next two lemmas respectively establish a form of anti-symmetry as well as transitivity of the order relation on characteristic trees.

Lemma 6.2.4 Let τ_1, τ_2 be two characteristic trees. Then $\tau_1 \equiv_\tau \tau_2$ iff $\tau_1 = \tau_2$.

Proof The if part is obvious because δ and δ' can be taken as prefixes of themselves for the two points of Definition 6.2.2.

For the only-if part, let us suppose that $\tau_1 \preceq_\tau \tau_2$ and $\tau_2 \preceq_\tau \tau_1$ but $\tau_1 \neq \tau_2$. This means that there must be a characteristic path δ in τ_1 which is not in τ_2 (otherwise we reverse the roles of τ_1 and τ_2). We know however, by point 1 of Definition 6.2.2, that an extension $\delta_x = \delta\gamma$ of δ must be in τ_2, as well as, by point 2 of the same definition, that a prefix δ_s of δ must be in τ_2. Therefore τ_2 contains the paths δ_x and δ_s, where $\delta_x = \delta_s\gamma'\gamma$ and $\delta_x \neq \delta_s$ (because $\delta \notin \tau_2$). But this is impossible by Lemma 5.2.10. □

Lemma 6.2.5 Let τ_1, τ_2 and τ_3 be characteristic trees. If $\tau_1 \preceq_\tau \tau_2$ and $\tau_2 \preceq_\tau \tau_3$ then $\tau_1 \preceq_\tau \tau_3$.

Proof Immediate from Definition 6.2.2 because a prefix of a prefix remains a prefix. □

We now present an algorithm to generalise two characteristic trees by computing the common intial subtree. We will later prove that this algorithm calculates the most specific generalisation.

The following notations will be useful in formalising the algorithm.

Definition 6.2.6 Let τ be a characteristic tree and δ a characteristic path. We then define the notations

$$\tau \downarrow \delta = \{\gamma \mid \delta\gamma \in \tau\}$$

and

$$top(\tau) = \{l \circ m \mid \langle l \circ m \rangle \in prefix(\tau)\}.$$

Note that, for a non-empty characteristic tree τ, $top(\tau) = \emptyset \Leftrightarrow \tau = \{\langle\rangle\}$.

Algorithm 6.2.7 (msg of characteristic trees)

Input
two non-empty characteristic trees τ_A and τ_B
Output
the *msg* τ of τ_A and τ_B
Initialisation
$i := 0$; $\tau_0 := \{\langle\rangle\}$;
while $\exists \delta \in \tau_i$ such that $top(\tau_A \downarrow \delta) = top(\tau_B \downarrow \delta) \neq \emptyset$ **do**
$\tau_{i+1} := (\tau_i \setminus \{\delta\}) \cup \{\delta\langle l \circ m \rangle \mid l \circ m \in top(\tau_A \downarrow \delta)\}$;
$i := i + 1$;
end while
return $\tau = \tau_i$

Example 6.2.8 Take the characteristic trees $\tau_A = \{\langle 1 \circ 1, 1 \circ 2\rangle\}$ and $\tau_B = \{\langle 1 \circ 1, 1 \circ 2\rangle, \langle 1 \circ 1, 1 \circ 3\rangle\}$. Then Algorithm 6.2.7 proceeds as follows:

- $\tau_0 = \{\langle\rangle\}$,
- $\tau_1 = \{\langle 1 \circ 1\rangle\}$ as for $\langle\rangle \in \tau_0$ we have $top(\tau_A \downarrow \langle\rangle) = top(\tau_B \downarrow \langle\rangle) = \{1 \circ 1\}$.
- $\tau = \tau_1$ as $top(\tau_A \downarrow \langle 1 \circ 1\rangle) = \{1 \circ 2\}$ and $top(\tau_B \downarrow \langle 1 \circ 1\rangle) = \{1 \circ 2, 1 \circ 3\}$ and the while loop terminates.

We first establish that Algorithm 6.2.7 indeed produces a proper characteristic tree as output.

Lemma 6.2.9 Any τ_i arising during the execution of Algorithm 6.2.7 satisfies the property that $\delta \in \tau_i \Rightarrow \delta \in prefix(\tau_A) \cap prefix(\tau_B)$.

Proof We prove this by induction on i.
Induction Hypothesis: For $i \leq k \leq max$ (where max is the maximum value that i takes during the execution of Algorithm 6.2.7) we have that $\delta \in \tau_i \Rightarrow \delta \in prefix(\tau_A) \cap prefix(\tau_B)$.
Base Case: For $i = 0$ we have $\tau_i = \{\langle\rangle\}$ and trivially $\langle\rangle \in prefix(\tau_A) \cap prefix(\tau_B)$ because $\tau_A \neq \emptyset$ and $\tau_B \neq \emptyset$.
Induction Step: Let $i = k+1 \leq max$. For $\delta \in \tau_{k+1}$ we either have $\delta \in \tau_k$, and we can apply the induction hypothesis to prove the induction step, or we have $\delta = \delta'\langle l \circ m\rangle$ with $l \circ m \in top(\tau_A \downarrow \delta')\}$ and $l \circ m \in top(\tau_B \downarrow \delta')\}$. By definition this implies that $\delta'\langle l \circ m\rangle\gamma \in \tau_A$ for some γ, i.e. $\delta'\langle l \circ m\rangle \in$

prefix(τ_A). The same holds for τ_B, i.e. $\delta'\langle l \circ m\rangle \in$ *prefix*(τ_B), and we have proven the induction step. □

Lemma 6.2.10 Algorithm 6.2.7 terminates and produces as output a characteristic tree τ such that if *chtree*$(G, P, U) = \tau_A$ (respectively τ_B), then for some U', *chtree*$(G, P, U') = \tau$. The same holds for any τ_i arising during the execution of Algorithm 6.2.7.

Proof Termination of Algorithm 6.2.7 is obvious as τ_A and τ_B are finite, τ cannot grow larger (e.g. in terms of the sum of the lengths of the characteristic paths) than τ_A or τ_B and τ_{i+1} is strictly larger than τ_i. For the remaining part of the lemma we proceed by induction. Note that Algorithm 6.2.7 is symmetrical wrt τ_A and τ_B and it suffices to show the result for τ_A.
Induction Hypothesis: For $i \leq k$ we have that if *chtree*$(G, P, U) = \tau_A$ then for some U' *chtree*$(G, P, U') = \tau_i$.
Base Case: $\tau_0 = \{\langle\rangle\}$ is a characteristic tree of every goal and the property trivially holds.
Induction Step: Let $i = k + 1$ and let $\tau_{k+1} = (\tau_k \setminus \{\delta\}) \cup \{\delta\langle l_1 \circ m_1\rangle, \ldots, \delta\langle l_n \circ m_n\rangle\}$ where $top(\tau_A \downarrow \delta)\} = \{l_1 \circ m_1, \ldots, l_n \circ m_n\}$. By Lemma 6.2.9 we know that $\delta\langle l_j \circ m_j\rangle \in$ *prefix*(τ_A). Because τ_A is a characteristic tree we therefore know that $l_1 = \ldots = l_n = l$. By the induction hypothesis we know that for some U' *chtree*$(G, P, U') = \tau_k$. Let G' be the goal of the SLDNF-derivation of $P \cup \{G\}$ whose characteristic path is δ (which must exist because *chtree*$(G, P, U') = \tau_k$) and let T be the SLDNF-tree for $P \cup \{G\}$ whose characteristic tree is τ_k. If we expand T by selecting the literal at position l in G we obtain a SLDNF-tree T' whose characteristic tree is $\tau' = (\tau_k \setminus \{\delta\}) \cup \{\delta\langle l \circ m_1\rangle, \ldots, \delta\langle l \circ m_n\rangle\} \cup \{\delta\langle l \circ m'_1\rangle, \ldots, \delta\langle l \circ m'_q\rangle\}$ (because $\delta\langle l_j \circ m_j\rangle \in$ *prefix*(τ_A) and because additional clauses might match). If a negative literal is selected we have that $n = 1$, $q = 0$, $\tau_{k+1} = \tau'$ and the induction step holds. If a positive literal is selected then we can further unfold the goals associated with derivations of $P \cup \{G\}$ whose characteristic paths are $\delta\langle l \circ m'_j\rangle$ and make them fail finitely (because $\delta\langle l \circ m'_j\rangle \notin \tau_A$). In both cases we can come up with an unfolding rule U'' such that *chtree*$(G, P, U'') = \tau_{k+1}$ □

Proposition 6.2.11 Let τ_A, τ_B be two non-empty characteristic trees. Then the output τ of Algorithm 6.2.7 is the (unique) most specific generalisation of τ_A and τ_B.

Proof We first prove that each τ_i arising during the execution of Algorithm 6.2.7 is a generalisation of both τ_A and τ_B. For this we have to show

that the two points of Definition 6.2.2 are satisfied; the fact that τ and all τ_i are proper characteristic trees (a fact required by Definition 6.2.2) is already established in Lemma 6.2.10.

1. We have to show that $\delta \in \tau_i \Rightarrow \delta \in \mathit{prefix}(\tau_A) \cap \mathit{prefix}(\tau_B)$. But this is already proven in Lemma 6.2.9.

2. We have to show that $\delta' \in \tau_A \cup \tau_B \Rightarrow \exists \delta \in \mathit{prefix}(\{\delta'\})$ such that $\delta \in \tau_i$. Again we prove this by induction on i.
   Induction Hypothesis: For $i \leq k \leq max$ (where max is the maximum value that i takes during the execution of Algorithm 6.2.7) we have that $\delta' \in \tau_A \cup \tau_B \Rightarrow \exists \delta \in \mathit{prefix}(\{\delta'\})$ such that $\delta \in \tau_i$.
   Base Case: For all δ' we have that $\langle\rangle \in \mathit{prefix}(\{\delta'\})$ and $\langle\rangle \in \tau_0$.
   Induction Step: Let $i = k + 1 \leq max$ and take a $\delta' \in \tau_A \cup \tau_B$. Let $\delta \in \mathit{prefix}(\{\delta'\})$ be such that $\delta \in \tau_k$, which must exist by the induction hypothesis. Either we have that $\delta \in \tau_{k+1}$, and the induction step holds for δ, or $\delta\langle l \circ m\rangle \in \tau_{k+1}$ for every $l \circ m \in \mathit{top}(\tau_A \downarrow \delta)$ where also $\mathit{top}(\tau_A \downarrow \delta) = \mathit{top}(\tau_B \downarrow \delta) \neq \emptyset$. Let γ be such that $\delta' = \delta\gamma$ (which must exist because $\delta \in \mathit{prefix}(\{\delta'\})$). By Lemma 5.2.10 and because $\mathit{top}(\tau_A \downarrow \delta)\ \mathit{top}(\tau_B \downarrow \delta) \neq \emptyset$ we know that γ cannot be empty and thus $\gamma = \langle l \circ m\rangle\gamma'$ where $l \circ m \in \mathit{top}(\tau_A \downarrow \delta)$. Hence, we have found a prefix $\delta\langle l \circ m\rangle$ of δ' such that $\delta\langle l \circ m\rangle \in \tau_{k+1}$.

We have thus proven that every τ_i, and thus also the output τ, is a generalisation of both τ_A and τ_B. We will now prove that τ is indeed the most specific generalisation. For this we will prove that, whenever $\tau^* \preceq_\tau \tau_A$ and $\tau^* \preceq_\tau \tau_B$ then $\tau^* \preceq_\tau \tau$. This is sufficient to show that τ is a most specific generalisation; uniqueness then follows from Lemma 6.2.4. To establish $\tau^* \preceq_\tau \tau$ we now show that the two points of Definition 6.2.2 are satisfied.

1. Let $\delta \in \tau^*$. We have to prove that $\delta \in \mathit{prefix}(\tau)$. As $\tau^* \preceq_\tau \tau_A$ and $\tau^* \preceq_\tau \tau_B$, we have by Definition 6.2.2 that $\delta \in \mathit{prefix}(\tau_A) \cap \mathit{prefix}(\tau_B)$. In order to prove that $\delta \in \mathit{prefix}(\tau)$ it is sufficient to prove that if $\delta = \delta'\langle l \circ m\rangle\delta''$ then $\mathit{top}(\tau_A \downarrow \delta') = \mathit{top}(\tau_B \downarrow \delta') \neq \emptyset$. Indeed, once we have proven this statement, we can proceed by induction (starting out with $\delta' = \langle\rangle$) to show that for some i we have $\delta \in \tau_i$ (because $\mathit{top}(\tau_A \downarrow \delta') = \mathit{top}(\tau_B \downarrow \delta') \neq \emptyset$ and $\delta \in \mathit{prefix}(\tau_A) \cap \mathit{prefix}(\tau_B)$ ensure that if $\delta' \in \tau_j$ then $\delta'\langle l \circ m\rangle \in \tau_{j+k}$ for some $k > 0$). From then on, the algorithm will either leave δ unchanged or extend it, and thus we will have established that $\delta \in \mathit{prefix}(\tau)$.

   (a) Let us first assume that $\mathit{top}(\tau_A \downarrow \delta') = \mathit{top}(\tau_B \downarrow \delta') = \emptyset$ and show that it leads to a contradiction. By this assumption we

know that no extension of δ' can be in τ_A or τ_B, which is in contradiction with $\delta = \delta'\langle l \circ m\rangle\delta'' \in \mathit{prefix}(\tau_A) \cap \mathit{prefix}(\tau_B)$.

(b) Now let us assume that $\mathit{top}(\tau_A \downarrow \delta') \neq \mathit{top}(\tau_B \downarrow \delta')$. This means that, for some $l \circ m'$, $\delta'\langle l \circ m'\rangle \in \mathit{prefix}(\tau_A)$ and $\delta'\langle l \circ m'\rangle \notin \mathit{prefix}(\tau_B)$ (otherwise we reverse the roles of τ_A and τ_B). We can thus deduce that, for any γ (even $\gamma = \langle\rangle$), $\delta'\langle l \circ m'\rangle\gamma \notin \tau^*$ (because otherwise, by point 1 of Definition 6.2.2, $\tau^* \not\preceq_\tau \tau_B$). Thus, in order to satisfy point 2 of Definition 6.2.2 for $\tau^* \preceq_\tau \tau_A$, we know that some prefix of δ' must be in τ^* (because $\delta'\langle l \circ m'\rangle\gamma' \in \tau_A$ for some γ'). But this is impossible by Lemma 6.2.4, because $\delta = \delta'\langle l \circ m\rangle\delta'' \in \tau^*$.

So, assuming either $\mathit{top}(\tau_A \downarrow \delta') = \mathit{top}(\tau_B \downarrow \delta') = \emptyset$ or $\mathit{top}(\tau_A \downarrow \delta') \neq \mathit{top}(\tau_B \downarrow \delta')$ leads to a contradiction and we have established the desired result.

2. Let $\delta' \in \tau$. We have to prove point 2 of Definition 6.2.2 (for $\tau^* \preceq_\tau \tau$), namely that $\exists\delta'' \in \mathit{prefix}(\{\delta'\})$ such that $\delta'' \in \tau^*$. We have already proven that τ is a generalisation of both τ_A and τ_B, and thus $\delta' \in \mathit{prefix}(\tau_A) \cap \mathit{prefix}(\tau_B)$. We also know, by the while-condition in Algorithm 6.2.7, that either

   a) $\mathit{top}(\tau_A \downarrow \delta') = \mathit{top}(\tau_B \downarrow \delta') = \emptyset$ or
   b) $\mathit{top}(\tau_A \downarrow \delta') \neq \mathit{top}(\tau_B \downarrow \delta')$.

   Let us examine each of these cases.

   (a) In that case we have $\delta' \in \tau_A \cap \tau_B$ which implies, by point 2 of Definition 6.2.2, that $\exists\delta'' \in \mathit{prefix}(\{\delta'\})$ such that $\delta'' \in \tau^*$ because $\tau^* \preceq_\tau \tau_A$ and $\tau^* \preceq_\tau \tau_B$.

   (b) In that case we can find $\delta'\langle l \circ m\rangle\gamma_A \in \tau_A$ and $l \circ m \notin \mathit{top}(\tau_B \downarrow \delta')$ (otherwise we reverse the roles of τ_A and τ_B). As $\tau^* \preceq_\tau \tau_A$, we know by point 2 of Definition 6.2.2, that a prefix δ'' of $\delta'\langle l \circ m\rangle\gamma_A$ is in τ^*. Finally, δ'' must be a prefix of δ' (otherwise $\delta'' = \delta'\langle l \circ m\rangle\gamma'_A \in \tau^*$ and, as $\delta'\langle l \circ m\rangle\gamma'_A \notin \mathit{prefix}(\tau_B)$ because $l \circ m \notin \mathit{top}(\tau_B \downarrow \delta')$, we cannot have $\tau^* \preceq_\tau \tau_B$ by point 1 of Definition 6.2.2).

□

For $\tau_A \neq \emptyset$ and $\tau_B \neq \emptyset$ we denote by $\mathit{msg}(\tau_A, \tau_B)$ the output of Algorithm 6.2.7. If both $\tau_A = \emptyset$ and $\tau_B = \emptyset$ then $\emptyset$ is the unique most specific generalisation and we therefore define $\mathit{msg}(\emptyset, \emptyset) = \emptyset$. Only in case one of the characteristic trees is empty while the other is not, do we leave the *msg* undefined.

Example 6.2.12 Given $\tau_1 = \{\langle 1 \circ 3\rangle\}$, $\tau_2 = \{\langle 1 \circ 3, 2 \circ 4\rangle\}$, $\tau_3 = \{\langle 1 \circ 3\rangle, \langle 1 \circ 4\rangle\}$, $\tau_4 = \{\langle 1 \circ 3, 2 \circ 4\rangle, \langle 1 \circ 3, 2 \circ 5\rangle\}$, we have that $msg(\tau_1, \tau_2) = \tau_1$, $msg(\tau_1, \tau_3) = msg(\tau_2, \tau_3) = \{\langle\rangle\}$ and $msg(\tau_2, \tau_4) = \tau_1$.

The above Lemma 6.2.10 and Proposition 6.2.11 can also be used to prove an interesting property about the $\preceq_\tau$ relation.

Corollary 6.2.13 Let τ_1 and τ_2 be characteristic trees such that $\tau_1 \preceq_\tau \tau_2$. If $chtree(G, P, U) = \tau_2$ then for some U', $chtree(G, P, U') = \tau_1$.

Proof First we have that Algorithm 6.2.7 will produce for τ_1 and τ_2 the output $\tau = \tau_1$ (because Algorithm 6.2.7 computes the most specific generalisation by Proposition 6.2.11). Hence we have the desired property by Lemma 6.2.10. □

Definition 6.2.14 ($\preceq_{ca}$) A characteristic atom (A_1, τ_1) is *more general* than another characteristic atom (A_2, τ_2), denoted by $(A_1, \tau_1) \preceq_{ca} (A_2, \tau_2)$, iff $A_1 \preceq A_2$ and $\tau_1 \preceq_\tau \tau_2$. Also (A_1, τ_1) is said to be a *variant* of (A_2, τ_2) iff $(A_1, \tau_1) \equiv_{ca} (A_2, \tau_2)$.

The following proposition shows that the above definition safely approximates the optimal but impractical "more general" definition based on the set of concretisations.

Proposition 6.2.15 Let (A, τ_A), (B, τ_B) be two characteristic atoms. If $(A, \tau_A) \preceq_{ca} (B, \tau_B)$ then $\gamma_P(A, \tau_A) \supseteq \gamma_P(B, \tau_B)$.

Proof Let C be a precise concretisation of (B, τ_B). By Definition 5.1.3, there must be an unfolding rule U such that $chtree(\leftarrow C, P, U) = \tau_B$. By Corollary 6.2.13 we can find an unfolding rule U', such that $chtree(\leftarrow C, P, U') = \tau_A$. Furthermore, by Definition 6.2.14, B is an instance of A and therefore C is also an instance of A. We can conclude that C is also a precise concretisation of (A, τ_A). In other words, any precise concretisation of (B, τ_B) is also a precise concretisation of (A, τ_A). The result for general concretisations follows immediately from this by Definition 5.1.3. □

The converse of the above proposition does of course not hold. Take the *member* program from Example 4.2.5 and let $A = member(a, [a])$, $\tau = \{\langle 1 \circ 1\rangle, \langle 1 \circ 2\rangle\}$ and $\tau' = \{\langle 1 \circ 1\rangle\}$. Then $\gamma_P(A, \tau) = \gamma_P(A, \tau') = \{member(a, [a])\}$ (because the resolvent $\leftarrow member(a, [])$ associated with $\langle 1 \circ 2\rangle$ fails finitely) but neither $(A, \tau) \preceq_{ca} (A, \tau')$ nor $(A, \tau') \preceq_{ca} (A, \tau)$ hold.

The following is an immediate corollary of Proposition 6.2.15.

Corollary 6.2.16 Let P be a program and CA, CB be two characteristic atoms such that $CA \preceq_{ca} CB$. If CB is a P-characteristic atom then so is CA.

Finally, we extend the notion of *most specific generalisation (msg)* to characteristic atoms:

Definition 6.2.17 Let (A_1, τ_1), (A_2, τ_2) be two characteristic atoms such that $msg(\tau_1, \tau_2)$ is defined.
Then $msg((A_1, \tau_1), (A_2, \tau_2)) = (msg(A_1, A_2), msg(\tau_1, \tau_2))$.

Note that the above *msg* for characteristic atoms is indeed a most specific generalisation (because $msg(A_1, A_2)$ and $msg(\tau_1, \tau_2)$ are most specific generalisations for the atom and characteristic tree parts respectively) and is still unique up to variable renaming. Its further extension to *sets* of characteristic atoms (rather than just pairs) is straightforward, and will not be included explicitly.

Well-quasi ordering of characteristic atoms.

We now proceed to introduce another order relation on characteristic atoms. It will be instrumental in guaranteeing termination of the refined partial deduction method to be presented.

We recall Definition 3.3.9 from Chapter 3:

Definition 3.3.9 (wqo) A poset $V, \leq_V$ is called *well-quasi-ordered (wqo)* iff for any infinite sequence of elements $e_1, e_2, \ldots$ in V there are $i < j$ such that $e_i \leq_V e_j$. We also say that $\leq_V$ is a *well-quasi order (wqo) on* V.

An interesting wqo is the homeomorphic embedding relation $\trianglelefteq$. It has been adapted from [81, 82], where it is used in the context of term rewriting systems, for use in supercompilation in [258]. Its usefulness as a stop criterion for partial evaluation is also discussed and advocated in [190]. Some complexity results can be found in [264] and [117] (also summarised in [190]).

Recall that expressions are formulated using the alphabet $\mathcal{A}_P$ which we implicitly assume underlying the programs and queries under consideration. Remember that it may contain symbols occurring in no program and query but that it contains only finitely many constant, function and predicate symbols (but always infinitely many variables). The latter property is of crucial importance for some of the propositions and proofs below. In Section 6.4.1 we will present a way to lift this restriction.

Definition 6.2.18 ($\trianglelefteq$) The *homeomorphic embedding* relation $\trianglelefteq$ on expressions is defined inductively as follows:

1. $X \trianglelefteq Y$ for all variables X, Y
2. $s \trianglelefteq f(t_1, \ldots, t_n)$ if $s \trianglelefteq t_i$ for some i
3. $f(s_1, \ldots, s_n) \trianglelefteq f(t_1, \ldots, t_n)$ if $\forall i \in \{1, \ldots, n\} : s_i \trianglelefteq t_i$.

Example 6.2.19 We have that: $p(a) \trianglelefteq p(f(a))$, $X \trianglelefteq X$, $p(X) \trianglelefteq p(f(Y))$, $p(X, X) \trianglelefteq p(X, Y)$ and $p(X, Y) \trianglelefteq p(X, X)$.

Proposition 6.2.20 The relation $\trianglelefteq$ is a wqo on the set of expressions over a finite alphabet.

Proof (Proofs similar to this one are standard in the literature. We include it for completeness.) We first need the following concept from [82]. Let $\leq$ be a relation on a set S of functors (of arity ≥ 0). Then the *embedding extension* of $\leq$ is a relation $\leq_{emb}$ on terms, constructed (only) from the functors in S, which is inductively defined as follows:

1. $s \leq_{emb} f(t_1, \ldots, t_n)$ if $s \leq_{emb} t_i$ for some i
2. $f(s_1, \ldots, s_n) \leq_{emb} g(t_1, \ldots, t_n)$ if $f \leq g$ and $\forall i \in \{1, \ldots, n\} : s_i \leq_{emb} t_i$.

The definition in [82] actually also allows functors of variable arity, but we will not need this in the following. We define the relation $\leq$ on the set $\mathcal{S} = \mathcal{V} \cup \mathcal{F}$ of symbols, containing the (infinite) set of variables $\mathcal{V}$ and the (finite) set of functors and predicates $\mathcal{F}$, as the least relation satisfying:

- $x \leq y$ if $x \in \mathcal{V} \wedge y \in \mathcal{V}$
- $f \leq f$ if $f \in \mathcal{F}$

This relation is a wqo on $\mathcal{S}$ (because $\mathcal{F}$ is finite) and hence by Higman-Kruskal's theorem ([121, 155], see also [82]), its embedding extension to terms, $\leq_{emb}$, which is by definition identical to $\trianglelefteq$, is a wqo on the set of expressions. □

The intuition behind Definition 6.2.18 is that when some structure reappears within a larger one, it is homeomorphically embedded by the latter. As is argued in [190] and [258], this provides a good starting point for detecting growing structures created by possibly non-terminating processes.

However, as can be observed in Example 6.2.19, the homeomorphic embedding relation $\trianglelefteq$ as defined in Definition 6.2.18 is rather crude wrt variables. In fact, all variables are treated as if they were the same variable, a practice which is clearly undesirable in a logic programming context. Intuitively, in the above example, $p(X, Y) \trianglelefteq p(X, X)$ is acceptable, while $p(X, X) \trianglelefteq p(X, Y)$ is not. Indeed $p(X, X)$ can be seen as standing for

something like $and(eq(X,Y),p(X,Y))$, which clearly embeds $p(X,Y)$, but the reverse does not hold.

To remedy the problem (as well as another one related to the *msg* which we discuss later), we refine the above introduced homeomorphic embedding as follows:

Definition 6.2.21 ($\trianglelefteq^*$) Let A, B be expressions. Then B *(strictly homeomorphically) embeds* A, written as $A \trianglelefteq^* B$, iff $A \trianglelefteq B$ and A is not a strict instance of B.

Example 6.2.22 We now still have that $p(X,Y) \trianglelefteq^* p(X,X)$ but not $p(X,X) \trianglelefteq^* p(X,Y)$. Note that still $X \trianglelefteq^* Y$ and $X \trianglelefteq^* X$.

An alternate approach might be based on numbering variables using some mapping $\#(.)$ and then stipulating that $X \trianglelefteq^{\#} Y$ iff $\#(X) \leq \#(Y)$. For instance in [190] a de Bruijn numbering of the variables is proposed. Such an approach, however, has a somewhat ad hoc flavor to it. Take for instance the terms $p(X,Y,X)$ and $p(X,Y,Y)$. Neither term is an instance of the other and we thus have $p(X,Y,X) \trianglelefteq^* p(X,Y,Y)$ and $p(X,Y,Y) \trianglelefteq^* p(X,Y,X)$. Depending on the particular numbering we will have either $p(X,Y,X) \not\trianglelefteq^{\#} p(X,Y,Y)$ or $p(X,Y,Y) \not\trianglelefteq^{\#} p(X,Y,X)$, while there is no apparent reason why one expression should be considered smaller than the other.[2]

Theorem 6.2.23 The relation $\trianglelefteq^*$ is a wqo on the set of expressions over a finite alphabet.

The following lemmas will enable us to prove Theorem 6.2.23.

Lemma 6.2.24 (wqo from wfo) Let $<_V$ be a well-founded order on V. Then $\preceq_V$, defined by $v_1 \preceq_V v_2$ iff $v_1 \not> v_2$, is a wqo on V.

Proof Suppose that there is an infinite sequence $v_1, v_2, \ldots$ of elements of V such that, for all $i < j$, $v_i \not\preceq_V v_j$. By definition this means that, for all $i < j$, $v_i >_V v_j$. In particular this means that we have an infinite sequence with $v_i >_V v_{i+1}$, for all $i \geq 1$. We thus have a contradiction with Definition 3.3.6 of a well-founded order and $\preceq_V$ must be a wqo on V. □

Lemma 6.2.25 Let $\preceq_V$ be a wqo on V and let $\sigma = v_1, v_2, \ldots$ be an infinite sequence of elements of V.

[2] [190] also proposes to consider all possible numberings, but (leading to $n!$ complexity, where n is the number of variables in the terms to be compared). It is unclear how such a relation compares to $\trianglelefteq^*$ of Definition 6.2.21.

1. There exists an $i > 0$ such that the set $\{v_j \mid i < j \wedge v_i \preceq_V v_j\}$ is infinite.

2. There exists an infinite subsequence $\sigma^* = v_1^*, v_2^*, \ldots$ of σ such that for all $i < j$ we have $v_i^* \preceq_V v_j^*$.

Proof The proof will make use of the axiom of choice at several places. Given a sequence ρ, we denote by $\rho_{v' \preceq_V \diamond}$ the subsequence of ρ consisting of all elements v'' which satisfy $v' \preceq_V v''$. Similarly, we denote by $\rho_{v' \not\preceq_V \diamond}$ the subsequence of ρ consisting of all elements v'' which satisfy $v' \not\preceq_V v''$.
Let us now prove point 1. Assume that such an i does not exist. We can then construct the following infinite sequence $\sigma_0, \sigma_1, \ldots$ of sequences inductively as follows:

- $\sigma^0 = \sigma$
- if $\sigma^i = v_i'.\rho^i$ then $\sigma^{i+1} = \rho^i_{v_i' \not\preceq_V \diamond}$

All σ_i are indeed properly defined because at each step only a finite number of elements are removed (by going from ρ^i to $\rho^i_{v_i' \not\preceq_V \diamond}$; otherwise we would have found an index i satisfying point 1). Now the infinite sequence $v_1', v_2', \ldots$ has by construction the property that, for $i < j$, $v_i' \not\preceq_V v_j'$. Hence $\preceq_V$ cannot be a wqo on V and we have a contradiction.
We can now prove point 2. Let us construct $\sigma^* = v_1^*, v_2^*, \ldots$ inductively as follows:

- $\sigma^0 = \sigma$
- if $\sigma^i = r_1, r_2, \ldots$ then $v^*_{i+1} = r_k$ and $\sigma^{i+1} = \rho^i_{r_k \preceq_V \diamond}$ where k is the first index satisfying the requirements of point 1 for the sequence σ^i (i.e. $\{r_j \mid k < j \wedge r_k \preceq_V r_j\}$ is infinite) and where $\rho^i = r_{k+1}, r_{k+2}, \ldots$.

By point 1 we know that each $\rho^i_{r_k \preceq_V \diamond}$ is infinite and σ^* is thus an infinite sequence which, by construction, satisfies $v_i^* \preceq_V v_j^*$ for all $i < j$. □

Lemma 6.2.26 (combination of wqo) Let $\preceq_V^1$ and $\preceq_V^2$ be wqo's on V. Then the quasi order $\preceq_V$ defined by $v_1 \preceq_V v_2$ iff $v_1 \preceq_V^1 v_2$ and $v_1 \preceq_V^2 v_2$, is also a wqo on V.

Proof Let σ be any infinite sequence of elements from V. We can apply point 2 of Lemma 6.2.25 to obtain the infinite subsequence $\sigma^* = v_1^*, v_2^*, \ldots$ of σ such that for all $i < j$ we have $v_i^* \preceq_V^1 v_j^*$. Now, as $\preceq_V^2$ is also a wqo we know that, for some $i < j$, $v_i^* \preceq_V^2 v_j^*$ holds as well. Hence, for these particular indices, $v_i^* \preceq_V v_j^*$ and $\preceq_V$ satisfies the requirements of a wqo on V. □

We can now actually prove Theorem 6.2.23.

Proof of Theorem 6.2.23. $\trianglelefteq^*$ can be expressed as a combination of two

quasi orders on expressions: $\trianglelefteq$ and $\preceq_{NotStrictInst}$ where $A \preceq_{NotStrictInst} B$ iff $B \not\prec A$ (i.e. B is not strictly more general than A or equivalently A is not a strict instance of B). By Lemma 5.3.6 we know that $\prec$ is a well-founded order on expressions. Hence by Lemma 6.2.24 $\preceq_{NotStrictInst}$ is a wqo on expressions. By Proposition 6.2.20 we also have that $\trianglelefteq$ is a wqo on expressions (given a finite underlying alphabet). Hence we can apply Lemma 6.2.26 to deduce that $\trianglelefteq^*$ is also a wqo on expressions over a finite alphabet. □

We now extend the embedding relation of Definition 6.2.21 to characteristic atoms. Notice that the relation $\preceq_\tau$ is not a wqo on characteristic trees, even in the context of a given fixed program P. Take for example the infinite sequence of characteristic trees depicted in Figure 6.1. None of these trees is an instance of any other tree.

One way to obtain a wqo is to first define a term representation of characteristic trees and then apply the embedding relation $\trianglelefteq^*$ to this term representation.

Definition 6.2.27 ($\lceil . \rceil$) By $\lceil . \rceil$ we denote a total mapping from characteristic trees to terms (expressible in some finite alphabet) such that:

- $\tau_1 \prec_\tau \tau_2 \Rightarrow \lceil \tau_1 \rceil \prec \lceil \tau_2 \rceil$ (i.e. $\lceil . \rceil$ is strictly monotonic) and
- $\lceil \tau_1 \rceil \trianglelefteq^* \lceil \tau_2 \rceil \Rightarrow msg(\tau_1, \tau_2)$ is defined.

The conditions of Definition 6.2.27 will be essential for the termination of a partial deduction algorithm to be presented later.

In the following we show that such a mapping $\lceil . \rceil$ actually exists.

Recall that $\tau \downarrow \delta = \{\gamma \mid \delta\gamma \in \tau\}$ and $prefix(\tau) = \{\delta \mid \exists\gamma$ such that $\delta\gamma \in \tau\}$. The following lemma will prove useful to establish the existence of a mapping $\lceil . \rceil$.

Lemma 6.2.28 Let τ_1 and τ_2 be two characteristic trees and let $\delta \in prefix(\tau_1)$ be a characteristic path. If $\tau_1 \preceq_\tau \tau_2$ then $\tau_1 \downarrow \delta \preceq_\tau \tau_2 \downarrow \delta$.

Proof If $\delta' \in \tau_1 \downarrow \delta$ then by definition $\delta\delta' \in \tau_1$. Therefore, by point 1 of Definition 6.2.2, $\delta\delta' \in prefix(\tau_2)$ because $\tau_1 \preceq_\tau \tau_2$. Thus $\delta' \in prefix(\tau_2 \downarrow \delta)$ and point 1 of Definition 6.2.2 is verified for $\tau_1 \downarrow \delta$ and $\tau_2 \downarrow \delta$.

Secondly, if $\delta' \in \tau_2 \downarrow \delta$ then $\delta\delta' \in \tau_2$ and we have, by point 2 of Definition 6.2.2, $\exists\hat{\delta} \in prefix(\{\delta\delta'\})$ such that $\hat{\delta} \in \tau_1$. Now, because $\delta \in prefix(\tau_1)$ we know that $\hat{\delta}$ must have the form $\hat{\delta} = \delta\gamma$ (otherwise we arrive at a contradiction with Lemma 5.2.10) where $\gamma \in prefix(\{\delta'\})$. Thus $\gamma \in \tau_1 \downarrow \delta$ (because $\delta\gamma \in \tau_1$) and also point 2 of Definition 6.2.2 is verified for $\tau_1 \downarrow \delta$ and $\tau_2 \downarrow \delta$. □

Proposition 6.2.29 A function $\lceil . \rceil$ satisfying Definition 6.2.27 exists.

Proof The strict monotonicity condition of Definition 6.2.27 is slightly tricky, but can be satisfied by representing leaves of the characteristic tree by variables. First, recall that $top(\tau) = \{l \circ m \mid \langle l \circ m \rangle \in prefix(\tau)\}$. Let us now define the representation $\lceil \tau \rceil$ of a non-empty characteristic tree τ inductively as follows (using a binary functor m to represent clause matches as well as the usual functors for representing lists):

- $\lceil \tau \rceil = X$ where X is a fresh variable **if** $top(\tau) = \emptyset$
- $\lceil \tau \rceil = [m(m_1, \lceil \tau \downarrow \langle l \circ m_1 \rangle \rceil), \ldots, m(m_k, \lceil \tau \downarrow \langle l \circ m_k \rangle \rceil)]$ **if** $top(\tau) = \{l \circ m_1, \ldots, l \circ m_k\}$ and where $m_1 < \ldots < m_k$.

For example, using the above definition, we have

$$\lceil \{\langle 1 \circ 3 \rangle\} \rceil = [m(3, X)]$$

and

$$\lceil \{\langle 1 \circ 3, 2 \circ 4 \rangle\} \rceil = [m(3, [m(4, X)])].$$

Note that $\{\langle 1 \circ 3 \rangle\} \prec_\tau \{\langle 1 \circ 3, 2 \circ 4 \rangle\}$ and indeed $\lceil \{\langle 1 \circ 3 \rangle\} \rceil \prec_\tau \lceil \{\langle 1 \circ 3, 2 \circ 4 \rangle\} \rceil$. Also note that, because there are only finitely many clause numbers, these terms can be expressed using a finite alphabet.[3]

Note that if $\tau_1 \prec_\tau \tau_2$ we immediately have by Definition 6.2.2 that $\tau_1 \neq \emptyset$ and $\tau_2 \neq \emptyset$. It is therefore sufficient to prove strict monotonicity for two non-empty characteristic trees $\tau_1 \prec_\tau \tau_2$. We will prove this by induction on the depth of τ_1, where the depth is the length of the longest characteristic path in τ_1. We also have to perform an auxiliary induction, showing that $\lceil \tau_1 \rceil \preceq_\tau \lceil \tau_2 \rceil$ whenever $\tau_1 \preceq_\tau \tau_2$.

Induction Hypothesis: For all characteristic trees τ_1 of depth $\leq d$ we have that $\lceil \tau_1 \rceil \prec_\tau \lceil \tau_2 \rceil$ whenever $\tau_1 \prec_\tau \tau_2$ and $\lceil \tau_1 \rceil \preceq_\tau \lceil \tau_2 \rceil$ whenever $\tau_1 \preceq_\tau \tau_2$

Base Case: τ_1 has a depth of 0, i.e. $top(\tau_1) = \emptyset$. This implies that $\lceil \tau_1 \rceil$ is a fresh variable X. If we have $\tau_1 \prec_\tau \tau_2$ then $top(\tau_2) \neq \emptyset$ and $\lceil \tau_2 \rceil$ will be a strict instance of X, i.e. $\lceil \tau_1 \rceil \prec \lceil \tau_2 \rceil$. If we just have $\tau_1 \preceq_\tau \tau_2$ we still have $\lceil \tau_1 \rceil \preceq \lceil \tau_2 \rceil$.

Induction Step: Let τ_1 have depth $d + 1$. This implies that $top(\tau_1) \neq \emptyset$ and, because $\tau_1 \prec_\tau \tau_2$ or $\tau_1 \preceq_\tau \tau_2$, we have by Definition 6.2.2 and Lemma 5.2.10 that $top(\tau_1) = top(\tau_2)$ (more precisely, by point 1 of Definition 6.2.2 we have $top(\tau_1) \subseteq top(\tau_2)$ and by point 2 of Definition 6.2.2

[3] We can make $\lceil . \rceil$ injective (i.e. one-to-one) by adding the numbers of the selected literals. But because these numbers are not a priori bounded we then have to represent them differently than the clause numbers, e.g. in the form of $s(\ldots(0)\ldots)$, in order to stay within a finite alphabet.

combined with Lemma 5.2.10 — the latter affirming that $\langle\rangle \notin \tau_1$ — we get $top(\tau_1) \supseteq top(\tau_2)$). Let $top(\tau_1) = \{l \circ m_1, \ldots, l \circ m_k\}$. Both $\lceil\tau_1\rceil$ and $\lceil\tau_2\rceil$ will by definition have the same top-level term structure — they might only differ in their respective subterms $\{\lceil\tau_1 \downarrow \langle l \circ m_i\rangle\rceil \mid 1 \leq i \leq k\}$ and $\{\lceil\tau_2 \downarrow \langle l \circ m_i\rangle\rceil \mid 1 \leq i \leq k\}$. We can now proceed by induction. First by Lemma 6.2.28 we have that $\tau_1 \downarrow \langle l \circ m_i\rangle \preceq_\tau \tau_2 \downarrow \langle l \circ m_i\rangle$. Furthermore, in case $\tau_1 \prec_\tau \tau_2$, there must be at least one index j such that $\tau_1 \downarrow \langle l \circ m_j\rangle \prec_\tau \tau_2 \downarrow \langle l \circ m_j\rangle$, otherwise $\tau_1 \equiv_\tau \tau_2$. For this index j we can apply the first part of the induction hypothesis (because the depth of the respective sub-trees is strictly smaller) to show that $\lceil\tau_1 \downarrow \langle l \circ m_j\rangle\rceil \prec \lceil\tau_2 \downarrow \langle l \circ m_j\rangle\rceil$. For the other indexes $i \neq j$ we can apply the second part of the induction hypothesis to show that $\lceil\tau_1 \downarrow \langle l \circ m_i\rangle\rceil \preceq \lceil\tau_2 \downarrow \langle l \circ m_i\rangle\rceil$. Finally, because all variables used are fresh and thus distinct, there can be no aliasing between the respective subterms and we can therefore conclude that $\lceil\tau_1\rceil \preceq \lceil\tau_2\rceil$ if $\tau_1 \preceq_\tau \tau_2$ as well as $\lceil\tau_1\rceil \prec \lceil\tau_2\rceil$ if $\tau_1 \prec_\tau \tau_2$.
Remember that $msg(\tau_1, \tau_2)$ is defined unless one of the characteristic trees is empty while the other one is not. Therefore, to guarantee that if $\lceil\tau_1\rceil \trianglelefteq^* \lceil\tau_2\rceil$ holds then $msg(\tau_1, \tau_2)$ is defined, we simply have to ensure that

- $\lceil\emptyset\rceil \trianglelefteq^* \lceil\tau\rceil$ iff $\tau = \emptyset$ and
- $\lceil\tau\rceil \trianglelefteq^* \lceil\emptyset\rceil$ iff $\tau = \emptyset$.

This can be done by defining $\lceil\emptyset\rceil = empty$ where the functor $empty$ is not used in the representation of characteristic trees different from $\emptyset$. □

The existence of such a mapping also implies, by Lemma 5.3.6, that there are no infinite chains of strictly more general characteristic trees. (This would not have been true for a more refined definition of "more general" based on just the set of concretisations.)

Also note that the inverse of the mapping $\lceil.\rceil$, introduced in the proof of Proposition 6.2.29, is not total (it seems to be very difficult or even impossible to find a mapping satisfying Definition 6.2.27 whose inverse is total) but still strictly monotonic. We will not need this property in the following however. But note that, because the inverse is not total, we cannot simply apply the *msg* to the term representation of characteristic trees in order to obtain a generalisation. In other words, the definition of $\lceil.\rceil$ does not make the notion of an *msg* for characteristic trees, as well as Algorithm 6.2.7, superfluous.

From now on, we fix $\lceil.\rceil$ to be a particular mapping satisfying Definition 6.2.27. The mapping developed in the proof of Proposition 6.2.29 is actually a good candidate, as it has the desirable property that the structure of a characteristic tree τ is reflected in the tree-structure of the term $\lceil\tau\rceil$, thus ensuring that the value of $\trianglelefteq$ for spotting non-terminating processes (see e.g. [190]) carries over to characteristic trees.

Definition 6.2.30 ($\trianglelefteq^*_{ca}$) Let $(A_1, \tau_1), (A_2, \tau_2)$ be characteristic atoms. We say that (A_2, τ_2) *embeds* (A_1, τ_1), denoted by $(A_1, \tau_1) \trianglelefteq^*_{ca} (A_2, \tau_2)$, iff $A_1 \trianglelefteq^* A_2$ and $\lceil \tau_1 \rceil \trianglelefteq^* \lceil \tau_2 \rceil$.

Proposition 6.2.31 Let $\tilde{\mathcal{A}}$ be a set of P-characteristic atoms. Then $\tilde{\mathcal{A}}, \trianglelefteq^*_{ca}$ is well-quasi-ordered.

Proof Let $\mathcal{E}$ be the set of expressions over the finite alphabet $\mathcal{A}_P$. By Theorem 6.2.23, $\trianglelefteq^*$ is a wqo on $\mathcal{E}$. Let $\mathcal{F}$ be the alphabet containing just one binary functor $ca/2$ as well as all the elements of $\mathcal{E}$ as constant symbols. Let us extend $\trianglelefteq^*$ from $\mathcal{E}$ to $\mathcal{F}$ by (only) adding that $ca \trianglelefteq^* ca$. $\trianglelefteq^*$ is still a wqo on $\mathcal{F}$, and hence, by Higman-Kruskal's theorem ([121, 155], see also [82]), its embedding extension (see proof of Proposition 6.2.20) to terms constructed from $\mathcal{F}$ is also a wqo. Let us also restrict ourselves to terms $ca(A, T)$ constructed by using this functor exactly once such that A is an atom and T the representation of some characteristic tree. For this very special case, the embedding extension $\trianglelefteq^*_{emb}$ of $\trianglelefteq^*$ coincides with Definition 6.2.30 (i.e. $ca(A_1, \lceil \tau_1 \rceil) \trianglelefteq^*_{emb} ca(A_2, \lceil \tau_2 \rceil)$ iff $(A_1, \tau_1) \trianglelefteq^*_{ca} (A_2, \tau_2)$) and hence $\tilde{\mathcal{A}}, \trianglelefteq^*_{ca}$ is well-quasi-ordered. □

6.2.3 Global trees

In this subsection, we adapt and instantiate the m-tree concept presented in [201] according to our particular needs.

Definition 6.2.32 (global tree) A *global tree* γ_P for a program P is a (finitely branching) tree where nodes can be either *marked* or *unmarked* and each node carries a label which is a P-characteristic atom.

In other words, a node in a global tree γ_P looks like $(n, mark, CA)$, where n is the node identifier, $mark$ an indicator that can take the values m or u, designating whether the node is marked or unmarked, and the P-characteristic atom CA is the node's label. Informally, a marked node corresponds to a characteristic atom which has already been treated by the partial deduction algorithm.

In the sequel, we consider a global tree γ partially ordered through the usual relationship between nodes: $ancestor\_node >_\gamma descendent\_node$. Given a node $n \in \gamma$, we denote by $Anc_\gamma(n)$ the set of its γ ancestor nodes (including itself).

We now introduce the notion of a global tree being well-quasi-ordered, and subsequently prove that it provides a sufficient condition for finiteness. Let γ_P be a global tree. Then we will henceforth denote as Lbl_{γ_P} the set of its labels. And for a given node n in a tree γ, we will refer to its label by lbl_n.

Definition 6.2.33 (label mapping) Let γ be a global tree. Then we define its *associated label mapping* f_γ as the mapping $f_\gamma : (\gamma, >_\gamma) \rightarrow (Lbl_\gamma, \trianglelefteq^*_{ca})$ such that $n \mapsto lbl_n$. f_γ will be called *quasi-monotonic* iff $\forall n_1, n_2$ $n_1 >_\gamma n_2 \Rightarrow lbl_{n_1} \not\trianglelefteq^*_{ca} lbl_{n_2}$.

Definition 6.2.34 We call a global tree γ *well-quasi-ordered* if f_γ is quasi-monotonic.

Theorem 6.2.35 A global tree γ is finite if it is well-quasi-ordered.

Proof Assume that γ is not finite. Then it contains (König's Lemma) at least one infinite branch $n_1 >_\gamma n_2 >_\gamma \ldots$. Consider the corresponding infinite sequence of elements $lbl_{n_1}, lbl_{n_2}, \ldots \in Lbl_\gamma, \trianglelefteq^*_{ca}$. From Proposition 6.2.31, we know that $Lbl_\gamma, \trianglelefteq^*_{ca}$ is wqo and therefore, there must exist $lbl_{n_i}, lbl_{n_j}, i < j$ in the above mentioned sequence such that $lbl_{n_i} \trianglelefteq^*_{ca} lbl_{n_j}$. But this implies that f_γ is not quasi-monotonic. □

6.2.4 A tree based algorithm

In this subsection, concluding Section 6.2, we present the actual refined partial deduction algorithm where global control is imposed through characteristic atoms in a global tree.

A formal description of the algorithm can be found in Figure 6.6. Please note that it is parametrised by an unfolding rule U, thus leaving the particulars of the local control unspecified. As for Algorithm 5.3.3, we need the notation $chatom(A, P, U)$ (see Definition 5.3.2). Also, without loss of generality, we suppose that the initial goal contains just a single atom (otherwise we get a global forest instead of a global tree).

As in e.g. [98, 201] (but unlike Algorithm 5.3.3), Algorithm 6.2.36 does not output a specialised program, but rather a set of (characteristic) atoms from which the actual code can be generated in a straightforward way. Most of the algorithm is self-explanatory, except perhaps the inner **while**-loop. In $\tilde{\mathcal{B}}$, all the characteristic atoms are assembled, corresponding to the atoms occurring in the leaves of the SLDNF-tree built for A_n according to τ_{A_n}. Elements of $\tilde{\mathcal{B}}$ are subsequently inserted into γ as (unmarked) child nodes of L if they do not embed the label of n or any of its ancestor nodes. If one does, and it is a variant of n's label or that of another node in γ, then it is simply not added to γ. (Note that one can change to an instance test by simply replacing $\preceq_{ca}$ by $\equiv_{ca}$.) Finally, if a characteristic atom $CA_B \in \tilde{\mathcal{B}}$ does embed an ancestor label, but there is no variant to be found in γ, then the most specific generalisation M of CA_B and all embedded ancestor labels $\tilde{\mathcal{H}}$ is re-inserted into $\tilde{\mathcal{B}}$. The latter case is of course the most interesting:

Algorithm 6.2.36

Input
a normal program P and goal $\leftarrow A$
Output
a set of characteristic atoms $\tilde{\mathcal{A}}$
Initialisation
$\gamma := \{(1, u, (A, \tau_A))\}$;
while γ contains an unmarked leaf **do**
let n be such an unmarked leaf in γ: $(n, u, (A_n, \tau_{A_n}))$;
mark n;
$\tilde{\mathcal{B}} := \{chatom(B, P, U) \mid B \in leaves_P(A_n, \tau_{A_n})\}$;
while $\tilde{\mathcal{B}} \neq \emptyset$ **do**
select $CA_B \in \tilde{\mathcal{B}}$;
remove CA_B from $\tilde{\mathcal{B}}$;
if $\tilde{\mathcal{H}} = \{CA_C \in Anc_\gamma(n) | CA_C \trianglelefteq^*_{ca} CA_B\} = \emptyset$ **then**
add (n_B, u, CA_B) to γ as a child of n;
else if $\nexists CA_D \in Lbl_\gamma$ such that $CA_D \equiv_{ca} CA_B$ **then**
add $msg(\tilde{\mathcal{H}} \cup \{CA_B\})$ to $\tilde{\mathcal{B}}$;
end while
end while
return $\tilde{\mathcal{A}} := Lbl_\gamma$

Figure 6.6: Partial deduction with global trees.

Simply adding a node labelled CA_B would violate the well-quasi ordering of the tree and thus endanger termination. Calculating the *msg* M (which always exists by the conditions of Definition 6.2.27) and trying to add it instead secures finiteness, as proven below, while trying to preserve as much information as seems possible (see however Sections 6.3 and 6.4). Note that, similarly to CA_B, we cannot add the *msg* M directly to the tree because this might produce a tree which is not well-quasi ordered (and termination would also be endangered). Indeed, although by generalising CA_B into M we are actually sure that M will not embed any characteristic atom in $\tilde{\mathcal{H}}$ (for a proof of this property see [178]) it might embed other characteristic atoms in the tree. Take for example a branch in a global tree which contains the characteristic atoms $(p(f(a)), \tau)$ and $(p(X), \tau)$. Then $CA_B = (p(f(f(a))), \tau)$ embeds $(p(f(a)), \tau)$ but not $(p(X), \tau)$. Thus $\tilde{\mathcal{H}} = \{(p(f(a)), \tau)\}$ and $msg(\{(p(f(a)), \tau), (p(f(f(a))), \tau)\}) = (p(f(X)), \tau)$,

which no longer embeds $(p(f(a)), \tau)$ but now embeds $(p(X), \tau)$! So, to ensure that the global tree remains well-quasi ordered it is important to re-check the *msg* for embeddings before adding it to the global tree.

We obtain the following theorems:

Theorem 6.2.37 Algorithm 6.2.36 always terminates.

Proof Upon each iteration of the outer **while**-loop in Algorithm 6.2.36, exactly one node in γ is marked, and zero or more (unmarked) nodes are added to γ. Moreover, Algorithm 6.2.36 never deletes a node from γ, neither does it ever "unmark" a marked node. Hence, since all branchings are finite, non-termination of the outer **while**-loop must result in the construction of an infinite branch. It is therefore sufficient to argue that the inner **while**-loop terminates and that after every iteration of the outer loop, γ is a wqo global tree.
First, this holds after initialisation. Also, obviously, a global tree will be converted into a new global tree through the outer **while**-loop. Now, a while-iteration adds zero or more, but finitely many, child nodes to a particular leaf n in the tree, thus creating a (finite) number of new branches that are extensions of the old branch leading to n. We prove that on all of the new branches, f_γ is quasi-monotonic. The branch extensions are actually constructed in the inner **while**-loop, at most one for every element of $\tilde{\mathcal{B}}$. So, let us take an arbitrary characteristic atom $CA_B \in \tilde{\mathcal{B}}$, then there are three cases to consider:

1. Either CA_B does not embed any label on the branch up to (and including) n. It is then added in a fresh leaf. Obviously, f_γ will be quasi-monotonic on the newly created branch.

2. Or some such label is embedded, but there is also a variant (or more general respectively, in case an instance test is used) label already in some node of the tree γ. Then, no corresponding leaf is inserted in the tree, and there is nothing left to prove.

3. Or, finally, some labels on the branch are embedded, but no variants (or more general characteristic atoms respectively) are to be found in γ. We then calculate the *msg* M of CA_B and all[4] the labels $\tilde{\mathcal{H}} = \{L_1, \ldots, L_k\}$ on the branch it embeds. In that case, M must be strictly more general than CA_B. Indeed, if M would be a variant of CA_B then CA_B must be more general than all the elements in $\tilde{\mathcal{H}}$ (by property of the *msg*), and even strictly more general because no label was found of which it was a variant (or instance respectively). This is

[4] The algorithm would also terminate if we only pick one such label at every step.

in contradiction with the definition of $\trianglelefteq^*$, which requires that each L_i is not a strict instance of CA_B for $L_i \trianglelefteq^* CA_B$ to hold. (More precisely, given $L_i = (A_i, \tau_i)$ and $CA_B = (A_B, \tau_B)$, $L_i \trianglelefteq^* CA_B$ implies that A_i is not a strict instance of A_B and that $\lceil \tau_i \rceil$ is not a strict instance of $\lceil \tau_B \rceil$; the latter implies, by strict monotonicity[5] of $\lceil . \rceil$ that τ_i is not a strict instance of τ_B and thus, by Definition 6.2.14, we have that L_i is not a strict instance of CA_B.) So in this step we have not modified the tree (and it remains wqo), but replaced an atom in $\tilde{\mathcal{B}}$ by a strictly more general one, which we can do only finitely many times (by Lemma 5.3.6) and thus termination of the inner **while**-loop is ensured (as in the other two cases above an element is removed from $\mathcal{B}$ and none are added).

Note that, when using the $\trianglelefteq$ relation instead of $\trianglelefteq^*$, M would not necessarily be more general than CA_B, i.e. the algorithm could loop. For example, take a global tree having the single node $(p(X,X), \tau)$ and where we try to add $\tilde{\mathcal{B}} = \{(p(X,Y), \tau)\}$. Now we have that $p(X,X) \trianglelefteq p(X,Y)$ and we calculate the *msg*

$$msg(\{(p(X\,,X), \tau), (p(X\,,Y), \tau)\}) = (p(X\,,Y), \tau)$$

and we have a loop.

□

Theorem 6.2.38 Let P be a program, input to Algorithm 6.2.36, and $\tilde{\mathcal{A}}$ the corresponding set of characteristic atoms produced as output. Then $\tilde{\mathcal{A}}$ is P-covered.

Proof First, it is straightforward to prove that throughout the execution of Algorithm 6.2.36, any unmarked node in γ must be a leaf. It therefore suffices to show, because the output contains only marked nodes, that after each iteration of the outer **while**-loop, only unmarked leaves in γ possibly carry a non-covered label.[6] Trivially, this property holds after initialisation. Now, in the outer **while**-loop, one unmarked leaf n is selected and marked. The inner **while**-loop then precisely proceeds to incorporate (unmarked leaf) nodes into γ such that all leaf atoms of n's label are concretisations of at least one label in the new, extended γ. □

The correctness of the specialisation now follows from Theorem 5.2.2 presented earlier.

[5] Note that the proof also goes through if $\lceil . \rceil$ is not strictly monotonic but just satisfies that whenever τ_i is a strict instance of τ_B then $\tau_i \not\trianglelefteq^* \tau_B$.

[6] I.e. a label with at least one leaf atom (in P) that is not a concretisation of any label in γ.

6.3 Post-processing and other improvements

6.3.1 Removing superfluous polyvariance

In this section we first outline a possible post-processing phase to remove superfluous polyvariance. As mentioned in Chapter 4.5, this might not always be a good idea, e.g. when the specialised program is further analysed by a *monovariant* abstract interpretation phase which cannot generate further polyvariance by itself. In that case, it is impossible for the partial deducer to know exactly which polyvariance is superfluous and which is not. However, if the specialised program is destined to be executed directly, without any further specialisation, then it is possible to reduce the polyvariance to some minimal level which does not remove any of the specialisation performed by the proper partial deduction phase described in Algorithm 6.2.36.

Indeed, unlike ecological partial deduction as presented in Section 5.3, Algorithm 6.2.36 will obviously often output several characteristic atoms with the same characteristic tree, each giving rise to a different specialised version of the same original predicate definition. Such "duplicated" polyvariance is however superfluous, in the context of simply running the resulting program, when it increases neither local nor global precision. As far as preserving local precision is concerned, matters are simple: One procedure per characteristic tree is what you want. The case of global precision is slightly more complicated: Generalising atoms with identical characteristic trees might lead to the occurrence of more general atoms in the leaves of the associated local tree. In other words, we might loose subsequent instantiation at the global level, possibly leading to a different and less precise set of characteristic atoms.

The polyvariance reducing post-processing that we propose in this section therefore avoids the latter phenomenon. In order to obtain the desired effect, it basically collapses and generalises several characteristic atoms with the same characteristic tree only if this does not modify the global specialisation. To that end we number the leaf atoms of each characteristic atom and then label the arcs of the global tree with the number of the leaf atom it refers to. We also add arcs in case a leaf atom is a variant of another characteristic atom in the tree and has therefore not been lifted to the global level. We thus obtain a *labelled global graph*. We then try to collapse nodes with identical characteristic trees using the well-known algorithm for minimisation of finite state automata [1, 128]: we start by putting all characteristic atoms with the same characteristic tree into the same class, and subsequently split these classes if corresponding leaf atoms fall into different classes. As stated in [128], the complexity of this algorithm is $O(kn^2)$

where n is the maximum number of states (in our case the number of characteristic atoms) and k the number of symbols (in our case the maximum number of leaf atoms).

The following example illustrates the use of this minimisation algorithm for removing superfluous polyvariance.

Example 6.3.1 Let us return to the *member* program of Example 4.2.5, augmented with one additional clause:

(1) $member(X, [X \mid T]) \leftarrow$
(2) $member(X, [Y \mid T]) \leftarrow member(X, T)$
(3) $t(T) \leftarrow member(a, [a, b, c, d \mid T]), member(b, T)$

Suppose that after executing Algorithm 6.2.36 we obtain the following set of characteristic atoms $\tilde{\mathcal{A}} = \{(member(b, L), \tau), (member(a, [a, b, c, d \mid T]), \tau), (member(a, [b, c, d \mid T]), \tau'), (member(a, L), \tau), (t(T), \{\langle 1 \circ 3 \rangle\})\}$ where $\tau = \{\langle 1 \circ 1 \rangle, \langle 1 \circ 2 \rangle\}$ and $\tau' = \{\langle 1 \circ 2, 1 \circ 2, 1 \circ 2 \rangle\}$. Depending on the particular renaming function, a partial deduction of P wrt $\tilde{\mathcal{A}}$ will look like:

$mem_{a,[a,b,c,d]}(T) \leftarrow$
$mem_{a,[a,b,c,d]}(T) \leftarrow mem_{a,[b,c,d]}(T)$
$mem_{a,[b,c,d]}(T) \leftarrow mem_a(T)$
$mem_a([a|T]) \leftarrow$
$mem_a([Y|T]) \leftarrow mem_a(T)$
$mem_b([b|T]) \leftarrow$
$mem_b([Y|T]) \leftarrow mem_b(T)$
$t(T) \leftarrow mem_{a,[a,b,c,d]}(T), mem_b(T)$

The labelled graph version of the corresponding global tree can be found in Figure 6.7. Adapting the algorithm from [1, 128] to our needs, we start out by generating three classes (one for each characteristic tree) of states:

- $C_1 = \{(member(a, [a, b, c, d \mid T]), \tau), (member(a, L), \tau), (member(b, L), \tau)\}$,
- $C_2 = \{(member(a, [b, c, d \mid T]), \tau')\}$ and
- $C_3 = \{(t(T), \{\langle 1 \circ 3 \rangle\})\}$.

The class C_1 must be further split, as the state $(member(a, [a, b, c, d \mid T]), \tau)$ has a transition via the label #1 to a state of C_2 while the other states of C_1, $(member(a, L), \tau)$ and $(member(b, L), \tau)$, do not. This means that by actually collapsing all elements of C_1 we would lose subsequent specialisation at the global level (namely the pruning and pre-computation that is performed within $(member(a, [b, c, d \mid T]), \tau')$).
We now obtain the following 4 classes:

- $C_2 = \{(member(a, [b, c, d \mid T]), \tau')\}$,
- $C_3 = \{(t(T), \{\langle 1 \circ 3 \rangle\})\}$,
- $C_4 = \{(member(a, [a, b, c, d \mid T]), \tau)\}$ and
- $C_5 = \{(member(a, L), \tau), (member(b, L), \tau)\}$.

The only class that might be further split is C_5, but both states of C_5 have transitions with identical labels leading to the same classes, i.e. no specialisation will be lost by collapsing the class. We can thus generate one characteristic atom per class (by taking the *msg* of the atom parts and keeping the characteristic tree component). The resulting, minimised program is thus:

$$
\begin{aligned}
&mem_{a,[a,b,c,d]}(T) \leftarrow \\
&mem_{a,[a,b,c,d]}(T) \leftarrow mem_{a,[b,c,d]}(T) \\
&mem_{a,[b,c,d]}(T) \leftarrow mem_x(a,T) \\
&mem_x(X,[X|T]) \leftarrow \\
&mem_x(X,[Y|T]) \leftarrow mem_x(X,T) \\
&t(T) \leftarrow mem_{a,[a,b,c,d]}(T), mem_x(b,T)
\end{aligned}
$$

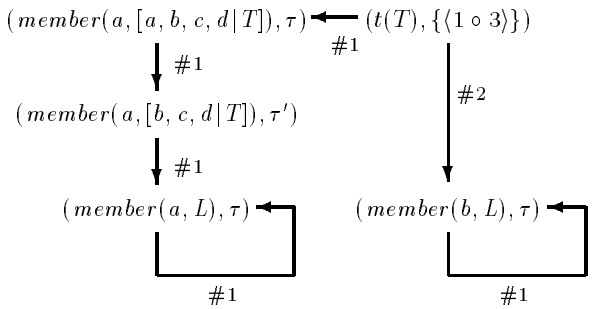


Figure 6.7: Labelled global graph of Example 6.3.1 before post-processing

A similar use of this minimisation algorithm is made in [285] and [237]. The former aims at minimising polyvariance in the context of multiple specialisation by optimising compilers. The latter, in a somewhat different context, studies polyvariant parallelisation and specialisation of logic programs based on abstract interpretation.

6.3.2 Other improvements

Upon close inspection, Algorithm 6.2.36 itself might be made more precise by choosing another, less general, label to insert upon generalisation. As it currently stands, the algorithm computes the *msg* of the characteristic atom to be added and the labels of all its embedded (future) ancestors. Now, there may be less general generalisations of CA_B that likewise do not embed any ancestor label, and can be added safely instead of CA_B.

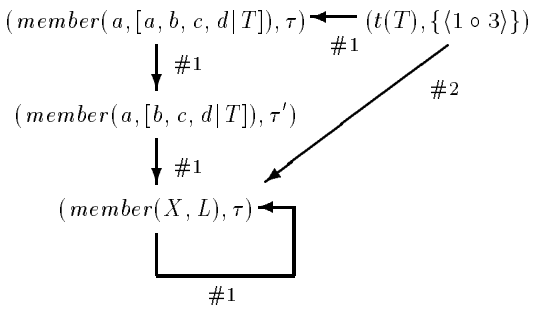


Figure 6.8: Labelled global graph of Example 6.3.1 after post-processing

For example, given $\tilde{\mathcal{H}} = \{(p(a,b), \tau_B)\}$ and $B = p(f(a), f(b))$, the atom part of $msg(\tilde{\mathcal{H}} \cup \{(B, \tau_B)\})$ is $p(X,Y)$. However, both $p(f(X), f(b))$ and $p(f(a), f(Y))$ are more specific generalisations of B that nevertheless do not embed $p(a,b)$. A similar consideration applies to the case where no node is added since some ancestor carries a label, more general than CA_B. To what extent such refinements would significantly influence the specialisation potential, and, if so, whether it is actually possible to incorporate them into a concrete algorithm, are topics for future research.

One further possibility for improvement lies in refining the ordering relation $\preceq_{ca}$ on characteristic atoms and the related *msg* operator, so that they more precisely capture the intuitive, but uncomputable order based on the set of concretisations.[7] Alternatively, one could try to use an altogether more accurate abstraction operator than taking an *msg* on characteristic atoms. For instance, one can endeavour to extend the constraint based abstraction operator proposed in [172] (and briefly discussed in Section 5.4.2) to normal programs and arbitrary unfolding rules. This would probably result in a yet (slightly) more precise abstraction, causing a yet smaller global precision loss.

Finally, one might also try to incorporate more detailed efficiency and cost estimates into the global control, e.g. based on [70, 71, 69], in order to analyse the trade-off between improved specialisation and increased polyvariance and code size.

[7] For example, suppose that a given predicate is defined by 3 clauses, numbered 1 to 3. Then the *msg* of $\tau = \{\langle 1 \circ 1\rangle\}$ and $\tau' = \{\langle 1 \circ 2\rangle\}$, according to Definitions 6.2.14 and 6.2.17, would be $\{\langle\rangle\}$, meaning that the associated specialised predicate definition would contain a resultant for clause (3) (see Definition 5.1.8). This could be remedied by using a more refined order relation $\preceq^*_{ca}$ for which $\tau^* = \{\langle 1 \circ 1\rangle, \langle 1 \circ 2\rangle\} \preceq^*_{ca} \tau$ and $\tau^* \preceq^*_{ca} \tau'$. Hence the *msg* of τ and τ' would no longer be $\{\langle\rangle\}$, but the more precise τ^*.

6.4 Experimental results and discussion

6.4.1 Systems

In this section we present an implementation of the ideas of the preceding sections, as well as an extensive set of experiments which highlight the practical benefits of the implementation over existing partial deduction systems.

The system which integrates the ideas of Chapters 5 and 6, called ECCE, is publicly available in [170] and is actually an implementation of a generic version of Algorithm 6.2.36 which allows the advanced user to change and even implement e.g. the unfolding rule as well as the abstraction operator and non-termination detection method. For instance, by adapting the settings of ECCE, one can obtain exactly Algorithm 6.2.36. But one can also simulate a (global tree oriented) version of Algorithm 5.3.3 using depth bounds to ensure termination. The system even offers some further possibilities, beyond (ecological) partial deduction. We will return to this aspect of the system and present the underlying generic algorithm of ECCE in Chapter 12.

All unfolding rules of ECCE were complemented by simple *more specific resolution* steps in the style of SP [97]. Constructive negation (see [49],[115]) has not yet been incorporated, but the selection of ground negative literals is allowed. Post-processing removal of unnecessary polyvariance, using the algorithm outlined in Section 6.3, determinate post-unfolding as well as redundant argument filtering (which we will present later in Chapter 11) were enabled throughout the experiments discussed below.

The ECCE system also handles a lot of Prolog built-ins, like for instance =, is, <, =<, <, >=, *nonvar*, *ground*, *number*, *atomic*, *call*, \==, \=. All built-ins are supposed to be declarative and their selection delayed until they are sufficiently instantiated (e.g. the built-in *ground* is supposed to be delayed until its argument is ground, i.e. *ground* just functions as a delay declaration and is probably the least interesting of the above built-ins). The method presented in this chapter is extended by also registering built-ins in the characteristic trees. One problematic aspect is that, when generalising calls to built-ins which generate bindings (like is/2 or =../2 but unlike >/2 or </2) and which are no longer executable after generalisation, these built-ins have to be removed from the generalised characteristic tree (i.e. they are no longer selected). With that, the concretisation definition for characteristic atoms scales up and the technique will ensure correct specialisation. It should also be possible to incorporate the `if-then-else` into characteristic trees.

Also, the embedding relation $\trianglelefteq$ of Definition 6.2.18 (and the relations

$\trianglelefteq^*$ and $\trianglelefteq^*_{ca}$ based on it) has to be adapted. Indeed, some built-ins (like $= ../2$ or $is/2$) can be used to dynamically construct infinitely many new constants and functors and thus $\trianglelefteq$ is no longer a wqo. To remedy this, the constants and functors are partitioned into the *static* ones, occurring in the original program and the partial deduction query, and the *dynamic* ones. (This approach is also used in [244, 245].) The set of dynamic constants and functors is possibly infinite, and we will therefore treat it like the infinite set of variables in Definition 6.2.18 by adding the following rule to the ECCE system:

$$f(s_1, \ldots, s_m) \trianglelefteq^* g(t_1, \ldots, t_n) \text{ if both } f \text{ and } g \text{ are dynamic}$$

Some dynamic functors, which already have a natural wqo (or a wfo, which can be turned into a wqo by Lemma 6.2.24) associated with them, might be treated in a more refined way. For instance for integers we can define:

$$i \trianglelefteq j \text{ if both } i \text{ and } j \text{ are integers and } i \leq j.$$

An even more refined solution (not implemented for the current experiments), might be based on using the *general* homeomorphic embedding relation of [155], which can handle infinitely many function symbols provided that a wqo $\preceq$ on the function symbols is given:[8]

$$f(s_1, \ldots, s_m) \trianglelefteq^* g(t_1, \ldots, t_n) \text{ if } f \preceq g \text{ and } \exists 1 \leq i_1 < \ldots i_m \leq n \text{ such that } \forall j \in \{1, \ldots, m\} : s_j \trianglelefteq^* t_{i_j}.$$

We present benchmarks using 3 different settings of ECCE, hereafter called ECCE-D, ECCE-X-10 and ECCE-X. The settings ECCE-D and ECCE-X use Algorithm 6.2.36, with a different unfolding rule, while ECCE-X-10 uses a (global tree oriented) version of Algorithm 5.3.3 with a depth bound of 10 to ensure termination. We also compare with MIXTUS [245, 244], PADDY [227, 228, 229] and SP [97, 98], of which the following versions have been used: MIXTUS 0.3.3, the version of PADDY delivered with ECLIPSE 3.5.1 and a version of SP dating from September 25$^{\text{th}}$, 1995.

Basically, the above mentioned systems use the following two different unfolding rules:

- *"MIXTUS-like" unfolding*: This is the unfolding strategy explained in [245, 244] which in general unfolds deeply enough to solve the "fully unfoldable" problems[9] but also has safeguards against excessive unfolding and code explosion. It requires however a number of ad hoc

[8] A simple one might be $f \preceq g$ if both f and g are dynamic or if $f = g$.

[9] I.e. those problems for which normal evaluation terminates (cf. the end of Section 3.1 in Chapter 3).

settings. (In the future, we plan experiments with unfolding along the lines of [199] which is free of such elements.) For instance, for ECCE-X and ECCE-X-10 we used the settings (see [245]) $max\_rec = 2$, $max\_depth = 2$, $maxfinite = 7$, $maxnondeterm = 10$ and only allowed non-determinate unfolding when no user predicates were to the left of the selected literal. The method ECCE-X-10 was also complemented by a level 10 depth bound. For MIXTUS and PADDY we used the respective default settings of the systems. Note that the "MIXTUS-like" unfolding strategies of ECCE, MIXTUS and PADDY differ slightly from each other, probably due to some details not fully elaborated in [245, 227] as well as the fact that the different global control regimes influence the behaviour of the "MIXTUS-like" local control.

- *Determinate unfolding*: Only (except once) select atoms that match a single clause head. The strategy is refined with a "lookahead" to detect failure at a deeper level (c.f. Definition 3.3.2). This approach is used by ECCE-D and SP. Note however that SP seems to employ a refined determinate unfolding rule (as indicated e.g. by the results for the benchmark *depth.lam* below).

Both of these unfolding rules actually ensure that neither the number nor the order of the solutions under Prolog execution are altered. Also, termination under Prolog will be preserved by these unfolding rules (termination behaviour might however be improved, as e.g. $\leftarrow loop, fail$ can be transformed into $\leftarrow fail$). For more details related to the preservation of (Prolog) termination we refer to [230, 27, 30].

6.4.2 Experiments

The benchmark programs can be found in [170], short descriptions are presented in Appendix C. In addition to the "Lam & Kusalik" benchmarks (originally in [159]) they contain a whole set of more involved and realistic examples, like e.g. a model-elimination theorem prover and a meta-interpreter for an imperative language. For some of these new benchmark tasks it is impossible to get (big) speedups — the goal of these tasks consists in testing whether no pessimisation, code explosion or non-termination occurs.

For the experimentation, we tried to be as realistic and practice-oriented as possible. In particular, we did not count the number of inferences, the cost of which varies a lot, or some other abstract measure, but the actual execution time and size of compiled code. The results are summarised in Table 6.1, while the full details can be found in Tables 6.2, 6.3 and 6.4. Further details and explanations about the benchmarks are listed below:

- Transformation times (TT):
  The transformation times of ECCE and MIXTUS also include the time to write the specialised program to file. Time for SP does not and for PADDY we do not know. We briefly explain the use of ∞ in the tables:

  - ∞, SP: this means "real" non-termination (based upon the description of the algorithm in [97])
  - ∞, MIXTUS: heap overflow after 20 minutes
  - ∞, PADDY: thorough system crash after 2 minutes

  In Tables 6.2, 6.3 and 6.4, the transformation times (TT) are expressed in seconds while the total transformation time in Table 6.1 is expressed in minutes (on a Sparc Classic running under Solaris, except for PADDY which for technical reasons had to be run on a Sun 4). Each system was executed using the Prolog system it runs on: Prolog by BIM for ECCE, SICStus Prolog for MIXTUS and SP and Eclipse for PADDY). So, except when comparing the different settings of ECCE, the transformation times should thus only be used for a rough comparison.

  It seems that the latest version 0.3.6 of MIXTUS does terminate for the missionaries example, but we did not yet redo the experiments. PADDY and SP did not terminate for one other example (memo-solve and imperative.power respectively) when we accidentally used *not* instead of \+ (*not* is not defined in SICStus Prolog; PADDY and SP follow this convention and interpreted *not* as an undefined, failing predicate). After changing to \+, both systems terminated.

- Relative Runtimes (RRT) of the specialised code:
  The timings are not obtained via a loop with an overhead but via special Prolog files, generated automatically by ECCE. These files call the original and specialised programs directly (i.e. without overhead), at least 100 times for the respective run-time queries, using the $time/2$ predicate of Prolog by BIM 4.0.12 on a Sparc Classic under Solaris. Sufficient memory was given to the Prolog system to prevent garbage collection. Runtimes in Tables 6.2, 6.3 and 6.4 are given *relative* to the runtimes of the original programs. In computing averages and totals, the time and size of the original program were taken in case of non-termination (i.e. we did not punish MIXTUS, PADDY and SP for the non-termination). The total speedups are obtained by the formula

$$\frac{n}{\sum_{i=1}^{n} \frac{spec_i}{orig_i}}$$

System	Total Speedup	Worst Speedup	FU Speedup	Not FU Speedup	Total Code Size in KB	Total TT in min
ECCE-D	1.90	0.85	2.57	1.74	166.69	2.64
ECCE-X-10	2.13	0.79	7.07	1.71	224.35	112.72
ECCE-X	2.51	0.92	8.36	2.02	135.91	2.70
MIXTUS	2.08	0.65	8.13	1.65	152.26	∞+2.49
PADDY	2.08	0.68	8.12	1.65	196.19	∞+0.28
SP	1.46	0.86	2.08	1.32	182.02	3∞+1.92

Table 6.1: Short summary of the results (higher speedup and lower code size is better)

where n is the number of benchmarks and $spec_i$ and $orig_i$ are the absolute execution times of the specialised and original programs respectively. In Table 6.1 the column for "FU" holds the total speedup for the fully unfoldable benchmarks (see Appendix C) while the column for "Not FU" holds the total speedup for the benchmarks which are not fully unfoldable.

All timings were for renamed queries, except for the original programs and for SP (which does not rename the top-level query — this puts SP at a disadvantage of about 10% in average for speed but at an advantage for code size). Note that PADDY systematically included the original program and the specialised part could only be called in a renamed style. We removed the original program whenever possible and added 1 clause which allows calling the specialised program also in an unrenamed style (just like MIXTUS and ECCE). This possibility was not used in the benchmarks but avoids distortions in the code size figures (wrt MIXTUS and ECCE).

Note that timing in Prolog (by BIM), especially on Sparc machines, can sometimes be problematic. It is not uncommon that a simple reordering of the predicates can lead to a 10% difference in speed (and in some rare cases even more, as we will see later in Chapter 12). The problem is probably due to the caching performed by the Sparc processor.

- Size of the specialised code:
  The compiled code size was obtained via *statistics*/4 and is expressed in units, were 1 unit = 4.08 bytes (in the current implementation of Prolog by BIM).

6.4.3 Analysing the results

The ECCE based systems did terminate on all examples, as would be expected by the results presented earlier in the chapter. To our surprise however, all the existing systems, MIXTUS, PADDY and SP, did not properly terminate for at least one benchmark each. Even ignoring this fact, the system ECCE-X clearly outperforms MIXTUS, PADDY and SP, as well for speed as for code size, while having the best worst case performance. Even though ECCE is still a prototype, the transformation times are reasonable and usually close to the ones of MIXTUS. ECCE can certainly be speeded up considerably, maybe even by using the ideas in [229] which help PADDY to be (except for one glitch) the fastest system overall.

Note that even the system ECCE-X-10 based on a depth bound of 10, outperforms the existing systems on account of the speed of the specialised programs. Its transformation times as well as the size of the specialised code are not so good however. Also note that for some benchmarks the ad-hoc depth bound of 10 was too shallow (e.g. relative) while for others it was too deep and resulted in excessive transformation times (e.g. model-elim.app). But by removing the depth bound (ECCE-X) we increase the total speedup from 2.13 to 2.51 while decreasing the size of the specialised code from 224 KB down to 136 KB. Also the total transformation time drastically decreases by a factor of 42. This clearly illustrates that getting rid of the depth bound is a very good idea in practice.

The difference between ECCE-D and ECCE-X in the resulting speedups shows that determinate unfolding, at least in the context of standard partial deduction,[10] is in general not sufficient for fully satisfactory specialisation. The "MIXTUS-like" unfolding seems to be a good compromise for standard partial deduction. Also note that for the not fully unfoldable benchmarks (most applications will fall into this category: if a benchmark is fully unfoldable one does not need a partial evaluator, normal evaluation suffices) ECCE-D actually outperforms MIXTUS, PADDY and SP (but not ECCE-X).

The total speedup of ECCE-X is 2.51 and the speedup for the fully unfoldable benchmarks is 2.02, i.e. by partial deduction we were able to cut execution time more than in half. Compared to speedups that are usually obtained by low-level compiler optimisations, these figures are extremely satisfactory. Taken on its own, however, these figures might look a bit disappointing as to the potential of partial deduction. But, as already mentioned, for some benchmark tasks it is impossible to get significant speedups. Also, partial deduction is of course not equally suited for all tasks; but for those tasks for which it is suited, partial deduction can be even

[10] See Chapter 12 for benchmarks on "conjunctive partial deduction", in which determinate unfolding is better than MIXTUS-like unfolding.

much more worthwhile. For instance for the benchmarks model_elim.app, liftsolve.app and liftsolve.db1 — which exhibit interpretation overhead — ECCE-X obtains speedup factors of about 8, 17 and 50 respectively. Getting rid of higher-order overhead also seems highly beneficial, yielding about one order of magnitude speedup. Another area in which partial deduction might be very worthwhile is handling overly general programs, e.g. getting rid of unnecessary intermediate variables or making use of the hidden part of abstract data types. We will return to some such programs (and the techniques required to adequately specialise them) in Part V of the thesis. So, given the difficulty of the benchmarks (and the worst case slowdown of only 8 %), the speedup figures are actually very satisfactory and we conjecture that it will be beneficial to integrate the techniques, developed in this part of the thesis, into a compiler.

In conclusion, the ideas presented in this chapter and the previous chapters do not only make sense in theory on accounts of precision and termination, but they also pay off in practice, resulting in a specialiser which improves upon some major existing systems on accounts of speed and size of the specialised programs.

6.4.4 Further discussion

Note that Algorithm 6.2.36, as well as the ECCE system based on it, structure the characteristic atoms in a global tree, and do not just put them in a set as in Algorithm 5.3.3 of Section 5.3. The following example shows that generalisation with non-ancestors may significantly limit specialisation potential.

Example 6.4.1 Let P be the usual *append* program where a type check on the second argument has been added:

(1) $app([], L, L) \leftarrow$
(2) $app([H|X], Y, [H|Z]) \leftarrow ls(Y), app(X, Y, Z)$
(3) $ls([]) \leftarrow$
(4) $ls([H|T]) \leftarrow ls(T)$

Let $A = app(X, [], Z)$ and $B = app(X, [Y], Z)$. When unfolding as depicted in Figure 6.9, we obtain $chtree(\leftarrow A, P, U) = \{\langle 1 \circ 1\rangle, \langle 1 \circ 2, 1 \circ 3\rangle\}$ $= \tau_A$ and $chtree(\leftarrow B, P, U) = \{\langle 1 \circ 1\rangle, \langle 1 \circ 2, 1 \circ 4, 1 \circ 3\rangle\} = \tau_B$. The only leaf atom of (A, τ_A) is $app(X', [], Z')$ and the sole leaf atom of (B, τ_B) is $app(X', [Y], Z')$. In other words, the set $\{(A, \tau_A), (B, \tau_B)\}$ is P-covered and no potential for non-termination exists. However, if we collect characteristic atoms in a set rather than a tree, we do not notice that (B, τ_B) does not descend from (A, τ_A). Consequently, a growing of characteristic

trees (and syntactic structure) will be detected, leading to an unnecessary generalisation of (B, τ_B), and an unacceptable loss of precision.

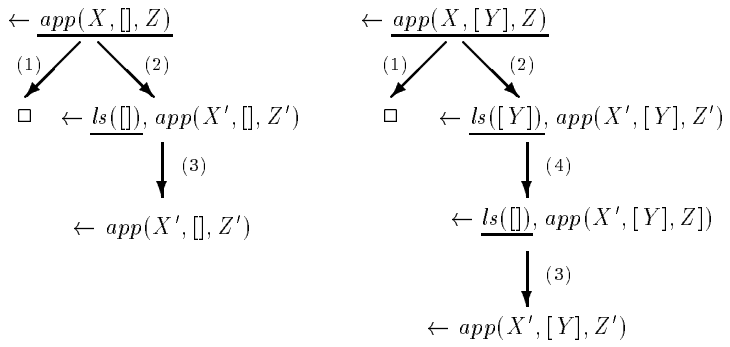


Figure 6.9: SLD-trees for Example 6.4.1

One might also wonder whether, in a setting where the characteristic atoms are structured in a global tree, it would not be sufficient to just test for homeomorphic embedding on the atom part. The intuition behind this would be that a growth of the structure of an atom part would for reasonable programs and unfolding rules lead to a growth of the associated characteristic tree as well — so, using characteristic trees for deciding when to abstract would actually be superfluous. For instance, in Example 6.1.1 we observe that $rev(L, [], R) \trianglelefteq^* rev(T, [H], R)$ and, indeed, for the corresponding characteristic trees $\lceil\{\langle 1 \circ 1\rangle, \langle 1 \circ 2, 1 \circ 3\rangle\}\rceil \trianglelefteq^* \lceil\{\langle 1 \circ 1\rangle, \langle 1 \circ 2, 1 \circ 4, 1 \circ 3\rangle\}\rceil$ holds. Nonetheless, the intuition turns out to be incorrect. The following examples illustrate this point.

Example 6.4.2 Let P be the following normal program searching for paths without loops:

(1) $path(X, Y, L) \leftarrow \neg member(X, L), arc(X, Y)$
(2) $path(X, Y, L) \leftarrow \neg member(X, L), arc(X, Z), path(Z, Y, [X|L])$
(3) $arc(a, b) \leftarrow$
(4) $arc(b, a) \leftarrow$
(5) $member(X, [X|T]) \leftarrow$
(6) $member(X, [Y|T]) \leftarrow member(X, T)$

Let $A = path(a, Y, [])$ and $B = path(a, Y, [b, a])$ and U an unfolding rule based on $\trianglelefteq^*$ (i.e. only allow the selection of an atom if it does not embed a covering ancestor). The SLDNF-tree accordingly built for $\leftarrow A$ is depicted

in Figure 6.10. B occurs in a leaf ($A \trianglelefteq^* B$) and will hence descend from A in the global tree. But the (term representation of) the characteristic tree $\tau_A = \{\langle 1\circ 1, 1\circ member, 1\circ 3\rangle, \langle 1\circ 2, 1\circ member, 1\circ 3, 1\circ 1, 1\circ member, 1\circ 4\rangle, \langle 1\circ 2, 1\circ member, 1\circ 3, 1\circ 2, 1\circ member, 1\circ 4\rangle\}$ is not embedded in (the representation of) $\tau_B = \emptyset$, and no danger for non-termination exists (more structure resulted in this case in failure and thus less unfolding). A method based on testing only $\trianglelefteq^*$ on the atom component would abstract B unnecessarily.

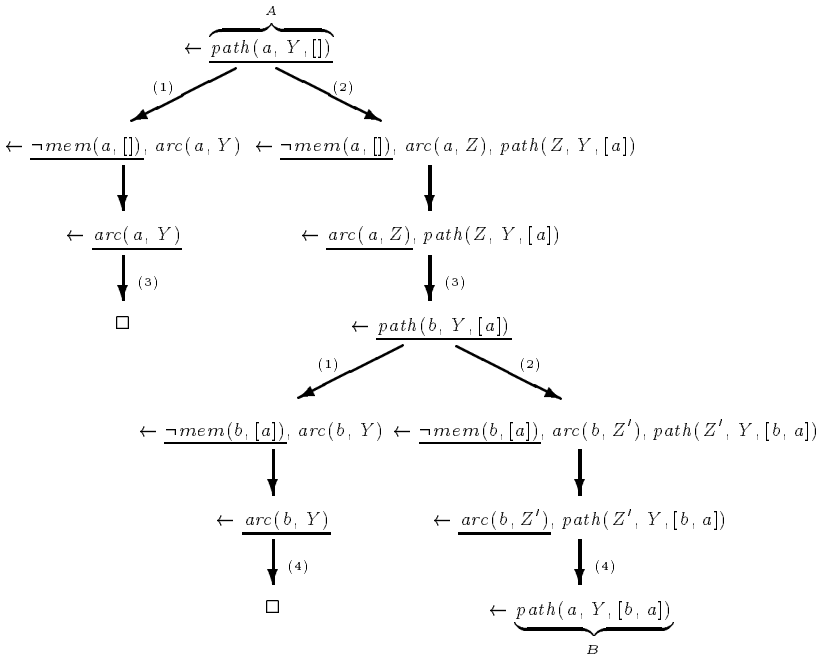


Figure 6.10: SLDNF-tree for Example 6.4.2

Example 6.4.3 Let P be the following *definite* program:

(1) $path([N]) \leftarrow$
(2) $path([X, Y|T]) \leftarrow arc(X, Y), path([Y|T])$
(3) $arc(a, b) \leftarrow$

Let $A = path(L)$. Unfolding $\leftarrow A$ (using an unfolding rule U based on $\trianglelefteq^*$) will result in lifting $B = path([b|T])$ to the global level. The characteristic trees are:
$\tau_A = \{\langle 1 \circ 1\rangle, \langle 1 \circ 2, 1 \circ 3\rangle\}$,
$\tau_B = \{\langle 1 \circ 1\rangle\}$. Again, $A \trianglelefteq^* B$ holds, but not $\lceil \tau_A \rceil \trianglelefteq^* \lceil \tau_B \rceil$.

In recent experiments, it also turned out that characteristic trees might be a vital asset when trying to solve the *parsing problem* [196], which appears when unfolding meta-interpreters with non-trivial object programs. In such a setting a growing of the syntactic structure also does not imply a growing of the characteristic tree.

Example 6.4.4 Take the vanilla meta-interpreter with a simple family database at the object level:

$solve(empty) \leftarrow$
$solve(A\&B) \leftarrow solve(A), solve(B)$
$solve(A) \leftarrow clause(A, B), solve(B)$
$clause(anc(X\,,\,Y), parent(X\,,\,Y)) \leftarrow$
$clause(anc(X\,,\,Z), parent(X\,,\,Y)\&anc(\,Y\,,\,Z)) \leftarrow$
$clause(parent(peter, paul), empty) \leftarrow$
$clause(parent(paul, mary), empty) \leftarrow$

Let $A = solve(anc(X\,,\,Z))$ and $B = solve(parent(X\,,\,Y)\&anc(\,Y\,,\,Z))$. We have $A \trianglelefteq^* B$ and without characteristic trees these two atoms would be generalised (supposing that both these atoms occur at the global level) by their *msg* $solve(G)$. If however, we take characteristic trees into account, we will notice that the object level atom $anc(Z, X)$ within A has more solutions than the object level atom $anc(\,Y\,,\,Z)$ within B (because in the latter one Y will get instantiated through further unfolding). I.e. the characteristic tree of B does not embed the one of A and no unnecessary generalisation will occur.

Returning to the global control method as laid out in Section 6.2, one can observe that a possible drawback might be its complexity. Indeed, first, ensuring termination through a well-quasi-ordering is structurally more costly than the alternative of using a well-founded ordering. The latter only requires comparison with a single "ancestor" object and can be enforced without any search through "ancestor lists" (see [199]). Testing for well-quasi-ordering, however, unavoidably does entail such searching and repeated comparisons with several ancestors. Moreover, in our particular case, checking $\trianglelefteq^*_{ca}$ on characteristic atoms might seem to be in itself a quite costly operation, adding to the complexity of maintaining a well-quasi-ordering. But as the experiments of the previous subsection show, in practice, the complexity of the transformation does not seem to be all that bad, especially since the experiments were still conducted with a prototype which was not yet tuned for transformation speed.

As already mentioned in Section 4.5 of Chapter 4, the partial deduction method of [97] was extended in [66] by adorning characteristic trees with a depth-k abstraction of the corresponding atom component. This was done

in order to increase the amount of polyvariance so that a post-processing abstract interpretation phase — detecting useless clauses — can obtain a more precise result. However, this k parameter is of course yet another ad hoc depth bound. Our method is free of any such bounds and, without the Section 6.3 post-processing, nevertheless obtains a similar effect.

Algorithm 6.2.36 can also be seen as performing an abstract interpretation on an *infinite domain* of *infinite height* (i.e. the ascending chain condition of [57] is not satisfied) and without a priori limitation of the precision (i.e., if possible, we do not perform any abstraction at all and obtain simply the concrete results). Very few abstract interpretations of logic programs use infinite domains of infinite height (some notable exceptions are [36, 134, 118]) and to our knowledge all of them have some a priori limitation of the precision, at least in practice.[11] An adaptation of Algorithm 6.2.36, with its non ad hoc termination and precise generalisations, might provide a good starting point to introduce similar features into abstract interpretation methods, where they might prove equally beneficial.[12]

We conclude this section with a brief discussion on the relation between our global control and what may be termed as such in supercompilation [273, 274, 258, 114]. (A distinction between local and global control is not yet made in supercompilation.) We already pointed out that the inspiration for using $\trianglelefteq$ derives from [190] and [258]. In the latter, a generalisation strategy for positive supercompilation (no negative information propagation while driving) is proposed. It uses $\trianglelefteq$ to compare nodes in a marked partial process tree; a notion originating from [111] and corresponding to global trees in partial deduction. These nodes, however, only contain syntactical information corresponding to ordinary atoms (or rather goals, see Chapter 10). It is our current understanding that both the addition of something similar to characteristic trees and the use of the refined $\trianglelefteq^*$ embedding can lead to improvements of the method proposed in [258]. Finally, it is interesting to return to an observation already made in Section 4 of [201]: Neighbourhoods of order "n", forming the basis for generalisation in full supercompilation ([274]), are essentially the same as classes of atoms or goals with an identical depth n characteristic tree. Adapting our technique to the supercompilation setting will therefore allow to remove the depth bound on neighbourhoods.

[11] In theory the type graphs of [134] can be as precise as one wants, but in practice widening is applied whenever an upper bound is computed.

[12] For instance, one might adapt the wqo $\trianglelefteq^*$ so that they work on the type graphs on [134] instead of characteristic trees. One could then combine the method of [134] with Algorithm 6.2.36 and obtain a more precise abstract interpretation method, which only selectively applies widening where it seems necessary for termination. See also Chapter 13 for a concrete cross-fertilisation between abstract interpretation and partial deduction.

6.5 Conclusion and future work

In this chapter, we have first identified the problems of imposing a depth bound on characteristic trees (or neighbourhoods for that matter) using some practical and realistic examples. We have then developed a sophisticated on-line global control technique for partial deduction of normal logic programs. Importing and adapting m-trees from [201], we have overcome the need for a depth bound on characteristic trees to guarantee termination of partial deduction. Plugging in a depth bound free local control strategy (see e.g. [37, 199]), we thus obtain a fully automatic, concrete partial deduction method that always terminates and produces precise and reasonable polyvariance, without resorting to any ad hoc techniques. To the best of our knowledge, this is the very first such method.

Along the way, we have defined generalisation and embedding on characteristic atoms, refining the homeomorphic embedding relation $\trianglelefteq$ from [81, 82, 190, 258] into $\trianglelefteq^*$, and showing that the latter is more suitable in a logic programming setting. To that end, we have also developed a way to combine two well-quasi orders into a more powerful one, as well as a way to obtain a well-quasi order from a well-founded one. We have also discussed an interesting post-processing intended to sift superfluous polyvariance, possibly produced by the main algorithm. Extensive experiments with an implementation of the method showed its practical value; outperforming existing partial deduction systems for speedup as well as code size while guaranteeing termination.

We believe that the global control proposed in this and the previous chapters is a very good one, but the quality of the specialisation produced by any fully concrete instance of Algorithm 6.2.36 will obviously also heavily depend on the quality of the specific local control used. At the local control level, a number of issues are still open: fully automatic satisfactory unfolding of meta-interpreters and a good treatment of truly non-determinate programs are among the most pressing.

Later in Chapters 10 and 11 we will also present an extension of partial deduction, called *conjunctive* partial deduction, which incorporates more powerful unfold/fold-like transformations [222], allowing for example to eliminate unnecessary variables from programs [231]. The extension boils down to the lifting of entire goals (instead of separate atoms) to the global level, as for instance in supercompilation (where non-atomic goals translate into nested function calls). This opens up a whole range of challenging new control issues. As we will see later in Chapter 12, the technique presented in this part of the thesis will significantly contribute in that context too.

	ECCE-X-10			ECCE-X		
Benchmark	RRT	Size	TT	RRT	Size	TT
advisor	0.32	809	1.01	0.31	809	0.78
contains.kmp	0.80	1974	11.53	0.09	685	4.48
depth.lam	0.06	1802	2.25	0.02	2085	1.91
doubleapp	0.95	216	0.59	0.95	216	0.53
ex_depth	0.32	350	1.73	0.32	350	1.58
grammar.lam	0.14	218	2.27	0.14	218	1.90
groundunify.complex	0.53	4511	29.10	0.53	4800	0.75
groundunify.simple	0.25	368	0.83	0.25	368	22.03
imperative.power	0.56	1578	18.14	0.54	1578	27.42
liftsolve.app	0.53	4544	16.70	0.06	1179	6.57
liftsolve.db1	0.02	2767	22.15	0.02	1326	7.33
liftsolve.db2	0.47	11303	154.94	0.61	4786	34.25
liftsolve.lmkng	1.02	2385	3.44	1.02	2385	2.75
map.reduce	0.08	348	0.84	0.08	348	0.86
map.rev	0.13	427	1.02	0.11	427	0.89
match.kmp	0.70	669	1.24	0.70	669	1.23
memo-solve	1.26	2033	10.56	1.09	2241	4.31
missionaries	1.03	2927	26.67	0.72	2226	9.21
model_elim.app	0.42	6092	5864.28	0.13	532	3.56
regexp.r1	0.29	435	6.77	0.29	435	0.98
regexp.r2	0.43	1373	8.63	0.51	1159	4.87
regexp.r3	0.48	2041	10.82	0.42	1684	14.92
relative.lam	0.02	709	506.89	0.00	261	4.06
rev_acc_type	0.99	2188	22.95	1.00	242	0.83
rev_acc_type.inffail	0.55	1503	21.47	0.60	527	0.80
ssuply.lam	0.14	426	7.84	0.06	262	1.18
transpose.lam	0.18	2312	8.75	0.17	2312	1.98
Average	0.47	2085	250.50	0.40	1263	6.00
Total	12.67	56308	6763.41	10.75	34177	161.96
Speedup	**2.13**			**2.51**		

Table 6.2: Detailed results for ECCE-X-10 and ECCE-X

	ECCE-D			SP		
Benchmark	RRT	Size	TT	RRT	Size	TT
advisor	0.47	412	0.79	0.40	463	0.29
contains.kmp	0.83	1363	2.90	0.75	985	1.13
depth.lam	0.94	1955	1.53	0.53	928	0.99
doubleapp	0.98	277	0.61	1.02	160	0.11
ex_depth	0.76	1614	2.78	0.27	786	1.35
grammar.lam	0.17	309	1.92	0.15	280	0.71
groundunify.complex	0.40	9502	25.04	0.73	4050	2.46
groundunify.simple	0.25	368	0.78	0.61	407	0.20
imperative.power	0.37	2401	61.28	0.97	1706	6.97
liftsolve.app	0.06	1179	5.42	0.23	1577	2.46
liftsolve.db1	0.01	1280	12.95	0.82	4022	3.95
liftsolve.db2	0.17	4694	14.95	0.82	3586	3.71
liftsolve.lmkng	1.07	1730	1.70	1.16	1106	0.37
map.reduce	0.07	507	0.84	0.09	437	0.23
map.rev	0.11	427	0.88	0.13	351	0.20
match.kmp	0.73	639	1.17	1.08	527	0.49
memo-solve	1.17	2318	4.22	1.15	1688	3.65
missionaries	0.81	2294	4.31	0.73	16864	82.59
model_elim.app	0.63	2100	2.83	-	-	∞
regexp.r1	0.50	594	1.29	0.54	466	0.37
regexp.r2	0.55	629	1.29	1.08	1233	0.67
regexp.r3	0.50	828	1.74	1.03	1646	1.20
relative.lam	0.82	1074	1.92	0.69	917	0.35
rev_acc_type	1.00	242	0.70	-	-	∞
rev_acc_type.inffail	0.60	527	0.71	-	-	∞
ssuply.lam	0.06	262	1.18	0.06	231	0.52
transpose.lam	0.17	2312	2.56	0.26	1267	0.52
Average	0.53	1550	5.86	0.68	1903	4.81
Total	14.19	41837	158.29	18.48	45683	115.5
Speedup	**1.90**			**1.46**		

Table 6.3: Detailed results for ECCE-D and SP

	MIXTUS			PADDY		
Benchmark	RRT	Size	TT	RRT	Size	TT
advisor	0.31	809	0.85	0.31	809	0.10
contains.kmp	0.16	533	2.48	0.11	651	0.55
depth.lam	0.04	1881	4.15	0.02	2085	0.32
doubleapp	1.00	295	0.30	0.98	191	0.08
ex_depth	0.40	643	2.40	0.29	1872	0.53
grammar.lam	0.17	841	2.73	0.43	636	0.22
groundunify.complex	0.67	5227	11.68	0.60	4420	1.53
groundunify.simple	0.25	368	0.45	0.25	368	0.13
imperative.power	0.57	2842	5.35	0.58	3161	2.18
liftsolve.app	0.06	1179	4.78	0.06	1454	0.80
liftsolve.db1	0.01	1280	5.36	0.02	1280	1.20
liftsolve.db2	0.31	8149	58.19	0.32	4543	1.60
liftsolve.lmkng	1.16	2169	4.89	0.98	1967	0.32
map.reduce	0.68	897	0.17	0.08	498	0.20
map.rev	0.11	897	0.16	0.26	2026	0.37
match.kmp	1.55	467	4.89	0.69	675	0.28
memo-solve	0.60	1493	12.72	1.48	3716	1.70
missionaries	-	-	∞	-	-	∞
model_elim.app	0.13	624	5.73	0.10	931	0.90
regexp.r1	0.20	457	0.73	0.29	417	0.13
regexp.r2	0.82	1916	2.85	0.67	3605	0.63
regexp.r3	0.60	2393	4.49	1.26	10399	1.35
relative.lam	0.01	517	7.76	0.00	517	0.42
rev_acc_type	1.00	497	0.99	0.99	974	0.33
rev_acc_type.inffail	0.97	276	0.77	0.94	480	0.28
ssuply.lam	0.06	262	0.93	0.08	262	0.08
transpose.lam	0.18	1302	3.89	0.18	1302	0.43
Average	0.48	1470	5.76	0.48	1894	0.64
Total	13.00	38214	149.7	12.96	49239	16.7
Speedup	**2.08**			**2.08**		

Table 6.4: Detailed results for MIXTUS and PADDY

Part III

Off-line Control of Partial Deduction: Achieving Self-Application

Chapter 7

Efficiently Generating Efficient Generating Extensions in Prolog

7.1 Introduction

7.1.1 Off-line vs. on-line control

The control problems of specialisation have been tackled from two different angles: the so-called *off-line* versus *on-line* approaches. The *on-line* approach performs all the control decisions *during* the actual specialisation phase. It is within this methodology that Part II of this thesis was situated. The *off-line* approach on the other hand performs an analysis phase *prior* to the actual specialisation phase, based on some rough descriptions of what kinds of specialisations will have to be performed. The analysis phase provides annotations which then guide the control aspect of the proper specialisation phase, often to the point of making it completely trivial.

Partial evaluation of functional programs [56, 138] has mainly stressed off-line approaches, while supercompilation of functional [273, 274, 258, 114] and partial deduction of logic programs [100, 245, 21, 37, 199, 201, 168, 178] have concentrated on on-line control. (Some exceptions are [284, 207, 173, 167].)

On-line methods, like the one presented in Part II of this thesis, usually obtain better specialisation, because no control decisions have to be taken beforehand, i.e. at a point where the full specialisation information

is not yet available. The main reason for using the off-line approach is to make specialisation more amenable to effective self-application [140, 141], as explained below.

7.1.2 The Futamura projections

A partial evaluation or deduction system is called *self-applicable* if it is able to effectively[1] specialise itself. The practical interests of such a capability are manifold. The most well-known lie with the so called second and third *Futamura projections* (a term coined in [91]; the idea of these projections as well as the idea of self-application in general originated in [96]). The general mechanism of the Futamura projections is depicted in Figure 7.1. The first Futamura projection consists in specialising an *interpreter* for a particular *object program*, thereby producing a specialised version of the interpreter which can be seen as a *compiled* version of the object program. If the partial evaluator is self-applicable then one can specialise the partial evaluator for performing the first Futamura projection, thereby obtaining a *compiler* for the interpreter under consideration. This process is called the second Futamura projection. The third Futamura projection now consists in specialising the partial evaluator to perform the second Futamura projection. By this process we obtain a *compiler generator* (cogen for short).

Guided by these Futamura projections a lot of effort, specially in the functional partial evaluation community, has been put into making systems self-applicable. First successful self-application was reported in [140], and later refined in [141] (see also [138]). The main idea which made this self-application possible was to separate the specialisation process into two phases:

- first a *binding-time analysis* (*BTA* for short) is performed which, based on some rough description of the specialisation task, safely approximates the values that will be known at specialisation time and
- a (simplified) *specialisation phase*, which is guided by the result of the *BTA*.

Such an approach is *off-line* because some, or even most, control decisions are taken beforehand. The interest for self-application lies with the fact that only the second, simplified phase has to be self-applied. On a more technical level, such an approach also avoids (due to the rough description of the specialisation task) the generation of overly general compilers and compiler generators. We refer to [140, 141, 138] for further details.

[1] This implies some efficiency considerations, e.g. the system has to terminate within reasonable time constrains, using an appropriate amount of memory.

In the context of logic programming languages the off-line approach was used in [207] and to some extent also in [115].

On-line methods are much more difficult to self-apply, and no results, comparable to the ones obtained for off-line methods, have been obtained so far. Note however the research in [107, 275] which tries to remedy this problem as well as recent promising results obtained in [213], [276], [260]. Finally, before discussing the issue of self-application for logic programming languages in more detail, we would like to mention other possibilities of self-applicable partial evaluators beyond the Futamura projections, namely the *specialiser projections* [108]. These allow e.g. the generation of new specialisers from interpreters.

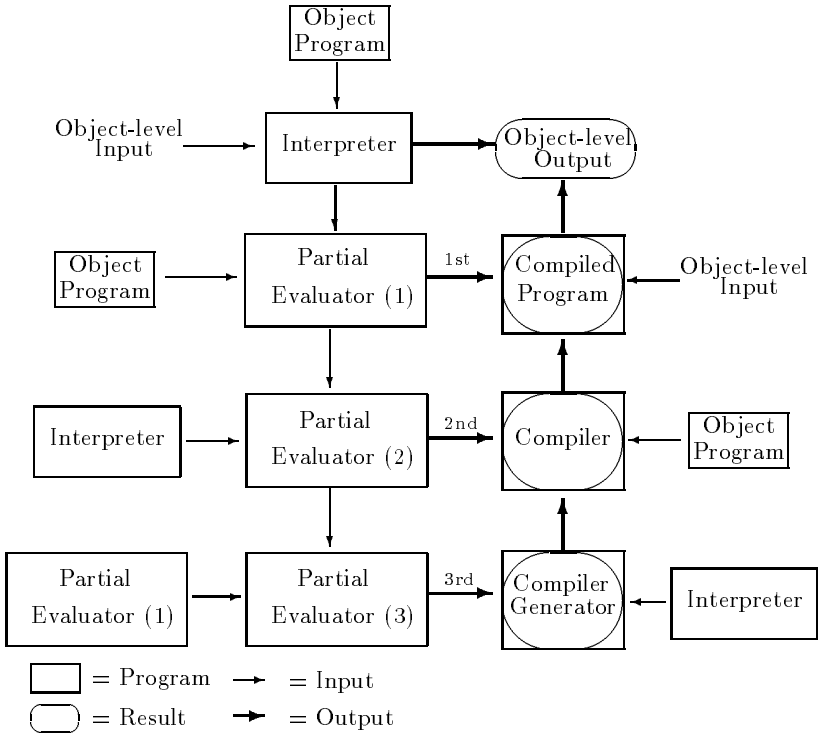


Figure 7.1: Illustrating the 3 Futamura projections

7.1.3 Self-application for logic programming languages and the cogen approach

From now on we will restrict our attention to self-application of specialisers for logic programming languages.

Above we have discussed the interest of self-applicable specialisers for automatically deriving compilers and compiler generators (cogen's). However, even when using an off-line approach, writing an effectively self-applicable specialiser is a non-trivial task — the more features one uses in writing the specialiser the more complex the specialisation process becomes, because the specialiser then has to handle these features as well. Also, when non-declarative features come into play, certain powerful optimisation might no longer be admissible, because the specialiser is now forced to preserve the operational semantics. All this explains why so far no partial evaluator for full Prolog (like MIXTUS [245], or PADDY [227]) has been made effectively self-applicable. On the other hand a partial deducer which specialises only purely declarative logic programs (like SAGE in [115] or the system in [26]) has itself to be written purely declaratively, currently leading to slow systems and impractical compilers and compiler generators.

So far, the only practical compilers and compiler generators have been obtained by striking a delicate balance between the expressivity of the underlying language and the ease with which it can be specialised. Two approaches for logic programming languages along this line are [95] and [207]. However, the specialisation in [95] is incorrect with respect to some of the extra-logical built-ins, leading to incorrect compilers and compiler generators when attempting self-application (a problem mentioned in [26], see also [207, 167]). The partial evaluator LOGIMIX of [207] does not share this problem, but gives only modest speedups (when compared to results for functional programming languages, see the remarks in [207]) when self-applied.

The actual creation of the cogen according to the third Futamura projection is not of much interest to users since, given the specialiser, cogen can be generated once and for all. Therefore, from a user's point of view, whether a cogen is produced by self-application or not is of little importance — it is important that it exists and that it has an improved performance over direct self-application. This is the background behind the approach to program specialisation called the *cogen approach*: instead of trying to write a partial evaluation system which is neither too inefficient nor too difficult to self-apply one simply writes a compiler generator directly. This is not as difficult as one might imagine at first sight: basically cogen turns out to be a rather straightforward extension of a binding-time analysis for logic programs (something first discovered for functional languages in [125]).

In this chapter we will describe the first cogen written in this way for a logic programming language: a small subset of Prolog.

The most noticeable advantages of the cogen approach is that the cogen and the compilers it generates can use all features of the implementation language. Therefore, no restrictions due to self-application have to be imposed (the compiler and the compiler generator don't have to be self-applied)! As we will see, this leads to extremely efficient compilers and compiler generators. So, in this case, being able to use extra-logical features actually makes the compilers more efficient as well as easier to generate.

Some general advantages of the cogen approach are: the cogen manipulates only syntax trees and there is no need to implement a self-interpreter[2]; values in the compilers are represented directly (there is no encoding overhead); and it becomes easier to demonstrate correctness for non-trivial languages (due to the simplicity of the transformation). In addition, the compilers are stand-alone programs that can be distributed without the cogen.

A further advantage of the cogen approach for logic languages is that the compilers and compiler generators can use the non-ground representation (as we will see even a compiled version of it). This is in contrast to self-applicable partial deducers which *must* use the ground representation in order to be declarative. We will return to this issue in Chapter 8. Furthermore, the non-ground representation executes several orders of magnitude faster than the ground representation (even after specialising, see [35]) and, as we will see later in Chapter 13, can be impossible to specialise satisfactorily by partial deduction alone. (Note that even [207] uses a "mixed" representation approach which lies in between the ground and non-ground style; see Chapter 8).

Although the Futamura projections focus on how to generate a compiler from an interpreter, the projections of course also apply when we replace the interpreter by some other program which is not an interpreter. In this case the program produced by the second Futamura projection is not called a compiler, but a *generating extension*. The program produced by the third Futamura projection could rightly be called a *generating extension generator* or gengen, but we will stick to the more conventional cogen.

The main contributions of this chapter are:

- the first description of a handwritten compiler generator (cogen) for a logic programming language which shows that such a program has quite an elegant and natural structure.

[2] I.e. a meta-interpreter for the underlying language. Indeed the cogen just transforms the program to be specialised, yielding a compiler which is then evaluated by the underlying system (and not by a self-interpreter).

- a formal specification of the concept of *binding-time analysis* (BTA) in a (pure) logic programming setting and a description of how to obtain a generic algorithm for partial deduction from such a BTA (by describing how to obtain an unfolding and a generalisation strategy from the result of a BTA).
- benchmark results showing the efficiency of the cogen, the generating extensions and the specialised programs.

The remainder of this chapter is organised as follows: In Section 7.2 we formalise the concept of off-line partial deduction and the associated binding-time analysis. In Section 7.3 we present and explain our cogen approach in a pure logic programming setting (details on how to extend this approach to handle some extra-logical built-ins and the if-then-else can be found in Appendix D). In Section 7.4 we present some examples and results underlining the efficiency of the cogen. We conclude with some discussions in Section 7.5.

7.2 Off-line partial deduction

As mentioned in the introduction, the main reason for using the off-line approach is to achieve effective self-application ([140, 141]). But the off-line approach is in general also more efficient, since many decisions concerning control are made before and not during specialisation. For the cogen approach to be efficient it is vital to use the off-line approach, since then the (local) control can be hard-wired into the generating extension.

7.2.1 Binding-time analysis

Most off-line approaches perform what is called a *binding-time analysis (BTA)* prior to the specialisation phase. This phase classifies arguments to predicate calls as either *static* or *dynamic*. The value of a static argument is definitely known (bound) at specialisation time whereas a dynamic argument is not definitely known (it might only be known at the actual run-time of the program). In the context of partial deduction, a static argument can be seen as a term which is guaranteed not to be more instantiated at run-time (it can never be less instantiated at run-time). For example, if we specialise a program for all instances of $p(a, X)$ then the first argument to p is static while the second one is dynamic — actual run-time instances might be $p(a, b), p(a, Z)$ or $p(a, X)$ but not $p(b, c)$. We will also say that an atom is static if all its arguments are static and likewise that a goal is static if it consist only of static (literals) atoms.

We will now formalise the concept of a binding-time analysis. For that we first define the concept of divisions which classify arguments into static and dynamic ones.

Definition 7.2.1 (division) A *division* Δ *of arity* n is a subset of the set $\{1,\ldots,n\}$ of integers. Given Δ, we denote by $\bar{\Delta}$ the set $\{1,\ldots,n\} \setminus \Delta$.

The elements of Δ represent the dynamic argument positions of an atom, while $\bar{\Delta}$ represents the static positions. As a notational convenience we will use $(\delta_1,\ldots,\delta_n)$ to denote a division Δ of arity n, where $\delta_i = s$ if $i \in \bar{\Delta}$ and $\delta_i = d$ if $i \in \Delta$. For example, (s,d) denotes the division $\{2\}$ of arity 2. From now on we will also use the notation $Pred(P)$ to denote the predicate symbols occurring inside a program P. We now define a division for a program P which divides the arguments of every predicate $p \in Pred(P)$ into the static and the dynamic ones:

Definition 7.2.2 (division for a program) A *division* Δ *for a program* P is a mapping from $Pred(P)$ to divisions having the arity of the corresponding predicates.
We say that an argument t_i of an atom $p(t_1,\ldots,t_n)$ is *dynamic wrt* Δ (respectively *static wrt* Δ) iff $i \in \Delta(p)$ (respectively $i \in \overline{\Delta(p)}$).

Example 7.2.3 (d,s) is a division of arity 2 and (s,d,d) a division of arity 3. Let P be a program containing the predicate symbols $p/2$ and $q/3$. Then $\Delta = \{p/2 \mapsto (d,s), q/3 \mapsto (s,d,d)\}$ is a division for P. Then the arguments b and X of $q(a,b,X)$ are dynamic wrt Δ while a is static wrt Δ.

Divisions can be ordered in a straightforward manner: a division of arity n is more general than another one if it classifies more arguments as dynamic. In the following we extend the order to divisions of programs.

Definition 7.2.4 (partial order of divisions) Let Δ and Δ' be divisions for a program P. We say that Δ' is *more general* than Δ, denoted by $\Delta \sqsubseteq \Delta'$,[3] iff for all predicates $p \in Pred(P)$: $\Delta(p) \subseteq \Delta'(P)$.

As already mentioned, a binding-time analysis will, given a program P (and some description of how P will be specialised), perform a preprocessing analysis and return a *division* for P describing when values will be bound (i.e. known). It will also return an *annotation* which will then guide the local unfolding process of the actual partial deduction. From a

[3] In fact we can even construct a lattice based on the *lub* for divisions of arity n defined by $\Delta \sqcup \Delta' = \Delta \cup \Delta'$, the *glb* defined by $\Delta \sqcap \Delta = \Delta \cap \Delta'$, as well as $\bot_n = \emptyset$ and $\top_n = \{1,\ldots,n\}$.

theoretical viewpoint an annotation restricts the possible unfolding rules that can be used (e.g. the annotation could state that predicate calls to p should never be unfolded whereas calls to q should always be unfolded). We therefore define annotations as follows:

Definition 7.2.5 (annotation) An *annotation* $\mathcal{U}$ is a set of unfolding rules (i.e. it is a subset of the set of all possible unfolding rules).

In order to be really off-line, the unfolding rules in the annotation should not take the unfolding history into account and should not depend "too much" on the actual values of the static (nor dynamic) arguments. We will come back in the following subsection on what annotations can look like from a practical viewpoint. We are now in a position to formally define a binding-time analysis in the context of (pure) logic programs:

Definition 7.2.6 (BTA, BTC) A *binding-time analysis* (BTA) yields, given a program P and an initial division Δ_0 for P, a couple $(\mathcal{U}, \Delta)$ consisting of an annotation $\mathcal{U}$ and a division Δ for P more general than Δ_0. We will call the result of a binding-time analysis a *binding-time classification* (BTC)

The initial division Δ_0 gives information about how the program will be specialised. In fact Δ_0 specifies what the initial atom(s) to be specialised (i.e. the ones in $\mathcal{A}_0$ of Algorithm 3.3.11) will look like (if p' does not occur in $\mathcal{A}_0$ we simply set $\Delta_0(p') = \emptyset$). The role of Δ is to give information about what the atoms in Algorithm 3.3.11 will look like at the global level. In that light, not all BTC as specified above are correct and we now develop a safety criterion for a BTC wrt a given program. Basically a BTC is safe iff every atom that can potentially appear in one of the sets $\mathcal{A}_i$ of Algorithm 3.3.11 (given the restrictions imposed by the annotation of the BTA) corresponds to the patterns described by Δ. Note that if a predicate p is always unfolded by the unfolding rule used in Algorithm 3.3.11 then it is irrelevant what the value of Δ_p is.

For simplicity, we will from now on impose that a *static* argument must be *ground*.[4] In particular this guarantees our earlier requirement that the argument will not be more instantiated at run-time.

Definition 7.2.7 (safe wrt Δ) Let P be a program and let Δ be a division for P and let $p(\bar{t})$ be an atom with $p \in Pred(P)$. Then $p(\bar{t})$ is *safe wrt* Δ iff all its static arguments wrt Δ are ground. A set of atoms S is *safe wrt* Δ iff every atom in S is safe wrt Δ. Also a goal G is *safe wrt* Δ iff all the atoms occurring in G are safe wrt Δ.

[4] This simplifies stating the safety criterion of a BTA because one does not have to reason about "freeness". In a similar vein this also makes the BTA itself easier.

For example, $p(a, X)$ is safe wrt $\Delta = \{p/2 \mapsto (s, d)\}$ while $p(X, a)$ is not.

Definition 7.2.8 (safe BTC, safe BTA) Let $\beta = (\mathcal{U}, \Delta)$ be a BTC for a program P and let $U \in \mathcal{U}$ be an unfolding rule. Then β is a *safe BTC for* P iff for every $U \in \mathcal{U}$ and for every goal G, which is safe wrt Δ, U returns an incomplete SLDNF-tree whose leaf goals are safe wrt Δ. A BTA is *safe* if for any program P it produces a safe BTC for P.

Example 7.2.9 Let P be the well known append program

(1) $app([], L, L) \leftarrow$
(2) $app([H|X], Y, [H|Z]) \leftarrow app(X, Y, Z)$

Let $\Delta = \{app \mapsto (s, d, d)\}$ and let $\mathcal{U}$ be the set of all unfolding rules. Then $(\mathcal{U}, \Delta)$ is a safe BTC for P. For example, the goal $\leftarrow app([a, b], Y, Z)$ is safe wrt Δ and an unfolding rule can either stop at $\leftarrow app([b], Y, Z)$, $\leftarrow app([], Y', Z')$ or at the empty goal $\Box$. All of these goals are safe wrt Δ. In general, unfolding a goal $\leftarrow app(t_1, t_2, t_3)$ where t_1 is ground, leads only to goals whose first arguments are ground.

So, the above Definition 7.2.8 requires atoms to be safe in the leaves of incomplete SLDNF-trees, i.e. at the point where the atoms get abstracted and then lifted to the *global* level.[5] So, in order for the above condition to ensure safety at all stages of Algorithm 3.3.11, the particular abstraction operator should not abstract atoms which are safe wrt Δ into atoms which are no longer safe wrt Δ. This motivates the following definition:

Definition 7.2.10 An abstraction operator *abstract* is *safe wrt a division* Δ for some program P iff for every finite set of atoms S, which is safe wrt Δ, $abstract(S)$ is also safe wrt Δ.

7.2.2 A particular off-line partial deduction method

In this subsection we define a specific off-line partial deduction method which will serve as the basis for the cogen developed in the remainder of this chapter. For simplicity, we will from now on restrict ourselves to definite programs. Negation will in practice be treated in the cogen either as a built-in or via the if-then-else construct (see Appendix D).

Let us first define a particular unfolding rule.

[5] Also, when leaving the pure logic programming context and allowing extra-logical built-ins (like = ../2) a *local* safety condition will also be required.

Definition 7.2.11 ($U_{\mathcal{L}}$) Let $\mathcal{L} \subseteq Pred(P)$ be a set of predicates, called the *reducible* predicates. Also an atom will be called *reducible* iff its predicate symbol is in $\mathcal{L}$ and *non-reducible* otherwise. We then define the unfolding rule $U_{\mathcal{L}}$ to be the unfolding rule which applies an unfolding step to the goal in the root and then unfolds the leftmost reducible atom in each goal.

We will use such unfolding rules in Algorithm 3.3.11 and we will restrict ourselves (to avoid distracting from the essential points) to safe BTA's which return results of the form $\beta = (\{U_{\mathcal{L}}\}, \Delta)$. In the actual implementation of the cogen (Appendix E) we use a slightly more liberal approach, in the sense that specific program points (calls to predicates) are annotated as either reducible or non-reducible. Also note that nothing prevents a BTA from having a pre-processing phase which splits the predicates according to their different uses.

Example 7.2.12 Let P be the following program

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

Let $\Delta = \{p \mapsto (s), q \mapsto (s,d)\}$. Then $\beta = (\{U_{\{q\}}\}, \Delta)$ is a safe BTC for P. For example, the goal $\leftarrow p(a)$ is safe wrt Δ and unfolding it according to $U_{\{q\}}$ will lead (via the intermediate goals $\leftarrow q(a,Y), q(Y,Z)$ and $\leftarrow q(b,Z)$) to the empty goal $\square$ which is safe wrt Δ. Note that every selected atom is safe wrt Δ.[6] Also note that $\beta' = (\{U_{\{\}}\}, \Delta)$ is a *not* a safe BTC for P. For instance, for the goal $\leftarrow p(a)$ the unfolding rule $U_{\{\}}$ just performs one unfolding step and thus stops at the goal $\leftarrow q(a,Y), q(Y,Z)$ which contains the unsafe atom $q(Y,Z)$.

The only thing that is missing in order to arrive at a concrete instance of Algorithm 3.3.11 is a (safe) abstraction operator, which we define in the following.

Definition 7.2.13 (gen_{Δ}, $abstract_{\Delta}$) Let P be a program and Δ be a division for P. Let $A = p(\bar{t})$ with $p \in Pred(P)$. We then denote by $gen_{\Delta}(A)$ an atom obtained from A by replacing all dynamic arguments of A (wrt Δ_p) by distinct, fresh variables not occurring in A.
We also define the abstraction operator $abstract_{\Delta}$ to be the natural extension of the function gen_{Δ}: $abstract_{\Delta}(\mathcal{A}) = \{gen_{\Delta}(A) \mid A \in \mathcal{A}\}$.

[6] As already mentioned, this is not required in Definition 7.2.8 but (among others) such a condition will have to be incorporated for the selection of extra-logical built-ins.

For example, if the division Δ is $\{p/2 \mapsto (s,d), q/3 \mapsto (d,s,s)\}$ then $gen_\Delta(p(a,b)) = p(a,X)$ and $gen_\Delta(q(a,b,c)) = q(X,b,c)$.
Then $abstract_\Delta(\{p(a,b), q(a,b,c)\}) = \{p(a,X), q(X,b,c)\}$. Note that, trivially, $abstract_\Delta$ is safe wrt Δ.

Observe that in Algorithm 3.3.11 the uncovered leaf atoms are all added and abstracted simultaneously, i.e. the algorithm progresses in a breadth-first manner. In general this will yield a different result from a depth-first progression (i.e. adding one leaf atom at a time). However, because $abstract_\Delta$ is a homomorphism[7] we can use a depth-first progression in Algorithm 3.3.11 and still get the same specialisation. This is something which we will actually do in the practical implementation.

In the remainder of this chapter we will use the following off-line partial deduction method:

Algorithm 7.2.14 (off-line partial deduction)

1. Perform a BTA (possibly by hand) returning results of the form $(\{U_\mathcal{L}\}, \Delta)$
2. Perform Algorithm 3.3.11 with $U_\mathcal{L}$ as unfolding rule and $abstract_\Delta$ as abstraction operator. The initial set of atoms $\mathcal{A}_0$ should only contain atoms which are safe wrt Δ.

Proposition 7.2.15 Let $(\{U_\mathcal{L}\}, \Delta)$ be a safe BTC for a program P. Let $\mathcal{A}_0$ be a set of atoms safe wrt Δ. If Algorithm 7.2.14 terminates then the final set $\mathcal{A}_i$ only contains atoms safe wrt Δ.

Proof Trivial, by induction on i. □

We will illustrate this particular partial deduction method on an example.

Example 7.2.16 We use a small generic parser for a set of languages which are defined by grammars of the form $S ::= aS|X$ (where X is a placeholder for a terminal symbol). The example is adapted from [152] and the parser P is depicted in Figure 7.2.

Given the initial division $\Delta_0 = \{nont/3 \mapsto (s,d,d),\ t/3 \mapsto \emptyset\}$ a BTA might return the following BTC $\beta = (\{U_{\{t/3\}}\}, \Delta)$ where $\Delta = \{nont/3 \mapsto (s,d,d),\ t/3 \mapsto (s,d,d)\}$. It can be seen that β is a safe BTC for P.

Let us now perform the proper partial deduction starting from $\mathcal{A}_0 = \{nont(c,R,T)\}$. Note that the atom $nont(c,R,T)$ is safe wrt Δ_0 (and hence also wrt Δ). Unfolding the atom in $\mathcal{A}_0$ yields the SLD-tree in Figure 7.3. We see that the set of leaf atoms is $\{nont(c,V,T)\}$. As

[7] I.e. $abstract_\Delta(\emptyset) = \emptyset$ and $abstract_\Delta(S \cup S') = abstract_\Delta(S) \cup abstract_\Delta(S')$.

$nont(c, V, T)$ is a variant of $nont(c, R, T)$ we obtain $\mathcal{A}_1 = \mathcal{A}_0$. The specialised program after renaming and filtering looks like:

$nont_c([a|V], R) \leftarrow nont_c(V, R)$
$nont_c([c|R], R) \leftarrow$

(1) $nont(X, T, R) \leftarrow t(a, T, V), nont(X, V, R)$
(2) $nont(X, T, R) \leftarrow t(X, T, R)$
(3) $t(X, [X|ES], ES) \leftarrow$

Figure 7.2: A parser

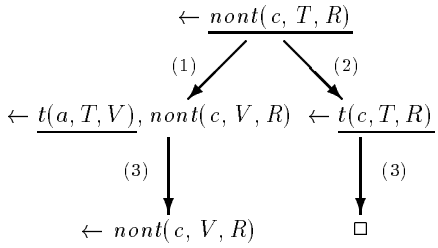


Figure 7.3: Unfolding the parser of Figure 7.2

7.3 The cogen approach for logic programming languages

For presentation purposes, and without loss of generality, we from now on suppose that in Algorithm 7.2.14 the initial set $\mathcal{A}_0$ consists of just a single atom A_0 (a convention followed by a lot of practical partial deduction systems). Thus, a *generating extension*, is a program that, given a safe BTC $(\{U_{\mathcal{L}}\}, \Delta)$ for P, performs part 2 of the off-line partial deduction Algorithm 7.2.14 for a given atom A_0 which is safe wrt Δ. For instance, in the case of the parser and the BTC from Example 7.2.16, a generating extension is a program that, when given the atom $A_0 = nont(c, R, T)$, produces the residual program shown in the example.

Given a program P and a safe BTC β for P, a *compiler generator*, cogen, is then simply a program that produces a generating extension of P wrt β.[8]

We will first examine in more detail the structure of an efficient generating extension. Based on this investigation the design of the cogen will be rather straightforward (given the BTC).

To perform the required specialisation, a generating extension first has to unfold the initial atom A_0 once (to ensure a non-trivial tree). In each branch, it then has to unfold the left-most reducible atom until no more reducible atoms can be found. The atoms in the leaves of the thus obtained SLDNF-tree have to be collected and generalised. This process is repeated for all the new generalised atoms which have not yet been unfolded, until no more new atoms are found.

The unfolding component of a generating extension might be implemented via a meta-program, working on a representation of the program P to be specialised. However, this would not be the most efficient way to tackle that problem. Indeed, the generating extension is tailored towards a particular program P (and a particular BTC) and we can use this information to get rid of some unnecessary overhead. The crucial idea is to write a specific predicate p_u for each predicate p of arity n, tailored towards unfolding atoms of the form $p(t_1, \ldots, t_n)$. This predicate p_u has $n+1$ arguments: the first n arguments contain the arguments $t_1, \ldots, t_n$ of the atom that has to be unfolded while the last argument collects the result of the unfolding process. More precisely, $p\_u(t_1, ..., t_n, B)$ will succeed for each branch of the incomplete SLDNF-tree obtained by applying the unfolding rule $U_{\mathcal{L}}$ to $p(t_1, ..., t_n)$, whereby it will return in B the atoms in the leaf of the branch[9] and also instantiate $t_1, ..., t_n$ via the composition of *mgu*'s of the branch. For complete SLDNF-trees (i.e. for atoms which get fully unfolded) the above can be obtained very *efficiently* by simply executing the original predicate definition of p for the goal $\leftarrow p(t_1, ..., t_n)$ (no atoms in the leaves have to be returned because there are none). To handle incomplete SLDNF-trees we just have to adapt the definition of p so that unfolding can be stopped (for non-reducible predicates) and so that the atoms in the leaves are collected.

This can be obtained by transforming every clause for p/n into a clause for $p\_u/(n+1)$, as done in the following definition.

Definition 7.3.1 Let P be a program and $C = p(\bar{t}) \leftarrow A_1, ..., A_k$ a clause of

[8] The BTA itself can also be seen as part of the cogen. In that case a cogen is a program that, given an initial division Δ_0 produces a generating extension of P wrt some $(\mathcal{U}, \Delta)$, where Δ is more general than Δ_0.

[9] For reasons of clarity and simplicity in unflattened form.

P defining a predicate symbol p/n. Let $\mathcal{L} \subseteq Pred(P)$ be a set of reducible predicate symbols. We then define the clause $C_u^{\mathcal{L}}$ to be:

$$p\_u(\bar{t}, [\mathcal{R}_1, ..., \mathcal{R}_k]) \leftarrow \mathcal{S}_1, ..., \mathcal{S}_k$$

where

1. $\mathcal{S}_i = q\_u(\bar{s}, \mathcal{R}_i)$ and $\mathcal{R}_i$ is a fresh unused variable, **if** $A_i = q(\bar{s})$ is reducible
2. $\mathcal{S}_i = true$ and $\mathcal{R}_i = A_i$, **if** A_i is non-reducible

We will denote by $P_u^{\mathcal{L}}$ the program obtained by applying the above transformation to every clause in P and removing all $true$ atoms from the bodies.

In the above definition inserting a literal of the form $q\_u(\bar{s}, \mathcal{R}_i)$ into the body corresponds to further unfolding whereas inserting $true$ corresponds to stopping the unfolding process. In the case of Example 7.2.16 with $\mathcal{L} = \{t/3\}$, applying the above to the program P of Figure 7.2 gives rise to the following program $P_u^{\mathcal{L}}$ (which could be further improved by a simple partial evaluation):

```
nont_u(X,T,R,[V1,nont(X,V,R)]) :- t_u(a,T,V,V1).
nont_u(X,T,R,[V1]) :- t_u(X,T,R,V1).
t_u(X,[X|R],R,[]).
```

This piece of code might actually be called a *compiled* non-ground representation, and contributes much to the final efficiency of the generating extensions. Evaluating it for the call `nont_u(c,T,R,Leaves)` yields two computed answers which correspond to the two branches in Figure 7.2:

```
> ?-nont_u(c,T,R,Leaves).
  T = [a | _52]
  Leaves = [[],nont(c,_52,R)]
Yes ;
  T = [c | R]
  Leaves = [[]]
Yes
```

The above code is of course still incomplete as it only handles the unfolding process and we have to extend it to treat the global level as well. Firstly, calling p_u only returns the atoms of one leaf of the SLDNF-tree, so we need to add some code that collects the information from all the leaves. This can be done very efficiently using Prolog's `findall` predicate. Note

that here we leave the purely declarative context. This poses no problem, because our generating extensions do not have to be self-applied. To stay declarative one would have to use something like meta-programming in the ground representation (see Chapter 8), which would severely undermine our efficiency (and simplicity) concerns. This is why the cogen approach is (probably) much more difficult to realise in a language like Gödel (maybe intensional sets can be used to achieve the above) and having some non-declarative features at our disposal is a definite advantage.

Using the call `findall(B,nont_u(c,R,T,B),Bs)`, the `Bs` will be instantiated to the list `[[[],nont(c,_48,_49)],[]]]`, which essentially corresponds to the leaves of the SLDNF-tree in Figure 7.3, since by flattening out we obtain: `[nont(c,_48,_49)]`. Furthermore, if we call

```
findall(clause(nont(c,T,R),Bdy),nont_u(c,T,R,Bdy),Cs)
```

we will even get in `Cs` a representation of the two resultants of Example 7.2.16.

Once all the resultants have been generated, the body atoms have to be generalised (using gen_Δ) and unfolded if they have not been encountered yet. The easiest way to achieve this is to add a function p_m, for each non-reducible predicate p, such that p_m implements the global control aspect of the specialisation. That is, for every atom $p(\bar{t})$, if one calls $p\_m(\bar{t}, R)$ then R will be instantiated to the residual call of $p(\bar{t})$ (i.e. the call after filtering and renaming, for instance the residual call of $p(a, b, X)$ might be $p_1(X)$). At the same time p_m also generalises this call, checks if it has been encountered before and if not, unfolds the atom, generates code and prints the resultants (residual code) of the atom. We have the following definition of p_m, where we denote the Prolog conditional by `If->Th;El`. An illustration of this definition, for the predicate *nont* of Example 7.2.16, can be found in Figure 7.4.

Definition 7.3.2 Let P be a program and p/n be a predicate defined in P. Let $\mathcal{L} \subseteq Pred(P)$ be a set of reducible predicate symbols. For $p \in Pred(P)$ we define the clause C^p_m, defining the predicate p_m, to be:

```
p_m(t̄,R) :-
   ( find_pattern(p(t̄),R) -> true
   ; ( insert_pattern(p(s̄),H),
       findall(C,(p_u(s̄,B), treat_clause(H,B,C)),Cs),
       pp(Cs),
       find_pattern(p(t̄),R) ) ).
```

where $\bar{t}$ is a sequence of n distinct, fresh variables and $p(\bar{s}) = gen_\Delta(p(\bar{t}))$. Finally we define $P^{\mathcal{L}}_m = \{C^p_m \mid p \in Pred(P) \setminus \mathcal{L}\}$.[10]

[10] This corresponds to saying that only reducible atoms can occur at the global level,

In the above, the predicate `find_pattern` checks whether its first argument is a call that has been encountered before, and if so its second argument will be instantiated to the corresponding residual call (with renaming and filtering performed). This is achieved by keeping a list of the predicates that have been encountered before along with their renamed and filtered calls. So if the call `find_pattern(p(`$\bar{t}$`),R)` succeeds, then `R` has been instantiated to the residual call of $p(\bar{t})$, if not then the other branch of the conditional is executed.

The predicate `insert_pattern` will add a new atom (its first argument) to the list of atoms encountered before and return (in its second argument `H`) the generalised, renamed and filtered version of the atom. The atom `H` will provide (maybe further instantiated) the head of the resultants to be constructed. This call to `insert_pattern` is put first to ensure that an atom is not specialised over and over again at the global level.

The call to `findall(C,(p_u(`$\bar{s}$`,B),treat_clause(H,B,C)),Cs)` unfolds the generalised atom $p(\bar{s})$ and returns a list of residual clauses for $p(\bar{s})$ (in `Cs`). The call `p_u(`$\bar{s}$`,B)` inside `findall` returns a leaf goal of the SLDNF-tree for $p(\bar{s})$, which is going to be the body of a residual clause with head `H`. For each of the atoms in the body of this clause two things have to be done. First, for each atom a specialised residual version has to be generated if necessary. Second, each atom has to be replaced by a call to a corresponding residual version. Both of these tasks can be performed by calling the corresponding "m" function of the atoms, so if a body contains an atom $p(\bar{t})$ then `p_m(`$\bar{t}$`,R)` is called and the atom is replaced by the value of `R`. This task is performed by the predicate `treat_clause`, whose third argument will contain the new clauses.

The predicate `pp` pretty-prints the clauses of the residual program. The last call to `find_pattern` will instantiate `R` to the residual call of the atom $p(\bar{t})$.

We can now define the generating extension of a program:

Definition 7.3.3 Let P be a program, $\mathcal{L} \in Pred(P)$ a set of predicates and $(\{U_{\mathcal{L}}\}, \Delta)$ a safe BTC for P, then the *generating extension* of P with respect to $(\{U_{\mathcal{L}}\}, \Delta)$ is the program $P_g = P_u^{\mathcal{L}} \cup P_m^{\mathcal{L}}$.

The complete generating extension for Example 7.2.16 is shown in Figure 7.4.

The generating extension is called as follows: if one wants to specialise an atom $p(\bar{t})$, where p is one of the non-reducible predicates of the subject program P, then one simply calls `p_m(`$\bar{t}$`,_)`.

---

and hence only reducible atoms can be put into the initial set of atoms $\mathcal{A}_0$ of Algorithm 3.3.11. To be able to put non-reducible atoms into $\mathcal{A}_0$, one just hast to replace "$p \in Pred(P) \setminus \mathcal{L}$" by "$p \in Pred(P)$".

```
nont_m(B,C,D,E) :-
    ( find_pattern(nont(B,C,D),E) -> true
    ; (insert_pattern(nont(B,F,G),H),
         findall(I,(nont_u(B,F,G,J),treat_clause(H,J,I)),K),
         pp(K),
         find_pattern(nont(B,C,D),E)
        )).
nont_u(B,C,D,[E,nont(B,G,D)]) :- t_u(a,C,G,E).
nont_u(H,I,J,[K]) :- t_u(H,I,J,K).
t_u(L,[L|M],M,[]).
```

Figure 7.4: The generating extension for the parser

The job of the cogen is now quite simple: given a program P and a safe BTC β for P, generate a generating extension for P consisting of the two parts described above. The code of the essential parts of our cogen is shown in Appendix E. The predicate `predicate` generates the definition of the global control m-predicates for each non-reducible predicate of the program, whereas the predicates `clause, bodys` and `body` take care of translating clauses of the original predicate into clauses of the local control u-predicates. Note how the second argument of `bodys` and `body` corresponds to code of the generating extension whereas the third argument corresponds to code produced at the next level, i.e. at the level of the specialised program. Further details on extending the cogen to handle built-ins and the if-then-else can be found in Appendix D.

7.4 Examples and results

In this section we present some experiments with our cogen implementation, henceforth referred to as LOGEN, as well as with some other specialisation systems. We will use three example programs to that effect.

The first program is the parser from Example 7.2.16. We will use the same annotation as in the previous sections: $nont \mapsto (s, d, d)$.

The second example program is the "mixed" meta-interpreter for the ground representation of [173] in which the goals are "lifted" to the non-ground representation for resolution. See Chapter 8 for further details. We will specialise this program given the annotation $solve \mapsto (s, d)$, i.e. we suppose that the object program is given and the query to the object program is dynamic.

Finally we also experimented with a regular expression parser, which tests whether a given string can be generated by a given regular expression. The example is taken from [207]. In the experiment we used $dgenerate \mapsto (s, d)$ for the initial division, i.e. the regular expression is fully known whereas the string is dynamic.

7.4.1 Experiments with LOGEN

The Tables 7.1, 7.2 and 7.3 summarise our benchmarks of the LOGEN system. The timings were obtained by using the $cputime/1$ predicate of Prolog by BIM on a Sparc Classic under Solaris (timings, at least for Table 7.1, were almost identical for a Sun 4).

Program	Time	Annotation
parser	0.02 s	$nont \mapsto (s, d, d)$
solve	0.06 s	$solve \mapsto (s, d)$
regexp	0.02 s	$dgenerate \mapsto (s, d)$

Table 7.1: Running LOGEN

Program	Time	Query
parser	0.01 s	$nont(c, T, R)$
solve	0.01 s	$solve(\text{“}\{q(X) \leftarrow p(X), p(a) \leftarrow\}\text{”}, Q)$
regexp	0.03 s	$dgenerate(\text{”}(a+b)*.a.a.b\text{”}, S)$

Table 7.2: Running the generating extension

Program	Speedup Factor	Runtime Query
parser	2.35	$nont(c, [\overbrace{a, \ldots, a}^{18}, c, b], [b])$
solve	7.23	$solve(\text{“}\{q(X) \leftarrow p(X), p(a) \leftarrow\}\text{”}, \text{“}\leftarrow q(a)\text{”})$
regexp	101.1	$dgenerate(\text{”}(a+b)*.a.a.b\text{”}, \text{”}abaaaabbaab\text{”})$

Table 7.3: Running the specialised program

The results depicted in Tables 7.1, 7.2 and 7.3 are very satisfactory. The generating extensions are generated very efficiently and also run very efficiently. Furthermore the specialised programs are also very efficient and the speedups are very satisfactory. The specialisation for the *parser* example corresponds to the one obtained in 7.2.16. By specialising *solve* our system LOGEN was able to remove almost all[11] the overhead of the

[11] To get rid also of the encoding overhead using $struct/2$ one would also have to apply

ground representation, something which has been achieved for the first time in [97]. In fact, the specialised program looks like this:

```
solve__0([]).
solve__0([struct(q,[B])|C]) :-
  solve__0([struct(p,[B])]), solve__0(C).
solve__0([struct(p,[struct(a,[])])|D]) :-
  solve__0([]), solve__0(D).
```

The specialised program obtained for the *regexp* example actually corresponds to a deterministic automaton, a feat that has also been achieved by the system LOGIMIX in [207]. For further details about the examples see Appendices F.1, F.2 and F.3.

7.4.2 Experiments with other systems

We also performed the experiments using other specialisation systems, some of which we already encountered in Chapter 6. All systems were able to satisfactorily handle the *parser* example and came up with (almost) the same specialised program as LOGEN. More specific information is presented in the following.

MIXTUS. MIXTUS ([245]) is a partial evaluator for full Prolog which is not (effectively) self-applicable. We experimented with version 0.3.3 of MIXTUS running under SICStus Prolog 2.1. MIXTUS came up with exactly the same specialisation as our LOGEN for the *parser* and *solve* examples. MIXTUS was also able to specialise the *regexp* program, but not to the extent of generating a deterministic automaton.

SP. We experimented with the SP system (see [97]), a specialiser for a subset of Prolog (comparable to our subset, with the exception that SP does not handle the if-then-else). For the *solve* example SP was able to obtain the same specialisation as LOGEN, but only after re-specialising the specialised program a second time (also SP does not perform filtering which might account for some loss in efficiency). Due to the heavy usage of the if-then-else the *regexp* example could not be handled (directly) by SP.

LOGIMIX. LOGIMIX ([207]) is a self-applicable partial evaluator for a subset of Prolog, containing if-then-else, side-effects and some built-ins. This

a technique called *constructor specialisation* (for functional programming languages, see e.g. [206] or [88]).

system incorporates ideas developed for functional programming and falls within the off-line setting and requires a binding time annotation. It is not (yet) fully automatic in the sense that the program has to be hand-annotated. For the *parser* and *regexp* examples, LOGIMIX came up with almost the same programs than LOGEN (a little bit less efficient because bindings were not back-propagated on the head of resultants). We were not able to annotate *solve* properly, in every case LOGIMIX aborted due to an "instantiation error" on the *=../2* built-in. This could either be due to a misunderstanding (on our part) of the annotations of LOGIMIX or simply due to a bug in LOGIMIX. It might also be that the example cannot be handled by LOGIMIX because the restrictions on the annotations are more severe than ours (in LOGEN the unfoldable predicates do not require a division and LOGEN allows non-deterministic unfolding — the latter seems to be crucial for the *solve* example).

LEUPEL. LEUPEL ([167, 173], see also Chapter 9) is a (not yet effectively self-applicable) partial evaluator for a subset of Prolog, very similar to the one treated by LOGIMIX. The system is guided by an annotation phase which is unfortunately also not automatic. The annotations are "semi-on-line", in the sense that conditions (tested in an on-line manner) can be given on when to make a call reducible, non-reducible or even unfoldable (given no loop is detected at on-line specialisation time). We will re-examine that system in Chapter 9. For the *parser* and *regexp* examples the system performed the same specialisation as LOGEN. For the *solve* example LEUPEL even came up with a better specialisation than LOGEN, in the sense that unfolding has also been performed at the object level:

```
solve__1([]).
solve__1([struct(q,[struct(a,[])])|A]) :- solve__1(A).
solve__1([struct(p,[struct(a,[])])|A]) :- solve__1(A).
```

Such optimisations depend on the particular object program and are therefore outside the reach of purely off-line methods.

ECCE. This is the system we already described and used in Chapter 6. It is a fully automatic on-line system for a declarative subset of Prolog (similar to the language handled by SP). We used the settings corresponding to ECCE-X, as described in Section 6.4. For the *parser* example ECCE produced the same specialisation as LOGEN. For the *solve* example the ECCE came up with a better specialisation than LOGEN, identical to the one obtained by LEUPEL (but this time fully automatically). Due to the heavy usage of

the if-then-else the *regexp* example could, similarly to SP, not be handled (directly) by ECCE.

PADDY. We also did some experiments with the PADDY system (see [227]) written for full Eclipse (a variant of Prolog). PADDY basically performed the same specialisation of *solve* as ECCE or LEUPEL, but left some useless tests and clauses inside. PADDY was also able to specialise the *regexp* program, but again not to the extent of generating a deterministic automaton.

SAGE. Finally, we tried out the self-applicable partial deducer SAGE (see [115, 116]) for the logic programming language Gödel. SAGE came up with (almost) the same specialised program for the *parser* example as LOGEN. SAGE performed little specialisation on the *solve* example, returning almost the unspecialised program back. Due to the heavy usage of the if-then-else the *regexp* example could not be handled by SAGE.

7.4.3 Comparing transformation times

We were able to measure transformation times (to be compared with the results of Table 7.2) for all systems except for SAGE. Also, all systems except PADDY were run on the same machine as LOGEN. In fact, PADDY runs under Eclipse and had for technical reasons to be executed on a Sun 4. MIXTUS and LOGIMIX were executed under SICStus Prolog 2.1, while SP was executed using SICStus Prolog 3 (for SP we only benchmarked one iteration, although two were required for an optimal result for *solve*). LEUPEL and ECCE were benchmarked using Prolog by BIM.[12]

As LOGIMIX is self-applicable, we were also able to produce generating extensions to perform the specialisation more efficiently. Producing these generating extensions by using $\text{LOGIMIX}_{\text{cogen}}$ (obtained via the third Futamura projection) took 1.103*s* for the *parser* example and 0.983*s* for the *regexp* example.[13] The corresponding generating extensions then performed the specialisation in 0.015*s* instead of 0.018*s* for the *parser* example, and in 0.078*s* instead of 0.093*s* for the *regexp* example (so only modestly faster than running LOGIMIX directly). We also tested the size of the LOGEN and $\text{LOGIMIX}_{\text{cogen}}$ using `statistics(program,S)` of SICStus Prolog. The result for $\text{LOGIMIX}_{\text{cogen}}$ (without front- and back-end) was 161616 bytes and the

[12] LEUPEL uses the ground representation and is therefore rather slow. Also, the timings of ECCE include the printing of tracing information as well as some run-time type checks. As mentioned in Chapter 6, ECCE is still a prototype and its transformation speed can still be (dramatically) improved.

[13] Producing these generating extensions via the second Futamura projection took 1.469*s* for the *parser* example and 1.277*s* for the *regexp* example.

size of LOGEN (without the interactive shell and various tools) was 20464 bytes, so about 1/8th of the size of $\text{LOGIMIX}_{\text{cogen}}$.

A summary of all the transformation times can be found in Table 7.4. The columns marked by *spec* contain the times needed to produce the specialised program (using the generating extensions in the case of LOGEN and $\text{LOGIMIX}_{\text{cogen}}$), whereas the columns marked by *genex* contain the times needed to produce the generating extensions. As can be seen, LOGEN is by far the fastest system overall, as well for specialisation as for compiler generation. Table 7.5 contains a comparison of the run-times of some of the residual programs for a variety of run-time queries.[14] It highlights the good specialisation obtained by LOGEN but also shows that for the *solve* example, as explained earlier, the on-line systems were able to produce better specialisation.

Specialiser	*parser genex*	*parser spec*	*solve genex*	*solve spec*	*regexp genex*	*regexp spec*
Off-line						
LOGEN	0.02 s	0.01 s	0.06 s	0.01 s	0.02 s	0.03 s
LOGIMIX	1.47 s	0.02 s	-	-	1.28 s	0.09 s
$\text{LOGIMIX}_{\text{cogen}}$	1.10 s	0.02 s	-	-	0.98 s	0.08 s
Semi On-line						
LEUPEL	-	0.11 s	-	0.64 s	-	4.00 s
On-line						
ECCE-X	-	0.09 s	-	3.48 s	-	-
MIXTUS	-	0.14 s	-	1.36 s	-	13.63 s
PADDY	-	0.05 s	-	0.80 s	-	3.17 s
SP	-	0.07 s	-	0.47 s	-	-

Table 7.4: Specialisation times

Finally the figures in Tables 7.1 and 7.2 really shine when compared to the compiler generator and the generating extensions produced by the self-applicable SAGE system. Unfortunately self-applying SAGE is currently not possible for normal users, so we had to take the timings from [115]: generating the compiler generator takes about 100 hours (including garbage collection), generating a generating extension took for the examples (which are probably more complex than the ones treated in this section) in [115]

[14] The *parser* example is not included as all systems performed practically identical specialisation. For SP, the residual program obtained after two specialisation phases was used. However, the residual programs produced by SP cannot be called in renamed way, thus leading to the higher run-time in Table 7.5.

Specialiser	*solve*	*regexp*
Off-line		
LOGEN	0.31 s	0.21 s
LOGIMIX	-	0.24 s
Semi On-line		
LEUPEL	0.16 s	0.21 s
On-line		
ECCE-X	0.16 s	-
MIXTUS	0.28 s	0.24 s
PADDY	0.30 s	0.27 s
PADDY	0.48 s	-

Table 7.5: Speed of the residual programs (for a large number of queries)

at least 7.9 hours (11.8 hours with garbage collection). The speedups by using the generating extension instead of the partial evaluator range from 2.7 to 3.6 but the execution times for the system (including pre- and post-processing) still range from $113s$ to $447s$.

7.5 Discussion and future work

In comparison to other partial deduction methods the cogen approach may, at least from the examples given in this chapter, seem to do quite well with respect to speedup and quality of residual code, and outperform any other system with respect to transformation speed. But this efficiency has a price. Firstly, since our approach is off-line it will of course suffer from the same deficiencies than other off-line systems when compared to on-line systems. For instance, an off-line system has to resort to a much simpler control strategy during specialisation, and incorporating refined control methods, like the ones developed in Part II of the thesis, is usually not possible. This means e.g. that off-line systems are not able to perform unfolding at the object level, as discussed in Sections 7.4.2 and 7.4.3). Secondly, no partially static structures were needed in the above examples and our system cannot handle these, so it will probably have difficulties with something like the *transpose* program (see [97] or Appendix C) or with a non-ground meta-interpreter. However, our notion of BTA and BTC is quite a coarse one and corresponds roughly to that used in early work on self-applicability of partial evaluators for functional programs, so one might expect that this could be refined considerably.

Although our approach is closely related to the one for functional programming languages there are still some important differences. Since computation in our cogen is based on unification, a variable is not forced to have a fixed binding-time assigned to it — the binding-time analysis is just required to be safe. Consider, for example, the following program:

```
g(X) :- p(X),q(X)
p(a).
q(a).
```

If the initial division Δ_0 states that the argument to `g` is dynamic, then Δ_0 is safe for the program and the unfolding rule that unfolds predicates `p` and `q`. The residual program that one gets by running the generating extensions is:

```
g__0(a).
```

In contrast to this any cogen for a functional language known to us will classify the variable `X` in the following analogue functional program (here exemplified in Scheme) as dynamic:

```
(define (g X) (and (equal? X a) (equal? X a)))
```

and the residual program would be identical to the original program.

One could say that our system allows divisions that are not uniformly congruent in the sense of Launchbury [162] and essentially, our system performs specialisation that a partial evaluation system for a functional language would need some form of *driving* to be able to do.

Whether application of the cogen approach is feasible for specialisation of other logical programming languages than Prolog is hard to say, but it seems essential that such languages have some meta-level built-in predicates, like Prolog's `findall` and `call` predicates, for the method to be efficient. This means that it is probably very difficult, or even impossible, to use the approach (efficiently) for Gödel. On the other hand, in a language like XSB [243, 50], the cogen might actually become simpler because one might be able to use the underlying tabling mechanism for the memoisation, which currently has to be done explicitly by the generating extensions. Further work will be needed to establish these conjectures.

7.5.1 *BTA* based on groundness analysis

We now present some remarks on the relation between groundness analysis and *BTA*.

Since we imposed that a *static* term must be ground, one might think that the BTA corresponds exactly to groundness analysis (via abstract interpretation [57] for instance). This is however not entirely true because a standard groundness analysis gives information about the arguments at the point where a call is selected (and often imposing left-to-right selection). In other words, it gives groundness information at the local level when using some standard execution. A BTA however requires groundness information about the arguments of calls in the leaves, i.e. at the point where these atoms are lifted to the global control level.

So what we actually need is a groundness analysis adapted for unfolding rules and not for standard execution of logic programs. However, we will see that, by re-using and running a standard groundness analysis on a transformed version of the program to be specialised, we can come up with a reasonable BTA.

The groundness analysis which we will re-use is based on the PLAI system (implemented in SICStus Prolog) which is a domain independent framework for developing global analysers based on abstract interpretation. It was originally developed in [120], was subsequently enhanced with a more efficient fix-point algorithm [209, 210, 211]. In our experiments we will use the set sharing domain [131] provided with PLAI (sharing allows to infer groundness in a straightforward way — basically if a variable does not share with any other variable nor with itself then it is ground).

Let us now examine the Example 7.2.16 again and perform some modifications to the program $P_u^{\mathcal{L}}$ we produced earlier:

```
nont_u(X,T,R) :- t_u(a,T,V),nont_g(X,V,R).
nont_u(X,T,R) :- t_u(X,T,R).
t_u(X,[X|R],R).
nont_g(X,V,R).
```

All we have done is to remove the extra argument collecting the atoms in the leaves and we have also replaced the literal `true` (which corresponds to stopping the unfolding process and lifting `nont(X,V,R)` to the global level) by a special call to a new predicate `nont_g(X,V,R)`. In this way the call patterns of `nont_g` correspond almost exactly to the atoms which are lifted to the global level in Algorithm 7.2.14.

If we now run the groundness analysis on this program, stating that the entry point is `nont(X,T,R)`, with `X` being ground, we will obtain as a result that all calls to `nont_g` have their first argument ground. Also for the *solve* example of Section 7.4 this approach (by removing negative goals so that the results of the abstract interpretation remain a safe approximation) we obtain a correct BTC telling us that calls to `solve_g` will have the first argument ground!

However, note that the groundness analysis supposes a left-to-right selection rule. This results in an analysis which supposes that the non-reducible atoms are lifted to the global level as soon as they become leftmost (and not after the whole unfolding as been done). This might result in the groundness analysis being too conservative wrt the actual partial deduction. We can remedy this to some extent by moving all g-calls to the end of the clause. The optimal solution would be, for the groundness analysis to delay calls to g-calls functions as long as possible. It will have to be studied whether this can be obtained via some adaptation of the PLAI algorithm. Another promising direction might be based on using the techniques for the "prop" domain developed in [54].

Also note that the above process still needs a set $\mathcal{L}$ of reducible predicates. The big question is, how do we come up with such a set. One might use a "standard" strategy from functional programming (see [24]): every predicate p that is not deterministic will be added to the set of residual predicates $\mathcal{L}$ (and then groundness analysis will have to be run again,..., until a fixpoint is reached). Further work will be required to work out the exact theoretical and practical details of this approach. It will also have to be studied, whether in the new logic programming language Mercury a BTA becomes much easier, due to the presence of the type and mode declarations.[15]

7.5.2 Related work in partial evaluation and abstract interpretation

The first hand-written cogen based on partial evaluation principles was, in all probability, the system *RedCompile* for a dialect of Lisp [16]. Since then successful compiler generators have been written for many different languages and language paradigms [241, 125, 127, 20, 5, 25, 109].

In the context of definite clause grammars and parsers based on them, the idea of hand writing the compiler generator has also been used in [214, 215].[16] However, it is not based on (off-line) partial deduction. The exact relationship to our work is currently being investigated.

Also the construction of our program $P_u^{\mathcal{L}}$ (Definition 7.3.1) seems to be related to the idea of *abstract compilation*, as defined for instance in [120]. In abstract compilation a program P is first transformed and abstracted. Running this transformed program then performs the actual abstract interpretation analysis of P. In our case concrete execution of $P_u^{\mathcal{L}}$ performs

[15] Note however that the type and mode declarations are specified for fully known input and not for partially known input.

[16] Thanks to Ulrich Neumerkel for pointing this out.

(part of) the partial deduction process. Another similar idea has also been used in [271] to calculate abstract answers.

Note that in [54], a different kind of abstract compilation is presented, in which a transformed program is analysed (and does not perform the analysis itself). This seems to be related to the idea outlined in Section 7.5.1 for obtaining a BTA from an existing groundness analysis.

7.5.3 Future work

The most obvious goal of the near future is to see if a complete and precise binding-time analysis can be developed, e.g. by extending or modifying an existing groundness/sharing analysis, as outlined above. On a slightly longer term one might try to extend the cogen and the binding-time analysis to handle partially static structures as well as allowing more than two binding-times, thus leading to a multi-level cogen (see [109]). It also seems natural to investigate to what extent more powerful control and specialisation techniques (like the unfold/fold transformations, [222]) can be incorporated into the cogen in the context of conjunctive partial deduction (which will be presented later in Chapter 10).

Part IV

Optimising Integrity Checking by Program Specialisation

Chapter 8

Integrity Checking and Meta-Programming

8.1 Introduction and motivation

Integrity constraints play a crucial role in (among others) deductive databases, abductive and inductive logic programs.[1] They ensure that no contradictory data can be introduced and monitor the coherence of the program or database. From a practical viewpoint, however, it can be quite expensive to check the integrity after each update. To alleviate this problem, special purpose integrity simplification methods have been proposed (e.g. [39, 73, 188, 186, 242, 59]), taking advantage of the fact that the database was consistent prior to the update, and only verifying constraints possibly affected by the new information. However, even these refined methods often turn out to be not efficient enough. We will show how program specialisation can be used to get rid of some of these inefficiencies. The basic idea will be to write the integrity checking procedure as a meta-interpreter, which will then be optimised for certain transaction patterns.

There have been two lines of motivation for this part of the thesis. A first motivation came from participation in the ESPRIT-project COMPULOG. This project brought together leading European research groups, both from the areas of deductive databases and logic programming. One idea, that was a frequent issue of discussion among the partners, was the potential of deriving highly specialised integrity checks for deductive databases, by

[1] In the remainder of this part of the thesis we will concentrate on deductive databases, but the results are also valid for inductive or abductive logic programs with integrity constraints.

partially evaluating integrity checking procedures, implemented as meta-interpreters, with respect to particular update patterns and with respect to the more static parts of the database.

A second line of motivation is to promote partial deduction toward a broader community in software development research and industry, as a (mature) technology for automatic optimisation of software. Although partial deduction is by now a well-accepted and frequently applied optimisation technique within the logic programming community, very few reports on successfully optimised applications have appeared in the literature. This is quite in contrast with work on partial evaluation for functional - and even imperative - languages, where several "success stories", e.g in the areas of ray tracing [6, 205], scientific computing [112], simulation and modelling [284], [3, 4] have been published. As such, a second motivation for this work has been to report on a very successful application of partial deduction: our experiments illustrating how specialisation can significantly improve on the best general purpose integrity checking techniques known in the literature.

First though, we present some essential background in deductive databases and integrity checking. We also present a new method for specialised integrity checking, which we will turn out to be well suited for program specialisation.

8.2 Deductive databases and specialised integrity checking

Definition 8.2.1 (deductive database) A *deductive database* is a set of clauses.

A *fact* is a clause with an empty body, while an *integrity constraint* is a clause of the form $false \leftarrow Body$. A *rule* is a clause which is neither a fact nor an integrity constraint. As is well-known, more general rules and constraints can be reduced to this format through the transformations proposed in [187]. Constraints in this format are referred to as inconsistency indicators in [251].

For the purposes of this and the following chapter, it is convenient to consider a database to be *inconsistent*, or violating the integrity constraints, iff *false* is derivable in the database via SLDNF-resolution. Other views of inconsistency exist and some discussions can for instance be found in [41]. Note that we do not require a deductive database to be range-restricted. This is because our notion of integrity is based on SLDNF-resolution, which always gives the same answer irrespective of the underlying language (see

e.g. [252]). However, range-restriction is still useful as it ensures that no SLDNF-refutation will flounder.

As pointed out above, integrity constraints play a crucial role in several logic programming based research areas. It is however probably fair to say that they received most attention in the context of (relational and) deductive databases. Addressed topics are, among others, constraint satisfiability, semantic query optimisation, system supported or even fully automatic recovery after integrity violation and efficient constraint checking upon updates. It is the latter topic that we focus on in this and the following chapter.

Two seminal contributions, providing first treatments of efficient constraint checking upon updates in a deductive database setting, based upon the original work by Nicolas for relational databases [217], are [73] and [188, 186]. In essence, what is proposed is reasoning forwards from an explicit addition or deletion, computing indirectly caused implicit *potential updates*. Consider the following clause:

$$p(X,Y) \leftarrow q(X), r(Y)$$

The addition of $q(a)$ might cause implicit additions of $p(a,Y)$-like facts. Which instances of $p(a,Y)$ will actually be derivable depends of course on r. Moreover, some or all such instances might already be provable in some other way. Propagating such potential updates through the program clauses, we might hit upon the possible addition of *false*. Each time this happens, a way in which the update might endanger integrity has been uncovered. It is then necessary to evaluate the (properly instantiated) body of the affected integrity constraint to check whether *false* is actually provable in this way.

Propagation of potential updates, along the lines proposed in [186], can be formalised as follows.

Definition 8.2.2 (database update) A *database update*, U, is a triple $\langle Db^+, Db^=, Db^- \rangle$ such that $Db^+, Db^=, Db^-$ are mutually disjoint deductive databases.
We say that δ is an *SLDNF-derivation after* U for a goal G iff δ is an SLDNF-derivation for $Db^+ \cup Db^= \cup \{G\}$.
Similarly δ is an *SLDNF-derivation before* U for G if δ is an SLDNF-derivation for $Db^- \cup Db^= \cup \{G\}$.

Db^- are the clauses removed by the update and Db^+ are the clauses which are added by the update. Thus, $Db^- \cup Db^=$ represents the database state before the update and $Db^+ \cup Db^=$ represents the database state after the update.

Below, $mgu^*(A, B)$ represents a particular idempotent and relevant mgu of $\{A, B'\}$, where B' is obtained from B by renaming apart (wrt A). If no such unifier exists then $mgu^*(A, B) = fail$. The operation mgu^* has the following interesting property:

Proposition 8.2.3 Let A, B be two expressions. Then $mgu^*(A, B) = fail$ iff A and B have no common instance.

Proof $\Leftarrow$: Suppose $mgu^*(A, B) = \theta \neq fail$. This means that $A\theta = B\gamma\theta$ for some γ and A and B have a common instance and we have a contradiction. $\Rightarrow$: Suppose that A and B have the common instance $A\theta = B\sigma$ and let γ be the renaming substitution for B used by mgu^*. This means that for some γ^{-1} we have $B\gamma\gamma^{-1} = B$ and $B\gamma\gamma^{-1}\sigma = A\theta$. Now as the variables of $B\gamma$ and A are disjoint the set of bindings $\theta^* = \theta \mid_{vars(A)} \cup (\gamma^{-1}\sigma) \mid_{vars(B\gamma)}$ is a well defined substitution and a unifier of A and $B\gamma$, i.e. $mgu^*(A, B) \neq fail$ and we have a contradiction. □

Definition 8.2.4 (potential updates) Given a database update $U = \langle Db^+, Db^=, Db^- \rangle$, we define the set of *positive potential updates* $pos(U)$ and the set of *negative potential updates* $neg(U)$ inductively as follows:

$$pos^0(U) = \{A \mid A \leftarrow Body \in Db^+\}$$
$$neg^0(U) = \{A \mid A \leftarrow Body \in Db^-\}$$

$$\begin{aligned} pos^{i+1}(U) = \{&A\theta \mid A \leftarrow Body \in Db^=,\ Body = \ldots, B, \ldots, \\ &C \in pos^i(U) \text{ and } mgu^*(B, C) = \theta\} \\ \cup\ \{&A\theta \mid A \leftarrow Body \in Db^=,\ Body = \ldots, \neg B, \ldots, \\ &C \in neg^i(U) \text{ and } mgu^*(B, C) = \theta\} \end{aligned}$$

$$\begin{aligned} neg^{i+1}(U) = \{&A\theta \mid A \leftarrow Body \in Db^=,\ Body = \ldots, B, \ldots, \\ &C \in neg^i(U) \text{ and } mgu^*(B, C) = \theta\} \\ \cup\ \{&A\theta \mid A \leftarrow Body \in Db^=,\ Body = \ldots, \neg B, \ldots, \\ &C \in pos^i(U) \text{ and } mgu^*(B, C) = \theta\} \end{aligned}$$

$$pos(U) = \bigcup_{i \geq 0} pos^i(U)$$
$$neg(U) = \bigcup_{i \geq 0} neg^i(U)$$

These sets can be computed through a bottom-up fixpoint iteration.

Example 8.2.5 Let $Db^+ = \{man(a) \leftarrow\}$, $Db^- = \emptyset$ and let the following clauses be the rules of $Db^=$:

$$mother(X, Y) \leftarrow parent(X, Y), woman(X)$$

$father(X, Y) \leftarrow parent(X, Y), man(X)$
$false \leftarrow man(X), woman(X)$
$false \leftarrow parent(X, Y), parent(Y, X)$

For the update $U = \langle Db^+, Db^=, Db^- \rangle$, we then obtain, independently of the facts in $Db^=$, that $pos(U) = \{man(a), father(a, \_), false\}$ and $neg(U)=\emptyset$.

Simplified integrity checking, along the lines of the Lloyd, Topor and Sonenberg (LST) method [188, 186], then essentially boils down to evaluating the corresponding (instantiated through θ) $Body$ every time a $false$ fact gets inserted into some $pos^i(U)$. In Example 8.2.5, one would thus only have to check $\leftarrow man(a), woman(a)$. In practice, these tests can be collected, those that are instances of others removed, and the remaining ones evaluated in a separate constraint checking phase. The main difference with the exposition in [188, 186] is that we calculate $pos(U)$ and $neg(U)$ in one step instead of in two.[2]

Note that in the above definition we do not test whether an atom $A \in pos(U)$ is a "real" update, i.e. whether A is actually derivable after the update (this is what is called the *phantomness* test) and whether A was indeed not derivable before the update (this is called the *idleness* test). A similar remark can be made about the atoms in $neg(U)$. Other proposals for simplified integrity checking often feature more precise (but more laborious) update propagation. The method by Decker [73], for example, performs the phantomness test and computes *induced updates* rather than just potential updates. Many other solutions are possible, all with their own weak as well as strong points. An overview of the work during the 80s is offered in [41]. A clear exposition of the main issues in update propagation, emerging after a decade of research on this topic, can be found in [156]. [47] compares the efficiency of some major strategies on a range of examples. Finally, recent contributions can be found in, among others, [46, 75, 164, 251, 165].

Concentrating on potential updates has the advantage that one does not have to access the entire database for the update propagation. In fact, Definition 8.2.4 does not reference the facts in $Db^=$ at all (only clauses with at least one literal in the body are used). For a lot of examples, like databases with a large number of facts, this leads to very good efficiency (see e.g. the experiments in [47, 251]). It, however, also somewhat restricts the usefulness of a method based solely on Definition 8.2.4 when the rules and integrity constraints change more often than the facts.

Based upon Definition 8.2.4 we now formalise a particular simplification method in more detail. The following definition uses the sets $pos(U)$ and

[2] This approach should be more efficient, while yielding the same result, because in each iteration step the influence of an atom C is independent of the other atoms currently in pos^i and neg^i.

$neg(U)$ to obtain more specific instances of goals and detect whether the proof tree of a goal is potentially affected by an update.

Definition 8.2.6 (Θ_U^+, Θ_U^-) Given a database update U and a goal $G = \leftarrow L_1, \ldots, L_n$, we define:

$$\begin{aligned}
\Theta_U^+(G) = \{\theta \mid & C \in pos(U),\ mgu^*(L_i, C) = \theta, \\
& L_i \text{ is a positive literal and } 1 \leq i \leq n\ \} \\
\cup\ \{\theta \mid & C \in neg(U),\ mgu^*(A_i, C) = \theta, \\
& L_i = \neg A_i \text{ and } 1 \leq i \leq n\} \\
\Theta_U^-(G) = \{\theta \mid & C \in neg(U),\ mgu^*(L_i, C) = \theta, \\
& L_i \text{ is a positive literal and } 1 \leq i \leq n\ \} \\
\cup\ \{\theta \mid & C \in pos(U),\ mgu^*(A_i, C) = \theta, \\
& L_i = \neg A_i \text{ and } 1 \leq i \leq n\}
\end{aligned}$$

We say that G is *potentially added by* U iff $\Theta_U^+(G) \neq \emptyset$. Also, G is *potentially deleted by* U iff $\Theta_U^-(G) \neq \emptyset$.

Note that trivially $\Theta_U^+(G) \neq \emptyset$ iff $\Theta_U^+(\leftarrow L_i) \neq \emptyset$ for some literal L_i of G. The method by Lloyd, Sonenberg and Topor in [186] can roughly be seen as calculating $\Theta_U^+(\leftarrow Body)$ for each body $Body$ of an integrity constraint and then evaluating the simplified constraint $false \leftarrow Body\theta$ for every $\theta \in \Theta_U^+(\leftarrow Body)$.

For the Example 8.2.5 above we obtain:

$$\Theta_U^+(\leftarrow man(X), woman(X)) = \{\{X/a\}\} \text{ and}$$
$$\Theta_U^+(\leftarrow parent(X, Y), parent(Y, X)) = \emptyset$$

and thus obtain the following set of simplified integrity constraints:

$$\{false \leftarrow man(a), woman(a)\}$$

Checking this simplified constraint is of course much more efficient than entirely re-checking the unsimplified integrity constraints of Example 8.2.5.

In the method presented in the remainder of this section, we will use the substitutions Θ_U^+ slightly differently. First though, we characterise derivations in a database after an update, which were not present before the update.

Definition 8.2.7 (incremental SLDNF-derivation)
Let $U = \langle Db^+, Db^=, Db^- \rangle$ be a database update and let δ be an SLDNF-derivation after U for a goal G. A derivation step of δ will be called *incremental* iff it resolves a positive literal with a clause from Db^+ or if it selects

a ground negative literal $\neg A$ such that $\leftarrow A$ is potentially deleted by U. We say that δ is *incremental* iff it contains at least one incremental derivation step.

The treatment of negative literals in the above definition is not optimal. In fact "$\leftarrow A$ is potentially deleted by U" does not guarantee that the same derivation does not exist in the database state prior to an update. However an optimal criterion, due to its complexity, has not been implemented in the current approach.

Lemma 8.2.8 Let G be a goal and U a database update. If there exists an incremental SLDNF-derivation after U for G, then G is potentially added by U.

Proof Let $U = \langle Db^+, Db^=, Db^- \rangle$ and let δ be the incremental SLDNF-derivation after U for G. We define δ' to be the incremental SLDNF-derivation for G after U obtained by stopping at the first incremental derivation step of δ. Let $G_0 = G, G_1, \ldots, G_k$, with $k > 0$, be the sequence of goals of δ'. We will now prove by induction on the length k of δ' that G_0 is potentially added by U.
Induction Hypothesis: For $1 \leq k \leq n$ we have that G_0 is potentially added by U.
Base Case: $(k = 1)$. This means that only one derivation step has been performed, which must therefore be incremental. There are two possibilities: either a positive literal $L_i = A_i$ or a negative literal $L_i = \neg A_i$ has been selected inside G_0. In the first case the goal G_0 has been resolved with a standardised apart[3] clause $A \leftarrow Body \in Db^+$ with $mgu(A_i, A) = \theta$. Thus by Definition 8.2.4 we have $A \in pos^0(U)$ and by Definition 8.2.6 we obtain $\theta \in \Theta_U^+(G_0)$. In the second case we must have $\Theta_U^-(\leftarrow A_i) \neq \emptyset$ and by Definition 8.2.6 there exists a $C \in neg(U)$ such that $mgu^*(A_i, C) = \theta$. Hence we know that $\theta \in \Theta_U^+(\leftarrow G_0)$. So, in both cases $\Theta_U^+(\leftarrow G_0) \neq \emptyset$, i.e. $G = G_0$ is potentially added by U.
Induction Step: $(k = n+1)$. We can first apply the induction hypothesis on the incremental SLDNF-derivation for G_1 after U consisting of the last n steps of δ (i.e. whose sequence of goals is $G_1, \ldots, G_{n+1}$) to deduce that G_1 is potentially added by U.
Let $G_1 = \leftarrow L_1, \ldots, L_n$. We know that for at least one literal L_i we have that $\Theta_U^+(\leftarrow L_i) \neq \emptyset$.
If a negative literal has been selected in the derivation step from G_0 to

[3] So far we have not provided a formal definition of the notion of "standardising apart". Several ones, correct and incorrect, exist in the literature (see e.g. the discussion in [149] or [84]). Just suppose for the remainder of this proof that fresh variables, not occurring "anywhere else", are used.

G_1 then G_0 is also potentially added, because all the literals L_i also occur unchanged in G_0.
If a positive literal L'_j has been selected in the derivation step from G_0 to G_1 and resolved with the (standardised apart) clause $A \leftarrow B_1, \ldots, B_q \in Db^=$, with $mgu(L'_j, A) = \theta$, we have: $G_0 = \leftarrow L'_1, \ldots, L'_j, \ldots, L'_r$, $G_1 = \leftarrow (L'_1, \ldots, L'_{j-1}, B_1, \ldots, B_q, L'_{j+1}, \ldots, L'_r)\theta$.
There are again two cases. Either there exists a L'_p, with $1 \leq p \leq r \wedge p \neq j$, such that $\Theta^+_U(\leftarrow L'_p\theta) \neq \emptyset$. In that case we have for the more general goal $\leftarrow L'_p$ that $\Theta^+_U(\leftarrow L'_p) \neq \emptyset$[4] and therefore G_0 is potentially added.
In the other case there must exist a B_p, with $1 \leq p \leq q$, such that $\Theta^+_U(\leftarrow B_p\theta) \neq \emptyset$. If B_p is a positive literal, we have by Definition 8.2.6 for some $C \in pos(U)$ that $mgu^*(B_p\theta, C) = \sigma$. Therefore, by Definition 8.2.4, we know that there is an element $A' \in pos(U)$ which is more general than $A\theta\sigma$. As $A\theta$ is an instance of L'_j, L'_j and A' have the common instance $A\theta\sigma$ and thus $mgu^*(L'_j, A')$ must exist (by Proposition 8.2.3) and we can thus conclude that $\Theta^+_U(\leftarrow L'_j) \neq \emptyset$ and that $G = G_0$ is potentially added.
The proof is almost identical for the case that B_p is a negative literal. □

Definition 8.2.9 (relevant SLDNF-derivation) Let δ be a (possibly incomplete) SLDNF-derivation after $U = \langle Db^+, Db^=, Db^- \rangle$ for G_0 and let $G_0, G_1, \ldots$ be the sequence of goals of δ. We say that δ is a *relevant SLDNF-derivation after* U for G_0 iff for each G_i we either have that G_i is potentially added by U or δ_i is incremental after U, where δ_i is the sub-derivation leading from G_0 to G_i.

A refutation being a particular derivation we can specialise the concept and define *relevant SLNDF-refutations*. The following theorem will form the basis of our method for performing specialised integrity checking.

Theorem 8.2.10 (incremental integrity checking)
Let $U = \langle Db^+, Db^=, Db^- \rangle$ be a database update such that there is no SLDNF-refutation before U for the goal $\leftarrow$ *false*.
Then $\leftarrow$ *false* has an SLDNF-refutation after U iff $\leftarrow$ *false* has a *relevant* refutation after U.

Proof $\Leftarrow$: If $\leftarrow$ *false* has a relevant refutation then it trivially has a refutation, namely the relevant one.
$\Rightarrow$: The refutation must be incremental, because otherwise the derivation

[4] Note that this is *not* the case if we use just the *mgu* without standardising apart inside Definition 8.2.6. This technical detail has been overlooked in [188, 186]. Take for instance $L'_p = p(X)$ and $\theta = \{X/f(Y)\}$. Then $L'_p\theta$ unifies with $p(f(X)) \in pos(U)$ and the more general L'_p does not!

is also valid for $Db^{=} \cup Db^{-}$ and we have a contradiction. Let $G_0 = \leftarrow false, G_1, \ldots, G_k = \square$ be the incremental refutation. For each G_i, we either have that G_i occurs after the first incremental derivation step and hence the sub-derivation δ_i, leading from G_0 to G_i, is incremental. If on the other hand G_i is situated before the first incremental derivation step, we can use Lemma 8.2.8 to infer that G_i is potentially added. Thus the derivation conforms to Definition 8.2.9 and is relevant. $\square$

In other words, if we know that the integrity constraints of a deductive database were not violated before an update, then we only have to search for a *relevant* refutation of $\leftarrow false$ in order to check the integrity constraints after the update. Observe that, by definition, once a derivation is not relevant, it cannot be extended into a relevant one.

The method can be illustrated by re-examining Example 8.2.5. The goals in the SLD-tree in Figure 8.1 are annotated with their corresponding sets of substitutions Θ_U^+. The SLD-derivation leading to $\leftarrow parent(X, Y)$, $parent(Y, X)$ is not relevant and can therefore be pruned. Similarly all derivations descending from the goal $\leftarrow man(X)$, $woman(X)$ which do not use $Db^+ = \{man(a) \leftarrow\}$ are not relevant either and can also be pruned. However, the derivation leading to $\leftarrow woman(a)$ is incremental and is relevant even though $\leftarrow woman(a)$ is not potentially added.

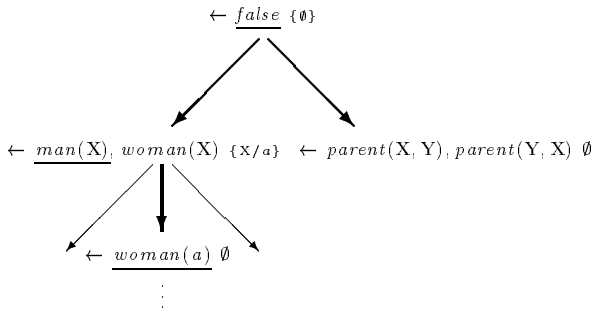


Figure 8.1: SLDNF-tree for Example 8.2.5

Note that the above method can be seen as an extension of the LST method in [186], because Θ_U^+ is not only used to simplify the integrity constraints at the topmost level (i.e. affecting the bodies of integrity constraints), but is used throughout the testing of the integrity constraints to prune non-relevant branches. An example where this aspect is important will be presented in Section 9.3. However, the LST method in [186] not

only removes integrity constraints but also instantiates them, possibly generating several specialised integrity constraints for a single unspecialised one. This instantiation often considerably reduces the number of matching facts and is therefore often vital for improving the efficiency of the integrity checks. The Definition 8.2.9 of relevant derivations does not use Θ_U^+ to instantiate intermediate goals. The reasons for this are purely practical, namely, to keep the method as simple as possible for effective partial evaluation. Definition 8.2.9 could actually be easily adapted to use Θ_U^+ for instantiating goals and Theorem 8.2.10 would still be valid. But, surprisingly, the instantiations will often be performed by the partial evaluation method itself and the results in Section 9.3 illustrate this. We will further elaborate on these aspects in Section 9.1.2.

8.3 Meta-interpreters and pre-compilation

A *meta-program* is a program which takes another program, the *object program*, as input, manipulating it in some way. Usually the object and meta-program are supposed to be written in (almost) the same language. Meta-programming can be used for (see e.g. [122, 14]) extending the programming language, modifying the control [60], debugging, program analysis, program transformation and, as we will see, specialised integrity checking.

Indeed, update propagation, constraint simplification and verification can be implemented through a meta-interpreter[5], manipulating updates and databases as object level expressions. A major benefit of such a meta-programming approach lies in the flexibility it offers: Any particular propagation and simplification strategy can be incorporated into the meta-program.

Furthermore, by partial evaluation of this meta-interpreter, we may (in principle) be able to pre-compile the integrity checking for certain update patterns. Let us re-examine Example 8.2.5. For the *concrete update* of Example 8.2.5, with $Db^+ = \{man(a) \leftarrow\}$, a meta-interpreter implementing the method of the previous section would try to find a refutation for $\leftarrow false$ in the manner outlined in Figure 8.1. By specialising this meta-interpreter for an *update pattern* $Db^+ = \{man(\mathcal{A}) \leftarrow\}$, $Db^- = \emptyset$, where $\mathcal{A}$ is not yet known, one might (hopefully) obtain a *specialised update procedure*, efficiently checking integrity, essentially as follows:

$$inconsistent(add(man(A))) \leftarrow evaluate(woman(A))$$

Given the *concrete* value for $\mathcal{A}$, this procedure will basically check consis-

[5] We sometimes also refer to a meta-program as a *meta-interpreter* if we want to emphasise the fact that the object program is executed rather than analysed.

tency in a similar manner to the unspecialised meta-interpreter, but will do this much more efficiently, because the propagation, simplification and evaluation process is already *pre-compiled*. For instance, the derivation in Figure 8.1 leading to $\leftarrow parent(X, Y)$, $parent(Y, X)$ has already been pruned at specialisation time. Similarly all derivations descending from the goal $\leftarrow man(X)$, $woman(X)$, which do not use Db^+, have also already been pruned at specialisation time. Finally, the specialised update procedure no longer has to calculate $pos(U)$ and $neg(U)$ for the concrete update U. All of this can lead to very high efficiency gains. Furthermore, for the integrity checking method we have presented in the previous section, the same specialised update procedure can be used as long as the rules and integrity constraints do not change (i.e. Db^+ and Db^- only contain facts). For example, after any concrete update which is an instance of the update pattern above, the specialised update procedure remains valid and does not have to be re-generated.

Both [282] and [251] explicitly address this compilation aspect. Their approaches are, however, more limited in some important respects and both use ad-hoc techniques and terminology instead of well-established and general apparatus provided by meta-interpreters and partial evaluation (the main concern of [251] is to show why using inconsistency indicators instead of integrity constraints is relevant for efficiency and a good idea in general).

In our approach we can consider any kind of update pattern, any kind of partial knowledge and any simplification method, simply by changing the meta-interpreter and the partial evaluation query — it is not fixed beforehand which part of the database is static and which part is subject to change.[6]

In the next chapter we will present a meta-interpreter for specialised integrity checking which is based on Theorem 8.2.10. This meta-interpreter will act on object level expressions which represent the deductive database under consideration. So, before presenting the meta-interpreter in more detail, it is advisable to discuss the issue of representing these object level expressions at the meta-level, i.e. inside the meta-interpreter.

[6] This can be very useful in practice. For instance, in [40], Bry and Manthey argue that for some applications facts change more often than rules and rules are updated more often than integrity constraints. As such, specialisation could be done with respect to dynamic EDBs, as well as with respect to dynamic IDBs, if so required by the application.

8.4 Some issues in meta-programming

8.4.1 The ground vs. the non-ground representation

In logic programming, there are basically two (fundamentally) different approaches to representing an object level expression, say the atom $p(X, a)$, at the meta-level. In the first approach one uses the term $p(X, a)$ as the meta-level representation. This is called a *non-ground representation*, because it represents an object level variable by a meta-level variable. In the second approach one would use something like the term $struct(p, [var(1), struct(a, [])])$ to represent the object level atom $p(X, a)$. This is called a *ground representation*, as it represents an object level variable by a ground term. Figure 8.2 contains some further examples of the particular ground representation which we will use throughout this thesis. From now on, we use "$\mathcal{T}$" to denote the ground representation of a term $\mathcal{T}$. Also, to simplify notations, we will sometimes use "p"$(t_1, \ldots, t_n)$ as a shorthand for $struct(p, [t_1, \ldots, t_n])$.

Object level	Ground representation
X	$var(1)$
c	$struct(c, [])$
$f(X, a)$	$struct(f, [var(1), struct(a, [])])$
$p \leftarrow q$	$struct(clause, [struct(p, []), struct(q, [])])$

Figure 8.2: A ground representation

The ground representation has the advantage that it can be treated in a purely declaratively manner, while for many applications the non-ground representation requires the use of extra-logical built-ins (like `var/1` or `copy/2`). The non-ground representation also has semantical problems (although they were solved to some extent in [64, 197, 198]). The main advantage of the non-ground representation is that the meta-program can use the "underlying"[7] unification mechanism, while for the ground representation an explicit unification algorithm is required. This (currently) induces a difference in speed reaching several orders of magnitude. The current consensus in the logic programming community is that both representations have their merits and the actual choice depends on the particular application. In the following subsection we discuss the differences between

[7] The term "underlying" refers to the system in which the meta-interpreter itself is written.

the ground and the non-ground representation in more detail. For further discussion we refer the reader to [122], [123, 34], the conclusion of [197].

Unification and collecting behaviour

As already mentioned, meta-interpreters for the non-ground representation can simply use the underlying unification. For instance, to unify the object level atoms $p(X, a)$ and $p(Y, Y)$ one simply calls $p(X, a) = p(Y, Y)$. This is very efficient, but after the call both atoms will have become instantiated to $p(a, a)$. This means that the original atoms $p(X, a)$ and $p(Y, Y)$ are no longer "accessible" (in Prolog for instance, the only way to "undo" these instantiations is via failing and backtracking), i.e. we cannot test in the same derivation whether the atom $p(X, a)$ unifies with another atom, say $p(b, a)$. This in turn means that it is impossible to write a breadth-first like or a collecting (i.e. performing something like `findall/3`[8]) meta-interpreter declaratively for the non-ground representation (it is possible to do this non-declaratively by using for instance Prolog's extra-logical `copy/2` built-in).

In the ground representation on the other hand, we cannot use the underlying unification (for instance $\text{"}p\text{"}(var(1), \text{"}a\text{"}) = \text{"}p\text{"}(var(2), var(2))$ will fail). The only declarative solution is to use an *explicit unification* algorithm. Such an algorithm, taken from [67], is included in Appendix H.1. (For the non-ground representation such an algorithm cannot be written declaratively; non-declarative features, like `var/1` and $= ../2$, have to be used.) For instance, $unify(\text{"}p\text{"}(var(1), \text{"}a\text{"}), \text{"}p\text{"}(var(2), var(2)), Sub)$ yields an explicit representation of the unifier in Sub, which can then be applied to other expressions. In contrast to the non-ground representation, the original atoms $\text{"}p\text{"}(var(1), \text{"}a\text{"})$ and $\text{"}p\text{"}(var(2), var(2))$ remain accessible in their original form and can thus be used again to unify with other atoms. Writing a declarative breadth-first like or a collecting meta-interpreter poses no problems.

Standardising apart and dynamic meta-programming

To standardise apart object program clauses in the non-ground representation, we can again use the underlying mechanism. For this we simply have to store the object program explicitly in meta-program clauses. For instance, if we represent the object level clause

[8] Note that the `findall/3` built-in is non-declarative, in the sense that the meaning of programs using it may depend on the selection rule. For example, given a program containing just the fact $p(a) \leftarrow$, we have that $\leftarrow findall(X, p(X), [A]), X = b$ succeeds (with the answer $\{A/p(a), X/b\}$) when executed left-to-right but fails when executed right-to-left.

$anc(X, Y) \leftarrow parent(X, Y)$

by the meta-level fact

$clause(1, anc(X, Y), [parent(X, Y)]) \leftarrow$

we can obtain a standardised apart version of the clause simply by calling $\leftarrow clause(1, Hd, Bdy)$. Similarly, we can resolve this clause with the atom $anc(a, B)$ by calling $\leftarrow clause(C, anc(a, B), Bdy)$.[9]

The disadvantage of this method, however, is that the object program is fixed, making it impossible to do "dynamic meta-programming" (i.e. dynamically change the object program, see [122]); this can be remedied by using a mixed meta-interpreter, as we will explain in Subsection 8.4.2 below). So, unless we resort to such extra-logical built-ins as `assert` and `retract`, the object program has to be represented by a term in order to do dynamic meta-programming. This in turn means that non-logical built-ins like `copy/2` have to be used to perform the standardising apart. Figure 8.3 illustrates these two possibilities. Note that without the `copy` in Figure 8.3, the second meta-interpreter would incorrectly fail for the given query. For our application this means that, on the one hand, using the non-logical copying approach unduly complicates the specialisation task while at the same time leading to a serious efficiency bottleneck. On the other hand, using the clause representation, implies that representing updates to a database becomes much more cumbersome. Basically, we also have to encode the updates explicitly as meta-program clauses, thereby making dynamic meta-programming impossible.

1. Using a Clause Representation	*2. Using a Term Representation*
$solve([]) \leftarrow$ $solve([H \mid T]) \leftarrow$ $\quad clause(H, B)$ $\quad solve(B), solve(T)$ $clause(p(X), []) \leftarrow$	$solve(P, []) \leftarrow$ $solve(P, [H \mid T]) \leftarrow$ $\quad member(Cl, P), copy(Cl, cl(H, B))$ $\quad solve(P, B), solve(P, T)$
$\leftarrow solve([p(a), p(b)])$	$\leftarrow solve([cl(p(X), [])], [p(a), p(b)])$

Figure 8.3: Two non-ground meta-interpreters with $\{p(X) \leftarrow\}$ as object program

For the ground representation, it is again easy to write an explicit standardising apart operator in a fully declarative manner. For instance, in the programming language Gödel [123] the predicate $RenameFormulas/3$ serves this purpose.

[9] However, we cannot generate a renamed apart version of $anc(a, B)$. The $copy/2$ built-in has to be used for that purpose.

Testing for variants or instances

In the non-ground representation we cannot test in a declarative manner whether two atoms are variants or instances of each other, and non-declarative built-ins, like `var/1` and `=../2`, have to be used to that end. Indeed, suppose that we have implemented a predicate $variant/2$ which succeeds if its two arguments represent two atoms which are variants of each other and fails otherwise. Then $\leftarrow variant(p(X), p(a))$ must fail and $\leftarrow variant(p(a), p(a))$ must succeed. This, however, means that the query $\leftarrow variant(p(X), p(a)), X = a$ fails when using a left to right computation rule and succeeds when using a right to left computation rule. Hence $variant/2$ cannot be declarative (the exact same reasoning holds for the predicate $instance/2$). Thus it is not possible to write declarative meta-interpreters which perform e.g. tabling, loop checks or subsumption checks.

Again, for the ground representation there is no problem whatsoever to write declarative predicates which perform variant or instance checks.

Specifying partial knowledge

One additional disadvantage of the non-ground representation is that it is more difficult to specify partial knowledge for partial evaluation. Suppose, for instance, that we know that a given atom (for instance the head of a fact that will be added to a deductive database) will be of the form $man(\mathcal{T})$, where $\mathcal{T}$ is a constant, but we don't know yet at partial evaluation time which particular constant $\mathcal{T}$ stands for. In the ground representation this knowledge can be expressed as $struct(man, [struct(C, [])])$. However, in the non-ground representation we have to write this as $man(X)$, which is unfortunately less precise, as the variable X now no longer represents only constants but stands for any term.[10]

Unfolding

Automatically unfolding a meta-interpreter in a satisfactory way is a non-trivial issue and has been the topic of a lot of contributions [158, 261, 220, 215, 21, 115, 196, 195] (see also [137] for an account about the specialisation of interpreters in functional programming languages). For the fully general case, this problem has not been solved yet. However, using a non-ground representation for goals in the meta-interpreter simplifies the

[10] A possible solution is to use the $=../2$ built-in to constrain X and represent the above atom by the conjunction $man(X), X = ..[C]$. This requires that the partial evaluator provides non-trivial support for the built-in $=../2$ and is able to specialise conjunctions instead of simply atoms, see Chapter 10.

control of unfolding. In fact, a simple variant test[11] inside the partial evaluator can sometimes be sufficient to guarantee termination when unfolding such a meta-interpreter. This is illustrated in Figure 8.4, where intermediate goals have been removed for clarity and where Prog represents an object program inside of which the predicate $p/1$ is recursive via $q/1$. The meta-interpreter unfolded in the left column uses a ground representation for resolution and a variant test of the partial evaluator will not detect a loop. The partial evaluator will have to abstract away the constants 1 and 3 in order to terminate and generate specialised code. However, if we unfold a meta-interpreter in which the goals are in non-ground form, the variant test is sufficient to detect the loop and no abstraction is needed to generate efficient specialised code. This point is completely independent of the internal representation the partial evaluator uses, i.e. of the fact whether the partial evaluator itself uses a ground or a non-ground representation — it might even be written in another programming language.

A ground solve	*Non-ground solve*
$solve(Prog, [struct(p, [var(1)])])$	$solve(Prog, [p(\_1)])$
$\downarrow$	$\downarrow$
$solve(Prog, [struct(q, [var(3)])])$	$solve(Prog, [q(\_3)])$
$\downarrow$	$\downarrow$
$solve(Prog, [struct(p, [var(3)])])$	$solve(Prog, [p(\_3)])$

Figure 8.4: Unfolding meta-interpreters

8.4.2 The mixed representation

Sometimes it is possible to combine the ground and the non-ground approaches. This was first exemplified by Gallagher in [97, 98], where a (declarative) meta-interpreter for the ground representation is presented. From an operational point of view, this meta-interpreter lifts the ground representation to the non-ground one for resolution. We will call this approach the *mixed* representation, as object level goals are in non-ground form while the object programs are in ground form. A similar technique was used in the self-applicable partial evaluator LOGIMIX [207, 138]. Hill and Gallagher [122] provide a recent account of this style of writing meta-interpreters. With that technique we can use the versatility of the ground representation for representing object level programs (but not goals), while

[11] Meaning that the partial evaluator does not unfold atoms which are variants of some covering ancestor.

still remaining reasonably efficient. Furthermore, as demonstrated by Gallagher in [97], and by the experiments in the next chapter, partial evaluation can in this way sometimes completely remove the overhead of the meta-interpretation. Performing a similar feat on a meta-interpreter using the full ground representation with explicit unification is much harder and has, to the best of our knowledge, not been accomplished yet (for some promising attempts see the partial evaluator SAGE [116, 115, 35], or the new scheme for the ground representation in [177]).

```
make_non_ground(GrTerm, NgTerm) ←
      mng(GrTerm, NgTerm, [], _Sub)

mng(var(N), X, [], [sub(N, X)]) ←
mng(var(N), X, [sub(M, Y)|T], [sub(M, Y)|T1]) ←
     (N = M → (T1 = T, X = Y) ; mng(var(N), X, T, T1))
mng(struct(F, GrArgs), struct(F, NgArgs), InSub, OutSub) ←
     l_mng(GrArgs, NgArgs, InSub, OutSub)

l_mng([], [], Sub, Sub) ←
l_mng([GrH|GrT], [NgH|NgT], InSub, OutSub) ←
     mng(GrH, NgH, InSub, InSub1),
     l_mng(GrT, NgT, InSub1, OutSub)
```

Figure 8.5: Lifting the ground representation

A meta-interpreter using the mixed representation contains a predicate *make_non_ground/2*, which lifts a ground term to a non-ground one. For instance, the query

$$\leftarrow make\_non\_ground(struct(f, [var(1), var(2), var(1)]), X)$$

succeeds with a computed answer similar to

$$\{X/struct(f, [\_49, \_57, \_49])\}.$$

The variables _49 and _57 are fresh variables (whose actual names may vary and are not important). The code for this predicate is presented in Figure 8.5 and a simple meta-interpreter based on it can be found in Figure 8.6. This code is a variation of the *Instance-Demo* meta-interpreter in [122], where the predicate *make_non_ground/2* is called *InstanceOf/2*. Indeed, although operationally *make_non_ground/2* lifts a ground term to a non-ground term, declaratively (when typing the predicates) the second argument of *make_non_ground/2* can be seen as representing all ground terms which are instances of the first argument; hence the names *In-*

$$
\begin{array}{l}
solve(Prog, []) \leftarrow \\
solve(Prog, [H|T]) \leftarrow \\
\quad non\_ground\_member(struct(clause, [H|Body]), Prog), \\
\quad solve(Prog, Body), solve(Prog, T) \\
\\
non\_ground\_member(NonGrTerm, [GrH|GrT]) \leftarrow \\
\quad make\_non\_ground(GrH, NonGrTerm) \\
non\_ground\_member(NonGrTerm, [GrH|GrT]) \leftarrow \\
\quad non\_ground\_member(NonGrTerm, GrT)
\end{array}
$$

Figure 8.6: An interpreter for the ground representation

stanceOf/2 and *Instance-Demo*. Also note that, in contrast to the original meta-interpreter presented in [97], these meta-interpreters can be executed without specialisation. One can even execute

$$\leftarrow make\_non\_ground(struct(man, [struct(C, [])]), X)$$

and obtain the computed answer $\{X/struct(man, [struct(C, [])])\}$. (However, the goal $\leftarrow make\_non\_ground(struct(man, [C]), X)$ has an infinite number of computed answers.) Observe that in Figure 8.5 we use an if-then-else construct, written as `(if -> then ; else)`, which for the time being we assume to be declarative (the Prolog if-then-else will, however, also behave properly because the first argument to *make_non_ground* will always be ground).

Behaviour of Meta-interpreter	Non-Ground	Mixed	Ground
Breadth-First/Findall	No	No	Yes
Dynamic Meta-Programming	No	Yes	Yes
Loop Checking/Tabling	No	No	Yes
Underlying Unification	Yes	Yes	No

Figure 8.7: Comparing the ground, non-ground and mixed representations

So, because the object program is in ground form, dynamic meta-programming is possible. Because the goals of the mixed representation are in non-ground form, features such as tabling, subsumption or loop checking cannot be added declaratively to a meta-interpreter for the mixed repre-

sentation. On the positive side, however, as discussed for the non-ground representation above, this also means that unfolding becomes simpler and sometimes a variant test can suffice.

So, for applications which do not require such features as tabling, the mixed representation is the most promising option.

Figure 8.7 summarises the possibilities and limitations of the different styles of writing meta-interpreters.

Chapter 9

Pre-Compiling Integrity Checks via Partial Evaluation of Meta-Interpreters

In this chapter we pursue our goal of pre-compiling the integrity checking procedure by writing it as a meta-interpreter which we then specialise for certain update patterns. As we have elaborated in the previous chapter, the mixed representation is the most promising option for the meta-interpreter, if we can avoid such features as subsumption or loop checking. It is clear that in the general setting of recursive databases this cannot be accomplished. However, for hierarchical databases, a loop or subsumption check is not needed to ensure termination of the integrity checking procedure. So, in a first approach we will restrict ourselves to hierarchical databases with negation and try to write the integrity checking procedure using the mixed representation. We will return to the issue of recursion in Section 9.4.

9.1 A meta-interpreter for integrity checking in hierarchical databases

9.1.1 General layout

We will now extend the meta-interpreter for the mixed representation of Figure 8.6 to perform specialised integrity checking, based upon Theo-

rem 8.2.10 presented earlier. This theorem tells us that we can stop resolving a goal G when it is not potentially added, unless we have performed an *incremental* resolution step earlier in the derivation. The program skeleton in Figure 9.1 implements this idea (the full Prolog code can be found in Appendix G.1).

The argument Updt contains the ground representation of the update $\langle Db^+, Db^=, Db^- \rangle$. The predicate *resolve_incrementally/3* performs incremental resolution steps (according to Definition 8.2.7). Non-incremental resolution steps are performed by *resolve_unincrementally/3*. The predicate *potentially_added/2* tests whether a goal is potentially added by an update based Definition 8.2.6. The implementations of *resolve_incrementally* and *resolve_unincrementally* are rather straightforward (see Appendix G.1). However, the implementation of *potentially_added* poses some subtle difficulties, as we will see in Subsection 9.1.2 below.

Specialised integrity checking now consists in calling

$$\leftarrow \mathit{incremental\_solve}\ (\ \text{“}\langle Db^+, Db^=, Db^- \rangle\text{”},\ \leftarrow \mathit{false})$$

The query will succeed if the integrity of the database has been violated by the update.

```
incremental_solve(Updt, Goal) ←
   potentially_added(Updt, Goal),
   resolve(Updt, Goal)

resolve(Updt, Goal) ←
   resolve_unincrementally(Updt, Goal, NewGoal),
   incremental_solve(Updt, NewGoal)
resolve(Updt, Goal) ←
   resolve_incrementally(Updt, Goal, NewGoal),
   Updt = "⟨Db+, Db=, Db-⟩",
   solve("Db= ∪ Db+", NewGoal)
```

Figure 9.1: Skeleton of the integrity checker

9.1.2 Implementing *potentially_added*

The rules of Definition 8.2.4, can be directly transformed into a simple logic program which detects in a naive top-down way whether a goal is potentially added or not. Making abstraction of the particular representation of clauses and programs, we might write *potentially_added* like this:

$$\begin{aligned}
&potentially_added(\langle DB^+, DB^=, DB^-\rangle, A) \leftarrow \\
&\qquad A \leftarrow \ldots \in DB^+ \\
&potentially_added(\langle DB^+, DB^=, DB^-\rangle, A) \leftarrow \\
&\qquad A \leftarrow \ldots, A', \ldots \in DB^=, \\
&\qquad potentially_added(\langle DB^+, DB^=, DB^-\rangle, A')
\end{aligned}$$

Such an approach terminates for hierarchical databases and is very easy to partially evaluate. It will, however, lead to a predicate which has multiple, possibly identical and/or subsumed[1] solutions. Also, in the context of the non-ground or the mixed representation, this predicate will instantiate the (non-ground) goal under consideration. This means that, to ensure completeness, we would either have to backtrack and try out a lot of useless instantiations[2], or collect all solutions and perform expensive subsumption tests to keep only the most general ones. The latter approach would have to make use of a `findall` primitive as well as an instance test, both of which are non-declarative (see Chapter 8) and very hard to partially evaluate satisfactorily; e.g. effective partial evaluation of `findall` has to the best of our knowledge not been accomplished yet. Let us illustrate this problem through an example.

Example 9.1.1 Let the following clauses be the rules of $Db^=$:

$$\begin{aligned}
&mother(X, Y) \leftarrow\ parent(X, Y), woman(X) \\
&father(X, Y) \leftarrow\ parent(X, Y), man(X) \\
&false \leftarrow\ mother(X, Y), father(X, Z)
\end{aligned}$$

Let $Db^- = \emptyset$ and $Db^+ = \{parent(a, b) \leftarrow, man(a) \leftarrow\}$ and as usual $U = \langle Db^+, Db^=, Db^-\rangle$. A naive top-down implementation will succeed 3 times for the query

$$\leftarrow potentially\_added(\text{“}U\text{”}, false)$$

and twice for the query

$$\leftarrow potentially\_added(\text{“}U\text{”}, father(X, Y))$$

with computed answers $\{X/a\}$ and $\{X/a, Y/b\}$. Note that the solution $\{X/a, Y/b\}$ is “subsumed” by $\{X/a\}$ (which means that, if floundering is not possible, it is useless to instantiate the query by applying $\{X/a, Y/b\}$).

The above example shows that using a naive top-down implementation of *potentially_added* inside the integrity checker of Figure 9.1 is highly inefficient, as it will result in a lot of redundant checking. A solution to this problem is to wrap calls to *potentially_added* into a *verify*(.) primitive,

[1] A computed answer θ of a goal G is called *subsumed* if there exists another computed answer θ' of G such that $G\theta$ is an instance of $G\theta'$.

[2] It would also mean that we would have to extend Theorem 8.2.10 to allow for instantiation, but this is not a major problem.

where $verify(G)$ succeeds once with the empty computed answer if the goal G succeeds in any way and fails otherwise. This solves the problem of duplicate and subsumed solutions. For instance, for Example 9.1.1 above, both

$\leftarrow verify(potentially\_added(\text{“}U\text{”}, false))$
$\leftarrow verify(potentially\_added(\text{“}U\text{”}, father(X, Y)))$

will succeed just once with the empty computed answer and no backtracking is required, as no instantiations are made.

The disadvantage of using *verify* is of course that no instantiations are performed (which in general cut down the search space dramatically). However, as will see later, these instantiations can often be performed by program specialisation.

Unfortunately, the $verify(.)$ primitive is not declarative. It can, however, be implemented with the Prolog if-then-else construct, whose semantics is still reasonably simple. Indeed, $verify(Goal)$ can be translated into

```
((Goal->fail;true)->fail;true).
```

Also, as we will see in the next section, specialisation of the if-then-else poses much less problems than for instance the full blown cut. This was already suggested by O'Keefe in [219] and carried out by Takeuchi and Furukawa in [268]. So, given the current speed penalty of the ground representation [35], employing the if-then-else seems like a reasonable choice. In the next section we will first present a subset of full Prolog, called If-Then-Else-Prolog or just ITE-Prolog, and discuss how ITE-Prolog-programs can be specialised. ITE-Prolog of course contains the if-then-else, but includes several built-ins as well. Indeed, once the if-then-else is added, there are no additional difficulties in handling some simple built-ins like `var/1`, `nonvar/1` and `=../2` (but not built-ins like `call/1` or `assert/1` which manipulate clauses and goals).

9.2 Partial evaluation of ITE-Prolog

9.2.1 Definition of ITE-Prolog

To define the syntax of ITE-Prolog-programs we partition the predicate symbols into two disjoint sets Π_{bi} (the predicate symbols to be used for built-ins) and Π_{cl} (the predicate symbols for user-defined predicates). A *normal atom* is then an atom which is constructed using a predicate symbol $\in \Pi_{cl}$. Similarly a *built-in atom* is constructed using a predicate symbol $\in \Pi_{bi}$. A *ITE-Prolog-atom* is either a normal atom, a built-in atom or it is an expression of the form $(if \rightarrow then; else)$ where $if, then$ and $else$ are

conjunctions of ITE-Prolog-atoms. A *ITE-Prolog-clause* is an expression of the form $Head \leftarrow Body$ where $Head$ is a normal atom and $Body$ is a conjunction of ITE-Prolog-atoms.

Operationally, the if-then-else of ITE-Prolog behaves like the corresponding construct in Prolog [262, 218]. The following informal Prolog clauses can be used to define the if-then-else [235]:

```
(If->Then;Else) :- If,!,Then.
(If->Then;Else) :- Else.
```

In other words, when the test `If` succeeds (for the first time) a local cut is executed and execution proceeds with the then part. Most uses of the cut can actually be mapped to if-then-else constructs [219] and the if-then-else can also be used to implement the negation.

Because the if-then-else contains a local cut, its behaviour is sensitive to the sequence of computed answers of the test-part. This means that the computation rule and the search rule have to be fixed in order to give a clear meaning to the if-then-else. From now on we will presuppose the Prolog left-to-right computation rule and the lexical search rule.

The two ITE-Prolog-programs hereafter illustrate this point.

Program P_1	*Program* P_2
$q(X) \leftarrow (p(X) \rightarrow X = c; fail)$	$q(X) \leftarrow (p(X) \rightarrow X = c; fail)$
$p(a) \leftarrow$	$p(c) \leftarrow$
$p(c) \leftarrow$	$p(a) \leftarrow$

Using the Prolog computation and search rules, the query $\leftarrow q(X)$ will fail for program P_1, whereas it will succeed for P_2. All we have done is change the order of the computed answers for the predicate $p/1$. This implies that a partial evaluator for ITE-Prolog has to ensure the preservation of the *sequence* of computed answers. This for instance is not guaranteed by any partial deduction method presented so far, all of which preserve the computed answers but not their sequence.

However, the semantics of the if-then-else remains reasonably simple and a straightforward denotational semantics [139], in the style of [15] and [230], can be given to ITE-Prolog. For example, suppose that we associate to each ITE-Prolog-literal L a denotation $[\![L]\!]_P$ in the program P, which is a possibly infinite sequence of computed answer substitutions with an optional element $\perp$ at the end of (a finite sequence) to denote looping. Some examples are then $[\![true]\!]_P = \langle \emptyset \rangle$, $[\![X = a]\!]_P = \langle \{X/a\} \rangle$ and $[\![fail]\!]_P = \langle \rangle$.

In such a setting the denotation of an if-then-else construct can simply

be defined by:

$$[\![(A \rightarrow B; C)]\!]_P = \begin{cases} [\![B\theta_1]\!]_P & \text{if } [\![A]\!]_P = \langle \theta_1, \ldots \rangle \\ [\![C]\!]_P & \text{if } [\![A]\!]_P = \langle\rangle \\ \langle \bot \rangle & \text{if } [\![A]\!]_P = \langle \bot \rangle \end{cases}$$

9.2.2 Specialising ITE-Prolog

In order to preserve the sequence of computed answers, we have to address a problem related to the *left-propagation* of bindings. For instance, unfolding a non-leftmost atom in a clause might instantiate the atoms to the left of it or the head of the clause . In the context of extra-logical built-ins this can change the program's behaviour. But even without built-ins, this left-propagation of bindings can change the order of solutions which, as we have seen above, can lead to incorrect transformations for programs containing the if-then-else. In the example below, P_4 is obtained from P_3 by unfolding the non-leftmost atom $q(Y)$, thereby changing the sequence of computed answers.

Program P_3	*Program* P_4
$p(X,Y) \leftarrow q(X), \underline{q(Y)}$	$p(X,a) \leftarrow q(X)$
$q(a) \leftarrow$	$p(X,b) \leftarrow q(X)$
$q(b) \leftarrow$	$q(a) \leftarrow$
	$q(b) \leftarrow$
Sequence of computed answers for $\leftarrow p(X,Y)$	
$\langle p(a,a), p(a,b), p(b,a), p(b,b) \rangle$	$\langle p(a,a), p(b,a), p(a,b), p(b,b) \rangle$

This problem of left-propagation of bindings has been solved in various ways in the partial evaluation literature [228, 227, 245, 244], as well as overlooked in some contributions (e.g. [95]). In the context of unfold/fold transformations of pure logic programs, preservation of the order of solutions, as well as left-termination, is handled e.g. in [230, 30].

In the remainder, we will use the techniques we described in [167] to specialise ITE-Prolog-programs. The method of [167] tries to strictly enforce the Prolog left-to-right selection rule. However, sometimes one does not want to select the leftmost atom, for instance because it is a built-in which is not sufficiently instantiated, or simply to ensure termination of the partial evaluation process. To cope with this problem, [167] extends the concept of LD-derivations and LD-trees to LDR-derivations and LDR-trees, which in addition to left-most resolution steps also contain *residualisation* steps. The latter remove the left-most atom from the goal and hide it

from the left-propagations of bindings. Generating the residual code from LDR-trees is discussed in [167].

Unfolding inside the if-then-else is also handled in a rather straightforward manner. This is in big contrast to programs which contain the full blown cut. The reason is that the full cut can have an effect on all subsequent clauses defining the predicate under consideration. By unfolding, the scope of a cut can be changed, thereby altering its effect. The treatment of cuts therefore requires some rather involved techniques (see [42], [245, 244] or [228, 227]). The cut inside the if-then-else, however, is local and does not affect the reachability and meaning of other clauses. It is therefore much easier to handle by a partial evaluator. The following example illustrates this. Unfolding $q(X)$ in the program P_5 with the if-then-else poses no problems and leads to a correct specialisation. However, unfolding the same atom in the program P_6 written with the cut leads to an incorrect specialised program in which e.g. $p(a)$ is no longer a consequence.

Program P_5	*Program* P_6
$p(X) \leftarrow (\underline{q(X)} \rightarrow fail; true)$	$p(X) \leftarrow \underline{q(X)}, !, fail$
	$p(X) \leftarrow$
$q(X) \leftarrow (X = a \rightarrow fail; true)$	$q(X) \leftarrow X = a, !, fail$
	$q(X) \leftarrow$

Unfolded Programs	
$p(X) \leftarrow ((X = a \rightarrow fail; true)$	$p(X) \leftarrow X = a, !, fail, !, fail$
$\rightarrow fail$	$p(X) \leftarrow !, fail$
$; \ true)$	$p(X) \leftarrow$

The exact details on how to unfold the if-then-else can be found in [167], where a freeness and sharing analysis is used to produce more efficient specialised programs by removing useless bindings as well as useless if-then-else tests. The latter often occur when predicates with output arguments are used inside the test-part of an if-then-else. A further improvement lies in generating multiple specialisations of a predicate call according to varying freeness information of the arguments.

In summary, partial evaluation of ITE-Prolog, can be situated somewhere between partial deduction of *pure* logic programs and partial evaluation of *full* Prolog (e.g. MIXTUS [245, 244] or PADDY [228, 229, 227] but also [279, 280]).

9.2.3 Some aspects of LEUPEL

The partial evaluation system LEUPEL, we have already encountered in Chapter 7, includes all the techniques sketched so far in this section. The

implementation originally grew out of [166]. In Section 9.3 we will apply this system to obtain specialised update procedures. Below we present two more aspects of that system, relevant for our application.

Safe left-propagation of bindings

At the end of Section 9.1.2 we pointed out that a disadvantage of using *verify* is that no instantiations are performed. Fortunately these instantiations can often be performed by the partial evaluation method, through pruning and safe left-propagation of bindings. Take for instance a look at the specialised update procedure in Figure 9.4 (to be presented later in Section 9.3) and generated for the update $Db^+ = \{man(a) \leftarrow\}, Db^- = \emptyset$. This update procedure tests directly whether $woman(\mathcal{A})$ is a fact, whereas the original meta-interpreter of Figure 9.1 would test whether there are facts matching $woman(X)$ and only afterwards prune all irrelevant branches. This instantiation performed by the partial evaluator is in fact the reason for the extremely high speedup figures presented in the results of Section 9.3. In a sense, part of the specialised integrity checking is performed by the meta-interpreter and part is performed by the partial evaluator.

The above optimisation was obtained by unfolding and pruning all irrelevant branches. In some cases we can also improve the specialised update procedures by performing a *safe* left-propagation of bindings. As we have seen in the previous subsection, left-propagation of bindings is in general unsafe in the context of ITE-Prolog (left-propagation is of course safe for purely declarative programs and is performed by ordinary partial deduction). There are, however, some circumstances where bindings can be left-propagated without affecting the correctness of the specialised program. The following example illustrates such a safe left-propagation, as well as its benefits for efficiency.

Example 9.2.1 Take the following clause, which might be part of a specialised update procedure.

$$incremental\_solve\_1(A) \leftarrow parent(X, Y), \\ (a\_test \rightarrow X = A, Y = b;\ X = A, Y = c)$$

Suppose that the computed answers of $parent/2$ are always grounding substitutions. This is always guaranteed for range restricted (see e.g. [41]) database predicates. In that case the binding $X = A$ can be left-propagated in the following way:

$$incremental\_solve\_1(A) \leftarrow \underline{X = A}, parent(X, Y), \\ (a\_test \rightarrow Y = b; Y = c)$$

This clause will generate the same sequence of computed answers than the original clause, but will do so much more efficiently. Usually, there

will be lots of *parent*/2 facts and *incremental_solve_*1 will be called with A instantiated. Therefore, the second clause will be much more efficient — the call to *parent* will just succeed once for every child of A instead of succeeding for the entire *parent* relation.

The LEUPEL partial evaluator contains a post-processing phase which performs conservative but safe left-propagation of bindings, common to *all* alternatives (like $X = A$ above). This guarantees that no additional choice points are generated but in the context of large databases this is usually not optimal. For instance, for the Example 9.2.1 above, we might also left-propagate the non-common bindings concerning X, thereby producing the following clauses:

$$\begin{aligned} incremental\_solve\_1(A) \leftarrow & \underline{X = A, Y = b}, parent(X, Y), \\ & (a\_test \rightarrow true; fail) \\ incremental\_solve\_1(A) \leftarrow & \underline{X = A, Y = c}, parent(X, Y), \\ & (a\_test \rightarrow fail; true) \end{aligned}$$

In general this specialised update procedure will be even more efficient. This indicates that the results in Section 9.3 can be even further improved.

Control of unfolding

The partial evaluator LEUPEL is decomposed into three phases:

- the *annotation phase*, which annotates the program to be specialised
- the *specialisation phase*, which performs the unfolding guided by the annotations of the first phase,
- the *post-processing phase*, which performs optimisation on the generated partial deductions (i.e. removes useless bindings, performs safe left-propagation of bindings, simplifies the residual code) and generates the residual program.

Such a decomposition has already proven to be useful for self-application in the world of functional programming (see e.g. [138]) as well as for logic programming ([207],[115]).

Unfortunately, the annotation phase of LEUPEL is not yet fully automatic and must usually be performed by hand. On the positive side, this gives the knowledgeable user very precise control over the unfolding, especially since some quite refined annotations are provided for. For instance, the user can specify that the atom $var(X)$ should be fully evaluated only if its argument is ground or if its argument is guaranteed to be free (at evaluation time) and that it should be residualised otherwise. Our partial evaluation method can thus be seen as being "semi on-line" in the sense that some unfolding decisions are made off-line while others are still made

on-line. This idea has recently been taken up for functional programming in [260] (see also the filters in [55]).

In our case, the approach of hand annotating the meta-interpreter is very sensible. Indeed, given proper care, the same annotated meta-program can be used for *any* kind of update pattern. Therefore, investing time in annotating the meta-interpreter of Appendix G.1 — which only has to be done *once* — gives high benefits for all consecutive applications and the second and third phases of LEUPEL will then be able to derive specialised update procedures **fully automatically**, as exemplified by the prototype [169].

9.3 Experiments and results

9.3.1 An example

Before showing the results of our method, let us first illustrate in what sense it *improves* upon the method of Lloyd, Sonenberg and Topor (LST) in [186].

Example 9.3.1 Let the rules in Figure 9.2 form the intensional part of $Db^{=}$ and let $Db^{+} = \{man(a) \leftarrow\}$, $Db^{-} = \emptyset$. Then, independently of the facts in $Db^{=}$, we have:

$$\begin{aligned} pos(U) = \{&man(a), father(a,\_), married\_to(a,\_), married\_man(a),\\ &married\_woman(\_), unmarried(a), false\} \end{aligned}$$

$$neg(U) = \{unmarried(a), false\}$$

The LST method of [186] will then generate the following simplified integrity constraints:

$$false \leftarrow man(a), woman(a)$$
$$false \leftarrow parent(a, Y), unmarried(a)$$

Given the available information, this simplification of the integrity constraints is not optimal. Suppose that some fact matching $parent(a, Y)$ exists in the database. Evaluating the second simplified integrity constraints above, then leads to the incomplete SLDNF-tree depicted in Figure 9.3 and subsequently to the evaluation of the goal:

$$\leftarrow woman(a), \neg married\_woman(a)$$

This goal is not *potentially added* and the derivation leading to the goal is not *incremental*. Hence, by Theorem 8.2.10, this derivation can be pruned and will never lead to a successful refutation, given the fact that the database was consistent before the update. The *incremental_solve*

meta-interpreter of Appendix G.1 improves upon this and does not try to evaluate $\leftarrow woman(a), \neg married\_woman(a)$.

$$
\begin{array}{l}
mother(X, Y) \leftarrow parent(X, Y), woman(X) \\
father(X, Y) \leftarrow parent(X, Y), man(X) \\
grandparent(X, Z) \leftarrow \ parent(X, Y), parent(Y, Z), \\
married\_to(X, Y) \leftarrow \\
\qquad parent(X, Z), parent(Y, Z), \\
\qquad man(X), woman(Y) \\
married\_man(X) \leftarrow married\_to(X, Y) \\
married\_woman(X) \leftarrow married\_to(Y, X) \\
unmarried(X) \leftarrow man(X), \neg married\_man(X) \\
unmarried(X) \leftarrow woman(X), \neg married\_woman(X) \\
\\
false \leftarrow man(X), woman(X) \\
false \leftarrow parent(X, Y), parent(Y, X) \\
false \leftarrow \ parent(X, Y), unmarried(X)
\end{array}
$$

Figure 9.2: Intensional part of $Db^=$

Actually, through partial evaluation of the *incremental_solve* meta-interpreter, this useless branch is already pruned at specialisation time. For instance, when generating a specialised update procedure for the update pattern $Db^+ = \{man(\mathcal{A}) \leftarrow\}$, $Db^- = \emptyset$, we obtain the update procedure presented in Figure 9.4.[3] This update procedure is very satisfactory and is in a certain sense optimal. The only way to improve it, would be to add the information that the predicates in the intensional and the extensional database are disjoint. For most applications this is the case, but it is not required by the current method. This explains the seemingly redundant test in Figure 9.4, checking whether there is a fact *married_to* in the database. Benchmarks concerning this example will be presented in Section 9.3.2.

Finally, we show that, although the LST method of [186] can also handle update patterns, the simplified integrity constraints that one obtains in that way are neither very interesting nor very efficient. Indeed, one might think that it is possible to obtain specialised update procedures immediately from

[3] The figure actually contains a slightly sugared and simplified version of the resulting update procedure. There is no problem whatsoever, apart from finding the time for coding, to directly produce the sugared and simplified version. Also, all the benchmarks were executed on un-sugared and un-simplified versions.

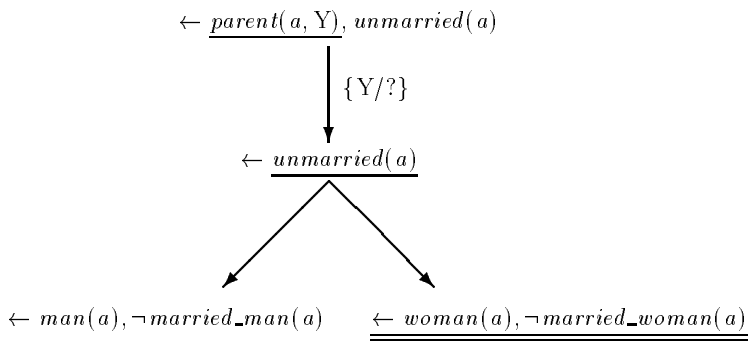


Figure 9.3: SLDNF-tree for Example 9.3.1

the LST method of [186], by running it on a generic update instead of a fully specified concrete update. Let us re-examine Example 9.3.1. For the generic update (pattern) $Db^+ = \{man(X) \leftarrow\}$,[4] $Db^- = \emptyset$, we obtain, independently of the facts in $Db^=$, the sets:

$$pos(U) = \{man(\_), father(\_,\_), married\_to(\_,\_), married\_man(\_), married\_woman(\_), unmarried(\_), false\}$$
$$neg(U) = \{unmarried(\_), false\}$$

The LST method of [186] will thus generate the following simplified integrity constraints:

$false \leftarrow man(X), woman(X)$
$false \leftarrow parent(X, Y), unmarried(X)$

So, by running LST on a generic update we were just able to deduce that the second integrity constraint cannot be violated by the update — the other integrity constraints remain unchanged. This means that the specialised update procedures we obtain by this technique will in general only be slightly faster than fully re-checking the integrity constraints after each update. For instance, the above procedure will run (cf. Table 9.1 in the next subsection) up to 3 orders of magnitude slower than the specialised update procedure in Figure 9.4 obtained by LEUPEL. So, although the pre-compilation process for such an approach will be very fast, the obtained

[4] The LST method of [186] (as well as any other specialised integrity checking method we know of) cannot (on its own) handle patterns of the form $man(\mathcal{A})$, where $\mathcal{A}$ stands for an as of yet unknown constant. We thus have to use the variable X to represent the update pattern for LST (replacing $\mathcal{A}$ by a constant would, in all but the simplest cases, lead to incorrect results).

```
incremental_solve_1(X1) :-
    fact(woman,[struct(X1,[])]).
incremental_solve_1(X1) :-
    fact(parent,[struct(X1,[]),X2]),
    (fact(married_to,[struct(X1,[]),X3])
     -> fail
     ;
      ((fact(parent,[struct(X1,[]),X4]),
        fact(parent,[X3,X4]),
        fact(woman,[X3]) )
        -> fail
        ; true
       )
     ).
```

Figure 9.4: Specialised update procedure for adding $man(\mathcal{A})$

update procedures usually have little practical value. In order for this approach to be more efficient, the LST method should be adapted so that it can distinguish between variables and unknown input stemming from the update pattern. But this is exactly what our approach based on partial evaluation of meta-interpreters achieves!

9.3.2 Comparison with other partial evaluators

In this subsection, we perform some experiments with the database of Example 9.3.1. The goal of these experiments is to compare the annotation based partial evaluation technique presented in Section 9.2 with some existing *automatic* partial evaluators for full Prolog and give a first impression of the potential of our approach. In Subsection 9.3.3, we will do a more extensive study on a more complicated database, with more elaborate transactions, and show the benefits compared to the LST method [186].

The times for the benchmark are expressed in seconds and were obtained by calling the $time/2$ predicate of Prolog by BIM, which incorporates the time needed for garbage collection, see [235]. We used sets of 400 updates and a fact database consisting of 108 facts and 216 facts respectively. The rule part of the database is presented in Figure 9.2. Also note that, in trying to be as realistic as possible, the fact part of the database has been simulated by Prolog facts (which are, however, still extracted a tuple at a time by the Prolog engine). The tests were executed on a Sun Sparc Classic running under Solaris 2.3.

The following different integrity checking methods were benchmarked:

1. **solve:** This is the naive meta-interpreter of Figure 8.6. It does not use the fact that the database was consistent before the update and simply tries to find a refutation for $\leftarrow false$.

2. **ic-solve:** This is the *incremental_solve* meta-interpreter performing specialised integrity checking, as described in Section 9.1. The skeleton of the meta-interpreter can be found in Figure 9.1, the full code is in Appendix G.1.

3. **ic-leupel:** These are the specialised update procedures obtained by specialising *ic-solve* with the partial evaluation system LEUPEL described in Section 9.2. A prototype, based on LEUPEL, performing these specialisations fully automatically, is publicly available in [169]. This prototype can also be used to get the timings for *solve* and *ic-solve* above as well as *ic-leupel*$^-$ below.

4. **ic-leupel$^-$:** These are also specialised update procedures obtained by LEUPEL, but this time with the safe left-propagation of bindings (see Section 9.2.3) disabled.

5. **ic-mixtus:** These specialised update procedures were obtained by specialising *ic-solve* using the automatic partial evaluator MIXTUS described in [244, 245]. Version 0.3.3 of MIXTUS, with the default parameter settings, was used in the experiments.

6. **ic-paddy:** These specialised update procedures were obtained by specialising *ic-solve* using the automatic partial evaluator PADDY presented in [227, 228, 229]. The resulting specialised procedures had to be slightly converted for Prolog by BIM: `get_cut/1` had to be transformed into `mark/1` and `cut_to/1` into `cut/1`. We also had to increase the "term_depth" parameter of PADDY from its default value. With the default value, PADDY actually slowed down the *ic-solve* meta-interpreter by about 30 %.

The first experiment we present consists in generating an update procedure for the update pattern:

$$Db^+ = \{man(\mathcal{A}) \leftarrow\},\ Db^- = \emptyset,$$

where $\mathcal{A}$ is unknown at partial evaluation time. The result of the partial evaluation obtained by LEUPEL can be seen in Figure 9.4 and the timings are summarised in Table 9.1. The first row of figures contains the absolute and relative times required to check the integrity for a database with 108 facts. The second row contains the corresponding figures for a database with 216 facts.

solve	*ic-solve*	*ic-leupel*	*ic-leupel*$^-$	*ic-mixtus*	*ic-paddy*
108 facts					
42.93 s	6.81 s	0.075 s	0.18 s	0.34 s	0.27 s
572.4	90.8	1	2.40	4.53	3.60
216 facts					
267.9 s	18.5 s	0.155 s	0.425 s	0.77 s	0.62 s
1728.3	119.3	1	2.74	4.96	4.00

Table 9.1: Results for $Db^+ = \{man(\mathcal{A}) \leftarrow\}$, $Db^- = \emptyset$

The times in Table 9.1 (as well as in Table 9.2) include the time to specialise the integrity constrains as well as the time to run them. Note that *solve* performs no specialisation, while all the other methods interleave execution and specialisation, as explained in Section 9.1. The times required to generate the specialised update procedures (*ic-leupel*, *ic-leupel*$^-$, *ic-mixtus* and *ic-paddy*) for the above update pattern are not included. In fact these update procedures only have to be regenerated when the rules or the integrity constraints change. The time needed to obtain the *ic-leupel* specialised update procedure was 78.19 s. The current implementation of LEUPEL has a very slow post-processor, displays tracing information and uses the ground representation. Therefore, it is certainly possible to reduce the time needed for partial evaluation by at least one order of magnitude. Still, even using the current implementation, the time invested into partial evaluation should pay off rather quickly for larger databases.

In another experiment we generated a specialised update procedure for the following update pattern:

$$Db^+ = \{parent(\mathcal{A}, \mathcal{B}) \leftarrow\},\ Db^- = \emptyset,$$

where $\mathcal{A}$ and $\mathcal{B}$ are unknown at partial evaluation time. This update pattern offers less opportunities for specialisation than the previous one. The speedup figures are still satisfactory but less spectacular. The results are summarised in Table 9.2.

In summary, the speedups obtained with the LEUPEL system are very encouraging. The specialised update procedures execute up to 2 orders of magnitude faster than the intelligent incremental integrity checker *ic-solve* and up to 3 orders of magnitude faster than the non-incremental *solve*. The latter speedup can of course be made to grow to almost any figure by using larger databases. Note that, according to our experience, specialising the *solve* meta-interpreter of Figure 8.6 usually yields speedups reaching at most 1 order of magnitude.

solve	*ic-solve*	*ic-leupel*	*ic-leupel*$^-$	*ic-mixtus*	*ic-paddy*
108 facts					
43.95 s	7.75 s	0.24 s	0.355 s	0.53 s	0.45 s
183.1	32.3	1	1.48	2.21	1.88
216 facts					
273.1 s	21.9 s	0.915 s	1.16 s	1.67 s	1.435 s
298	23.9	1	1.26	1.82	1.57

Table 9.2: Results for $Db^+ = \{parent(\mathcal{A}, \mathcal{B}) \leftarrow\}$, $Db^- = \emptyset$

Also, the specialised update procedures obtained by using the LEUPEL system performed between 1.6 and 5 times faster than the ones obtained by the fully automatic systems MIXTUS and PADDY. This shows that using annotations, combined with restricting the attention to ITE-Prolog instead of full Prolog, pays off in better specialisation. Finally, note that the safe left-propagation of bindings described in Section 9.2.3 has a definite, beneficial effect on the efficiency of the specialised update procedures.

9.3.3 A more comprehensive study

In this subsection, we perform a more elaborate study of the specialised update procedures generated by LEUPEL and compare their efficiency with the one of the LST technique [186], which often performs very well in practice, even for relational or hierarchical databases [74]. To that end we will use a more sophisticated database and more complicated transactions. The rules and integrity constraints $Db^=$ of the database are taken from in [251] and can be found in Appendix G.3.

For the benchmarks of this subsection, *solve*, *ic-solve* and *ic-leupel* are the same as in the Subsection 9.3.2. In addition we also have the integrity checking method *ic-lst*, which is an implementation of the LST method [186] and whose code can be found in Appendix G.2.[5]

For the more elaborate benchmarks, we used 5 different update patterns. The results are summarised in the Tables 9.3, 9.4, 9.5, 9.6 and 9.7. The particular update pattern, for which the specialised update procedures were generated, can be found in the table description. Different concrete updates, all instances of the given update pattern, were used to measure the efficiency of the methods. The second column of each table contains the integrity

[5] We also tried a "dirty" implementation using assert and retracts to store the potential updates. But, somewhat surprisingly, this solution ran slower than the one shown in Appendix G.2, which stores the potential updates in a list.

constraints violated by each concrete update. For each particular concrete update, the first row contains the absolute times for 100 updates and the second row contains the relative time wrt to LEUPEL. Each table is divided into sub-tables for databases of different sizes (and in case of *ic-leupel* the *same* specialised update procedure was used for the different databases). As in the previous experiments, the times to simplify and run the integrity constraints, given the concrete update, were included. The time to generate the specialised updated procedures (*ic-leupel*) is not included. As justified in Section 9.3.1, *ic-lst* is run on the concrete update and not on the update pattern.

As can be seen from the benchmark tables, the update procedures generated by LEUPEL perform extremely well. In Table 9.6, LEUPEL detected that there is no way this update can violate the integrity constraints — hence the "infinite" speedup. For 238 facts, the speedups in the other tables range from 99 to 918 over *solve*, from 89 to 324 over *ic-solve* and from 18 to 157 over *ic-lst*. These speedups are very encouraging and lead us to conjecture that the approach presented in this paper can be very useful in practice and lead to big efficiency improvements.

Of course, the larger the database becomes, the more time will be needed on the actual evaluation of the simplified constraints and not on the simplification. That is why the relative difference between *ic-lst* and *ic-leupel* diminishes with a growing database. However, even for 838 facts in Table 9.7, *ic-leupel* still runs 25 times faster than *ic-lst*. So for all examples tested so far, using an evaluation mechanism which is slower than in "real" database systems and therefore exaggerates the effect of the size of the database on the benchmark figures (but is still tuple-oriented — future work will have to examine how the current technique and experiments carry over to a set-oriented environment), the difference remains significant.

We also measured heap consumption of *ic-leupel*, which used from 43 to 318 times less heap space than *ic-solve*. Finally, in a small experiment we also tried to specialise *ic-lst* for the update patterns using MIXTUS, but without much success. Speedups were of the order of 10%.

Update	ICs viol	*solve*	*ic-solve*	*ic-leupel*	*ic-lst*
138 facts					
1	{2,2}	33.20 s 474	18.72 s 267	0.07 s 1	6.62 s 95
2	{8}	32.60 s 815	18.35 s 262	0.04 s 1	6.51 s 163
3	{}	33.10 s 473	18.54 s 265	0.07 s 1	6.51 s 93
238 facts					
1	{2,2}	69.60 s 535	32.56 s 250	0.13 s 1	6.75 s 52
2	{8}	68.30 s 683	32.40 s 324	0.10 s 1	6.51 s 65
3	{}	67.90 s 522	31.90 s 245	0.13 s 1	6.51 s 50

Table 9.3: Results for $Db^+ = \{father(\mathcal{X}, \mathcal{Y}) \leftarrow\}$, $Db^- = \emptyset$

Update	ICs viol	*solve*	*ic-solve*	*ic-leupel*	*ic-lst*
128 facts					
1	{5, 6}	28.80 s 169	35.40 s 208	0.17 s 1	13.12 s 77
2	{a1, a1, 5, 10}	29.70 s 87	37.21 s 109	0.34 s 1	12.58 s 37
3	{}	28.70 s 110	36.50 s 140	0.26 s 1	11.69 s 45
238 facts					
1	{5, 6}	67.50 s 225	77.61 s 259	0.30 s 1	12.88 s 43
2	{a1, a1, 5, 10}	68.00 s 99	80.20 s 116	0.69 s 1	12.66 s 18
3	{}	67.30 s 122	80.49 s 145	0.55 s 1	11.76 s 21

Table 9.4: Results for $Db^+ = \{civil\_status(\mathcal{X}, \mathcal{Y}, \mathcal{Z}, \mathcal{T}) \leftarrow\}$, $Db^- = \emptyset$

Update	ICs viol	*solve*	*ic-solve*	*ic-leupel*	*ic-lst*
138 facts					
1	{}	36.30 s 173	47.80 s 228	0.21 s 1	19.68 s 94
238 facts					
2	{}	78.10 s 190	95.30 s 232	0.41 s 1	20.32 s 50

Table 9.5: Results for $Db^+ = \{father(\mathcal{F}, \mathcal{X}),\ civil\_status(\mathcal{X}, \mathcal{Y}, \mathcal{Z}, \mathcal{T}) \leftarrow\}$, $Db^- = \emptyset$

Update	ICs viol	*solve*	*ic-solve*	*ic-leupel*	*ic-lst*
138 facts					
1	{}	59.60 s "∞"	3.50 s "∞"	0.00 s 1	6.50 s "∞"
2	{}	59.40 s "∞"	3.50 s "∞"	0.00 s 1	6.54 s "∞"
238 facts					
1	{}	59.70 s "∞"	3.50 s "∞"	0.00 s 1	6.55 s "∞"
2	{}	59.10 s "∞"	3.60 s "∞"	0.00 s 1	6.53 s "∞"

Table 9.6: Results for $Db^+ = \emptyset$, $Db^- = \{father(\mathcal{X}, \mathcal{Y}) \leftarrow\}$

Update	ICs viol	*solve*	*ic-solve*	*ic-leupel*	*ic-lst*
138 facts					
1	{8,8,9a}	59.90 s 333	13.45 s 75	0.18 s 1	16.60 s 92
2	{8,8,9a}	59.75 s 398	11.25 s 75	0.15 s 1	12.08 s 81
3	{}	60.70 s 759	11.00 s 69	0.08 s 1	11.96 s 150
4	{}	59.90 s 545	13.50 s 123	0.11 s 1	16.47 s 150
238 facts					
1	{8,8,9a}	74.10 s 390	17.50 s 92	0.19 s 1	17.38 s 91
2	{8,8,9a}	73.10 s 457	14.30 s 89	0.16 s 1	12.64 s 67
3	{}	73.50 s 918	14.00 s 175	0.08 s 1	12.56 s 157
4	{}	72.60 s 660	17.20 s 156	0.11 s 1	17.25 s 157
338 facts					
1	{8,8,9a}	114.20 s 407	22.30 s 80	0.28 s 1	17.09 s 61
438 facts					
1	{8,8,9a}	161.60 s 448	27.30 s 76	0.36 s 1	17.06 s 47
838 facts					
1	{8,8,9a}	391.50 s 576	48.10 s 71	0.68 s 1	17.26 s 25

Table 9.7: Results for $Db^+ = \emptyset$, $Db^- = \{civil\_status(\mathcal{X}, \mathcal{Y}, \mathcal{Z}, \mathcal{T}) \leftarrow\}$

9.4 Moving to recursive databases

The ITE-Prolog approach

As we have seen in the previous section, we were able to produce highly efficient update procedures for hierarchical databases by partial evaluation of meta-interpreters. The question is whether these results can be extended in a straightforward way to recursive, stratified databases, maybe by (slightly) adapting the meta-interpreter.

In theory the answer should be yes. The "only" added complication is that, in a top-down method, a loop check has to be incorporated into *potentially_added* to ensure termination. This loop check must act on the non-ground representation of the goal and can, for Datalog programs, be based on keeping a history of goals and using the instance or variant check to detect loops. As we have already seen in Section 8.4.1 such a loop check will be non-declarative for the mixed representation, but can be expressed within ITE-Prolog.

Unfortunately, we found out, through initial experiments, that such a loop check is very difficult to partially evaluate satisfactorily: often partial evaluation leads to an explosion of alternatives in the residual code, many of which are actually unreachable. Indeed, the partial evaluator should e.g. be able to detect that, if X is guaranteed to be free, then $p(a)$ is always an instance of $p(X)$. This requires intricate specialisation of built-ins using freeness information (something that might still be feasible with the LEUPEL system). Furthermore, the partial evaluator should be capable to detect that something like $p(f(\mathcal{A}))$ is always an instance of $p(\mathcal{A})$, no matter what $\mathcal{A}$ stands for. However, such reasoning requires analysing infinitely many different computations, something which standard partial evaluation cannot do.

In order to overcome this problem, stronger partial evaluation methods are needed. In Chapter 13 we will present such a technique, which combines partial deduction with a bottom-up abstract interpretation. Combining LEUPEL and its freeness analysis with the approach of Chapter 13 might result in a system which can obtain specialised update procedures for recursive, stratified databases. Further work will be needed to establish this.

The declarative approach

Another approach has been pursued in [176, 177]. There it was attempted to use *pure* logic programs and partial deduction in the hope of overcoming the above mentioned problems.

However, as we already mentioned in Section 8.4.1, any loop check for the non-ground representation or the mixed representation is non-declar-

ative by nature. But when using the ground representation, contrary to what one might expect, partial deduction is unable to specialise this meta-interpreter in an interesting way and no specialised update procedures can be obtained. To solve this problem, again an infinite number of different computations have to be analysed. We will return to this issue in Chapter 13 and show how a combination of partial deduction and abstract interpretation offers a solution. Another approach was presented in [176, 177], based on a new implementation of the ground representation combined with a custom specialisation technique. Some promising results were obtained, but the results are still far away from the good results obtained for hierarchical databases in this chapter. One reason was that the problems mentioned above for the instance check using the non-ground representation also persist to some extent with the ground representation.

The tabling approach

Finally, a third solution might lie with writing the integrity checking in a logic programming language with tabling (such as XSB [243, 50]) or within the deductive database itself (just like the meta-interpreters for magic-sets or abduction in [38, 263, 124]). In such a setting, the loop check does not have to be written within the meta-interpreter but can be (partly) delegated to the underlying system. This approach would enable the use of the non-ground or mixed representation and might even get rid of the need of the non-declarative *verify* primitive for the hierarchical case. However, specialisation of tabled logic programs is a largely unexplored area and poses some subtle difficulties. Determinate unfolding for instance, is no longer guaranteed to be beneficial and might even transform a terminating program into a non-terminating one (see [180, 76]). So, although this approach seems promising, it will require further research to gauge its potential.

9.5 Conclusion and future directions

Conclusion and discussion

We presented the idea of obtaining specialised update procedures for deductive databases in a principled way: namely by writing a meta-interpreter for specialised integrity checking and then partially evaluating this meta-interpreter for certain update patterns. The goal was to obtain specialised update procedures which perform the integrity checking much more efficiently than the generic integrity checking methods.

In Chapter 8 we have first described this approach in general and then presented a new integrity checking method well suited for partial evalu-

ation. We then discussed several implementation details of this integrity checking method and gained insights into issues concerning the ground versus the non-ground representation. We notably argued for the use of a "mixed" representation, in which the object program is represented using the ground representation, but where the goals are lifted to the non-ground representation for resolution. This approach has the advantage of using the flexibility of the ground representation for representing knowledge about the object program (in our case the deductive database along with the updates), while using the efficiency of the non-ground representation for resolution. The mixed representation is also much better suited for partial evaluation than the full ground representation.

In a first approach, we have restricted ourselves to normal, hierarchical databases in this Chapter. Also, for efficiency reasons, the meta-interpreter for specialised integrity checking had to make use of a non-declarative *verify* construct. This *verify* primitive can be implemented via the if-then-else construct. We have then presented an extension of pure Prolog, called ITE-Prolog, which incorporates the if-then-else and we have presented how this language can be specialised. We have drawn upon the techniques in [167] and presented the partial evaluator LEUPEL and a prototype [169] based upon it, which can generate specialised update procedure fully automatically.

This prototype has been used to conduct extensive experiments, the results of which were very encouraging. Speedups reached and exceeded 2 orders of magnitude when specialising the integrity checker for a given set of integrity constraints and a given set of rules. These high speedups are also due to the fact that the partial evaluator performs part of the integrity checking. We also compared the specialised update procedures with the well known approach by Lloyd, Sonenberg and Topor in [186], and the results show that big performance improvements, also reaching and exceeding 2 orders of magnitude, can be obtained.

Within this chapter, we have restricted our attention to hierarchical databases, which provided a conceptually simpler presentation, highlighting the basic issues in a clearer way. Still, one could argue that our experimental results should therefore not have been compared with alternative approaches for integrity checking in deductive databases, but compared to those for relational databases instead. However, even for relational databases — with complex views — the deductive database approaches, such as the LST method [186], seem to be the most competitive ones available. The simpler alternative of mapping down intensional database relations to extensional ones (through full unfolding of the views) and performing relational integrity checking on the extensional database predicates, tends to create extensive redundant multiplication of checks.

Example 9.5.1 Let us try to fully unfold the integrity constraints of the simple database in Figure 9.2. Note that in general, unfolding inside negation is tricky. Here however, we are lucky as all negated rules are just defined by 1 clause, and full unfolding will give:

$$
\begin{array}{l}
false \leftarrow man(X), woman(X) \\
false \leftarrow parent(X, Y), parent(Y, X) \\
false \leftarrow parent(X, Y), man(X), \neg parent(X, \_) \\
false \leftarrow parent(X, Y), man(X), \neg parent(\_Z, \_) \\
false \leftarrow parent(X, Y), man(X), \neg man(X) \\
false \leftarrow parent(X, Y), man(X), \neg woman(\_Z) \\
false \leftarrow parent(X, Y), woman(X), \neg parent(\_Z, \_) \\
false \leftarrow parent(X, Y), woman(X), \neg parent(X, \_) \\
false \leftarrow parent(X, Y), woman(X), \neg man(\_Z) \\
false \leftarrow parent(X, Y), woman(X), \neg woman(X)
\end{array}
$$

So even for this rather simple example, the fully unfolded integrity constraints become quite numerous.

For the update $Db^+ = \{man(a)\}$, $Db^- = \emptyset$ we then get, by unifying the update literals with the body atoms, the following specialised integrity constraints:

$$
\begin{array}{l}
false \leftarrow woman(a) \\
false \leftarrow parent(a, Y), \neg parent(a, \_) \\
false \leftarrow parent(a, Y), \neg parent(\_Z, \_) \\
false \leftarrow parent(a, Y), \neg man(a) \\
false \leftarrow parent(a, Y), \neg woman(\_Z)
\end{array}
$$

Note that the $parent(a, Y)$ has been duplicated 4 times! The resulting simplified integrity checks are thus not nearly as efficient as our specialised update procedure in Figure 9.4 (especially if a has a lot of children), which executes this $parent(a, Y)$ only once. For more complicated examples, entire conjunctions will get duplicated, making the situation worse. This is the price one has to pay for getting rid of the rules: rules capture common computations and avoid that they get repeated (memoisation will only solve this problem for atomic queries).

As such, even in the context of hierarchical databases, our comparisons are with respect to the best alternative approaches [74].

To summarise, it seems that partial evaluation is capable of automatically generating highly specialised update procedures for hierarchical databases with negation.

Future directions

In the current chapter we have already discussed how one might extend the current techniques and results to recursive, stratified databases. One might also apply the techniques of this chapter to other meta-interpreters, which have a more flexible way of specifying static and dynamic parts of the database and are less entrenched in the concept that facts change more often than rules and integrity constraints.

Another important point is the efficiency of generating the specialised update procedures, as opposed to running them. For the examples presented in this chapter, the update procedures have to be re-generated when the rules or the integrity constraints change (but not when the facts change). A technique, based on work by Benkerimi and Shepherdson [19], could be used to incrementally adapt the specialised update procedure whenever the rules or integrity constraints change. Another approach might be based on using a *self-applicable* partial evaluation system in order to obtain efficient update procedure compilers by self-application.

On the level of practical applications, one might try to apply the methods of this chapter to abductive and inductive logic programs. For instance, we conjecture that solvers for abduction, like the SLDNFA procedure [79], can greatly benefit in terms of efficiency, by generating specialised integrity checking procedures for each abducible predicate.

Finally, it might also be investigated whether partial evaluation *alone* is able to derive specialised integrity checks. In other words, is it possible to obtain specialised integrity checks by specialising a simple solve meta-interpreter, like the one of Figure 8.6. In that case, the assumption that the integrity constraints were satisfied before the update has to be handed to the specialiser, for instance in the form of a constraint. The framework of constrained partial deduction [172], which we mentioned in Section 5.4.2, could be used to that effect. In such a setting, self-applicable constrained partial deduction could be used to obtain specialised update procedures by performing the second Futamura projection [96, 91] and update procedure compilers by performing the third Futamura projection.

Part V

Conjunctive Partial Deduction

Chapter 10

Foundations of Conjunctive Partial Deduction

10.1 Partial deduction vs. unfold/fold

The partial evaluation and partial deduction approach, sometimes also referred to — in slightly different contexts — as program specialisation, has been our center of attention from Chapter 3 onwards. It falls within the more general area of *program transformation* techniques. Another approach to program transformation, which has (also) received considerable attention over the last few decades, is the *unfold/fold* approach (see e.g. [43],[93], [269], [249, 250], [222, 223, 232, 161, 233, 234, 231, 230]).

The relation between these two streams of work has been a matter of research, discussion and controversy. Some illuminating discussions, in the context of logic programming, can be found in [220], [278], [157], [232, 222], [250] and [29].

At first sight, their relation seems clear: partial deduction is a strict subset of the unfold/fold transformation. In essence, partial deduction refers to the class of unfold/fold transformations in which "unfolding" is the only basic transformation rule. Other basic unfold/fold rules, such as "folding", "definition", "lemma application" or "goal replacement", or — in more general unfold/fold contexts — "clause replacement" and others, are not supported.

This is however only a rough representation of the relationship between

the two approaches. To refine it, first, it should be noted that any partial deduction algorithm imposing the Lloyd-Shepherdson closedness condition ([185], cf. Definition 3.2.8), or a similar coveredness condition (cf. Definition 3.2.11 or Definition 5.2.1), implicitly achieves the effect of a folding transformation. If the condition holds, each atom in a body of a transformed clause refers back to one of the heads of the transformed clauses, and a limited form of implicit folding is obtained.

Moreover, most partial deduction methods make use of renaming transformations (cf. Section 3.3.2 and Definition 5.1.12) to ensure the Lloyd-Shepherdson independence condition (cf. Definition 3.2.8) while avoiding precision losses. Again, renaming is closely related to unfold/fold. Roughly stated, it can be formalised as a two-step basic transformation involving a "definition" step (the new predicate is defined to have the truth-value of the old one), immediately followed by a number of folding steps (appropriate occurrences of the old predicate are replaced by the corresponding new one).

In spite of these additional connections, there are still important differences between the unfold/fold and partial deduction methods. One is that there is a large class of transformations which are achievable through unfold/fold, but not through partial deduction. Typical instances of this class are transformations that eliminate "redundant variables" (see [222, 231, 233]). Two types of redundant variables are often distinguished in the literature. The first refer to those cases in which the same input datastructure is consumed twice. As an example, consider the predicate

$$max\_length(X, M, L) \leftarrow max(X, M), length(X, L)$$

which is true if M is the maximum of the list of numbers X, and if L is the length of the list. By unfold/fold, the definitions for $max/2$ and $length/2$ can be merged, producing a definition for $max\_length/3$, which only traverses X once. In the functional community, such a transformation is referred to as *tupling*.

A second type of redundant variables turns up in cases where a datastructure is first constructed by some procedure, and, in a next part of the computation, decomposed again. As an example, consider the predicate

$$double\_app(X, Y, Z, R) \leftarrow app(X, Y, I), app(I, Z, R)$$

which holds if the list R can be obtained by concatenating the lists X, Y and Z. Again, unfold/fold allows to merge the two calls to $app/3$ and to eliminate the construction of the intermediate datastructure I. In this case, in the functional community, the transformation is referred to as *deforestation* [281]. Neither of these transformations are achievable through partial deduction alone.

On the other hand, partial deduction has some advantages over unfold/fold as well. First, it should be noted that, in order to achieve effective specialisation, unfold/fold has to be augmented with some form of "dead code elimination". More importantly, however, due to its more limited applicability, and its resulting lower complexity, partial deduction can be more effectively and easily controlled. These control issues have obtained considerable attention in partial deduction research, and, in the current state-of-the-art, have obtained a level of refinement which goes beyond mere heuristic strategies, as we find in unfold/fold. Formal frameworks have been developed, analysing issues of termination, code- and search-explosion and obtained efficiency gains ([37, 199], [100, 98], [168, 178]). Several fully automated systems have been developed (SP [97, 98], SAGE [115, 116], PADDY [227, 228, 229], MIXTUS [245, 244], ECCE in Chapter 6) as well as semi-automated ones (LOGIMIX [207], LEUPEL [167, 173], see also Chapter 9, LOGEN in Chapter 7) have been developed and successfully applied to at least medium-size applications ([68], [158] and Chapter 9). As a result, partial deduction has reached a degree of maturity that brings it to the edge of wide-scale applicability, which is beyond what any other transformation technology for logic programs has achieved today.

The aim of this chapter is to provide a basic starting point to bring the advantages of these two transformation methods together. In order to do so, we only rely and build on two well-understood concepts, which have been at the basis of this thesis so far: the Lloyd-Shepherdson framework (cf. Chapter 3) and renaming transformations (cf. Chapters 3 and 5). (We will leave the more sophisticated control issues, related e.g. to characteristic trees and their preservation, as discussed in Chapters 4–6, aside for the time being.) No explicit new basic transformation rules, such as folding or definition, are introduced. Nevertheless, we provide a framework in which most tupling and deforestation transformations, in addition to the current partial deduction transformations, can be achieved.

More precisely, we propose two minimal extensions to partial deduction methods, prove their correctness and illustrate how they achieve removal of unnecessary variables within a framework of *conjunctive partial deduction*.

One of these minimal extensions is on the level of the Lloyd-Shepherdson framework itself: we will consider sets $\mathcal{Q}$ of conjunctions of atoms instead of the usual sets of atoms. A second extension is on the level of the renaming transformations, where we will use renamings of conjunctions of atoms, instead of renamings of single atoms. Together with a post-processing to be presented in Chapter 11, they provide for a setting of conjunctive partial deduction that — based on our current empirical evaluation — seems powerful enough to achieve the results of most unfold/fold transformations involving unfolding, folding and definition only.

We already pointed out that, at least on the technical level, most of these ideas have already been raised in the context of unfold/fold (e.g. [232, 222], [250]). In these papers, the step from partial deduction to (certain strategies in) unfold/fold has also been characterised as essentially moving from sets of atoms to sets of conjunctions. Therefore, technically, the current chapter is strongly related to these works, and for a number of our proofs we actually apply the results of [222]. The objectives, however, differ. Where [232, 222] aim to clarify the relation between the two approaches, by characterising partial deduction within the unfold/fold framework, we will provide minimal extensions to partial deduction with the aim of including a large part of the unfold/fold power. We show how correctness results for partial deduction can be reformulated in the extended context. We thus provide a generalised framework that allows to easily extend current results on control of partial deduction and current partial deduction systems. Particular instances of this framework, reusing the automatic methods for on-line control presented in Chapters 4–6 will be presented in Chapter 12. The so obtained method approaches more closely techniques for the specialisation and transformation of functional programs, such as deforestation [281], and supercompilation [273, 274, 258, 114]. Especially the latter constituted, together with unfold/fold transformations, a source of inspiration for the conception and design of conjunctive partial deduction.

Another related work is extended OLDT [33], which, in the context of abstract interpretation, extends OLDT [270, 145] to handle conjunctions. We will return to the relation of [33] to our work in Chapter 13.

10.2 Conjunctive partial deduction

In this section we provide extensions of the basic definitions in the Lloyd-Shepherdson framework [185] with renaming as presented in Chapter 3. We also illustrate how these extensions are sufficient to support the transformations referred to in the introduction. Throughout the chapter, we generally restrict our attention to definite programs and goals. In Section 10.4.1 we discuss extensions to the normal case.

10.2.1 Resultants

A crucial concept for partial deduction is the one of a *resultant*, which we defined in Definitions 3.2.3 and 3.2.4 of Chapter 3 for finite SLDNF-derivations and trees respectively. This allowed us to construct a specialised program by extracting clauses from finite, but possibly incomplete SLDNF-trees.

Let us briefly recall that the resultant of derivation for $P \cup \{\leftarrow Q\}$ leading to $\leftarrow B$ via the computed answer θ was defined to be $Q\theta \leftarrow B$. Note that in general such a resultant $Q\theta \leftarrow B$ is not a clause: the left-hand side Q may consist of a conjunction of atoms. In the presentation so far, based on the Lloyd-Shepherdson framework, the SLDNF-trees and derivations were restricted to having only *atomic* top-level goals. This restriction ensured that the resultants are indeed clauses. But as we will see later on, this also severely restricts the specialisation potential of partial deduction. We therefore omit this restriction here.

We first define partial deduction of a single conjunction.

Definition 10.2.1 (conjunctive partial deduction of Q) Let P be a program and Q a conjunction. Let τ be a finite, non-trivial and possibly incomplete SLD-tree for $P \cup \{\leftarrow Q\}$. Then the set of resultants $resultants(\tau)$ is called a *conjunctive partial deduction of Q in P*.

Let us immediately illustrate this notion with a simple example we already referred to at the beginning of the chapter. The example, as well as the *double_app* Example 10.2.7, is fairly trivial. This does not relate to any limitations of the proposed framework, but to a deliberate choice of selecting minimally complex examples for illustrating the proposed concepts and method. Also, in the following, we will use the connective $\wedge$ to avoid confusion between conjunction and the set punctuation symbol ",". We implicitly assume associativity of the connective $\wedge$, i.e. $Q_1 \wedge Q_2 \wedge Q_3$, $(Q_1 \wedge Q_2) \wedge Q_3$ and $Q_1 \wedge (Q_2 \wedge Q_3)$ all denote the same conjunction and we will usually use the first notation.

Example 10.2.2(max_length) Let P be the following program.

(C_1) $max\_length(X, M, L) \leftarrow max(X, M) \wedge length(X, L)$
(C_2) $max(X, M) \leftarrow max\_1(X, 0, M)$
(C_3) $max1([\,], M, M) \leftarrow$
(C_4) $max1([H \mid T], N, M) \leftarrow H \leq N \wedge max1(T, N, M)$
(C_5) $max1([H \mid T], N, M) \leftarrow H > N \wedge max1(T, H, M)$
(C_6) $length([\,], 0) \leftarrow$
(C_7) $length([H \mid T], L) \leftarrow length(T, K) \wedge L\ is\ K + 1$

Let $\mathcal{Q} = \{max\_length(X, M, L),\ max1(X, N, M) \wedge length(X, L)\}$. Assume that we construct the finite SLD-trees τ_1, τ_2 — depicted in Figure 10.1 — for the elements of $\mathcal{Q}$. The associated conjunctive partial deductions are then $resultants(\tau_1) = \{R_{1,1}\}$ and $resultants(\tau_2) = \{R_{2,1}, R_{2,2}, R_{2,3}\}$ respectively, where the individual resultants are as follows:

$(R_{1,1})$ $max\_length(X, M, L) \leftarrow max1(X, 0, M) \wedge length(X, L)$

$$
\begin{array}{ll}
(R_{2,1}) & max1([\,],N,N) \wedge length([\,],0) \leftarrow \\
(R_{2,2}) & max1([H\,|\,T],N,M) \wedge length([H\,|\,T],L) \leftarrow \\
 & \quad H \leq N \wedge max1(T,N,M) \wedge length(T,K) \wedge L \ is\ K+1 \\
(R_{2,3}) & max1([H\,|\,T],N,M) \wedge length([H\,|\,T],L) \leftarrow \\
 & \quad H > N \wedge max1(T,H,M) \wedge length(T,K) \wedge L \ is\ K+1
\end{array}
$$

If we take the union of the conjunctive partial deductions of the elements of $\mathcal{Q}$ we obtain the set of resultants $P_{\mathcal{Q}} = \{R_{1,1}, R_{2,1}, R_{2,2}, R_{2,3}\}$. Clearly $P_{\mathcal{Q}}$ is not a Horn clause program. Apart from that, with the exception that the redundant variable still has multiple occurrences, $P_{\mathcal{Q}}$ has the desired tupling structure. The two functionalities ($max/3$ and $length/2$) in the original program have been merged into single traversals.

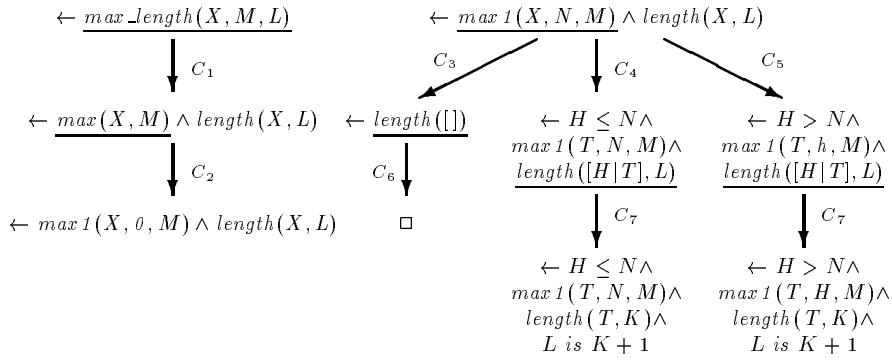


Figure 10.1: SLD-trees τ_1 and τ_2 for Example 10.2.2

10.2.2 Partitioning and renaming

In order to convert resultants into a standard logic program, we will rename conjunctions of atoms by new atoms. Such renamings require some care. For one thing, given a set of resultants $P_{\mathcal{Q}}$, obtained by taking the conjunctive partial deduction of the elements of a set $\mathcal{Q}$, there may be ambiguity concerning which conjunctions in the bodies to rename. For instance, if $P_{\mathcal{Q}}$ contains the clause $p(X,Y) \leftarrow r(X) \wedge q(Y) \wedge r(Z)$ and $\mathcal{Q}$ contains $r(U) \wedge q(V)$, then either the first two, or the last two atoms in the body of this clause are candidates for renaming. To formally fix such choices, we introduce the notion of a *partitioning function*.

Below $\mathcal{M}(A)$ denotes all multisets composed of elements of a set A and $=_r$ denotes identity of conjunctions, up to reordering. If M is a multiset

then we also use notations like $\wedge_{Q \in M} Q$ to denote a particular conjunction constructed from the elements in M, taking their multiplicity into account. For instance, for the multiset $M = \{p, p\}$, $\wedge_{Q \in M} Q$ refers to the conjunction $p \wedge p$.

Definition 10.2.3 (partitioning function) Let $\mathcal{C}$ denote the set of all conjunctions of atoms over the given alphabet. A *partitioning function* is a mapping $p : \mathcal{C} \rightarrow \mathcal{M}(\mathcal{C})$, such that for any $C \in \mathcal{C}$: $C =_r \wedge_{Q \in p(C)} Q$.

For the *max_length* example, let p be the partitioning function which maps any conjunction $C =_r max1(X, N, M) \wedge length(X, L) \wedge B_1 \wedge \ldots \wedge B_n$ to $\{max1(X, N, M) \wedge length(X, L), B_1, \ldots, B_n\}$, where $B_1, \ldots, B_n$ are $n \geq 0$ atoms with predicates different from *max1* and *length*. We leave p undefined on other conjunctions.

Note that the multiplicity of literals is relevant for the c.a.s. semantics[1] and we therefore have to use multisets — instead of just simple sets — for full generality in Definition 10.2.3 above.

Even with a fixed partitioning function, a range of different renaming functions could be introduced, all fulfilling the purpose of converting conjunctions into atoms (and therefore, resultants into Horn clauses). The differences are related to potentially added functionalities of these renamings, such as:

- elimination of multiply occurring variables (e.g. $p(X, X) \mapsto p'(X)$),
- elimination of redundant data structures (e.g. $q(a, f(Y)) \mapsto q'(Y)$),
- elimination of existential or unused variables.

Below we introduce a class of generalised renaming functions, supporting the first two functionalities stated above, but making abstraction of whether and how they are performed. We will also present a post-processing, supporting the third functionality, in Chapter 11.

The following is inspired from Definition 5.1.12 in Chapter 5.

Definition 10.2.4 (atomic renaming) An *atomic renaming* α for a given set of conjunctions $\mathcal{Q}$ is a mapping from $\mathcal{Q}$ to atoms such that

- for each $Q \in \mathcal{Q}$: $vars(\alpha(Q)) = vars(Q)$ and
- for $Q, Q' \in \mathcal{Q}$ such that $Q \neq Q'$: the predicate symbols of $\alpha(Q)$ and $\alpha(Q')$ are distinct (but not necessarily fresh).

Note that with this definition, we are actually also renaming the atomic elements of $\mathcal{Q}$. This is not really essential for converting generalised programs into standard ones, but, as already discussed in Chapter 3, proves useful for various other aspects (e.g. dealing with independence).

[1] Take for example $P = \{p(a, X) \leftarrow, p(X, b) \leftarrow\}$. Then $P \cup \{\leftarrow p(X, Y), p(X, Y)\}$ has an SLD-refutation with c.a.s. $\{X/a, Y/b\}$ while $P \cup \{\leftarrow p(X, Y)\}$ has not.

Definition 10.2.5 (renaming function) Let α be an atomic renaming for $\mathcal{Q}$ and p a partitioning function. A *renaming function* $\rho_{\alpha,p}$ for $\mathcal{Q}$ (based on α and p) is a mapping from conjunctions to conjunctions such that:

$\rho_{\alpha,p}(B) =_r \bigwedge_{C_i \in p(B)} \alpha(Q_i)\theta_i$ where each $C_i = Q_i\theta_i$ for some $Q_i \in \mathcal{Q}$.

If some $C_i \in p(B)$ is not an instance of an element in $\mathcal{Q}$ then $\rho_{\alpha,p}(B)$ is undefined. Also, for a goal $\leftarrow Q$, we define $\rho_{\alpha,p}(\leftarrow Q) = \leftarrow \rho_{\alpha,p}(Q)$.

Observe that we do not necessarily have that $\alpha(Q) = \rho_{\alpha,p}(Q)$. Indeed, there are two degrees of non-determinism for defining $\rho_{\alpha,p}$ once α and p are fixed. First, if $\mathcal{Q}$ contains elements Q and Q' which share common instances, then there are several possible ways to rename these common instances and a multitude of renaming functions based on the same atomic renaming α and partitioning p exist. Secondly, the order in which the atoms $\alpha(Q_i)\theta_i$ occur in $\rho_{\alpha,p}(B)$ is not fixed beforehand and may therefore vary from one renaming function to another. Usually one would like to preserve the order in which the unrenamed atoms occurred in the original conjunction B. This is however not always possible, namely when the partitioning function assembles non-contiguous chunks from B. Take for instance the conjunction $B = q_1 \wedge q_2 \wedge q_3$, a partitioning p such that $p(B) = \{q_1 \wedge q_3, q_2\}$ and an atomic renaming α such that $\alpha(q_1 \wedge q_3) = qq$ and $\alpha(q_2) = q$. Then $\rho_{\alpha,p}(B) = qq \wedge q$ and $\rho'_{\alpha,p}(B) = q \wedge qq$ are the only possible renamings and in both of them q_2 has changed position. Fortunately the order of the atoms is of no importance for the usual declarative semantics, i.e. it does neither influence the least Herbrand model, the computed answers obtainable by SLD-resolution nor the set of finitely failed queries. The order might however matter if we restrict ourselves to some specific selection rule (like LD-resolution). We will return to this issue in Chapter 12.

Definition 10.2.6 (conjunctive partial deduction wrt $\mathcal{Q}$) Let P be a program, $\mathcal{Q} = \{Q_1, \ldots, Q_n\}$ be a finite set of conjunctions and let $\rho_{\alpha,p}$ be a renaming function for $\mathcal{Q}$ based on the atomic renaming α and the partitioning function p. For each $i \in \{1, \ldots, n\}$, let P_{Q_i} be a conjunctive partial deduction of Q_i in P and let $P_{\mathcal{Q}} = \bigcup_{i \in \{1,\ldots,n\}} P_{Q_i}$.[2]
Then the program $\{\alpha(Q_i)\theta \leftarrow \rho_{\alpha,p}(B) \mid Q_i\theta \leftarrow B \in P_{Q_i} \wedge 1 \leq i \leq n \wedge \rho_{\alpha,p}(B) \text{ is defined } \}$ is called the *conjunctive partial deduction of P wrt $\mathcal{Q}$, $P_{\mathcal{Q}}$ and $\rho_{\alpha,p}$*.

Returning to Example 10.2.2, we introduce a different predicate for each of the two elements in $\mathcal{Q}$ via the atomic renaming α:

[2] We also implicitly assume that we know for which $Q_i \in \mathcal{Q}$ a particular resultant in $P_{\mathcal{Q}}$ was produced. Also, the same resultant might occur in different P_{Q_i}, and thus $P_{\mathcal{Q}}$ actually has to be a multiset of resultants (which poses no difficulties, however).

- $\alpha(max\_length(X, M, L)) = max\_length(X, M, L)$ and
- $\alpha(max1(X, N, M) \wedge length(X, L)) = ml(X, N, M, L)$.

$\mathcal{Q}$ does not contain elements with common instances and for the resultants at hand there exists only one renaming function $\rho_{\alpha,p}$ for $\mathcal{Q}$ based on α and p. The conjunctive partial deduction wrt $\mathcal{Q}$ is now obtained as follows. The head $max\_length(X, M, L)$ in the resultant $R_{1,1}$ is replaced by itself. The head-occurrences $max1([\,], N, N) \wedge length([\,], 0)$ and $max1([H|T], N, M) \wedge length([H|T], L)$ are replaced by $ml([\,], N, N, 0)$ and $ml([H|T], N, M, L)$. The body occurrences $max1(X, 0, M) \wedge length(X, L)$, $max1(T, N, M) \wedge length(T, K)$ as well as $max1(T, H, M) \wedge length(T, K)$ are replaced by the atoms $ml(X, 0, M, L)$, $ml(T, N, M, K)$ and $ml(T, H, M, K)$ respectively. The resulting program is:

$max\_length(X, M, L) \leftarrow ml(X, 0, M, L)$
$ml([\,], N, N, 0) \leftarrow$
$ml([H|T], N, M, L) \leftarrow H \leq N \wedge ml(T, N, M, K) \wedge L \; is \; K + 1$
$ml([H|T], N, M, L) \leftarrow H > N \wedge ml(T, H, M, K) \wedge L \; is \; K + 1$

Example 10.2.7(double append) Let $P = \{C_1, C_2\}$ be the by now well known *append* program.

(C_1) $app([\,], L, L) \leftarrow$
(C_2) $app([H|X], Y, [H|Z]) \leftarrow app(X, Y, Z)$

Let us employ this program to concatenate three lists, by using the goal $G = \leftarrow app(X, Y, I) \wedge app(I, Z, R)$, which concatenates the three lists X, Y, Z yielding as result R. This is achieved via two calls to *app* and the local variable I. The first call to *app* constructs from the lists X and Y an intermediate list I, which is then traversed when appending Z. While the use of the goal G is simple and elegant, it is rather inefficient since construction and traversal of such intermediate data structures is expensive.

Partial deduction within the framework of Lloyd and Shepherdson [185] *cannot* substantially improve the program since the atoms $app(X, Y, I)$, $app(I, Z, R)$ are transformed independently. We will now show, that conjunctive partial deduction *can* indeed remove the unnecessary variable I and get rid of the associated inefficiencies.

Let $\mathcal{Q} = \{app(X, Y, I) \wedge app(I, Z, R), app(X, Y, Z)\}$ and assume that we construct the finite SLD-tree τ_1 depicted in Figure 10.2 for the query $\leftarrow app(X, Y, I) \wedge app(I, Z, R)$ as well as a simple tree τ_2 with a single unfolding step for $\leftarrow app(X, Y, Z)$. Let $P_{\mathcal{Q}}$ consist of the clauses $resultants(\tau_2) = \{C_1, C_2\}$ as well as the resultants $resultants(\tau_1)$:

(R_1) $app([\,], Y, Y) \wedge app(Y, Z, R) \leftarrow app(Y, Z, R)$
(R_2) $app([H|X'], Y, [H|I']) \wedge app([H|I'], Z, [H|R']) \leftarrow$
$app(X', Y, I') \wedge app(I', Z, R')$

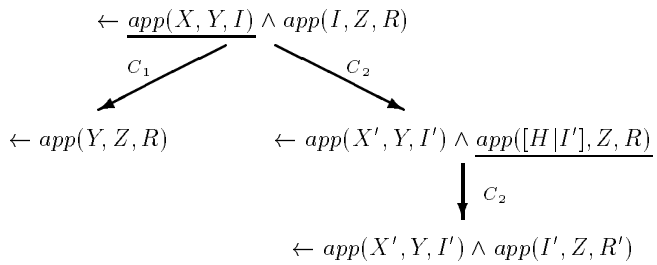


Figure 10.2: SLD-tree for Example 10.2.7

Suppose that we use a partitioning function p such that $p(B) = \{B\}$ for all conjunctions B. If we now take an atomic renaming α for $\mathcal{Q}$ such that $\alpha(app(X,Y,I) \wedge app(I,Z,R)) = da(X,Y,I,Z,R)$ and $\alpha(app(X,Y,Z)) = app(X,Y,Z)$ (i.e. the distinct variables have been collected and have been ordered according to their first appearance), the conjunctive partial deduction P' of P wrt $\mathcal{Q}$, $P_{\mathcal{Q}}$ and $\rho_{\alpha,p}$ will contain the clauses C_1, C_2 as well as:

$$(C_3') \quad da([\,],Y,Y,Z,R) \leftarrow app(Y,Z,R)$$
$$(C_4') \quad da([H|X'],Y,[H|I'],Z,[H|R']) \leftarrow da(X',Y,I',Z,R')$$

Executing $G = \leftarrow app(X,Y,I) \wedge app(I,Z,R)$ in the original program leads to the construction of an intermediate list I by $app(X,Y,I)$, which is then traversed again (consumed) by $app(I,Z,R)$. In the conjunctive partial deduction, the inefficiency caused by the unnecessary traversal of I is avoided as the elements encountered while traversing X and Y are stored directly in R. However, the intermediate list I is still constructed, and if we are not interested in its value, then this is an unnecessary overhead. We can remedy this by a rather straightforward post-processing phase, which we will present later in Chapter 11. The resulting specialised program then contains the clauses C_1, C_2 as well as:

$$(C_3) \quad da([\,],Y,Z,R) \leftarrow app(Y,Z,R)$$
$$(C_4) \quad da([H|X'],Y,Z,[H|R']) \leftarrow da(X',Y,Z,R')$$

The unnecessary variable I, as well as the inefficiencies caused by it, have now been completely removed.

10.3 Correctness results

In this section we will state and prove correctness results for conjunctive partial deduction.

10.3.1 Mapping to transformation sequences

As already mentioned in the introduction, standard partial deduction is a strict subset of the (full) unfold/fold transformation technique as defined for instance in the survey paper [222] by Pettorossi and Proietti. It is therefore not surprising that correctness can be established by showing that a conjunctive partial deduction can (almost) be obtained by a corresponding unfold/fold transformation sequence and then re-using correctness results from [222].

Note that, in contrast to [222], we treat programs as sets of clauses and not as sequences of clauses. The order (and multiplicity) of clauses makes no difference for the semantics we are interested in. In the remainder of this chapter we will use the notations $hd(C), bd(C)$ of [222] to refer to the head and the body of a clause C respectively.

As stated in [222], a program transformation process starting from an initial program P_0 is a sequence of programs $P_0, \ldots, P_n$, called a *transformation sequence*, such that program P_{k+1}, with $0 \leq k < n$, is obtained from P_k by the application of a *transformation rule*, which may depend on $P_0, \ldots, P_k$. We need the following four transformation rules from [222].

We start out by defining the unfolding rule [222, (R1)].

Definition 10.3.1 (Unfolding rule) Let P_k contain the clause $C = H \leftarrow F \wedge A \wedge G$, where A is a positive literal and where F and G are (possibly empty) conjunctions. Suppose that:

1. $\{D_1, \ldots, D_n\}$, with $n \geq 0$,[3] are all the clauses in a program P_j, with $0 \leq j \leq k$, such that A is unifiable with $hd(D_1), \ldots, hd(D_n)$, with most general unifiers $\theta_1, \ldots, \theta_n$, and
2. C_i is the clause $(H \leftarrow F \wedge bd(D_i) \wedge G)\theta_i$, for $i = 1, \ldots, n$.

If we *unfold* C *wrt* A *(using* $D_1, \ldots, D_n$*) in* P_j, we derive the clauses $C_1, \ldots, C_n$ and we get the new program $P_{k+1} = (P_k \setminus \{C\}) \cup \{C_1, \ldots, C_n\}$.

[3] [222, (R1)] actually stipulates that $n > 0$ and thus does not allow the selection of an atom which unifies with no clause — a situation which naturally arises when performing partial deduction. However, the "deletion of clauses with finitely failed body" rule [222, (R12)] can be used in those circumstances instead. Furthermore, all the correctness results that we will use from [222] allow both these rules to be applied, and we can thus effectively allow the case $n = 0$ as well.

For example, given $P_0 = \{p \leftarrow q \wedge r, q \leftarrow r\}$, we can unfold $p \leftarrow q \wedge r$ wrt q using P_0, deriving the clause $p \leftarrow r \wedge r$ and we get $P_1 = \{p \leftarrow r \wedge r, q \leftarrow r\}$.

The following is a simplified form of the folding rule [222, (R2)], sufficient for our needs.

Definition 10.3.2 (Folding rule) Let P_k contain the clause C and let D be a clause in a program P_j, with $0 \leq j \leq k$. Suppose that there exists a substitution θ such that:

1. C is the clause $H \leftarrow Bdy$, where $Bdy =_r bd(D)\theta \wedge F$ and where F is a (possibly empty) conjunction, and
2. for any clause $D' \neq D$ in P_j, $hd(D')$ is not unifiable with $hd(D)\theta$.

If we *fold* C *(wrt* $bd(D)\theta$*) using* D *in* P_j, we derive a clause $C' = H \leftarrow Bdy'$, with $Bdy' =_r hd(D)\theta \wedge F$, and we get the new program $P_{k+1} = (P_k \setminus \{C\}) \cup \{C'\}$.

For example, we can fold $p \leftarrow q \wedge r$ wrt r using $q \leftarrow r$ in P_0 above, giving as result the clause $p \leftarrow q \wedge q$ and the program $P_1 = \{p \leftarrow q \wedge q, q \leftarrow r\}$.

The following defines the Tamaki&Sato-folding (or T&S-folding) rule [222, (R3)], which is a restricted form of the folding rule above (the T&S-folding and -definition rules are initially from the paper [269] by Tamaki and Sato).

Definition 10.3.3 (T&S-folding rule) An application of the folding rule of Definition 10.3.2 is an application of *T&S-folding rule* if the following additional requirements are verified:

1. θ restricted to $vars(bd(D)) \setminus vars(hd(D))$ is a variable renaming[4] whose image (i.e. $ran(\theta \downarrow_{vars(bd(D)) \setminus vars(hd(D))})$) has an empty intersection with the set $vars(H) \cup vars(F) \cup vars(hd(D)\theta)$, and
2. the predicate symbol of $hd(D)$ occurs in P_j only once, that is, in the head of the clause D (thus, D is not recursive).

In that case we say that P_{k+1} is obtained by *T&S-folding* C *(wrt* $bd(D)\theta$*) using* D *in* P_j.

We now present the T&S-definition rule [222, (R15)] which, under certain conditions, allows to introduce new definitions. For this we assume that all predicate symbols occurring in a transformation sequence $P_0, \ldots, P_k$ are

[4] I.e. a variable pure substitution which is a one-to-one and onto mapping from its domain to itself.

partitioned into the set of *new predicates* and the set of *old predicates*. New predicates are the ones which occur either in the head of exactly one clause in P_0, and nowhere else in P_0, or they occur in the head of a clause introduced by the T&S-definition rule below. Note that this definition of old and new predicates from [222] differs slightly from the one in [269].

Definition 10.3.4 (T&S-definition rule) Given a transformation sequence $P_0, \ldots, P_k$, we may get a new program P_{k+1} by adding to program P_k a clause $H \leftarrow Bdy$ such that:

1. the predicate of H does not occur in $P_0, \ldots, P_k$, and
2. Bdy is made out of literals with old predicates occurring in $P_0, \ldots, P_k$.

We say that P_{k+1} is obtained from P_k by the *T&S-definition* rule. If point 2, but not necessarily point 1, is verified we will say that P_{k+1} is obtained from P_k by the *definition* rule.

In Definition 10.3.5 below, we map a conjunctive partial deduction to a transformation sequence. Basically the conjunctive partial deduction P' of P wrt $\mathcal{Q}$, $P_{\mathcal{Q}}$ and $\rho_{\alpha,p}$ can be obtained from P using 4 transformation phases. In the first phase, one introduces definitions for every conjunction in $\mathcal{Q}$, using the same predicate symbol as in α. In the second phase, these new definitions get unfolded according to the SLD-trees for the elements in $\mathcal{Q}$: exactly one unfolding step for each corresponding resolution step in the SLD-trees. In the third phase, conjunctions in the bodies of clauses are folded using the definitions introduced in phase 1. Finally, in the fourth phase, the original definitions in P are removed. The first three phases can be mapped to the unfold/fold transformation framework of [222] in a straightforward manner. Phase 4 will have to be treated separately (because the clause removals do not meet the requirements of definition elimination transformations as defined in [222]).

In Definition 10.2.4 of an atomic renaming, we did not require that the predicate symbols of the renamings were fresh, i.e. it is possible to reuse predicate symbols that occur in the original program P. This is of no consequence, because the original program is "thrown away". However, in unfold/fold, the original program is not systematically thrown away and in definition steps one can usually only define fresh predicates. To simplify the presentation, we restrict ourselves in a first phase to atomic renamings which only map to fresh predicate symbols, not occurring in the original program P. Those atomic renamings will be called *fresh*. At a later stage, we will extend the result to any atomic renaming satisfying Definition 10.2.4.

Definition 10.3.5 (transformation sequence) Let P' be the conjunctive partial deduction of P wrt $\mathcal{Q}$, $P_{\mathcal{Q}}$ and $\rho_{\alpha,p}$. A *transformation sequence for* P' (given P, $\mathcal{Q}$, $P_{\mathcal{Q}}$ and $\rho_{\alpha,p}$) is a transformation sequence $P_0, \ldots, P_d, \ldots, P_u, \ldots, P_f$, such that $P_0 = P$, and

1. $P_0, \ldots, P_d$ is obtained by performing (only) definition introductions, namely exactly one for every element $Q \in \mathcal{Q}$: $P_i = P_{i-1} \cup \{\alpha(Q) \leftarrow Q\}$.

2. $P_d, \ldots, P_u$ is obtained by performing (only) unfolding steps using clauses of P_0, namely exactly one for every resolution step in the SLD-trees constructed (in order to obtain $P_{\mathcal{Q}}$) for the elements of $\mathcal{Q}$: i.e. if this resolution step in a tree for $Q \in \mathcal{Q}$ resolves a selected literal A with clauses $D_1, \ldots, D_n$, we perform an unfolding step of a clause $\alpha(Q)\theta \leftarrow F, A, G$ in some P_i wrt A, using $D_1, \ldots, D_n$ in P_0.

3. $P_u, \ldots, P_f$ is obtained by performing (only) folding steps, namely exactly one for every renamed conjunction C in the body of a clause of $P_{\mathcal{Q}}$: i.e. for $C = Q\theta$, such that $Q \in \mathcal{Q} \wedge C \in p(B) \wedge \rho_{\alpha,p}(C) = \alpha(Q)\theta$, where $H \leftarrow B \in P_{\mathcal{Q}}$, we fold a corresponding clause $H \leftarrow B'$ wrt $Q\theta$, (where $B' =_r Q\theta \wedge R$) using the definition $\alpha(Q) \leftarrow Q$ in P_d, yielding the new clause $H \leftarrow B''$ (with $B'' =_r \alpha(Q)\theta \wedge R$).

The following example illustrates the above definition.

Example 10.3.6 (double append, revisited) Let $P = P_0 = \{C_1, C_2\}$ be the *append* program of Example 10.2.7 and let $\mathcal{Q}$, $\rho_{\alpha,p}$, $P_{\mathcal{Q}}$ and P' be defined as in that example except that we adapt α slightly such that $\alpha(app(X,Y,Z)) = app'(X,Y,Z)$ (to make α fresh). Then the transformation sequence $P_0, P_1, P_2, P_3, P_4, P_5, P_6, P_7, P_f$, shown below, is a transformation sequence for P'. $P_1 = P_0 \cup \{Def_1\}$ and $P_2 = P_1 \cup \{Def_2\}$ are obtained by a definition introduction, where

(Def_1) $\quad da(X,Y,I,Z,R) \leftarrow app(X,Y,I) \wedge app(I,Z,R)$
(Def_2) $\quad app'(X,Y,Z) \leftarrow app(X,Y,Z)$

$P_3 = P_0 \cup \{U_1, U_2, Def_2\}$ is obtained by unfolding the clause Def_1 above wrt $app(X,Y,I)$, using P_0, where

(U_1) $\quad da([\,],Y,Y,Z,R) \leftarrow app(Y,Z,R)$
(U_2) $\quad da([H|X'],Y,[H|I'],Z,R) \leftarrow app(X',Y,I') \wedge app([H|I'],Z,R)$

$P_4 = P_0 \cup \{U_1, U_3, Def_2\}$ is obtained by unfolding the clause U_2 above wrt $app([H|I'],Z,R)$, using P_0, where

(U_3) $\quad da([H|X'],Y,[H|I'],Z,[H|R']) \leftarrow app(X',Y,I') \wedge app(I',Z,R')$

$P_5 = P_0 \cup \{U_1, U_3, U_4, U_5\}$ is now obtained by unfolding clause Def_2 wrt $app(X,Y,Z)$ using P_0, where

(U_4) $app'([], L, L) \leftarrow$
(U_5) $app'([H|X], Y, [H|Z]) \leftarrow app(X, Y, Z)$

$P_6 = P_0 \cup \{U_1, U_3', U_4, U_5\}$ is obtained by folding the clause U_3 wrt $(app(X, Y, I) \wedge app(I, Z, R))\theta$, using the clause Def_1 from P_1 above, where $\theta = \{X/X', I/I', R/R'\}$ and

(U_3') $da([H|X'], Y, [H|I'], Z, [H|R']) \leftarrow da(X', Y, I', Z, R')$

Finally, after two more folding steps using Def_2 from P_1 we obtain the final program $P_f = P_0 \cup \{U_1', U_3', U_4, U_5'\}$:

(C_1) $app([], L, L) \leftarrow$
(C_2) $app([H|X], Y, [H|Z]) \leftarrow app(X, Y, Z)$
(U_1') $da([\,], Y, Y, Z, R) \leftarrow app'(Y, Z, R)$
(U_3') $da([H|X'], Y, [H|I'], Z, [H|R']) \leftarrow da(X', Y, I', Z, R')$
(U_4) $app'([], L, L) \leftarrow$
(U_5') $app'([H|X], Y, [H|Z]) \leftarrow app'(X, Y, Z)$

Note that the steps from P_0 to P_1 and P_1 to P_2 are applications of the T&S definition introduction rule. Note that the last 3 steps are T&S-folding steps (see Definition 10.3.3). However, e.g. the folding step from P_5 to P_6 is not an instance of the reversible folding rule (R13) of [222] (which would require $app(X', Y, I') \wedge app(I', Z, R')$ to be folded with a clause in P_5 and different from U_3). Also note that $P_f \setminus P = P'$.

In Lemma 10.3.7 we establish a necessary condition in order to apply some of the theorems from [222], namely that the definition steps in Definition 10.3.5 are *T&S-definition* steps.

Lemma 10.3.7 Let $P_0, \ldots, P_f$ be a transformation sequence for P' constructed using a fresh atomic renaming. All the definition introduction steps of $P_0, \ldots, P_f$ are T&S definition steps.

Proof All definition steps are of the form: $P_k = P_{k-1} \cup \{\alpha(Q) \leftarrow Q\}$. The conditions imposed on α guarantee that the predicate of $\alpha(Q)$ does not occur in $P_0, \ldots, P_k$ (point 1 of Definition 10.3.4). Point 2 of Definition 10.3.4 requires that all the predicates in the body of the introduced clause are only old predicates. According to the definitions in [269], all predicates in P_0 are old and hence this condition is trivially satisfied. However, for the slightly modified definitions used in [222], this is not always the case, but we can use the following simple construction to make every predicate in P an old predicate. Let the predicates occurring in P be $p_1, \ldots, p_j$, and let $fresh$ and $fail$ be distinct propositions not occurring in P, nor in the image of α. We simply define $P_0 = P \cup \{C_f\}$ where $C_f = fresh \leftarrow fail, p_1(\bar{t_1}), \ldots, p_j(\bar{t_j})$ and the t_i are sequences of terms of correct length. By doing so, we do not modify any of the semantics we are interested in, but ensure that all the

predicates in P are old according to the definition in [222]. We implicitly assume the presence of such a C_f in the following lemmas and propositions as well (in case we want to apply the modified definitions used in [222]). □

10.3.2 Fair and weakly fair partial deductions

In Definition 10.2.1 (as well as in standard partial deduction, cf. Chapter 3), we required the SLD-trees to be non-trivial. In the context of standard partial deduction of atoms, this condition avoids problematic resultants of the form $A \leftarrow A$ and is fully sufficient for total correctness (given independence and coveredness). In the context of conjunctive partial deductions, we need (for correctness wrt the finite failure semantics) an extension of this condition:

Definition 10.3.8 (inherited, fair) Let the goal $G' = \leftarrow (A_1 \wedge \ldots A_{i-1} \wedge B_1 \wedge \ldots B_k \wedge A_{i+1} \wedge \ldots A_n)\theta$ be derived via an SLD-resolution step from the goal $G = \leftarrow A_1 \wedge \ldots A_i \wedge \ldots A_n$, and the clause $H \leftarrow B_1 \wedge \ldots B_k$, with selected atom A_i. We say that the atoms $A_1\theta, \ldots, A_{i-1}\theta, A_{i+1}\theta, \ldots, A_n\theta$ in G are *inherited from G in G'*. We extend this notion to derivations by taking the transitive and reflexive closure.

A finite SLD-tree τ for $P \cup \{G\}$ is said to be *fair* iff no atom in a dangling leaf goal L of τ is inherited from G in L.

The conjunctive partial deduction P' of P wrt $\mathcal{Q}$, $P_{\mathcal{Q}}$ and $\rho_{\alpha,p}$ is *fair* iff all the SLD-trees used to construct $P_{\mathcal{Q}}$ are fair.

The above means that every atom occurring in the top-level goal of an SLD-tree has to be selected at some point in every non-failing branch. For SLD-trees for atomic goals this notion coincides with the one of non-trivial trees. Also, for the folding steps that we will perform (in the transformation sequence associated with a conjunctive partial deduction), this corresponds to conditions of *fold-allowing* in [222, Definition 7] or *inherited* in [249]. All these conditions ensure that we do not encode an unfair selection rule in the transformation process, which is vital when trying to preserve the finite failure semantics (for a more detailed discussion see e.g. [249]).

Sometimes however, this definition, as well as the one of fold-allowing in [222] or the one of inherited in [249], imposes more unfolding than strictly necessary. In some cases this even forces one to perform non-leftmost, non-determinate unfolding. As already discussed in Chapter 3, notably in Example 3.3.3, this can have disastrous effects on the efficiency of the specialised program. Also, the tree τ_1 of Example 10.2.7 depicted in Figure 10.2 does not satisfy Definition 10.3.8, although the resulting program is actually totally correct. In order to make τ_1 fair one would have to perform one more unfolding step on $\leftarrow app(Y, Z, R)$.

The following, weaker notion of fairness remedies this problem.

Definition 10.3.9 (weakly fair) Let P' be the conjunctive partial deduction of P wrt $\mathcal{Q}$, $P_{\mathcal{Q}}$ and $\rho_{\alpha,p}$. For $Q \in \mathcal{Q}$ let $Leaves_Q$ denote the dangling leaf goals of the SLD-tree for $P \cup \{\leftarrow Q\}$ used to construct the corresponding resultants in $P_{\mathcal{Q}}$.
We first define the following (increasing) series of subsets of $\mathcal{Q}$:

- $Q \in \mathcal{WF}_0$ iff $Q \in \mathcal{Q}$ and for each $L \in Leaves_Q$ no atom is inherited from $\leftarrow Q$ in L.
- $Q \in \mathcal{WF}_{k+1}$ iff $Q \in \mathcal{Q}$ and for each $L \in Leaves_Q$ and each $C \in p(L)$ which contains an atom inherited from $\leftarrow Q$ in L and which gets renamed into $\alpha(Q')\theta$ (with $C = Q'\theta$ and $Q' \in \mathcal{Q}$) inside $\rho_{\alpha,p}(L)$ we have that $Q' \in \mathcal{WF}_k$.

Then P' is *weakly fair* iff there exists a number $0 \leq k < \infty$ such that $\mathcal{Q} = \mathcal{WF}_k$.

Note that if every SLD-tree τ_Q is fair (i.e. P' is fair) then P' is weakly fair, independently of the renaming function $\rho_{\alpha,p}$ (because no atom in a leaf is inherited from the root goal and thus $\mathcal{WF}_0 = \mathcal{Q}$). Intuitively, the above definition ensures that every atom in a conjunction Q in $\mathcal{Q}$ is either unfolded directly in the tree $\tau(Q)$ or it is folded on a conjunction Q' in which the corresponding atom is guaranteed to be unfolded (again either directly or indirectly by folding and so on in a well-founded manner).[5]

Example 10.3.10 Let $P_{\mathcal{Q}}$ be the resultants for the set of conjunctions $\mathcal{Q} = \{app(X,Y,I) \wedge app(I,Z,R),\ app(X,Y,Z)\}$ of Example 10.2.7. The simple tree for $P \cup \{\leftarrow app(X,Y,Z)\}$ is fair. Therefore $app(X,Y,Z) \in \mathcal{WF}_0$ independently of $\rho_{\alpha,p}$.

Let τ_1 be the SLD-tree of Figure 10.2 for $P \cup \{G\}$, with $G = \leftarrow app(X,Y,I) \wedge app(I,Z,R)$. The conjunctions in the dangling leaves of τ are $\{L_1, L_2\}$, with $L_1 = app(Y,Z,R)$ and $L_2 = app(X',Y,I') \wedge app(I',Z,R')$. The SLD-tree τ_1 is not fair, but for $\rho_{\alpha,p}$ of Example 10.2.7, we have that $app(X,Y,I) \wedge app(I,Z,R) \in \mathcal{WF}_1$:

- $app(X',Y,I')$ and $app(I',Z,R')$ are not inherited from G in $\leftarrow L_2$.
- $app(Y,Z,R)$ is inherited from G in $\leftarrow L_1$, but we have $\rho_{\alpha,p}(L_1) = \alpha(app(X,Y,Z))\theta$ and, as we have seen above, $app(X,Y,Z) \in \mathcal{WF}_0$.

So for $k = 1$ we have that $\mathcal{Q} = \mathcal{WF}_k$ and P' is thus weakly fair.

[5] It would be possible to further refine Definition 10.3.9 by treating each atom in a conjunction individually. However, this makes the correctness proofs much more intricate and the need for this refinement has not (yet) arisen in practice.

The following shows that, due to our particular way of defining renamings, the folding steps in a transformation sequence are *T&S-folding* steps.

Lemma 10.3.11 Let $P_0, \ldots, P_f$ be a transformation sequence for the conjunctive partial deduction P' of P_0 wrt $\mathcal{Q}$, $P_{\mathcal{Q}}$ and $\rho_{\alpha,p}$ based on a fresh atomic renaming. Then the folding steps in $P_0, \ldots, P_f$ are T&S-folding steps which satisfy the requirements of Theorems 8 and 10 in [222]. If in addition P' is fair, then the T&S-folding steps also satisfy the requirements of Theorem 12 in [222].

Proof In order to prove that the folding steps of Definition 10.3.5 are T&S-folding steps, we have to show that points 1 and 2 of Definition 10.3.3 hold. Point 2 states that the predicate symbol of $\alpha(Q)$ should occur only once in P_d (where P_d is the program of Definition 10.3.5 obtained after all definitions have been introduced), which holds trivially by construction of the definitions in P_d and because the atomic renaming α is fresh. Also, point 1 of Definition 10.3.3 states that variables removed by the renaming should be existential variables. Because we imposed $vars(\alpha(Q)) = vars(Q)$ for atomic renamings, no variables are removed and the criterion is trivially satisfied.
The fact that we have non-trivial SLD-trees ensures that at least one atom in B is fold-allowing (see [222, Definition 7]) and hence, the requirements of Theorem 8 and 10 hold. Furthermore, if P' is fair, then every atom in B wrt which T&S-folding is performed (i.e. every atom in Q) is fold-allowing, and the requirements of Theorem 12 are met. □

If P' is only weakly fair then the requirements of Theorem 12 in [222] are not met. We will have to deal with that special case separately later on.

Proposition 10.3.12 Let $P_0, \ldots, P_f$ be a transformation sequence for the conjunctive partial deduction P' of P_0 wrt $\mathcal{Q}$, $P_{\mathcal{Q}}$ and $\rho_{\alpha,p}$ based on a fresh atomic renaming. Then, for every goal G, such that its predicates occur in P_0, we have that

- $P_0 \cup \{G\}$ has an SLD-refutation with computed answer θ iff $P_f \cup \{G\}$ has.

If in addition P' is weakly fair then

- $P_0 \cup \{G\}$ has a finitely failed SLD-tree iff $P_f \cup \{G\}$ has.

Proof Lemmas 10.3.7 and 10.3.11 ensure that the prerequisites of Theorem 10 in [222] are met. Hence the computed answer semantics Sem_{CA} is

preserved under the above conditions and $P_0 \cup \{G\}$ has an SLD-refutation with computed answer θ iff $P_f \cup \{G\}$ has.[6]

If P' is fair we can use the same Lemmas 10.3.7 and 10.3.11 combined with Theorem 12 of [222] to deduce that the finite failure semantics Sem_{FF} is preserved, i.e. $P_0 \cup \{G\}$ has a finitely failed SLD-tree iff $P_f \cup \{G\}$ has.

As already mentioned earlier, in case P' is only weakly fair we cannot directly apply Theorem 12 of [222]. We therefore do a specific proof by induction on the minimum number min such that $\mathcal{Q} = \mathcal{WF}_{min}$, where the $\mathcal{WF}_i$ are defined as in Definition 10.3.9.

Induction Hypothesis: The finite failure semantics is preserved far all P' which are weakly fair and such that $min \leq k$.

Base Case: If $min = 0$ then for every $Q \in \mathcal{Q}$ no leaf contains an atom inherited from $\leftarrow Q$ and thus P' is fair. Hence, by the above reasoning, we can deduce the preservation of finite failure.

Induction Step: Let $min = k + 1$ and let $\mathcal{W} \subset \mathcal{Q}$ be defined as $\mathcal{W} = \mathcal{WF}_{k+1} \setminus \mathcal{WF}_k$. The idea of the proof is to unfold the clauses for the elements of $\mathcal{W}$ so that, according to Definition 10.3.9, they become elements of $\mathcal{WF}_k$ in the unfolded program P'_f. This will allow us to apply the induction hypothesis on P'_f. The details are elaborated in the following. As in Definition 10.3.9, we denote by $Leaves_Q$ (with $Q \in \mathcal{Q}$) the dangling leaf goals of the SLD-tree for $P \cup \{\leftarrow Q\}$ used to construct the corresponding resultants in $P_{\mathcal{Q}}$. Let P'_f be obtained from P_f by performing the following unfolding steps for every element $Q \in \mathcal{W}$:

> for each $L \in Leaves_Q$ and each $C \in p(L)$ which contains an atom inherited from $\leftarrow Q$ in L and which gets renamed into $\alpha(Q')\theta$, unfold the clause corresponding to L wrt $\alpha(Q')\theta$ (i.e. the renamed version of C inside $\rho_{\alpha,p}(L)$) using the definition of $\alpha(Q')$ in P_f.

Note that P'_f can be obtained by a transformation sequence for a partial deduction P'' based on the same atomic renaming α and the same set $\mathcal{Q}$ as P' but based on SLD-trees with a deeper unfolding for the elements of $\mathcal{W}$.[7] Each element of $\mathcal{Q} \setminus \mathcal{W}$ is still in $\mathcal{WF}_k$ (as well as in $\mathcal{WF}_i, i < k$ if it was in $\mathcal{WF}_i$ for P_f) because the associated trees and resultants remain unchanged. We also know that every Q' above must be in $\mathcal{Q} \setminus \mathcal{W} = \mathcal{WF}_k$, because the second rule of Definition 10.3.9 could be applied to deduce

[6] Note that Lemmas 10.3.7 and 10.3.11 also ensure that Theorem 8 of [222] can be applied and thus, given a fixed first-order language $\mathcal{L}_P$, the least Herbrand model semantics Sem_H is also preserved (restricted to the predicates occurring in the original program P_0). We will not use this property in the remainder of this chapter however.

[7] Possibly a slightly adapted renaming function is needed to ensure that the renamings of the new leaves of these deeper SLD-trees coincide with the clause bodies obtained by the unfolding performed on P_f. See the discussion about admissible renamings in Section 5.2.2 of Chapter 5.

that $Q \in \mathcal{WF}_{k+1}$. Hence in P'' associated with P'_f, each element of $\mathcal{W}$ is now in $\mathcal{WF}_k$, due to the unfolding. Hence we can apply the induction hypothesis to deduce that finite failure is preserved in P'_f wrt P_0. Now because unfolding is totally correct wrt the finite failure semantics Sem_{FF}, we know that P_f and P'_f are equivalent under Sem_{FF}. Thus the induction hypothesis holds for $min = k + 1$. □

We are now in position to state a correctness result similar to the ones in Chapters 3 and 5. In contrast to these results however, we do not need an independence condition (because of the renaming), but we still need an adapted coveredness condition:

Definition 10.3.13($\mathcal{Q}$-covered wrt p) Let p be a partitioning function and $\mathcal{Q}$ a set of conjunctions. We say that a conjunction Q is *$\mathcal{Q}$-covered wrt p* iff every conjunction $Q' \in p(Q)$ is an instance of an element in $\mathcal{Q}$. Furthermore a set of resultants R is *$\mathcal{Q}$-covered wrt p* iff every body of every resultant in R is $\mathcal{Q}$-covered wrt p.

The above coveredness condition ensures that the renamings performed in Definition 10.2.6 are always defined and that the original program P can be thrown away from the end result of a transformation sequence for the associated conjunctive partial deduction.

Example 10.3.14 Let $\mathcal{Q} = \{q(x) \wedge r, q(a)\}$, $Q = q(a) \wedge q(b) \wedge r$. Then, for a partitioning function p such that $p(Q) = \{q(b) \wedge r, q(a)\}$, Q is $\mathcal{Q}$-covered wrt p. However, for p' with $p'(Q) = \{q(a) \wedge r, q(b)\}$, Q is not $\mathcal{Q}$-covered wrt p'.

Proposition 10.3.15 establishes a correspondence between the result P_f of the above transformation sequence and the corresponding conjunctive partial deduction.

Proposition 10.3.15 Let P' be the conjunctive partial deduction of P wrt $\mathcal{Q}$, $P_{\mathcal{Q}}$ and $\rho_{\alpha,p}$ such that $P_{\mathcal{Q}}$ is $\mathcal{Q}$-covered wrt p. Also let $P_0, \ldots, P_f$ be a transformation sequence for P'. Then $P_f \setminus P = P'$, where P is the original program.

Theorem 10.3.16 Let P' be the conjunctive partial deduction of P wrt $\mathcal{Q}$, $P_{\mathcal{Q}}$ and $\rho_{\alpha,p}$. If $P_{\mathcal{Q}} \cup \{G\}$ is $\mathcal{Q}$-covered wrt p then

- $P \cup \{G\}$ has an SLD-refutation with c.a.s. θ iff $P' \cup \{\rho_{\alpha,p}(G)\}$ has an SLD refutation with c.a.s. θ.

If in addition P' is weakly fair then

- $P \cup \{G\}$ has a finitely failed SLD-tree iff $P' \cup \{\rho_{\alpha,p}(G)\}$ has.

Proof Let us first prove the theorem for conjunctive partial deductions constructed using a fresh atomic renaming α.
Let $x_1, \ldots, x_n$ be the variables of G ordered according to their first appearance and let $query$ be a fresh predicate of arity n. We then define $P_0 = P \cup \{query(x_1, \ldots, x_n) \leftarrow Q_G\}$ where $G = \leftarrow Q_G$. The conjunctive partial deduction of P_0 will be identical to the one of P except for the extra clause for $query$. We can now construct a transformation sequence $P_0, \ldots, P_f$ for the conjunctive partial deduction of P_0 and then apply Proposition 10.3.12 to deduce that $query(x_1, \ldots, x_n)$ has the same computed answer and finite failure behaviour in P_0 and P_f. Note that $query$ is defined in P_f by the clause $query(x_1, \ldots, x_n) \leftarrow \rho_{\alpha,p}(Q_G)$. Hence $P \cup \{G\}$ has the same behaviour wrt computed answers and finite failure as $P_f \cup \{\rho_{\alpha,p}(G)\}$. Finally $P' = P_f \setminus \{P \cup query(x_1, \ldots, x_n) \leftarrow \rho_{\alpha,p}(Q)\}$. Hence, the theorem follows from the fact that, due to $\mathcal{Q}$-coveredness wrt p, the predicates defined in P, as well as the predicate $query$, are inaccessible from $\rho_{\alpha,p}(Q)$ in the predicate dependency graph.
Let us now prove the result for unrestricted renaming.
For that we simply introduce a fresh intermediate renaming and prove the result by two applications of the above theorem. More precisely, let α' and α'' be such that $\alpha(Q) = \alpha''(\alpha'(Q))$ for every $Q \in \mathcal{Q}$ and such that α' is a fresh atomic renaming for $P_{\mathcal{Q}}$ and also such that α'' is a fresh atomic renaming wrt the range of α'. Such renamings can always be constructed. We can now apply the above result to deduce that the conjunctive partial deduction P'', obtained from $P_{\mathcal{Q}}$ under $\rho_{\alpha',p}$, is totally correct for the query $\rho_{\alpha',p}(G)$. The conjunctive partial deduction P' (as well as the query $\rho_{\alpha,p}(G)$) can be obtained from P'' by performing a (standard) partial deduction wrt the set $\mathcal{A} = \{\alpha'(Q) \mid Q \in \mathcal{Q}\}$ and by unfolding every atom in $\mathcal{A}$ exactly once. Hence we can re-apply the above theorem and we obtain total correctness of P' wrt P. □

Let us briefly illustrate Theorem 10.3.16 on a small example.

Example 10.3.17 (double append, revisited) Let $P_{\mathcal{Q}}$, P and P' be taken from Example 10.2.7. Let $G = \leftarrow app([1,2],[3],I) \wedge app(I,[4],R)$. We have that $\rho_{\alpha,p}(G) = da([1,2],[3],I,[4],R)$. It can be seen that $P_{\mathcal{Q}} \cup \{G\}$ is $\mathcal{Q}$-covered wrt p and indeed, as predicted by Theorem 10.3.16, $P \cup \{G\}$ and $P_{\rho_{\alpha,p}} \cup \{\rho_{\alpha,p}(G)\}$ have the same set of computed answers: $\{\{I/[1,2,3], R/[1,2,3,4]\}\}$.
Note that P', as mentioned in Example 10.3.10 above, is weakly fair and therefore finite failure is also preserved.

10.4 Discussion and conclusion

10.4.1 Negation and normal programs

When extending conjunctive partial deduction for negation two issues come up: one is correctly handling normal logic programs instead of definite programs and the second one is to allow the selection of ground negative literals (i.e. moving from SLD$^+$-trees to SLDNF-trees).

The former is not so difficult, as a lot of results from the literature can be reused. For instance, we can recycle results from the unfold/fold literature to prove preservation of the perfect model semantics for stratified programs [249] and preservation of the well-founded semantics for normal programs [250]. If in addition we have fair SLD$^+$-trees, the conditions of modified (T&S) folding of [249] hold, and we can use preservation of the SLDNF success and finite failure set for stratified programs. Some further results from [13] can also be applied. In [31], the correctness results of [249] are adapted for Fitting's semantics and the results might also be applicable in our case. The results of [104] do not seem to be applicable because they use a different folding rule (which requires the clauses involved in the folding process to be all in the same program P_i).

Allowing SLDNF-trees instead of SLD$^+$-trees is more difficult. Note that [249, 250, 222, 13, 31] do not allow the unfolding of negative literals. Selecting negative literals might be obtained by *goal replacement* or *clause replacement*, but Theorems 15 and 16 from [222] cannot be applied because different folding rules are used. [246] allows unfolding inside negation and works with first order formulas, but it still has to be investigated whether its results (for Kleene's 3-valued logic) can be used. Also [144] has a negative unfolding rule (under certain termination conditions), but this rule (along with the correctness theorems) is situated in the context of deriving *definite* logic programs from first order specifications. So, for the moment, there seems to be no conjunctive equivalent to the correctness theorem of [185] for normal logic programs and partial deduction based on constructing finite SLDNF-trees. Further work will be needed to extend the correctness results of the previous section.

10.4.2 Preliminary results and potential

Some preliminary and promising experiments with conjunctive partial deduction have been conducted in [175] (in Chapter 12 we will present more elaborate and extensive benchmarks). They were based on a system for standard partial deduction described in [168, 178]. The adaptation to conjunctive partial deduction was pretty straightforward, further underlining

our earlier claim that conjunctive partial deduction is just a simple, but powerful, extension of the standard partial deduction concept.

The good results that were obtained in [175] (as well as the ones we will obtain later in Chapter 12) seem to indicate that folding combined with a rather simple unfolding rule can solve some of the traditional local control problems for meta-interpreters. This is also related to the fact that the local precision problem outlined in Chapter 3 disappears, at least to some extent: there is no longer an automatic, unavoidable abstraction associated with stopping the unfolding. In other words, given a good global control, the local control no longer has to worry about precision: it can concentrate on the efficiency of the resultants it generates.

Folding also solves a problem already raised in [220]. Take for example a meta-interpreter containing the following clauses, where $noloop(A, H)$ is an expensive test which cannot be (fully) unfolded:

$$\begin{array}{l} solve([], H) \leftarrow \\ solve([A|T], H) \leftarrow \\ \qquad noloop(A, H) \wedge clause(A, B) \wedge \\ \qquad solve(B, [A|H]) \wedge solve(T, H) \end{array}$$

Here standard partial deduction faces a dilemma when specialising the atom $solve([\bar{d}], H)$. After resolving with the second clause and performing one determinate unfolding step (on $solve([], H)$) we obtain the following goal

$$\leftarrow noloop(\bar{d}, H) \wedge clause(\bar{d}, B) \wedge solve(B, [\bar{d}|H])$$

Either we unfold $clause(\bar{d}, B)$, thereby propagating the partial input $\bar{d}$ over to B in $solve(B, [\bar{d}|H])$, but at the cost of duplicating $noloop(\bar{d}, H)$ and most probably leading to a slowdown. Or standard partial deduction can stop the unfolding, but then the partial input $\bar{d}$ can no longer be propagated over to B inside $solve(B, [\bar{d}|H])$. Using conjunctive partial deduction however, we can be efficient *and* propagate information at the same time, simply by stopping unfolding and specialising the conjunction $clause(\bar{d}, B) \wedge solve(B, [\bar{d}|H])$.[8]

10.4.3 Conclusion

In conclusion, we have presented a simple but powerful extension of partial deduction, showed that it can perform tupling and, when combined with a suitable post-processing to be presented in the next Chapter, also deforestation. We proved correctness results with respect to computed answer

[8] The Paddy system [227] uses non-atomic, non-recursive folding to avoid backpropagation of bindings (to preserve Prolog semantics). In general this is sufficient to solve the above problem. Apart from that, Paddy only performs folding of atoms, i.e. the same implicit folding used in *standard* partial deduction.

semantics, least Herbrand model semantics and finite failure semantics. For the latter we also presented improved conditions, which, in contrast to current results for unfold/fold, allow the preservation of finite failure while not imposing certain potentially disastrous (non-determinate) unfolding steps.

Chapter 11

Redundant Argument Filtering

11.1 Introduction

Automatically generated programs often contain redundant parts. For instance, programs produced by standard partial deduction [185] often have *useless clauses* and *redundant structures*, see e.g. [98]. This has motivated uses of *regular approximations* to detect useless clauses [101, 68, 66] and *renaming* (or *filtering*) transformations [99, 17] which remove redundant structures. We have already discussed the latter transformations in Chapter 3, where they also proved useful to ensure the independence condition, and we have then widely used such transformations, notably in Chapters 5 and 10.

In this chapter we are concerned with yet another notion of redundancy which may remain even after these transformations have been applied, viz. *redundant arguments*. These seem to appear particularly often in programs produced by *conjunctive partial deduction* as presented in Chapter 10.

Consider the program P' we obtained in Example 10.2.7 by conjunctive partial deduction of the *append* program for the goal $G = \leftarrow app(X, Y, I) \wedge app(I, Z, R)$ (renamed to $\leftarrow da(X, Y, I, Z, R)$ in P'):

$app([\,], L, L) \leftarrow$
$app([H|X], Y, [H|Z]) \leftarrow app(X, Y, Z)$
$da([\,], Y, Y, Z, R) \leftarrow app(Y, Z, R)$
$da([H|X'], Y, [H|I'], Z, [H|R']) \leftarrow da(X', Y, I', Z, R')$

Here the concatenation of the lists Xs and Ys is still stored in T, but is not used to compute the result in R. Instead the elements encountered

while traversing Xs and Ys are stored directly in R. Informally, the third argument of da is redundant. Thus, although this program represents a step in the right direction, we would rather prefer the following program:

$$app([\,], L, L) \leftarrow$$
$$app([H|X], Y, [H|Z]) \leftarrow app(X, Y, Z)$$
$$da'([\,], Y, Z, R) \leftarrow app(Y, Z, R)$$
$$da'([H|X'], Y, Z, [H|R']) \leftarrow da'(X', Y, Z, R')$$

The step from $da/5$ to $da'/4$ was left open in Chapter 10. The step cannot be obtained by the renaming operation in [99, 17] which only improves programs where some atom in some body contains functors or multiple occurrences of the same variable. In fact, this operation has *already* been employed by conjunctive partial deduction to arrive at the program with $da/5$. The step also cannot be obtained by other transformation techniques, such as partial deduction itself, or the more specific program construction of [191, 192] (cf. Chapter 13). Indeed, any method which preserves the least Herbrand model, or the computed answer semantics for all predicates, is incapable of transforming $da/5$ to $da'/4$. The point is that although the list I is redundant in some sense—which is made precise below—the change of arity also changes the semantics.

Redundant arguments also appear in a variety of other situations. For instance, they appear in programs generated by standard partial deduction when conservative unfolding rules are used.

As another example, redundant arguments arise when one re-uses generic predicates for more specific purposes. For instance, let us define a $member/2$ predicate by re-using a generic $delete/3$ predicate:

$$member(X, L) \leftarrow delete(X, L, DL)$$
$$delete(X, [X\,|\,T], T) \leftarrow$$
$$delete(X, [Y|\,T], [Y|DT]) \leftarrow delete(X, T, DT)$$

Here the third argument of *delete* is redundant but cannot be removed by any of the techniques cited above.

In this chapter we rigorously define the notion of a redundant argument, and show that the problem of removing all redundant arguments is undecidable. We then present an efficient algorithm which computes a safe approximation of the redundant arguments and removes them. Correctness of the technique is also established. On a range of programs produced by conjunctive partial deduction (with renaming), an implementation of our algorithm reduces code size and execution time by an average of approximately 20%. The algorithm never increases code size nor execution time.

11.2 Correct erasures

In the remainder, $Pred(P)$ denotes the set of predicates occurring in a logic program P, $arity(p)$ denotes the arity of a predicate p, $Clauses(P)$ denotes the set of clauses in P and $Def(p, P)$ denotes the definition of p in P. An atom, conjunction, goal, or program, in which every predicate has arity 0 is called *propositional*.

In this section we formalise *redundant arguments* in terms of *correct erasures*.

Definition 11.2.1 Let P be a program.

1. An *erasure* of P is a set of tuples (p, k) with $p \in Pred(P)$, and $1 \leq k \leq arity(p)$.
2. The *full erasure* for P is $\top_P = \{(p, k) \mid p \in Pred(P) \wedge 1 \leq k \leq arity(p)\}$.

The effect of applying an erasure to a program is to erase a number of arguments in every atom in the program. For simplicity of the presentation we assume that, for every program P and goal G of interest, each predicate symbol occurs only with one particular arity (this will later ensure that there are no unintended name clashes after erasing certain argument positions).

Definition 11.2.2 Let G be a goal, P a program, and E an erasure of P.

1. For an atom $A = p(t_1, \ldots, t_n)$ in P, let $1 \leq j_1 < \ldots < j_k \leq n$ be all the indexes such that $(p, j_i) \notin E$. We then define $A|E = p(t_{j_1}, \ldots, t_{j_k})$.
2. $P|E$ and $G|E$ arise by replacing every atom A by $A|E$ in P and G, respectively.

How are the semantics of P and $P|E$ of Definition 11.2.2 related? Since the predicates in P may have more arguments than the corresponding predicates in $P|E$, the two programs have incomparable semantics. Nevertheless, the two programs may have the same semantics for some of their arguments.

Example 11.2.3 Consider the definite program P:

$p(0, 0, 0) \leftarrow$
$p(s(X), f(Y), g(Z)) \leftarrow p(X, Y, Z)$

The goal $G = \leftarrow p(s(s(0)), B, C)$ has exactly one SLD-refutation, with computed answer $\{B/f(f(0)), C/g(g(0))\}$. Let $E = \{(p, 3)\}$, and hence $P|E$ be:

$p(0,0) \leftarrow$
$p(s(X), f(Y)) \leftarrow p(X,Y)$

Here $G|E = \leftarrow p(s(s(0)), B)$ has exactly one SLD-refutation, with computed answer $\{B/f(f(0))\}$. Thus, although we have erased the third argument of p, the computed answer for the variables in the remaining two arguments is not affected. Taking finite failures into account too, this suggests a notion of equivalence captured in the following definition.

Definition 11.2.4 An erasure E is *correct* for a definite program P and a definite goal G iff

1. $P \cup \{G\}$ has an SLD-refutation with computed answer θ with $\theta' = \theta \mid_{vars(G|E)}$ iff $P|E \cup \{G|E\}$ has an SLD-refutation with computed answer θ'.
2. $P \cup \{G\}$ has a finitely failed SLD-tree iff $P|E \cup \{G|E\}$ has.

Given a definite goal G and a definite program P, we may now say that the i'th argument of a predicate p is *redundant* if there is an erasure E which is correct for P and G and which contains (p,i). However, we will continue to use the terminology with correct erasures, rather than redundant arguments.

Usually there is a certain set of argument positions I which we do not want to erase. For instance, for $G = app([a],[b],R)$ and the append program, the erasure $E = \{(app,3)\}$ is correct, but applying the erasure will also make the result of the computation invisible. In other words, we wish to retain some arguments because we are interested in their values (see also the examples in Section 11.4). Therefore we only consider subsets of $\top_P \setminus I$ for some I. Not all erasures included in $\top_P \setminus I$ are of course correct, but among the correct ones we will prefer those that remove more arguments. This motivates the following definition.

Definition 11.2.5 Let G be a goal, P a program, $\mathcal{E}$ a set of erasures of P, and $E, E' \in \mathcal{E}$.

1. E is *better than* E' iff $E \supseteq E'$.
2. E is *strictly better than* E' iff E is better than E' and $E \neq E'$.
3. E is *maximal* iff no other $E' \in \mathcal{E}$ is strictly better than E.

Proposition 11.2.6 Let G be a definite goal, P a definite program and $\mathcal{E}$ a collection of erasures of P. Among the correct erasures for P and G in $\mathcal{E}$ there is a maximal one.

Proof There are only finitely many erasures in $\mathcal{E}$ that are correct for P and G. Just choose one which is not contained in any other. □

Maximal correct erasures are not always unique. For $G = \leftarrow p(1,2)$ and the following program P:

$p(3,4) \leftarrow q$

$q \leftarrow$

both $\{(p,1)\}$ and $\{(p,2)\}$ are maximal correct erasures, but $\{(p,1),(p,2)\}$ is incorrect.

The remainder of this section is devoted to proving that finding best correct erasures is undecidable. The idea is as follows. It is decidable whether $P \cup \{G\}$ has an SLD-refutation for propositional P and G, but not for general P and G. The full erasure of any P and G yields propositional $P|\top_P$ and $G|\top_P$. The erasure is correct iff both or none of $P \cup \{G\}$ and $P|\top_P \cup \{G|\top_P\}$ have an SLD-refutation. Thus a test to decide correctness, together with the test for SLD-refutability of propositional formulae, would give a general SLD-refutability test.

Lemma 11.2.7 There is an effective procedure that decides, for a propositional definite program P and definite goal G, whether $P \cup \{G\}$ has an SLD-refutation.

Proof By a well-known [184, Corollary7.2, Theorem 8.4] result, $P \cup \{G\}$ has an SLD-refutation iff $P \cup \{G\}$ is unsatisfiable. The latter problem is decidable, since P and G are propositional. □

Lemma 11.2.8 Let G be a definite goal, P a definite program, and E an erasure of P. If $P \cup \{G\}$ has an SLD-refutation, then so has $P|E \cup \{G|E\}$.

Proof By induction on the length of the SLD-derivation of $P \cup \{G\}$. □

Lemma 11.2.9 Let P be a definite program and G a definite goal. If $P \cup \{G\}$ has an SLD-refutation, then $P \cup \{G\}$ has no finitely failed SLD-tree.

Proof By [184, Theorem 10.3]. □

Proposition 11.2.10 There is no effective procedure that tests, for a definite program P and definite goal G, whether $\top_P$ is correct for P and G.

Proof Suppose such an effective procedure exists. Together with the effective procedure from Lemma 11.2.7 this would give an effective procedure

to decide whether $P \cup \{G\}$ has an SLD-refutation, which is known to be an undecidable problem:[1]

1. If $P|\top_P \cup \{G|\top_P\}$ has no SLD-refutation, by Lemma 11.2.8 neither has $P \cup \{G\}$.
2. If $P|\top_P \cup \{G|\top_P\}$ has an SLD-refutation then:
   (a) If $\top_P$ is correct then $P \cup \{G\}$ has an SLD-refutation by Definition 11.2.4.
   (b) If $\top_P$ is incorrect then $P \cup \{G\}$ has no SLD-refutation. Indeed, if $P \cup \{G\}$ had an SLD-refutation with computed answer θ, then Definition 11.2.4 (1) would be satisfied with $\theta' = \theta\,|_{vars(G|\top_P)} = \emptyset$. Moreover, by Lemma 11.2.9 none of $P \cup \{G\}$ and $P|\top_P \cup \{G|\top_P\}$ would have a finitely failed SLD-tree, so Definition 11.2.4(2) would also be satisfied. Thus $\top_P$ would be correct, a contradiction.

□

Corollary 11.2.11 There is no effective function that maps any definite program P and definite goal G to a maximal, correct erasure for P and G.

Proof $\top_P$ is the maximal among all erasures of P, so such a function f would satisfy:

$$f(P,G) = \top_P \quad \Leftrightarrow \quad \top_P \text{ is correct for } P \text{ and } G$$

giving an effective procedure to test correctness of $\top_P$, contradicting Proposition 11.2.10. □

11.3 Computing correct erasures

In this section we present an algorithm which computes correct erasures. Corollary 11.2.11 shows that we cannot hope for an algorithm that always computes *maximal* correct erasures. We therefore derive an approximate notion which captures some interesting cases. For this purpose, the following examples illustrate some aspects of correctness.

The first example shows what may happen if we try to erase a variable that occurs several times in the body of a clause.

[1] $G|\top_P$ may contain variables, namely those occurring in atoms with predicate symbols not occurring in P. However, such atoms are equivalent to propositional atoms not occurring in P.

Example 11.3.1 Consider the following program P:

$p(X) \leftarrow r(X,Y), q(Y)$
$r(X,1) \leftarrow$
$q(0) \leftarrow$

If $E = \{(r,2)\}$ then $P|E$ is the program:

$p(X) \leftarrow r(X), q(Y)$
$r(X) \leftarrow$
$q(0) \leftarrow$

In P the goal $G = \leftarrow p(X)$ fails finitely, while in $P|E$ the goal $G|E = \leftarrow p(X)$ succeeds. Thus E is not correct for P and G. The source of the problem is that the existential variable Y links the calls to r and q with each other. By erasing Y in $\leftarrow r(X,Y)$, we also erase the synchronisation between r and q. Also, if $E = \{(q,1),(r,2)\}$ then $P|E$ is the program:

$p(X) \leftarrow r(X), q$
$r(X) \leftarrow$
$q \leftarrow$

Again, $G|E = \leftarrow p(X)$ succeeds in $P|E$, so the problem arises independently of whether the occurrence of Y in $q(Y)$ is itself erased or not.

In a similar vein, erasing a variable that occurs several times within the same call, but is not linked to other atoms, can also be problematic.

Example 11.3.2 If P is the program:

$p(a,b) \leftarrow$
$p(f(X),g(X)) \leftarrow p(Y,Y)$

and $E = \{(p,2)\}$ then $P|E$ is the program:

$p(a) \leftarrow$
$p(f(X)) \leftarrow p(Y)$

Here $G = \leftarrow p(f(X),Z)$ fails finitely in P, while $G|E = \leftarrow p(f(X))$ succeeds (with the empty computed answer) in $P|E$.

Note that, for $E = \{(p,1),(p,2)\}$, $P|E$ is the program:

p
$p \leftarrow p$

Again $G|E = \leftarrow p$ succeeds in $P|E$ and the problem arises independently of whether the second occurrence of Y is erased or not.

Still another problem is illustrated in the next example.

Example 11.3.3 Consider the following program P:

$p([],[]) \leftarrow$
$p([X|Xs],[X|Ys]) \leftarrow p(Xs,[0|Ys])$

If $E = \{(p,2)\}$ then $P|E$ is the program:

$p([]) \leftarrow$
$p([X|Xs]) \leftarrow p(Xs)$

In P, the goal $G =\leftarrow p([1,1],Y)$ fails finitely, while in $P|E$ the goal $G|E =\leftarrow p([1,1])$ succeeds. This phenomenon can occur when erased arguments of predicate calls contain non-variable terms.

Finally, problems may arise when erasing in the body of a clause a variable which also occurs in a non-erased position of the head of a clause:

Example 11.3.4 Let P be the following program:

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

If $E = \{(p,2)\}$ then $P|E$ is the program:

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

Here $G =\leftarrow p(c,Y)$ fails (infinitely) in P while $G|E =\leftarrow p(c)$ succeeds in $P|E$. The synchronisation of the alternating arguments X and Y is lost by the erasure.

The above criteria lead to the following definition, in which (1) rules out Example 4, (2) rules out Examples 2 and 3, and (3) rules out Example 5.

Definition 11.3.5 Let P be a definite program and E an erasure of P. E is *safe* for P iff for all $(p,k) \in E$ and all $H \leftarrow C, p(t_1,\ldots,t_n), C' \in \mathit{Clauses}(P)$, it holds that:

1. t_k is a variable X.
2. X occurs only once in $C, p(t_1,\ldots,t_n), C'$.
3. X does not occur in $H|E$.

This in particular applies to goals:

Definition 11.3.6 Let P be a definite program and E an erasure of P. E is *safe* for a definite goal G iff for all $(p,k) \in E$ where $G =\leftarrow C, p(t_1,\ldots,t_n), C'$ it holds that:

1. t_k is a variable X.
2. X occurs only once in in $C, p(t_1,\ldots,t_n), C'$.

We will now show that the conditions in Definitions 11.3.5 and 11.3.6 are actually sufficient to ensure correctness. We already mentioned on page 222 in Chapter 10 that renaming is closely related to unfold/fold and that it can be formalised as a two-step basic transformation involving a definition step, immediately followed by a number of folding steps. This observation also applies to erasures, whose application can be seen as an, albeit powerful, renaming transformation. The conditions in Definitions 11.3.5 and 11.3.6 occur, in a less obvious formulation, within the formalisation of T&S-folding (see Definition 10.3.3). This will allow us to reuse correctness results from the unfold/fold literature in the proof below. Indeed, the method of this chapter can be seen as a novel application of T&S-folding using a particular control strategy.

Proposition 11.3.7 Let G be a definite goal, P a definite program, and E an erasure of P. If E is safe for P and for G then E is correct for P and G.

Proof We will show that $P|E$ can be obtained from P by a sequence of T&S-definition, unfolding and T&S-folding steps (see Section 10.3.1 in Chapter 10). Let $P_0 = P \cup query(\bar{X}) \leftarrow Q$ be the initial program of our transformation sequence, where $G = \leftarrow Q$ and $\bar{X}$ is the sequence of distinct variables occurring in $G|E$. First, for each predicate defined in P such that $A \neq A|E$, where $A = p(X_1, \ldots, X_n)$ is a maximally general atom, we introduce the definition $Def_p = A|E \leftarrow A$. The predicate of A occurs in P_0, and is therefore old according to the definitions in [269] (if one wants to use the definitions in [222] one can use exactly the same "trick" explained in the proof of Lemma 10.3.7). By the conditions we imposed earlier on P (namely that each predicate symbol occurs with only 1 arity, see the discussions just after Definition 11.2.1) we also know that the predicate of $P|E$ does not occur in P_0. Thus these definition steps are T&S-definition introduction steps. We now unfold every definition $A|E \leftarrow A$ wrt A using the clauses defining A in P_0, giving us the program P_k (where k is the number of definitions that have been unfolded). For every atom $p(t_1, \ldots, t_n)$ in the body of a clause C of P_k, for which a definition Def_p has been introduced earlier, we perform a folding step of C wrt $p(t_1, \ldots, t_n)$ using Def_p. Note that every such atom $p(t_1, \ldots, t_n)$ is fold-allowing (because either it has been obtained by unfolding a definition $A|E \leftarrow A$ and is not inherited from A or it stems from the original program). The result of the folding step is that of replacing $p(t_1, \ldots, t_n)$ by $p(t_1, \ldots, t_n)|E$. This means that after having performed all the resolution steps we obtain a program $P' = P|E \cup query(\bar{X}) \leftarrow Q|E \cup P''$ where P'' are the original definitions of those predicates for which we have introduce a definition Def_p. Now, as already

mentioned earlier, the conditions in Definition 11.3.5 and Definition 11.3.6 are equivalent to the conditions of T&S-folding and therefore P' can be obtained from P_k by a sequence of T&S-unfolding, unfolding and then T&S-folding steps on fold-allowable atoms. Note that here it is vital to define $\bar{X}$ to be the variables of $G|E$ in the clause $query(\bar{X}) \leftarrow Q$ of P_0, otherwise the folding steps performed on the atoms of Q would not be T&S-folding steps. We can thus apply Theorems 10 and 12 in [222] to deduce preservation of the computed answers and of finite failure. Finally, as P'' is unreachable from $\leftarrow query(\bar{X})$ we can remove P'' and because $G|E = \leftarrow Q|E$ the conditions of Definition 11.2.4 are verified for P and G. □

The following algorithm constructs a safe erasure for a given program.

Algorithm 11.3.8 (RAF)

Input: a definite program P, an initial erasure E_0.

Output: an erasure E with $E \subseteq E_0$.

Initialisation: $i := 0$;

while there exists a $(p,k) \in E_i$ and
a $H \leftarrow C, p(t_1, \ldots, t_n), C' \in Clauses(P)$ such that:

1. t_k is not a variable; **or**
2. t_k is a variable that occurs more than once in $C, p(t_1, \ldots, t_n), C'$; **or**
3. t_k is a variable that occurs in $H|E_i$ **do**

$E_{i+1} := E_i \setminus \{(p,k)\}$;

$i := i + 1$;

end while

return E_i

The above algorithm starts out from an initial erasure E_0, usually contained in $\top_P \setminus I$, where I are positions of interest (i.e. we are interested in the computed answers they yield). Furthermore E_0 should be so as to be safe for any goal of interest (see the example in the next section).

Proposition 11.3.9 With input E_0, RAF terminates, and output E is a unique erasure, which is the maximal safe erasure for P contained in E_0.

Proof The proof consists of four parts: *termination* of RAF, *safety* of E for P, *uniqueness* of E, and *optimality* of E. The two first parts are obvious; termination follows from the fact that each iteration of the while loop decreases the size of E_i, and safety is immediate from the definition.

To prove uniqueness, note that the non-determinism in the algorithm is the choice of which (p,k) to erase in the while loop. Given a logic program P, let the *reduction* $F \rightarrow_{(p,k)} G$ denote the fact that F is not safe for P and that an iteration of the while loop may chose to erase (p,k) from F yielding $G = F \setminus \{(p,k)\}$.
Now suppose $F \rightarrow_{(p,k)} G$ and $F \rightarrow_{(q,j)} H$, with $(p,k) \neq (q,j)$. Then by analysis of all the combinations of reasons that (p,k) and (q,j) could be removed from F it follows that $G \rightarrow_{(q,j)} I$ and $G \rightarrow_{(p,k)} I$ with $I = F \setminus \{(p,k),(q,j)\}$. In other words the reduction relation is locally confluent (see e.g. [82]). So, because the relation is also terminating, we can apply the diamond lemma [216] (also in [82]) to deduce that it is confluent. Hence the final output E is unique.
To see that E is the maximal one among the safe erasures contained in E_0, note that $F \rightarrow_{(p,k)} G$ implies that no safe erasure contained in F contains (p,k). □

11.4 Applications and benchmarks

We first illustrate the usefulness of the RAF algorithm in the transformation of double-append from Section 11.1. Recall that we want to retain the semantics (and so all the arguments) of *doubleapp*, but want to erase as many arguments in the auxiliary calls to *app* and *da* as possible. Therefore we start RAF with

$$E_0 = \{(da,1),(da,2),(da,3),(da,4),(da,5),(app,1),(app,2),(app,3)\}$$

Application of RAF to E_0 yields $E = \{(da,3)\}$, representing the information that the third argument of *da* can be safely removed, as desired. By construction of E_0, we have that $E \subseteq E_0$ is safe for any goal which is an instance of $\leftarrow doubleapp(Xs, Ys, Zs, R)$. Hence, as long as we consider only such goals, we get the same answers from the program with $da'/4$ as we get from the one with $da/5$.

Let us also treat the member-delete problem from Section 11.1. If we start RAF with

$$E_0 = \{(delete,1),(delete2),(delete,3)\}$$

indicating that we are only interested in computed answers to $member/2$, then we obtain $E = \{(delete,3)\}$ and the following more efficient program $P|E$:

$$\begin{array}{lcl} member(X,L) & \leftarrow & delete(X,L) \\ delete(X,[X|T]) & \leftarrow & \\ delete(X,[Y|T]) & \leftarrow & delete(X,T) \end{array}$$

To investigate the effects of Algorithm 11.3.8 more generally, we have incorporated it into the ECCE partial deduction system [170], already used in Chapter 6. The details about how the system was extended for conjunctive partial deduction are presented in the next Chapter.

We ran the system *with* and *without* redundant argument filtering (but always *with* renaming in the style of [99]) on a series of benchmarks of the DPPD library [170] (a brief description can also be found in Appendix C). An unfolding rule allowing determinate unfolding and leftmost "indexed" non-determinate unfolding (using the homeomorphic embedding relation on covering ancestors to ensure finiteness) was used.[2] As in Chapter 6, the timings were obtained via the *time*/2 predicate of Prolog by BIM 4.0.12 (on a Sparc Classic under Solaris) using the "benchmarker" files generated by ECCE. The compiled code size was obtained via *statistics*/4 and is expressed in units, were 1 unit corresponds to approximately 4.08 bytes (in the current implementation of Prolog by BIM). The total speedups were obtained by the same formula as in Section 6.4.

The results are summarised in Tables 11.2 and 11.1. As can be seen, RAF reduced code size by an average of 21% while at the same time yielding an average additional speedup of 18%. Note that 13 out of the 29 benchmarks benefited from RAF, while the others remained unaffected (i.e. no redundant arguments where detected). Also, none of the programs were deteriorated by RAF. Except for extremely large residual programs, the execution time of the RAF algorithm was insignificant compared to the total partial deduction time. Note that the RAF algorithm was also useful for examples which have nothing to do with deforestation and, when running the same benchmarks with standard partial deduction based on e.g. determinate unfolding, RAF also turned out to be useful, albeit to a lesser extent. In conclusion, RAF yields a practically significant reduction of code size and a practically significant speedup (e.g. reaching a factor of 4.29 for *depth*).

[2] The full system options were: Abs:j, InstCheck:a, Msv:s, NgSlv:g, Part:f, Prun:i, Sel:l, Whistle:d, Poly:y, Dpu: yes, Dce:yes, MsvPost: no.

Benchmark	Code Size	
	without RAF	with RAF
advisor	809 u	809 u
applast	188 u	145 u
contains.kmp	2326 u	1227 u
contains.lam	2326 u	1227 u
depth.lam	5307 u	1848 u
doubleapp	314 u	277 u
ex_depth	874 u	659 u
flip	573 u	493 u
grammar.lam	218 u	218 u
groundunify.simple	368 u	368 u
liftsolve.app	1179 u	1179 u
liftsolve.db1	1326 u	1326 u
liftsolve.lmkng	2773 u	2228 u
map.reduce	348 u	348 u
match.kmp	543 u	543 u
match.lam	543 u	543 u
maxlength	1083 u	1023 u
model_elim.app	444 u	444 u
regexp.r1	457 u	457 u
regexp.r2	831 u	799 u
regexp.r3	1229 u	1163 u
relative.lam	261 u	261 u
remove	2778 u	2339 u
rev_acc_type	242 u	242 u
rev_acc_type.inffail	1475 u	1475 u
rotateprune	4088 u	3454 u
ssuply.lam	262 u	262 u
transpose.lam	2312 u	2312 u
Average Size	1204.91 u	952.64 u (79.06%)

Table 11.1: Code size (in units)

Benchmark	Execution Time			Extra
	Original	without RAF	with RAF	Speedup
advisor	0.68	0.21	0.21	1.00
applast	0.44	0.17	0.10	1.70
contains.kmp	1.03	0.28	0.10	2.80
contains.lam	0.53	0.15	0.11	1.36
depth.lam	0.47	0.30	0.07	4.29
doubleapp	0.44	0.42	0.35	1.20
ex_depth	1.14	0.37	0.32	1.16
flip	0.61	0.66	0.58	1.14
grammar.lam	1.28	0.18	0.18	1.00
groundunify.simple	0.28	0.07	0.07	1.00
liftsolve.app	0.81	0.04	0.04	1.00
liftsolve.db1	1.00	0.01	0.01	1.00
liftsolve.lmkng	0.45	0.54	0.44	1.23
map.reduce	1.35	0.11	0.11	1.00
match.kmp	2.28	1.49	1.49	1.00
match.lam	1.60	0.95	0.95	1.00
maxlength	0.10	0.14	0.12	1.17
model_elim.app	1.43	0.19	0.19	1.00
regexp.r1	1.67	0.33	0.33	1.00
regexp.r2	0.51	0.25	0.18	1.39
regexp.r3	1.03	0.45	0.30	1.50
relative.lam	3.56	0.01	0.01	1.00
remove	4.66	3.83	3.44	1.11
rev_acc_type	3.39	3.39	3.39	1.00
rev_acc_type.inffail	3.39	0.96	0.96	1.00
rotateprune	5.84	6.07	5.82	1.04
ssuply.lam	0.65	0.05	0.05	1.00
transpose.lam	1.04	0.18	0.18	1.00
Total Speedup	1	2.11	2.50	1.18

Table 11.2: Execution times (in s)

11.5 Polyvariance and negation

In this section we discuss some natural extensions of our technique.

11.5.1 A polyvariant algorithm

The erasures computed by RAF are *monovariant*: an argument of some predicate has to be erased in all calls to the predicate or not at all. It is sometimes desirable that the technique be more precise and erase a certain argument only in certain contexts (this might be especially interesting when a predicate also occurs inside a negation, see the next subsection below).

Example 11.5.1 Consider the following program P:

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

For $E_0 = \{(p,2)\}$ (i.e. we are only interested in the first argument to p), RAF returns $E = \emptyset$ and hence $P|E = P$. The reason is that the variable Z in the call $p(X,Z)$ in the third clause of P cannot be erased. Therefore no optimisation can occur at all. To remedy this, we need a *polyvariant algorithm* which, in the process of computing a safe erasure, generates duplicate versions of some predicates, thereby allowing the erasure to behave differently on different calls to the same predicate. Such an algorithm might return the following erased program:

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

The rest of this subsection is devoted to the development of such a *polyvariant* RAF algorithm.

First, the following, slightly adapted, definition of erasing is needed. The reason is that several erasures might now be applied to the same predicate, and we have to avoid clashes between the different specialised versions for the same predicate.

Definition 11.5.2 Let E be an erasure of P. For an atom $A = p(t_1, \ldots, t_n)$, we define $A||E = p_E(t_{j_1}, \ldots, t_{j_k})$ where $1 \leq j_1 < \ldots < j_k \leq n$ are all the indexes such that $(p, j_i) \notin E$ and where p_E denotes a predicate symbol of arity j_k such that $\forall p, q, E_1, E_2$ ($p_{E_1} = q_{E_2}$ iff $(p = q \wedge E_1 = E_2)$).

For example, we might have that $p(X,Y)||\{(p,1)\} = p'(X)$ together with $p(X,Y)||\{(p,2)\} = p''(Y)$, thereby avoiding the name clash that occurs when using the old scheme of erasing.

Algorithm 11.5.3 (polyvariant RAF)

Input: a definite program P, an initial erasure E_p for some predicate p.

Output: a new program P' which can be called with $\leftarrow p(t_1,\ldots,t_n)||E_p$ and which is correct[3] if E_p is safe for $\leftarrow p(t_1,\ldots,t_n)$.

Initialisation: $New := \{(E_p,p)\}, S := \emptyset$, $P' = \emptyset$;

while not $New \subseteq S$ **do**

let $S := S \cup New$, $S' := New \setminus S$ and $New := \emptyset$

for every element (E_p,p) of S' **do**

for every clause $H \leftarrow A_1,\ldots,A_n \in Def(p,P)$ **do**

let $E_{A_i} = \{(q_i,k) \mid A_i = q_i(t_1,\ldots,t_m)$ and $1 \leq k \leq m$ and (q_i,k) satisfies

1. t_k is a variable X; **and**
2. X occurs exactly once in $A_1,\ldots,A_n$; **and**
3. X does not occur in $H||E_p$ $\}$

let $New := New \cup \{(E_{A_i},q_i) \mid 1 \leq i \leq n\}$

let $P' := P' \cup \{H||E_p \leftarrow A_1||E_{A_1},\ldots,A_n||E_{A_n}\}$

end for

end for

end while

return P'

Note that, in contrast to monovariant RAF, in the polyvariant RAF algorithm there is no operation that removes a tuple from the erasure E_p. So one may wonder how the polyvariant algorithm is able to produce a correct program. Indeed, if an erasure E_p contains the tuple (p,k) this means that this particular version of p will only be called with the k-th argument being an existential variable. So, it is always correct to erase the position k in the head of a clause C for that particular version of p, because no bindings for the body will be generated by the existential variable and because we are not interested in the computed answer bindings for that variable. However, the position k in a call to p somewhere else in the program, e.g. in the body of C, might not be existential. But in contrast to the monovariant RAF algorithm, we do not have to remove the tuple (p,k): we simply generate another version for p where the k-th argument is not existential.

[3] In the sense of Definition 11.2.4, by simply replacing $P|E$ by P' and $|$ by $||$.

Example 11.5.4 Let us trace Algorithm 11.5.3 by applying it to the program P of Example 11.5.1 above and with the initial erasure $E_p = \{(p, 2)\}$ for the predicate p. For this example we can suppose that p_{E_p} is the predicate symbol p with arity 1 and $p_\emptyset$ is simply p with arity 2.

1. After the first iteration we obtain $New = \{(\emptyset, p), (\{(p, 2)\}, p)\}$, as well as $S = \{(\{(p, 2)\}, p)\}$ and $P' =$

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

2. After the second iteration we have that $New = \{(\emptyset, p)\}$ as well as $S = \{(\{(p, 2)\}, p), (\emptyset, p)\}$, meaning that we have reached the fixpoint. Furthermore P' is now the desired program of Example 11.5.1 above, i.e. the following clauses have been added wrt the previous iteration:

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

The erasure E_p is safe for e.g. the goal $G = \leftarrow p(a, X)$, and the specialised program P' constructed for E_p is correct for $G||E_p = \leftarrow p(a)$ (in the sense of Definition 11.2.4, by simply replacing $P|E$ by P' and $|$ by $||$). For instance, $P \cup \{\leftarrow p(a, X)\}$ has the computed answer $\{X/b\}$ with $\theta' = \{X/b\}|_\emptyset = \emptyset$ and indeed $P' \cup \{\leftarrow p(a)\}$ has the computed answer $\emptyset$.

Termination of the algorithm follows from the fact that there are only finitely many erasures for every predicate. The result of Algorithm 11.5.3 is identical to the result of Algorithm 11.3.8 applied to a suitably duplicated and renamed version of the original program. Hence correctness follows from correctness of Algorithm 11.3.8 and of the duplication/renaming phase.

11.5.2 Handling normal programs

When treating *normal* logic programs an extra problem arises: erasing an argument in a negative goal might modify the floundering behaviour wrt SLDNF. In fact, the conditions of safety of Definition 11.3.5 or Definition 11.3.6 would ensure that the negative call will always flounder! So it does not make sense to remove arguments to negative calls (under the conditions of Definition 11.3.5, Definition 11.3.6) and in general it would even be incorrect to do so. Take for example the goal $\leftarrow ni$ and program P:

$int(0) \leftarrow$
$int(s(X)) \leftarrow int(X)$
$ni \leftarrow \neg int(Z)$
$p(a) \leftarrow$

By simply ignoring the negation and applying the RAF Algorithm 11.3.8 for $E_0 = \{(int, 1)\}$ we obtain $E = E_0$ and the following program $P|E$ which behaves incorrectly for the query $G =\leftarrow ni$ (i.e. $G|E$ fails and thereby falsely asserts that everything is an integer)[4]:

$int \leftarrow$
$int \leftarrow int$
$ni \leftarrow \neg int$
$p(a) \leftarrow$

This problem can be solved by adopting the pragmatic but safe approach of keeping all argument positions for predicates occurring inside negative literals. Hence, for the program P above, we would obtain the correct erasure $E = \emptyset$. This technique was actually used for the benchmark programs with negation of the previous section.

11.6 Reverse filtering (FAR)

In some cases, the conditions of Definition 11.3.6 can be relaxed. For instance, the erasure $\{(p, 1), (q, 1)\}$ is safe for the goal $p(X)$ and program:

$p(X) \leftarrow q(f(X))$
$q(Z) \leftarrow$

The reason is that, although the erased argument of $q(f(X))$ is a non-variable, the value is never used. So, whereas the RAF Algorithm 11.3.8 detects existential arguments (which might return a computed answer binding), the above is an argument which is non-existential and non-ground but whose value is never used (and for which no computed answer binding will be returned).

11.6.1 The FAR algorithm

Those kind of arguments can be detected by another post-processing phase, executing in a similar fashion as RAF, but using reversed conditions. The

[4] For instance, in the programming language Gödel, the query $\leftarrow ni$ flounders in P while $\leftarrow ni|E =\leftarrow ni$ fails in $P|E$. Note however that in Prolog, with its unsound negation, the query $\leftarrow ni$ fails both in P and $P|E$. So this approach to erasing inside negation is actually sound wrt unsound Prolog. Furthermore, in Mercury, the clause defining ni actually stands for $ni \leftarrow \neg\exists Z(int(Z))$. The approach is thus also sound for Mercury (thanks to Filip Ghyselen for pointing this out).

algorithm is presented in the following.

Algorithm 11.6.1 (FAR)

Input: a definite program P.

Output: a correct erasure E for P (and any G).

Initialisation: $i := 0$; $E_0 = \top_P$;

while there exists a $(p,k) \in E_i$ and a $p(t_1,\ldots,t_n) \leftarrow B \in Clauses(P)$ such that

1. t_k is not a variable; **or**
2. t_k is a variable that occurs more than once in $p(t_1,\ldots,t_n)$; **or**
3. t_k is a variable that occurs in $B|E_i$ **do**

$E_{i+1} := E_i \setminus \{(p,k)\}$;

$i := i + 1$;

end while

return E_i

The justifications for the points 1–3 in the FAR algorithm are as follows:

1. If t_k is a non-variable term this means that the value of the argument will be unified with t_k. This might lead to failure or to a computed answer binding being returned. So the value of the argument is used after all and might even be instantiated.

2. If t_k is repeated variable in the head of a clause it will be unified with another argument leading to the same problems as in point 1.

3. If t_k is a variable which occurs in non-erased argument in the body of a clause then it is passed as an argument to another call in which the value might be used after all and even be instantiated.

These conditions guarantee that an erased argument is never inspected or instantiated and is only passed as argument to other calls in positions in which it is neither inspected nor instantiated.

Note that this algorithm looks very similar to the RAF Algorithm 11.3.8, except that the roles of the head and body of the clauses have been reversed. This has as consequence that, while RAF detects the arguments which are existential (and in a sense propagates unsafe erasures top-down, i.e. from the head to the body of a clause), FAR detects arguments which are never used (and propagates unsafe erasures bottom-up, i.e. from the body to the head of a clause). Also, because the erasures calculated by this algorithm do *not* change the computed answers, we can safely start the algorithm

with the complete erasure $E_0 = \top_P$. It can again be seen that the outcome of the algorithm is unique.

Also note that the two algorithms RAF and FAR cannot be put into one algorithm in a straightforward way, because erasures have different meanings in the two algorithms. We can however get an optimal (monovariant) result by running sequences of FAR and RAF alternately — until a fix-point is reached (this process is well-founded as only finitely many additional argument positions can be erased). Unfortunately, as the following examples show, one application each of RAF and FAR is not sufficient to get the optimal result.

Example 11.6.2 Let P be the following program:

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

Applying FAR does not give any improvement because of the multiple occurrence of the variable X in the head of the second clause. After RAF we obtain:

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

Now applying FAR we get the optimally erased program:

$p \leftarrow q$
$q \leftarrow$

So in this example the FAR algorithm benefited from erasure performed by the RAF algorithm. The following example shows that the converse can also hold.

Example 11.6.3 Take the following program:

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

Applying RAF does not give any improvement because of the multiple occurrence of the variable X (but this time inside a call and not as in the Example 11.6.2 above inside the head). However, applying FAR gives the following:

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

And now RAF can give an improvement, leading to the optimal program:

$p \leftarrow q$
$q \leftarrow$

The reason that each of the algorithms can improve the result of the other is that RAF cannot erase multiply occurring variables in the body while FAR cannot erase multiply occurring variables in the head. So, one can easily extend Examples 11.6.2 and 11.6.3 so that a sequence of applications of RAF and FAR is required for the optimal result. We have not yet examined whether the RAF and FAR algorithm can be combined in a more refined way, e.g. obtaining the optimal program in one pass and maybe also weakening the respective safety conditions by using information provided by the other algorithm.

11.6.2 Polyvariance for FAR

The RAF algorithm looks at *every call* to a predicate p to decide which arguments can be erased. Therefore, the polyvariant extension was based on producing specialised (but still safe) erasures for *every distinct use* of the predicate p. The FAR algorithm however looks at *every head of a clause* defining p to decide which arguments can be erased. This means that an argument might be erasable wrt one clause while not wrt another. We clearly cannot come up with a polyvariant extension of FAR by generating different erasures for every clause. But one could imagine detecting for every call the clauses that match this call and then derive different erased versions of the same predicate. In the context of optimising residual programs produced by (conjunctive) partial deduction this does not seem to be very interesting. Indeed, every call will usually match every clause of the specialised predicate (especially for partial deduction methods which preserve characteristic trees like the ones presented in Chapters 5 and 6).

11.6.3 Negation and FAR

In contrast to RAF, the erasures obtained by FAR *can* be applied inside negative calls. The conditions of the algorithm ensure that any erased variable never returns any interesting[5] computed binding. Therefore removing such arguments, in other words allowing the selection of negative goals even for the case that this argument is non-ground, is correct wrt the completion semantics by correctness of SLDNFE (see Section 2.3.2).

Take for example the following program P:

$r(X) \leftarrow \neg p(X)$
$p(X) \leftarrow q(f(X))$
$q(Z) \leftarrow$

[5] An erased variable V might only return bindings of the form V/F where F is a fresh existential variable.

By ignoring the negation and applying the FAR algorithm, we get the erasure $E = \{(q,1),(p,1),(r,1)\}$ and thus $P|E$:

$r \leftarrow \neg p$
$p \leftarrow q$
$q \leftarrow$

Using $P|E \cup \{G|E\}$ instead of $P \cup \{G\}$ is correct. In addition $P|E \cup \{G|E\}$ will never flounder when using standard SLDNF, while P will flounder for any query $G =\leftarrow r(t)$ for which t is not ground. In other words, the FAR algorithm not only improves the efficiency of a program, but also its "floundering behaviour" under standard SLDNF.

11.6.4 Implementation of FAR

The FAR algorithm has also been implemented (by slightly re-writing the RAF algorithm) and incorporated into the ECCE system [170]. Preliminary experiments indicate that, when executed once after RAF, it is able to remove redundant arguments much less often than RAF, although in some cases it can be highly beneficial (e.g. bringing execution time of the final specialised program from 6.3 s down to 4.1 s for the memo-solve example of the DPPD library [170]). Also, it seems that an optimisation similar to FAR has recently been added to the Mercury compiler, where it is e.g. useful to get rid of arguments carrying unused type information.

11.7 Related work and conclusion

Our algorithm RAF for removal of redundant arguments, together with conjunctive partial deduction of Chapter 10, is related to Proietti and Pettorossi's *Elimination Procedure* (EP) for removal of *unnecessary variables*. Both can be used to perform tupling and deforestation and the proofs in this thesis show that conjunctive partial deduction and RAF can (partially) be mapped to sequences of Tamaki-Sato definition steps and unfolding steps followed by Tamaki-Sato folding steps. There are however some subtle differences between control in conjunctive partial deduction and control in the unfold/fold approach. Indeed, an unfold/fold transformation is usually described as doing the definition steps first (and at that point one knows which arguments are existential because existentiality can be propagated top-down — but one does not yet have the whole specialised program available) while conjunctive partial deduction can be seen as doing the definition introduction and the corresponding folding steps only at the very end (when producing the residual code). Therefore the use of an algorithm like RAF

is required for conjunctive partial deduction to detect the existential variables for the respective definitions. But on the other hand this also gives conjunctive partial deduction the possibility to base its choice on the *entire* residual program. For instance, one may use a monovariant algorithm (to limit the code size) or an algorithm like FAR which, due to its bottom-up nature, has to examine the entire residual program.

Another related work is [77], which provides some pragmatics for removing unnecessary variables in the context of optimising binarised Prolog programs.

Yet another related work is that on *slicing* [247], useful in the context of debugging. RAF can also be used to perform a simple form of program slicing; for instance, one can use RAF to find the sub-part of a program which affects a certain argument. However, the slice so obtained is usually less precise than the one obtained by the specific slicing algorithm of [247] which takes Prolog's left-to-right execution strategy into account and performs a data-dependency analysis.

Similar work has also been considered in other settings than logic programming. Conventional compiler optimisations use data-flow analyses to detect and remove *dead code*, i.e. commands that can never be reached and assignments to variables whose values are not subsequently required, see [1]. These two forms of redundancy are similar to useless clauses and redundant variables.

Such techniques have also appeared in functional programming. For instance, Chin [51] describes a technique to remove *useless variables*, using an abstract interpretation (forwards analysis). A concrete program is translated into an abstract one working on a two-point domain. The least fix-point of the abstract program is computed, and from this an approximation of the set of useless variables can be derived.

Hughes [130] describes a backwards analysis for *strictness analysis*. Such analyses give for each parameter of a function the information either that the parameter *perhaps is not* used, or that the parameter *definitely is* used. The analysis in [130] can in addition give the information that a parameter *definitely is not* used, in which case it can be erased from the program.

Another technique can be based on Seidl's work [248]. He shows that the corresponding question for higher-level grammars, *parameter-reducedness*, is decidable. The idea then is to approximate a functional program by means of a higher-level grammar, and decide parameter-reducedness on the grammar.

Most work on program slicing has been done in the context of imperative programs [272]. Reps [239] describes program slicing for functional programs as a backwards transformation.

Compared to all these techniques our algorithm is strikingly simple,

very efficient, and easy to prove correct. The obvious drawback of our technique is that it is less precise. Nevertheless, the benchmarks show that our algorithm performs well on a range of mechanically generated programs, indicating a good trade-off between complexity and precision.

Chapter 12

Conjunctive Partial Deduction in Practice

12.1 Controlling conjunctive partial deduction for pure Prolog

In conjunctive partial deduction, a conjunction can be abstracted by either splitting it into subconjunctions, or generalising syntactic structure, or through a combination of both. In classical partial deduction, on the other hand, any conjunction is always split (i.e. abstracted) into its constituent atoms before lifting the latter to the global level.

Apart from this aspect, the conventional control notions described in Chapters 3–6 also apply in a conjunctive setting. Notably, the concept of characteristic atoms can be generalised to *characteristic conjunctions*, which are just pairs consisting of a conjunction and an associated characteristic tree.

We will for the remainder of the chapter only be concerned with conjunctive partial deduction for pure Prolog. This means, besides disallowing non-pure features (see also Section 2.3.3), that we will restrict ourselves to LD-derivations, i.e. using the static (unfair) left-to-right selection rule (see Section 2.3.1). We will also demand preservation of termination under that selection rule.

12.1.1 Splitting and abstraction

A termination problem specific to conjunctive partial deduction lies in the possible appearance of ever growing conjunctions at the global level. To

cope with this, abstraction in the context of conjunctive partial deduction must include the ability to *split* a conjunction into several parts, thus producing *subconjunctions* of the original one. A method to deal with this problem has been developed in [110, 62]. and we will present the essentials below. See also e.g. [232] for a related generalisation operation in the context of an unfold/fold transformation technique.

First, to detect potential non-termination, the homeomorphic embedding, defined for expressions in Chapter 6, can be extended to conjunctions [110, 62]. For this the following notations will prove useful. Given a conjunction $Q = A_1 \wedge \ldots \wedge A_k$ any conjunction $Q' = A_{i_1} \wedge \ldots \wedge A_{i_j}$ such that $1 \leq i_1 < \ldots < i_j \leq k$ is called an *ordered subconjunction* of Q. If in addition $i_{l+1} = i_l + 1$ for $1 \leq l < j$ then Q' is called a *contiguous ordered subconjunction* of Q.

Definition 12.1.1 (homeomorphic embedding) Let $Q = A_1 \wedge \ldots \wedge A_n$ and Q' be conjunctions. We say that Q is *embedded* in Q', denoted by $Q \trianglelefteq^* Q'$, iff $Q' \not\trianglelefteq Q$ and there exists an ordered subconjunction $A'_1 \wedge \ldots \wedge A'_n$ of Q' such that $A_i \trianglelefteq^* A'_i$ for all $i \in \{1, \ldots, n\}$.

Proposition 12.1.2 The relation $\trianglelefteq^*$ is a well-quasi order for the set of conjunctions.

Proof Without the "strictly more general part" ($Q' \not\trianglelefteq Q$) this can be proven using Higman-Kruskal's theorem ([121, 155], see also [82]) by taking the embedding extension ([82], see also Chapter 6) of $\trianglelefteq^*$ for atoms (which is a wqo according to Theorem 6.2.23) and by considering $\wedge$ as a functor of variable arity (i.e. an associative operator). This embedding extension is then identical to $\trianglelefteq^*$ for conjunctions, except for the test "$Q' \not\trianglelefteq Q$". The fact that with the test "$Q' \not\trianglelefteq Q$" $\trianglelefteq^*$ is still a wqo can be proven exactly like in Theorem 6.2.23, i.e. by using Lemma 6.2.24 to construct a wqo from the well-founded order $\prec$ and then integrate it using Lemma 6.2.26. □

Example 12.1.3 Let us present a simple example. Consider the two conjunctions Q_1 and Q_2:

$$Q_1 = p(X, Y) \wedge q(Y, Z)$$
$$Q_2 = p(f(X), Y) \wedge r(Z, R) \wedge q(Y, Z)$$

If specialisation of Q_1 leads to specialisation of Q_2, there is a danger of non-termination as we have $Q_1 \trianglelefteq^* Q_2$ (because $p(X, Y) \trianglelefteq^* p(f(X), Y)$, $q(Y, Z) \trianglelefteq^* q(Y, Z)$ and Q_2 is not strictly more general than Q_1).

Once we have detected such potential non-termination we have to avoid

specialising Q_2 directly and abstract it first.[1] For this a generalisation of syntactic structure based on the *msg*, as used in the context of "classical" partial deduction, is not sufficient. Indeed, for two conjunctions $Q = A_1 \wedge \ldots \wedge A_n$ and $Q' = A'_1 \wedge \ldots \wedge A'_n$ a most specific generalisation $msg(Q, Q')$ exists only (which is then unique modulo variable renaming) if A_i and A'_i have the same predicate symbols for all i. So, when Q_1 and Q_2 differ in the number of atoms or in their respective predicate symbols the *msg* cannot be taken. This is for instance the case in Example 12.1.3 and one then has to split Q_2 up into subconjunctions (possibly combined with a generalisation of syntactic structure).

These observations lead to the following definition of abstraction in the context of conjunctive partial deduction.

Definition 12.1.4 (abstraction) A multiset of conjunctions $\{Q_1, \ldots, Q_k\}$ is an *abstraction* of a conjunction Q iff for some substitutions $\theta_1, \ldots, \theta_k$ we have that $Q =_r Q_1\theta_1 \wedge \ldots \wedge Q_k\theta_k$. An *abstraction operator* is an operator which maps every conjunction to an abstraction of it.

A particular abstraction operator

One constituent of the abstraction operation of [110] is to always split a conjunction into *maximal connected subconjunctions* first. This notion is closely related to those of "variable-chained sequence" and "block" in [231, 232], and the definition is as follows. Recall that we assume associativity of $\wedge$ and that $=_r$ denotes identity up to reordering (cf. Section 10.2.2).

Definition 12.1.5 (maximal connected subconjunctions) Given a conjunction $Q = A_1 \wedge \ldots \wedge A_n$, where $A_1, \ldots, A_n$ are literals, we define the binary relation $\downarrow_Q$ over the literals in Q as follows: $A_i \downarrow_Q A_j$ iff $vars(A_i) \cap vars(A_j) \neq \emptyset$. By $\Downarrow_Q$ we denote the reflexive and transitive closure of $\downarrow_Q$. The *maximal connected subconjunctions* of Q, denoted by $mcs(Q)$, are defined to be the multiset of conjunctions $\{Q_1, \ldots, Q_m\}$ such that

1. $Q =_r Q_1 \wedge \ldots \wedge Q_m$,
2. $A_i \Downarrow_Q A_j$ iff A_i and A_j occur in the same Q_k and
3. for every Q_k there exists a sequence of indices $j_1 < j_2 < \ldots < j_l$ such that $Q_k = A_{j_1} \wedge \ldots \wedge A_{j_l}$.

[1] One could also try to generalise Q_1 and then restart the specialisation with this more general conjunction. Note however that this is not always possible, e.g. when Q_1 is already maximally general (take for example $Q_1 = p(X)$ and $Q_2 = p(Z) \wedge p(Z)$). The ECCE system which we will use later for the benchmarks actually has a setting in which it will try to generalise Q_1 first and only if this is not possible generalise Q_2.

Note that each Q_i is also sometimes called a block [232]. Intuitively, no precision (in the sense of e.g. detecting failed branches) is lost by splitting into maximal connected subconjunctions and termination of the specialisation process is improved. However, termination is not ensured. For instance, in Example 12.1.3 we have that $mcs(Q_2) = p(f(X), Y) \wedge r(Z, R) \wedge q(Y, Z) = Q_2$ and the potential non-termination remains. So, we still have to solve the following problem: how to split up a conjunction Q_2, which embeds an earlier conjunction Q_1, in a sensible manner. This can be achieved as follows (taken from [110, 62] — we will show that this method actually ensures termination of the specialisation process later in Subsection 12.2.1).

Definition 12.1.6 Let Q be a conjunction and $\mathcal{Q}$ be a set of conjunctions. A *best matching conjunction for* Q *in* $\mathcal{Q}$ is a minimally general element of the set $\{ msg(Q, Q') \mid Q' \in \mathcal{Q} \text{ and } msg(Q, Q') \text{ exists } \}$.

By $bmc(Q, \mathcal{Q})$ we denote one particular best matching conjunction for Q in $\mathcal{Q}$. It might for example be chosen as follows. Consider graphs representing conjunctions where nodes represent occurrences of variables and there is an edge between two nodes iff they refer to occurrences of the same variable. A best match is then a Q' with a maximal number of edges in the graph for $msg(Q, Q')$.

Definition 12.1.7 (splitting) Let $Q = A_1 \wedge \ldots \wedge A_n$, Q' be conjunctions such that $Q \trianglelefteq^* Q'$. Let $\mathcal{Q}$ be the set of all ordered subconjunctions Q'' of Q' consisting of n atoms such that $Q \trianglelefteq^* Q''$. Then $split_Q(Q')$ is the pair (B, R) where $B = bmc(Q, \mathcal{Q})$ and R is the ordered subconjunction of Q' such that $Q' =_r B \wedge R$.

So, for Example 12.1.3 the potential non-termination is resolved by first splitting Q_2 into $Q = p(f(X), Y) \wedge q(Y, Z)$ and $r(Z, R)$ and subsequently taking the *msg* of Q_1 and Q. As a result, only $r(Z, R)$ will be considered for further specialisation (because $msg(Q_1, Q)$ is a variant of Q_1).

Note that $Q \trianglelefteq^* Q'$ in the above definition ensures that $\mathcal{Q}$ is not empty and $split_Q(Q')$ is thus properly defined.

Preservation of termination

Now, given a left-to-right computation rule, the above operation alters the sequence in which goals are executed. Indeed, the p- and q-subgoals will henceforth be treated jointly and will probably be renamed to a single atom. Consequently, there is no way an r-call can be interposed. A similar thing can happen just by splitting a conjunction into maximal connected subconjunctions.

From a purely declarative point of view, there is of course no reason why goals should not be interchanged, but under a fixed unfair computation rule, such *non-contiguous* splitting can worsen program performance (similar to non-leftmost, non-determinate unfolding, cf. Example 3.3.3), and even destroy termination. Consider for instance the following program:

```
flipallint(XT,TT) :- flip(XT,TT),allint(TT).
flip(leaf(X),leaf(X)).
flip(tree(XT,Info,YT),tree(FYT,Info,FXT)) :-
             flip(XT,FXT), flip(YT,FYT).
allint(leaf(X)) :- int(X).
allint(tree(L,Info,R)) :- int(Info), allint(L), allint(R).
int(0).
int(s(X)) :- int(X).
```

The deforested version, obtained by conjunctive partial deduction using the control of [110], would be:

```
flipallint(leaf(X),leaf(X)) :- int(X).
flipallint(tree(XT,Info,YT),tree(FYT,Info,FXT)) :-
        int(Info), flipallint(XT,FXT), flipallint(YT,FYT).
```

where the transformed version of `int` is unchanged. Under a left-to-right computation rule, the query

```
?-flipallint(tree(leaf(Z),0,leaf(a)),Res).
```

terminates with the original, but not with the deforested program.

In fact, the latter point has already been addressed in the context of unfold/fold transformations (see e.g. [32, 27, 30, 28]). To the best of our knowledge, however, no satisfactory solutions, suitable to be incorporated in a fully automatic system, have yet been proposed.

12.1.2 Contiguous splitting

For this reason, in the benchmarks below, we have in all but two cases *limited splitting to be contiguous*, that is, we split into contiguous subconjunctions only. (This can be compared with the outruling of goal switching in [27].)

As a consequence, compared to the basic method in [110] based on Definition 12.1.7, on the one hand, some opportunities for fruitful program transformation are left unexploited, but, on the other hand, Prolog programs are significantly less prone to actual deterioration rather than optimisation.

First, instead of systematically splitting a conjunction into maximal contiguous subconjunctions (*mcs*) we ensure contiguous splitting by splitting into contiguous connected subconjunctions (*ccs*), formalised in Definition 12.1.8 below. In addition, we replace the use of "ordered subconjunction" in Definition 12.1.7 by "contiguous ordered subconjunction", resulting in Definition 12.1.9 further below.

Definition 12.1.8 (contiguous connected subconjunctions) For a conjunction $Q = A_1 \wedge \ldots \wedge A_n$, a sequence of *contiguous connected subconjunctions* of Q is a sequence $\langle Q_1, \ldots, Q_m \rangle$ of conjunctions such that:

1. $Q = Q_1 \wedge \ldots \wedge Q_m$ and
2. $mcs(Q_i) = \{Q_i\}$

The conjunctions $\langle p(X) \wedge p(Y) \wedge q(X,Y), r(Z,T), p(Y) \rangle$ are (maximal) contiguous connected subconjunctions of $p(X) \wedge p(Y) \wedge q(X,Y) \wedge r(Z,T) \wedge p(Y)$. In the particular implementation, contiguous connected subconjunctions are obtained by scanning B from left-to-right and splitting when a literal does not share variables with the current block to the left. Other definitions of contiguous subconjunctions could disallow built-ins and/or negative literals in the subconjunctions or allow unconnected atoms inside the subconjunctions, e.g. like $p(S)$ in $p(X) \wedge p(S) \wedge q(X,Y)$.

Definition 12.1.9 (contiguous splitting) Let $Q = A_1 \wedge \ldots \wedge A_n$, Q' be conjunctions such that $Q \trianglelefteq^* Q'$. Let $\mathcal{Q}$ be the set of all contiguous ordered subconjunctions Q'' of Q' consisting of n atoms such that $Q \trianglelefteq^* Q''$. If $\mathcal{Q} \neq \emptyset$ then $csplit_Q(Q')$ is the triple (L, B, R) where $B = bmc(Q, \mathcal{Q})$ and L and R are the (possibly empty) contiguous ordered subconjunctions of Q' such that $Q' = L \wedge B \wedge R$. If $\mathcal{Q} = \emptyset$ then $csplit_Q(Q')$ is undefined.

Note that, in contrast to Definition 12.1.7, $Q \trianglelefteq^* Q'$ no longer ensures that $\mathcal{Q} \neq \emptyset$. For example, for $Q = p \wedge q$, $Q' = p \wedge r \wedge q$ we have $Q \trianglelefteq^* Q'$ but $csplit_Q(Q')$ is undefined (even though $split_Q(Q') = (p \wedge q, r)$ is defined). In the implementation to be used for the experiments, we simply split off the first atom of Q' if $csplit_Q(Q')$ is undefined. This simple-minded approach worked reasonably well, but there is definitely room for improvement.

12.1.3 Static conjunctions

Actually, the global control regime used in some of our experiments deviates from the one described by [110] in one further aspect. Even though abstraction by splitting ensures that the length of conjunctions (the number of its atoms) remains finite, there are realistic examples where the length

gets very large. This, combined with the use of homeomorphic embeddings (or lexicographical orderings for that matter), leads to very large global trees, large residual programs and a bad transformation time complexity.

Take for example global trees just containing atomic goals with predicates of arity k and having as argument just ground terms $s(s(...s(0)...))$ representing the natural numbers up to a limit n. Then we can construct branches in the global tree having as length $l = (n+1)^k$. Indeed for $n = 1, k = 2$ we can construct a branch of length $2^2 = 4$: $\langle p(s(0),s(0)), p(s(0),0), p(0,s(0)), p(0,0)\rangle$ while respecting homeomorphic embedding or lexicographical ordering.[2]

When going to conjunctive partial deduction the number of argument positions k is no longer bounded, meaning that, even when the terms are restricted to some natural depth, the size of the global tree can be arbitrarily large. Such a kind of explosion can actually occur for realistic examples, notably for meta-interpreters manipulating the ground representation and specialised for partially known queries (see the benchmarks, e.g. `groundunify.complex` and `liftsolve.db2`).

One way to ensure that this does not happen is to limit the conjunctions that may occur at the global level. For this we have introduced the notion of *static conjunctions*. A static conjunction is any conjunction that can be obtained by *non-recursive* unfolding of the goal to be partially evaluated (or a generalisation thereof). The idea is then, by a static analysis, to compute a set of static conjunctions $\mathcal{S}$ from the program and the goal. During partial deduction we then only allow conjunctions at the global level that are abstracted by one of the elements of $\mathcal{S}$. This is ensured by possibly further splitting of the disallowed conjunctions. (A related technique is used in [232].) In our implementation, we use a simple-minded method of approximating the set of static conjunctions, based on counting the maximum number of occurrences of each predicate symbol in a conjunction in the program or in the goal to be partially deduced, and disallowing conjunctions surpassing these numbers. In the example above, the maximum for `flip` and `allint` is 2, while for the other predicates it is 1.

Another approach, investigated in the experiments, is to avoid using homeomorphic embeddings on conjunctions, but go to a less explosive strategy, e.g. requiring a decrease in the total term size.[3] As we will see later in the benchmarks (e.g. Csc-th-t in Table 12.4), the combination of these

[2] Of course, by not restricting oneself to natural numbers up to a limit n, we can construct arbitrarily large branches starting from the same $p(s(0),s(0))$: $\langle p(s(0),s(0)), p(s(s(....s(0)...))),0), p(s(s(....0...))),0), ...\rangle$.

[3] Note that this also implies that Definition 12.1.7 of $split_Q(Q')$ has to be adapted, i.e. replacing $Q \trianglelefteq^* Q'$ by $s(Q) \leq s(Q'')$ and $Q \trianglelefteq^* Q'$ by $s(Q) \leq s(Q'')$, where $s(.)$ measures the term size of conjunctions.

two methods leads to reasonable transformation times and code size while maintaining good specialisation.

12.2 The system and its methods

We use the same partial evaluation system ECCE [170] as in Section 6.4. The system contains a generic algorithm to which one may add one's own methods for unfolding, partitioning, abstraction, etc. In the following we will give a short description of the different methods that we used in the experiments.

12.2.1 The algorithm

The system implements an extension of Algorithm 6.2.36, extended for conjunctive partial deduction and incorporating some ideas from [110, 62]. The algorithm uses a global tree γ with nodes labelled with (characteristic) conjunctions. When a conjunction Q gets unfolded, then the conjunctions in the bodies of the resultants of Q (maybe further split by the abstraction) are added as child nodes (leaves) of Q in the global tree.

Algorithm 12.2.1

Input: a program P and a goal $\leftarrow Q$
Output: a set of conjunctions $\mathcal{Q}$
Initialisation: $i := 0$; $\gamma_0 :=$ the global tree with a single node, labelled Q
repeat

1. **for** all leaves L in γ_i labelled with conjunction Q_L and for all leaf goals $\leftarrow B$ of $U(P, \leftarrow Q_L)$ **do**
   (a) $\mathcal{Q} = partition(B)$
   (b) **while** $\mathcal{Q} \neq \emptyset$ **do**
   select $Q_i \in \mathcal{Q}$
   $\mathcal{Q} := \mathcal{Q} \setminus Q_i$
   if Q_i is not an instance (respectively variant) of a node in γ_i **then**
   if $whistle(\gamma_i, L, Q_i)$ **then** $\mathcal{Q} := \mathcal{Q} \cup abstract(\gamma_i, L, Q_i)$
   else add a child L' to L with label Q_i
2. $i := i + 1$

until $\gamma_i = \gamma_{i-1}$
output the set of nodes in γ_i

The unfolding rule U performs the local control. It takes a program and a conjunction and produces a finite, but possibly incomplete SLD(NF)-tree. The function *partition* does the initial splitting of the bodies into either plain atoms for standard partial deduction or into contiguous connected subconjunctions (*ccs*) or maximal connected subconjunctions (*mcs*) for conjunctive partial deduction. In addition for the latter, the size of conjunctions can be limited by using static conjunctions (static vs. dyn.).

Then for each of the subconjunctions it is checked if there is a risk of non-termination. This is done by the function *whistle*. The whistle will look at the labels (conjunctions) on the branch in the global tree to which the new conjunction Q_i is going to be added as a leaf and if Q_i is "larger" than one of these, it returns true. Finally, if the "whistle blows" for some subconjunction Q_i, then Q_i is abstracted by using the function *abstract*. For the conjunctive methods this is achieved by calculating $split_{Q_\gamma}(Q_i)$ (possibly adapted for contiguous ordered subconjunction, see Subsection 12.1.2 above), where Q_γ is the conjunction "larger" than Q_i (depending on the actual whistle, e.g. $Q_\gamma \trianglelefteq^* Q_i$). This abstraction, combined with the $\trianglelefteq^*$ whistle of Definition 12.1.1, ensures termination of the inner **for** loop of Algorithm 12.2.1. Indeed, upon every iteration, a conjunction (Q_i) is removed from $\mathcal{Q}$, and either replaced by finitely many strictly smaller conjunctions (i.e. with fewer atoms) or is replaced by a conjunction which is strictly more general. As there are no infinite chains of strictly more general expressions (cf. Lemma 5.3.6), termination follows. The detailed proof can be found in [62].

After the algorithm terminates the residual program is obtained from the output by unfolding and renaming (cf. Chapters 10 and 11).

12.2.2 Concrete settings

We have concentrated on four local unfolding rules:

1. safe determinate (t-det): do determinate unfolding, (allowing one left-most non-determinate step) using homeomorphic embedding with covering ancestors of selected atoms to ensure finiteness.
2. safe indexed unfolding (l-idx). The difference with t-det is that more than one left-most non-determinate unfolding step is allowed. However, only "indexed" unfolding is then allowed, i.e. it is ensured that the unification work that might get duplicated (cf. Example 3.3.4) is captured by the Prolog indexing (which may depend on the particular compiler). Again, homeomorphic embeddings are used to ensure finiteness.
3. homeomorphic embedding and reduction of search space (h-rs): non-left-most unfolding is allowed if the search space is reduced by the

unfolding. In other words, an atom $p(\bar{t})$ can be selected if it does not match all the clauses defining p. Again, homeomorphic embeddings are used to ensure finiteness. Note that, in contrast to 2 and 3, this method might worsen the backtracking behaviour.

4. "MIXTUS-like" unfolding (x): See [245] for further details (we used $max\_rec = 2$, $max\_depth = 2$, $maxfinite = 7$, $maxnondeterm = 10$ and only allowed non-determinate unfolding when no user predicates were to the left of the selected literal).

Let us now turn our attention to the global control. The measures that we have used in whistles are the following:

1. homeomorphic embedding ($\trianglelefteq^*$) on the conjunctions
2. termsize ($s(.)$ as defined in Definition 5.3.5) on the conjunctions
3. homeomorphic embedding on the conjunctions and homeomorphic embedding on the associated characteristic trees
4. termsize on the conjunctions and homeomorphic embedding on the characteristic trees

Abstraction is always done by possibly splitting conjunctions further and then taking the *msg* as explained in Subsection 12.1.1. One standard partial deduction method (SE-hh-x, called ECCE-X in Chapter 6) also uses the ecological partial deduction principle of Chapters 5 and 6 to ensure preservation of characteristic trees upon generalisation. The latter becomes more tricky in the presence of splitting and we have therefore not yet implemented it for the conjunctive methods.

The following extensions, already used in Chapter 6, were always enabled:

- simple more specific resolution steps in the style of SP [97] and
- the selection of ground negative literals for the local control;
- the removal of unnecessary polyvariance of Section 6.3,
- determinate post-unfolding, as well as the
- redundant argument filtering of Chapter 11 in the post-processing phase.

12.3 Benchmarks and conclusion

The benchmarks were conducted in the same manner as in Chapter 6. The benchmark programs are available in [170]; Short descriptions are given in Appendix C. Note that we have added some specific deforestation and tupling benchmarks over the experiments in Chapter 6. Tables 12.3 – 12.10 show the results of the experiments. The relative runtimes (RRT), transformation times (TT, in seconds) and code size (in units of 4.08 bytes) are exactly like for the Tables 6.2, 6.3 and 6.4 in Chapter 6. Runtimes (RRT)

are given relative to the runtimes of the original programs. In computing averages and totals, time and size of the original program were taken in case of non-termination or an error occurring during transformation. Just like in Chapter 6, the total speedups are obtained by the formula

$$\frac{n}{\sum_{i=1}^{n} \frac{spec_i}{orig_i}}$$

where $n = 36$ is the number of benchmarks and $spec_i$ and $orig_i$ are the absolute execution times of the specialised and original programs respectively. The weighted total speedups are obtained by using the code size $size_i$ of the original program as a weight for computing the average:

$$\frac{\sum_{i=1}^{n} size_i}{\sum_{i=1}^{n} size_i \frac{spec_i}{orig_i}}$$

Some additional details can be found in Section 6.4.2.

The results are summarised in Tables 12.1 and 12.2. An overview of the speedups and average code sizes of some systems can be found in Figure 12.1. As in Chapter 6, we also compared to the existing systems MIXTUS [245, 244], PADDY [227, 228, 229] and SP [97, 98]. The same versions as in Chapter 6 have been used and for these systems ∞ means abnormal termination (crash or heap overflow) occurred, as explained in Section 6.4.2. As already mentioned in Chapter 6, the unfolding used by SP seems to be based on a refined determinate unfolding (look e.g. at the results for *depth.lam*), hence the "+" in Table 12.1. For the ECCE based systems the indication $> 12h$ means that the specialisation was interrupted after 12 hours (though, theoretically, it should have terminated by itself when granted sufficient time to do so). **bi err** means that an error occurred while running the program due to a call of a built-in where the arguments were not sufficiently instantiated.

Even more so than in Chapter 6, benchmarking Prolog (by BIM) programs on Sparc machines, turned out to be problematic at times. For instance, for `maxlength`, deforestation does not seem to pay off. However, with reordering of clauses we go from a relative time of 1.4 (i.e. a slowdown) to a relative time of 0.9 (i.e. a speedup)! On Sicstus Prolog 3, we even get a 20 % speedup for this example (without reordering)! The problem is probably due to the caching behaviour of the Sparc processor.

12.3.1 Analysing the results

One conclusion of the experiments is that conjunctive partial deduction (using determinate unfolding and contiguous splitting) pays off while guaranteeing no (serious) slowdown. In fact, the cases where there is a slowdown

System	Partition		Whistle		Unf	Total
	contig	s/d	conj	chtree		TT (min)
Conjunctive Partial Deduction						
Cdc-hh-t	*ccs*	dyn.	$\trianglelefteq^*$	$\trianglelefteq^*$	t-det	62.46
Cdc-th-t	*ccs*	dyn.	termsize	$\trianglelefteq^*$	t-det	31.18
Csc-hh-t	*ccs*	static	$\trianglelefteq^*$	$\trianglelefteq^*$	t-det	29.72
Csc-th-t	*ccs*	static	termsize	$\trianglelefteq^*$	t-det	5.95
Csc-hn-t	*ccs*	static	$\trianglelefteq^*$	none	t-det	35.49
Csc-tn-t	*ccs*	static	termsize	none	t-det	2.67
Csc-th-li	*ccs*	static	termsize	$\trianglelefteq^*$	l-idx	$> 12h$+12.95
Cdm-hh-t	*mcs*	dyn.	$\trianglelefteq^*$	$\trianglelefteq^*$	t-det	$> 12h$+110.49
Csm-hh-h	*mcs*	static	$\trianglelefteq^*$	$\trianglelefteq^*$	h-rs	$> 12h$+73.55
Standard Partial Deduction						
S-hh-t	-	-	homeo	homeo	t-det	3.00
S-hh-li	-	-	homeo	homeo	l-idx	14.95
SE-hh-x	-	-	homeo	homeo	mixtus	2.96
Existing Systems						
MIXTUS	-	-	mixtus	none	mixtus	∞+2.71
PADDY	-	-	mixtus	none	mixtus	∞+0.31
SP	-	-	pred =	=	det^+	$3^*\infty$+1.99

Table 12.1: Overview: systems and transformation times

System	Total Speedup	Weighted Speedup	Fully Unfoldable Speedup	Not Fully Unfoldable Speedup	Average Relative Size (orig = 1)
Conjunctive Partial Deduction					
Cdc-hh-t	1.93	2.44	5.90	1.66	2.39
Cdc-th-t	1.96	2.49	5.90	1.69	2.27
Csc-hh-t	1.89	2.38	5.90	1.62	2.02
Csc-th-t	1.92	2.44	5.90	1.65	1.68
Csc-hn-t	1.89	2.40	5.90	1.62	1.67
Csc-tn-t	1.76	2.18	4.48	1.54	1.53
Csc-th-li	1.89	2.38	7.07	1.61	1.79
Cdm-hh-t	2.00	2.39	5.90	1.72	3.17
Csm-hh-h	0.77	0.52	6.16	0.63	3.91
Standard Partial Deduction					
S-hh-t	1.56	1.86	2.57	1.42	1.60
S-hh-li	1.65	2.09	4.88	1.42	1.61
SE-hh-x	1.76	2.24	8.36	1.48	1.46
Existing Systems					
MIXTUS	1.65	2.11	8.13	1.38	1.67
PADDY	1.65	2.00	8.12	1.38	2.49
SP	1.34	1.54	2.08	1.23	1.18

Table 12.2: Summary of benchmarks (higher speedup and lower code size is better)

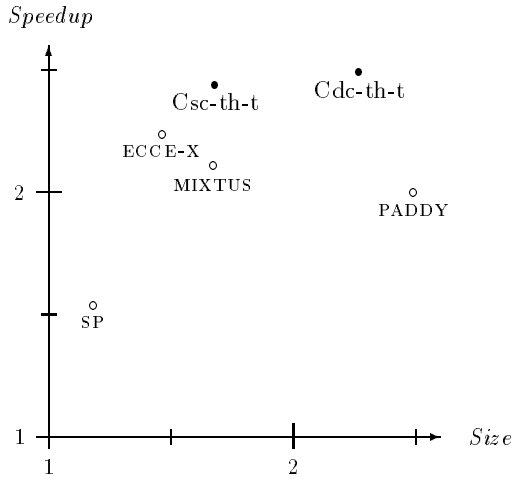


Figure 12.1: Weighted speedups and average code size for some systems

are some of those that were designed to show the effect of deforestation (`flip`, `match-append`, `maxlength` and `upto.sum2`). Two of these are handled well by the methods using non-contiguous splitting. On the fully unfoldable benchmarks, S-hh-t gave a speedup of 2.57 while Csc-hh-t achieved a speedup of 5.90. A particularly interesting example is `contains.kmp` benchmark, where Csc-hh-t is about 8 times faster than S-hh-t. *contains* has long been a difficult benchmark, especially for determinate unfolding rules. So, the results clearly illustrate that conjunctive partial deduction diminishes the need for aggressive unfolding.

Notice that MIXTUS and PADDY have very aggressive unfolding rules and fare well on the fully unfoldable benchmarks. However, on the non-fully unfoldable ones, even S-hh-t, based on determinate unfolding, is already better. The best standard partial deduction method, for both runtime and (apart from SP) code size, is SE-hh-x. Still, compared to any of the standard partial deduction methods, our conjunctive methods (except for Csm-hh-h, and also Csc-tn-t, which are not meant to be competitors anyway) have a better average speedup.

Furthermore, the experiments also show that the process of performing conjunctive partial deduction can be made efficient, especially if one uses determinate unfolding combined with a termsize measure on conjunctions (Csc-th-t and Csc-tn-t), in which case the average transformation time is comparable with that of standard partial deduction. Of course only further

experiments may show how the transformation times grow with the size of programs. In fact, the system itself was not written with efficiency as a primary concern and there is a lot of room for improvement on this point.

Next, the experiments demonstrate that using the termsize measure instead of homeomorphic embedding on conjunctions clearly improves the average transformation time without loosing too much specialisation. But they also show that if one uses the termsize measure, then the use of characteristic trees becomes vital (compare Csc-th-t and Csc-tn-t). However, methods with homeomorphic embedding on conjunctions (e.g. Csc-hn-t), do not seem to benefit from adding homeomorphic embedding on characteristic trees as well (e.g. Csc-hh-t). This, at first sight somewhat surprising phenomenon, can be explained by the fact that, for the benchmarks at hand, the homeomorphic embedding on conjunctions, in a global tree setting, is already a very generous whistle, and, in the absence of negation or e.g. the parsing problem (cf. the discussions in Section 6.4.4), a growing of the conjunction will often result in a growing of the characteristic tree as well, especially when the latter are based on determinate unfolding (see also the discussion at the end of this section concerning the match-append benchmark).

Comparing Csc-hh-t and Cdc-hh-t, one can see that using static conjunctions also pays off in terms of faster transformation time without much loss of specialisation. If one looks more closely at the results for both methods, then the speedup and the transformation times are more or less the same for the two methods except for the rather few cases where static conjunctions were really needed: `groundunify.complex`, `liftsolve.db2`, `regexp.r2`, `regexp.r3`, `remove2` and `imperative.power`. For those cases, the loss of speedup due to the use of static conjunctions was small or insignificant while the improvement in transformation time was considerable.

Comparing Csc-th-li to Csc-th-t, one sees that indexed unfolding does not seem to have a definite effect for conjunctive partial deduction. In some case the speedup is better and in some cases worse. Only for `relative.lam` is indexed unfolding much better than determinate, but this corresponds to a case where the program can be completely unfolded. This is of course partially due to the fact that conjunctive partial deduction diminishes the need for aggressive unfolding. For standard partial deduction however, indexed (as well as "MIXTUS-like") unfolding leads to a substantial improvement over determinate unfolding. Note that the "MIXTUS-like" unfolding used by SE-hh-x does not seem to pay off for conjunctive partial deduction at all. In a preliminary experiment, the method Csc-th-x only produced a total speedup of 1.69, i.e. only slightly better than MIXTUS or PADDY and worse than SE-hh-x. Still, for some examples it would be highly beneficial to allow more than "just" determinate unfolding (notably `depth.lam`, `grammar.lam`

and `relative.lam` — examples where SE-hh-x performs much better than all the conjunctive methods based on determinate unfolding). In future work, we will examine how more eager unfolding rules can be more successfully used for conjunctive partial deduction.

For some benchmarks, the best speedup is obtained by the non-safe methods Cdm-hh-t or Csm-hh-h based on non-contiguous *mcs* splitting. But in some cases, these methods, for reasons explained earlier, indeed lead to a considerable slowdown (`missionaries` and `remove`) and sometimes even to errors (`imperative.power` and `upto.sum1`). This shows that methods based on non-contiguous splitting can lead to better specialisation due to *tupling* and *deforestation*, but that we need some method to control the splitting and unfolding to ensure that no slowdown, or change in termination can occur.

Finally, a brief remark on the match-append benchmark. The bad figures of most systems seem to be due to a bad choice of the filtering, further work will be needed the avoid this kind of effect. Also, none of the presented methods was able to deforest this particular example. However, if we run for instance Csc-hh-t twice on match-append we get the desired deforestation and a much improved performance (relative time of 0.03 !). The same effect can be obtained by using the refinement described earlier in Section 5.4.1 (and available within the ECCE system), which consists in using a deeper characteristic tree for the control of polyvariance than the tree used for constructing the resultants. For instance, adding such a feature to Csc-hh-t solves the match-append problem and increases the total speedup from 1.89 to 2.04, but unfortunately at the cost of a larger transformation time (liftsolve.db2 no longer terminates in less than 12 hours).

12.3.2 Conclusion

It looks like conjunctive partial deduction can be performed with acceptable efficiency and pays off with respect to standard partial deduction, but there are still many unsolved problems. Indeed, the speedups compared to standard partial deduction are significant on average, but less dramatic and less systematic than we initially expected. Apparently, this is partly due to the fact that non-contiguous conjunctive partial deduction on the one hand often leads to substantial slowdowns and is not really practical for most applications, while contiguous conjunctive partial deduction on the other hand is in general too weak to deforest or tuple datastructures.

Therefore it is vital, if one wants to more heavily exploit the advantages of conjunctive partial deduction, to add non-contiguous splitting (i.e. reordering) in a safe way which guarantees no serious slowdown. A first step towards a solution is presented in [30], but it remains quite restric-

tive and considers only ground queries. Another, more pragmatic approach might be based on making use of some mode system to allow reordering of literals as long as the resulting conjunction remains well-moded. This would be very similar to the way in which the compiler for Mercury [257] reorders literals to create different modes for the same predicate. For the semantics of Mercury any well-moded re-ordering of the literals is allowed. Although this approach does not ensure the preservation of termination, it is then simply considered a programming error if one well-moded query terminates while the other does not. So, conjunctive partial deduction, not unlike program transformation in general, may be much more viable in a truly declarative setting. Of course, also in that context, finding a good way to prevent slowdowns remains a pressing open question. A promising direction might be to incorporate more detailed efficiency and cost estimation into the global and local control of conjunctive partial deduction, e.g. based on [70, 71, 69]. Other topics for further work include implementation improvements, research into more sophisticated local control, and methods for improved information passing between the local and global control levels.

Finally, we perceive the extensive experimentation in itself and the set of benchmark assembled to that end, as a noteworthy contribution of this chapter. Indeed we feel that, if one wants to move towards more practical or even industrial applications of program transformation, extensive and realistic empirical experiments are called for. The benchmark suite used in this chapter (and in Chapter 6), containing some difficult benchmarks especially designed to put transformation methods at a stress, might form a suitable basis to gauge progress along the difficult path towards full practical applicability. We invite the interested reader to consult [170] for further details.

	Cdc-hh-t			Csc-hh-t		
Benchmark	RRT	Size	TT	RRT	Size	TT
advisor	0.47	412	0.90	0.47	412	0.86
applast	0.36	202	0.92	0.36	202	0.80
contains.kmp	0.11	1039	5.61	0.11	1039	5.41
depth.lam	0.15	1837	4.11	0.15	1837	4.01
doubleapp	0.80	362	0.85	0.80	362	0.88
ex_depth	0.26	508	3.30	0.29	407	1.62
flip	1.33	686	1.41	1.33	686	1.25
grammar.lam	0.16	309	1.94	0.16	309	1.84
groundunify.complex	0.40	6247	118.69	0.47	6277	19.47
groundunify.simple	0.25	368	0.78	0.25	368	0.80
imperative.power	0.40	36067	906.60	0.40	3132	71.37
liftsolve.app	0.05	1179	5.75	0.05	1179	5.98
liftsolve.db1	0.01	1280	22.39	0.01	1280	14.23
liftsolve.db2	0.16	17472	2599.03	0.21	21071	1594.90
liftsolve.lmkng	1.02	1591	3.09	1.02	1591	2.66
map.reduce	0.07	507	0.78	0.07	507	0.85
map.rev	0.11	427	0.83	0.11	427	0.82
match-append	1.21	406	1.29	1.21	406	1.14
match.kmp	0.73	639	1.16	0.73	639	1.15
maxlength	1.40	620	1.22	1.40	620	1.14
memo-solve	0.81	1095	5.88	0.81	1095	2.53
missionaries	0.69	2960	7.93	0.69	2960	7.59
model_elim.app	0.12	451	2.65	0.12	451	2.66
regexp.r1	0.39	557	1.76	0.39	557	1.36
regexp.r2	0.41	833	3.57	0.53	692	1.52
regexp.r3	0.31	1197	6.85	0.44	873	1.82
relative.lam	0.07	1011	5.80	0.07	1011	5.39
remove	0.62	1774	5.34	0.62	1774	4.89
remove2	0.87	1056	3.42	0.92	831	2.08
rev_acc_type	1.00	242	1.01	1.00	242	0.91
rev_acc_type.inffail	0.63	864	3.21	0.63	864	3.01
rotateprune	0.71	1165	3.08	0.71	1165	2.80
ssuply.lam	0.06	262	1.31	0.06	262	1.17
transpose.lam	0.17	2312	2.87	0.17	2312	2.45
upto.sum1	1.20	848	4.00	1.20	848	3.64
upto.sum2	1.12	623	1.48	1.12	623	1.46
Average	0.52	2484	103.91	0.53	1648	49.35
Total	18.66	89408	3740.8	19.10	59311	1776.5
Total Speedup	**1.93**			**1.89**		

Table 12.3: ECCE Determinate conjunctive partial deduction (A)

	Csc-th-t			Cdc-th-t		
Benchmark	RRT	Size	TT	RRT	Size	TT
advisor	0.47	412	0.87	0.47	412	0.82
applast	0.36	202	0.67	0.36	202	0.86
contains.kmp	0.11	1039	5.44	0.11	1039	5.22
depth.lam	0.15	1837	3.82	0.15	1837	3.53
doubleapp	0.80	362	0.84	0.80	362	0.86
ex_depth	0.29	407	1.60	0.27	508	3.08
flip	1.33	686	1.02	1.33	686	1.41
grammar.lam	0.16	309	1.82	0.16	309	1.76
groundunify.complex	0.47	6277	19.08	0.40	6247	81.35
groundunify.simple	0.25	368	0.75	0.25	368	0.73
imperative.power	0.40	3293	42.85	0.40	37501	1039.48
liftsolve.app	0.05	1179	5.74	0.05	1179	5.43
liftsolve.db1	0.01	1280	13.33	0.01	1280	20.39
liftsolve.db2	0.17	5929	198.19	0.17	11152	628.47
liftsolve.lmkng	1.02	1591	2.63	1.02	1591	2.89
map.reduce	0.07	507	0.80	0.07	507	0.77
map.rev	0.11	427	0.80	0.11	427	0.80
match-append	1.21	406	1.17	1.21	406	1.18
match.kmp	0.73	639	1.15	0.73	639	1.22
maxlength	1.40	620	1.17	1.40	620	1.13
memo-solve	0.81	1095	4.54	0.81	1095	10.32
missionaries	0.69	2960	7.13	0.69	2960	7.86
model_elim.app	0.12	451	2.58	0.12	451	2.60
regexp.r1	0.39	557	1.41	0.39	557	1.76
regexp.r2	0.53	692	1.55	0.43	833	3.55
regexp.r3	0.44	873	1.87	0.30	1197	6.01
relative.lam	0.07	1011	5.32	0.07	1011	5.74
remove	0.62	1774	4.92	0.62	1774	5.39
remove2	0.92	831	2.13	0.87	1056	3.45
rev_acc_type	1.00	242	0.96	1.00	242	1.04
rev_acc_type.inffail	0.63	864	3.09	0.63	864	3.24
rotateprune	0.71	1165	2.81	0.71	1165	3.10
ssuply.lam	0.06	262	1.19	0.06	262	1.32
transpose.lam	0.17	2312	2.53	0.17	2312	2.51
upto.sum1	0.88	734	3.03	0.88	734	3.40
upto.sum2	1.12	623	1.48	1.12	623	1.49
Average	0.52	1228	9.73	0.51	2345	51.78
Total	18.73	44216	350.3	18.35	84408	1864.16
Total Speedup	**1.92**			**1.96**		

Table 12.4: ECCE Determinate conjunctive partial deduction (B)

	Csc-hn-t			Csc-tn-t		
Benchmark	RRT	Size	TT	RRT	Size	TT
advisor	0.47	412	0.88	0.47	412	1.20
applast	0.36	202	0.64	0.36	202	0.73
contains.kmp	0.11	1039	5.19	0.63	862	1.16
depth.lam	0.15	1837	3.95	0.15	1837	4.18
doubleapp	0.80	362	0.86	0.80	362	1.13
ex_depth	0.29	407	1.60	0.29	407	1.75
flip	1.33	686	1.07	1.33	686	1.14
grammar.lam	0.16	309	1.77	0.16	309	1.98
groundunify.complex	0.40	4869	15.03	0.47	5095	18.67
groundunify.simple	0.25	368	0.76	0.25	368	0.94
imperative.power	0.37	2881	49.33	0.37	2881	32.88
liftsolve.app	0.05	1179	5.50	0.05	1179	5.65
liftsolve.db1	0.01	1280	13.60	0.01	1280	13.56
liftsolve.db2	0.17	10146	1974.50	0.33	3173	19.84
liftsolve.lmkng	1.09	1416	1.82	1.07	1416	2.09
map.reduce	0.07	507	0.83	0.07	507	1.09
map.rev	0.11	427	0.79	0.11	427	1.04
match-append	1.21	406	0.91	1.21	406	1.06
match.kmp	0.73	639	1.11	0.73	613	1.69
maxlength	1.40	620	1.20	1.40	620	1.31
memo-solve	0.81	1095	4.28	1.38	1709	6.73
missionaries	0.69	2960	7.06	0.71	3083	6.03
model_elim.app	0.12	451	2.63	0.12	451	2.75
regexp.r1	0.39	557	1.38	0.39	557	1.84
regexp.r2	0.53	692	1.55	0.53	692	1.66
regexp.r3	0.44	873	1.81	0.44	873	2.00
relative.lam	0.07	1011	5.34	0.45	1252	6.78
remove	0.65	1191	2.42	0.65	1191	2.19
remove2	0.92	831	2.04	0.92	831	1.89
rev_acc_type	1.00	242	0.67	1.00	242	0.84
rev_acc_type.inffail	0.63	598	0.87	0.63	598	0.98
rotateprune	0.71	1165	2.79	0.71	1165	2.55
ssuply.lam	0.06	262	1.15	0.06	262	1.32
transpose.lam	0.17	2312	2.46	0.17	2312	2.62
upto.sum1	1.20	848	3.77	0.88	734	2.83
upto.sum2	1.12	623	1.45	1.12	623	1.39
Average	0.53	1270	58.97	0.57	1100	4.37
Total	19.04	45703	2123.01	20.40	39617	157.49
Total Speedup	**1.89**			**1.76**		

Table 12.5: ECCE Determinate conjunctive partial deduction (C)

	Cdm-hh-t			Csm-hh-h		
Benchmark	RRT	Size	TT	RRT	Size	TT
advisor	0.47	412	0.80	0.46	647	1.00
applast	0.36	202	0.97	0.36	145	0.85
contains.kmp	0.11	1039	5.16	0.10	814	5.77
depth.lam	0.15	1837	3.99	0.15	1848	5.91
doubleapp	0.80	362	0.84	0.82	277	0.83
ex_depth	0.26	508	3.15	0.34	1240	6.18
flip	0.75	441	1.11	0.69	267	0.81
grammar.lam	0.16	309	1.74	0.16	309	2.21
groundunify.complex	0.40	6247	137.68	0.47	8113	77.21
groundunify.simple	0.25	368	0.75	0.25	399	1.27
imperative.power	0.37	103855	4548.51	bierr	85460	1617.93
liftsolve.app	0.05	1179	5.44	0.06	1210	6.26
liftsolve.db1	0.01	1280	20.85	0.02	1311	18.10
liftsolve.db2	0.17	17206	1813.49	-	-	$> 12h$
liftsolve.lmkng	1.02	1591	2.85	1.24	1951	8.89
map.reduce	0.07	507	0.85	0.08	348	0.84
map.rev	0.11	427	0.88	0.13	285	0.75
match-append	1.21	406	1.20	1.36	362	0.98
match.kmp	0.73	639	1.18	0.65	543	0.94
maxlength	1.40	620	1.18	1.10	314	1.19
memo-solve	1.12	1294	22.69	1.50	3777	34.84
missionaries	-	-	$> 12h$	21.17	43268	2537.39
model_elim.app	0.12	451	2.77	0.12	451	3.20
regexp.r1	0.39	557	1.91	0.20	457	1.21
regexp.r2	0.41	833	3.65	0.57	1954	6.43
regexp.r3	0.31	1197	6.84	1.89	9124	31.07
relative.lam	0.07	1011	5.84	0.01	954	7.31
remove	0.62	1774	5.51	6.40	4116	7.69
remove2	0.87	1056	3.65	0.94	862	2.34
rev_acc_type	1.00	242	1.08	1.00	242	1.73
rev_acc_type.inffail	0.63	864	3.26	0.61	786	1.94
rotateprune	0.71	1165	4.01	0.17	691	2.33
ssuply.lam	0.06	262	1.36	0.06	262	1.69
transpose.lam	0.17	2312	2.77	0.19	2436	4.34
upto.sum1	bierr	448	3.43	bierr	479	3.79
upto.sum2	0.65	394	1.50	0.58	242	1.11
Average	0.50	4380	189.23	1.30	5027	125.90
Total	18.01	153295	6622.90	46.84	175944	4406.33
Total Speedup	**2.00**			**0.77**		

Table 12.6: ECCE Non-contiguous conjunctive partial deduction

	Csc-th-li			S-hh-li		
Benchmark	RRT	Size	TT	RRT	Size	TT
advisor	0.32	809	0.85	0.31	809	0.85
applast	0.34	145	0.67	1.48	314	0.75
contains.kmp	0.10	1227	15.73	0.55	1294	5.97
depth.lam	0.15	1848	12.65	0.62	1853	3.76
doubleapp	0.82	277	0.98	0.95	216	0.58
ex_depth	0.27	659	6.31	0.44	1649	4.26
flip	0.95	493	1.26	1.03	313	0.65
grammar.lam	0.14	218	2.15	0.14	218	2.43
groundunify.complex	0.47	19640	117.12	0.47	8356	50.63
groundunify.simple	0.25	368	0.76	0.25	368	0.77
imperative.power	0.69	3605	155.55	0.58	2254	62.97
liftsolve.app	0.05	1179	6.04	0.06	1179	6.40
liftsolve.db1	0.02	1326	19.94	0.02	1326	20.82
liftsolve.db2	-	-	$> 12h$	0.76	3751	242.86
liftsolve.lmkng	1.00	1591	4.22	1.07	1730	2.12
map.reduce	0.08	348	0.78	0.08	348	0.82
map.rev	0.13	285	0.79	0.13	285	0.88
match-append	1.36	362	1.02	1.36	362	0.75
match.kmp	0.65	543	1.65	0.65	543	1.77
maxlength	1.30	620	1.22	1.10	715	1.16
memo-solve	0.95	1015	3.69	1.20	2238	4.96
missionaries	0.54	15652	348.68	0.66	13168	430.99
model_elim.app	0.13	444	3.11	0.13	444	3.18
regexp.r1	0.20	457	1.04	0.20	457	1.03
regexp.r2	0.41	831	4.98	0.61	737	4.67
regexp.r3	0.31	1041	14.70	0.38	961	14.00
relative.lam	0.00	261	6.19	0.00	261	5.88
remove	0.87	1369	7.31	0.68	659	1.02
remove2	0.93	862	3.51	0.75	453	1.30
rev_acc_type	1.00	242	1.21	1.00	242	0.92
rev_acc_type.inffail	0.66	700	2.24	0.80	850	1.25
rotateprune	0.80	1470	4.62	1.02	779	1.10
ssuply.lam	0.06	262	1.41	0.06	262	1.51
transpose.lam	0.18	2312	2.99	0.17	2312	2.99
upto.sum1	0.88	734	2.81	1.07	581	1.80
upto.sum2	1.00	654	1.48	1.05	485	1.38
Average	0.53	1824	21.70	0.61	1466	24.70
Total	19.03	63849	759.66	21.86	52772	889.18
Total Speedup	**1.89**			**1.65**		

Table 12.7: ECCE Partial deduction based on indexed unfolding

	S-hh-t			SE-hh-x		
Benchmark	RRT	Size	TT	RRT	Size	TT
advisor	0.47	412	0.87	0.31	809	0.78
applast	1.05	343	0.71	1.48	314	0.70
contains.kmp	0.85	1290	2.69	0.09	685	4.48
depth.lam	0.94	1955	1.47	0.02	2085	1.91
doubleapp	0.95	277	0.65	0.95	216	0.53
ex_depth	0.76	1614	2.54	0.32	350	1.58
flip	1.05	476	0.77	1.03	313	0.53
grammar.lam	0.16	309	1.91	0.14	218	1.90
groundunify.complex	0.40	5753	13.47	0.53	4800	0.75
groundunify.simple	0.25	368	0.78	0.25	368	22.03
imperative.power	0.42	2435	75.10	0.54	1578	27.42
liftsolve.app	0.05	1179	6.05	0.06	1179	6.57
liftsolve.db1	0.01	1280	13.27	0.02	1326	7.33
liftsolve.db2	0.18	3574	16.86	0.61	4786	34.25
liftsolve.lmkng	1.07	1730	1.80	1.02	2385	2.75
map.reduce	0.07	507	0.91	0.08	348	0.86
map.rev	0.11	427	0.83	0.11	427	0.89
match-append	1.21	406	0.64	1.36	362	0.68
match.kmp	0.73	639	1.16	0.70	669	1.23
maxlength	1.20	715	1.07	1.10	421	0.95
memo-solve	1.17	2318	4.74	1.09	2241	4.31
missionaries	0.81	2294	5.11	0.72	2226	9.21
model_elim.app	0.63	2100	2.82	0.13	532	3.56
regexp.r1	0.50	594	1.28	0.29	435	0.98
regexp.r2	0.57	629	1.28	0.51	1159	4.87
regexp.r3	0.50	828	1.74	0.42	1684	14.92
relative.lam	0.82	1074	1.89	0.00	261	4.06
remove	0.71	955	1.46	0.68	659	0.90
remove2	0.74	508	1.15	0.80	440	1.00
rev_acc_type	1.00	242	0.70	1.00	242	0.83
rev_acc_type.inffail	0.63	864	1.48	0.60	527	0.80
rotateprune	0.71	1165	1.77	1.02	779	0.88
ssuply.lam	0.06	262	1.15	0.06	262	1.18
transpose.lam	0.17	2312	2.49	0.17	2312	1.98
upto.sum1	1.06	581	2.18	1.20	664	3.11
upto.sum2	1.10	623	1.50	1.05	485	0.94
Average	0.64	1196	4.90	0.57	1071	4.77
Total	23.12	43038	176.29	20.47	38614	171.65
Total Speedup	**1.56**			**1.76**		

Table 12.8: ECCE Standard partial deduction methods

	MIXTUS			PADDY		
Benchmark	RRT	Size	TT	RRT	Size	TT
advisor	0.31	809	0.85	0.31	809	0.10
applast	1.27	309	0.28	1.30	309	0.08
contains.kmp	0.16	533	2.48	0.11	651	0.55
depth.lam	0.04	1881	4.15	0.02	2085	0.32
doubleapp	1.00	295	0.30	0.98	191	0.08
ex_depth	0.40	643	2.40	0.29	1872	0.53
flip	1.03	495	0.37	1.02	290	0.12
grammar.lam	0.17	841	2.73	0.43	636	0.22
groundunify.complex	0.67	5227	11.68	0.60	4420	1.53
groundunify.simple	0.25	368	0.45	0.25	368	0.13
imperative.power	0.56	2842	5.35	0.58	3161	2.18
liftsolve.app	0.06	1179	4.78	0.06	1454	0.80
liftsolve.db1	0.01	1280	5.36	0.02	1280	1.20
liftsolve.db2	0.31	8149	58.19	0.32	4543	1.60
liftsolve.lmkng	1.16	2169	4.89	0.98	1967	0.32
map.reduce	0.68	897	0.17	0.08	498	0.20
map.rev	0.11	897	0.16	0.26	2026	0.37
match-append	0.47	389	0.27	0.98	422	0.12
match.kmp	1.55	467	4.89	0.69	675	0.28
maxlength	1.20	594	0.72	0.90	398	0.17
memo-solve	0.60	1493	12.72	1.48	3716	1.70
missionaries	-	-	∞	-	-	∞
model_elim.app	0.13	624	5.73	0.10	931	0.90
regexp.r1	0.20	457	0.73	0.29	417	0.13
regexp.r2	0.82	1916	2.85	0.67	3605	0.63
regexp.r3	0.60	2393	4.49	1.26	10399	1.35
relative.lam	0.01	517	7.76	0.00	517	0.42
remove	0.81	715	0.49	0.71	437	0.12
remove2	1.01	715	0.84	0.84	756	0.12
rev_acc_type	1.00	497	0.99	0.99	974	0.33
rev_acc_type.inffail	0.97	276	0.77	0.94	480	0.28
rotateprune	1.02	756	0.49	1.01	571	0.12
ssuply.lam	0.06	262	0.93	0.08	262	0.08
transpose.lam	0.18	1302	3.89	0.18	1302	0.43
upto.sum1	0.96	556	1.80	1.08	734	0.30
upto.sum2	1.06	462	0.44	1.06	462	0.13
Average	0.61	1234	4.44	0.61	1532	0.51
Total	21.83	43205	155.37	21.87	53618	17.95
Total Speedup	**1.65**			**1.65**		

Table 12.9: Some existing systems (A)

	SP		
Benchmark	RRT	Size	TT
advisor	0.40	463	0.29
applast	0.84	255	0.15
contains.kmp	0.75	985	1.13
depth.lam	0.53	928	0.99
doubleapp	1.02	160	0.11
ex_depth	0.27	786	1.35
flip	1.02	259	0.13
grammar.lam	0.15	280	0.71
groundunify.complex	0.73	4050	2.46
groundunify.simple	0.61	407	0.20
imperative.power	1.16	1706	6.97
liftsolve.app	0.23	1577	2.46
liftsolve.db1	0.82	4022	3.95
liftsolve.db2	0.82	3586	3.71
liftsolve.lmkng	1.16	1106	0.37
map.reduce	0.09	437	0.23
map.rev	0.13	351	0.20
match-append	0.99	265	0.18
match.kmp	1.08	527	0.49
maxlength	0.90	367	0.31
memo-solve	1.15	1688	3.65
missionaries	0.73	16864	82.59
model_elim.app	-	-	∞
regexp.r1	0.54	466	0.37
regexp.r2	1.08	1233	0.67
regexp.r3	1.03	1646	1.20
relative.lam	0.69	917	0.35
remove	0.75	561	0.29
remove2	0.82	386	0.25
rev_acc_type	-	-	∞
rev_acc_type.inffail	-	-	∞
rotateprune	1.00	725	0.31
ssuply.lam	0.06	231	0.52
transpose.lam	0.26	1267	0.52
upto.sum1	1.05	467	0.48
upto.sum2	1.01	431	0.21
Average	0.75	1497	3.57
Total	26.86	49399	117.80
Total Speedup	**1.34**		

Table 12.10: Some existing systems (B)

Part VI

Combining Abstract Interpretation and Partial Deduction

Chapter 13

Logic Program Specialisation: How to Be More Specific

13.1 Partial deduction vs. abstract interpretation

The heart of any technique for logic program specialisation, is a program analysis phase. Given a program P and a goal $\leftarrow Q$, one aims to analyse the computation-flow of P for all instances $\leftarrow Q\theta$ of $\leftarrow Q$. Based on the results of this analysis, new program clauses are synthesised.

As we have seen on many occasions in this thesis, in partial deduction, such an analysis is based on the construction of finite but possibly incomplete SLD(NF)-trees. More specifically, following the foundations for standard partial deduction we have presented in Chapter 3, one constructs

- a finite set of atoms $\mathcal{A} = \{A_1, \ldots, A_n\}$, and
- a finite (possibly incomplete) SLD(NF)-tree τ_i for each $P \cup \{\leftarrow A_i\}$,

such that:

1) each atom in the initial goal $\leftarrow Q$ is an instance of some $A_i \in \mathcal{A}$, and
2) for each goal $\leftarrow B_1, \ldots, B_k$ labelling a leaf of some SLD(NF)-tree τ_l, each B_i is an instance of some $A_j \in \mathcal{A}$.

The conditions 1) and 2) (of $\mathcal{A}$-coveredness, cf. Definition 3.2.11) ensure that *together* the SLD(NF)-trees $\tau_1, \ldots, \tau_n$ form a *complete description* of all possible computations that can occur for all concrete instances $\leftarrow Q\theta$ of the goal of interest. At the same time, the point is to propagate the available

input data in $\leftarrow Q$ as much as possible through these trees, in order to obtain sufficient accuracy. The outcome of the analysis is precisely the set of SLD(NF)-trees $\{\tau_1, \ldots, \tau_n\}$: a complete, and as precise as possible, description of the computation-flow.

Finally, a code generation phase produces a *resultant clause* for each non-failing branch of each tree, which synthesises the computation in that branch.

Algorithm 3.3.11, presented in Chapter 3, describes the basic layout of practically all proposed algorithms for computing $\mathcal{A}$ and $\{\tau_1, \ldots, \tau_n\}$. An analysis following this scheme focusses exclusively on a top-down propagation of call-information. In the separate SLD-trees τ_i, this propagation is performed through repeated unfolding steps. The propagation over different trees is achieved by the fact that for each atom in a leaf of a tree there exists another tree with (a generalisation of) this atom as its root. The decision to create a *set* of different SLD-trees — instead of just creating one single tree, which would include both unfolding steps *and* generalisation steps — is motivated by the fact that these individual trees determine how to generate the new clauses.

The starting point for this chapter is that the described analysis scheme suffers from some clear imprecision problems. It has some obvious drawbacks compared to top-down abstract interpretation schemes, such as for instance the one in [36]. These drawbacks are related to two issues:

- the lack of *success-propagation*, both upwards and side-ways,
- the lack of inferring *global* success-information.

We discuss these issues in more detail. In the remainder of this chapter we will restrict our attention to *definite* logic programs (possibly with some declarative built-ins like $\backslash ==$, cf. Section 2.3.3).

13.1.1 Lack of success-propagation

Consider the following tiny program:

Example 13.1.1

$p(X) \leftarrow q(X), r(X)$
$q(a) \leftarrow$
$r(a) \leftarrow$
$r(b) \leftarrow$

For a given query $\leftarrow p(X)$, one possible (although very unoptimal) outcome of the Algorithm 3.3.11 is the set $\mathcal{A} = \{p(X), q(X), r(X)\}$ and the SLD-trees τ_1, τ_2 and τ_3 presented in Figure 13.1.

With this result of the analysis, the transformed program would be identical to the original one. Note that in τ_2 we have derived that the

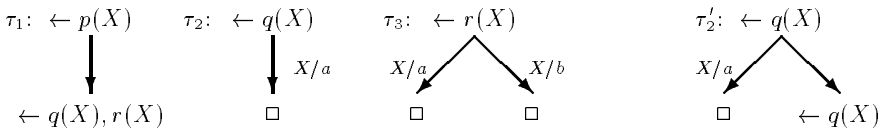


Figure 13.1: A possible outcome of Algorithm 3.3.11 for Examples 13.1.1 and 13.1.2

only answer for $\leftarrow q(X)$ is X/a. An abstract interpretation algorithm such as the one in [36] would propagate this success-information to the leaf of τ_1, yielding that (under the left-to-right selection rule) the call $\leftarrow r(X)$ becomes more specific, namely $\leftarrow r(a)$. This information would then be used in the analysis of the $r/1$ predicate, allowing to remove the redundant branch. Finally, the success-information, X/a, would be propagated up to the $\leftarrow p(X)$ call, yielding a specialised program:

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

which is correct for all instances of the considered query $\leftarrow p(X)$.

Note that this particular example could be solved by the techniques in [97]. There, a limited success-propagation, restricted to only one resolution step, is introduced and referred to as a *more specific resolution step*. The particular example can of course also be solved by standard partial deduction and a sufficiently refined unfolding rule. More difficult and realistic examples, which can be solved by neither [97] nor standard partial deduction alone, will be presented later on in the chapter.

13.1.2 Lack of inference of global success-information

Example 13.1.2 Assume that we add the clause $q(X) \leftarrow q(X)$ to the program in Example 13.1.1. A possible outcome of Algorithm 3.3.11 for the query $\leftarrow p(X)$ now is $\mathcal{A} = \{p(X), q(X), r(X)\}$ along with the SLD-trees τ_1, τ_2', τ_3, depicted in Figure 13.1.

Again, standard partial deduction produces a resulting program which is identical to the input program. In this case, simple bottom-up propagation of successes is insufficient to produce a better result. An additional fix-point computation is needed to detect that X/a is the only answer substitution. Methods as the one in [36] integrate such fix-point computations in the

top-down analysis. As a result, the same more specialised program as for Example 13.1.1 can be obtained.

In addition to pointing out further imprecision problems of the usual analysis scheme, in this chapter:

1. We propose a more refined analysis scheme, building on the notions of *conjunctive partial deduction* (cf. Chapters 10, 11 and 12) and *more specific programs* (see [191, 192]),[1] that solves the above mentioned problems.
2. We illustrate the *applicability* of the new scheme and to describe a *class of applications* in which they are *vital* for successful specialisation.

One class of problems is related to the specialisation of meta-programs that use the *ground-representation* for representing object programs (cf. Chapter 8). In Section 9.4 of Chapter 9 we already pointed out (in the context of pre-compiling integrity checking) that specialising such program satisfactorily often requires the analysis of an infinite number of different computations (like e.g. in Example 13.1.2), something which partial deduction techniques alone cannot do. In this chapter we provide a general solution for this problem.

The remainder of the chapter is organised as follows. In Section 13.2 we present the intuitions behind the proposed solution and illustrate the extensions on a few simple examples. In Section 13.3 we present more realistic, practical examples and and we justify the need for a more refined algorithm. This more refined Algorithm is then presented in Section 13.4 and used to specialise the ground representation in Section 13.5. We conclude with some discussions in Section 13.6.

13.2 Introducing more specific programs

There are different ways in which one could enhance the analysis to cope with the problems mentioned in the introduction. A solution that seems most promising is to just apply the abstract interpretation scheme of [36] to replace Algorithm 3.3.11. Unfortunately, this analysis is based on an AND-OR-tree representation of the computation, instead of an SLD-tree representation. As a result, applying the analysis for partial deduction causes considerable problems for the code-generation phase. It becomes very complicated to extract the specialised clauses from the tree. The alternative of adapting the analysis of [36] in the context of an SLD-tree

[1] The method of [191, 192] is the most straightforward to integrate with partial deduction because both these methods use the same abstract domain: a set of concrete atoms (or goals) is represented by all the instances of a given atom (or goal).

representation causes considerable complications as well. The analysis very heavily exploits the AND-OR-tree representation to enforce termination.

As mentioned above, the solution we propose here is based on the combination of two existing analysis schemes, each of which is underlying to a specific specialisation technique: the one of *conjunctive partial deduction* introduced in Chapter 10 and the one of *more specific programs* [191, 192].

Let us first present an abstract interpretation method based on [191, 192] which calculates more specific versions of programs.

We first recall the two following notations. $Pred(P)$ denotes the set of predicates occurring in a set of logic formulas P. By $mgu^*(A, B)$ we denote a particular idempotent and relevant most general unifier of A and some B', obtained from B by renaming apart wrt A. We also define the predicate-wise application msg^* of the msg: $msg^*(S) = \{msg(S_p) \mid p \in Pred(P)\}$, where S_p are all the atoms of S having p as predicate.

In the following we define the well-known *non-ground* T_P operator (cf. Definition 2.2.7 for the ground T_P operator) along with an abstraction U_P of it.

Definition 13.2.1 (T_P,U_P) For a definite logic program P and a set of atoms $\mathcal{A}$ we define:
$T_P(\mathcal{A}) = \{H\theta_1 \ldots \theta_n \mid H \leftarrow B_1, \ldots, B_n \in P \wedge \theta_i = mgu^*(B_i\theta_1 \ldots \theta_{i-1}, A_i)$ with $A_i \in \mathcal{A}\}$.
We also define $U_P(\mathcal{A}) = msg^*(T_P(\mathcal{A}))$.

One of the abstract interpretation methods of [191, 192] can be seen (for maximally general, atomic goal tuples, see Section 13.6) as calculating $lfp(U_P) = U_P \uparrow^\infty (\emptyset)$.[2] In [191, 192] more specific versions of clauses and programs are obtained in the following way:

Definition 13.2.2 (more specific version) Let $C = H \leftarrow B_1, \ldots, B_n$ be a definite clause and $\mathcal{A}$ a set of atoms. We define

$$msv_{\mathcal{A}}(C) = \{C\theta_1 \ldots \theta_n \mid \theta_i = mgu^*(B_i\theta_1 \ldots \theta_{i-1}, A_i) \text{ with } A_i \in \mathcal{A}\}.$$

The *more specific version* $msv(P)$ of a program P is then obtained by replacing every clause $C \in P$ by $msv_{lfp(U_P)}(C)$ (note that $msv_{lfp(U_P)}(C)$ contains at most 1 clause).

In the light of the stated problems, an integration of partial deduction with the more specific program transformation seems a quite natural solution. In [191, 192] such an integration was already suggested as a promising future direction. Take for instance the following program.

[2] This in turn can be seen as an abstract interpretation method which infers top level functors for every predicate.

Example 13.2.3 (eqlist)

$eqlist(X, Z) \leftarrow append(X, [\,], Z)$
$append([\,], L, L) \leftarrow$
$append([H|X], Y, [H|Z]) \leftarrow append(X, Y, Z)$

Partial deduction for the goal $\leftarrow eqlist(X, Z)$ results in the following specialised program P':

$eqlist(X, Z) \leftarrow app_{[\,]}(X, Z)$
$app_{[\,]}([\,], [\,]) \leftarrow$
$app_{[\,]}([H|X], [H|Z]) \leftarrow app_{[\,]}(X, Z)$

The least fixpoint of $U_{P'}$ is obtained as follows:

$U_{P'} \uparrow^1 (\emptyset) = \{app_{[\,]}([\,], [\,])\}$
$U_{P'} \uparrow^2 (\emptyset) = msg^*(\{app_{[\,]}([\,], [\,]), app_{[\,]}([H], [H]), eqlist([\,], [\,])\}) =$
$= \{app_{[\,]}(X, X), eqlist([\,], [\,])\}$
$U_{P'} \uparrow^3 (\emptyset) = U_{P'} \uparrow^4 (\emptyset) = \{app_{[\,]}(X, X), eqlist(X, X)\}$

The more specific version $msv(P')$ of P' is thus:

$eqlist(X, X) \leftarrow app_{[\,]}(X, X)$
$app_{[\,]}([\,], [\,]) \leftarrow$
$app_{[\,]}([H|X], [H|X]) \leftarrow app_{[\,]}(X, X)$

So, by combining partial deduction with the more specific program transformation, we are able to deduce that $\leftarrow eqlist(X, X)$ is a more specific version of $\leftarrow eqlist(X, Z)$, something which $msv(.)$ alone is incapable of doing.

The following example reveals that, in general, this combination is still too weak to deal with side-ways information propagation.

Example 13.2.4 (append-last)

$app\_last(L, X) \leftarrow append(L, [a], R), last(R, X)$
$append([\,], L, L) \leftarrow$
$append([H|X], Y, [H|Z]) \leftarrow append(X, Y, Z)$
$last([X], X) \leftarrow$
$last([H|T], X) \leftarrow last(T, X)$

The hope is that the specialisation techniques are sufficiently strong to infer that a query $\leftarrow app\_last(L, X)$ produces the answer $X = a$. Partial deduction on its own is incapable of producing this result. An SLD-tree for the query $\leftarrow app\_last(L, X)$ takes the form of τ_1 in Figure 13.2. Although the success-branch of the tree produces $X = a$, there are infinitely many possibilities for L and, without a bottom-up fixed-point computation, $X = a$ cannot be derived for the entire computation. At some point the unfolding

needs to terminate, and additional trees for *append* and *last*, for instance τ_2 and τ_3 in Figure 13.2, need to be constructed. The resulting program is:

$app\_last([\,], a) \leftarrow$
$app\_last([H|L'], X) \leftarrow app_a(L', [a], R'), last(R', X)$
$app_a([\,], [a], [a]) \leftarrow$
$app_a([H|X], [a], [H|Z]) \leftarrow app_a(X, [a], Z)$

in addition to the original clauses for $last/2$.

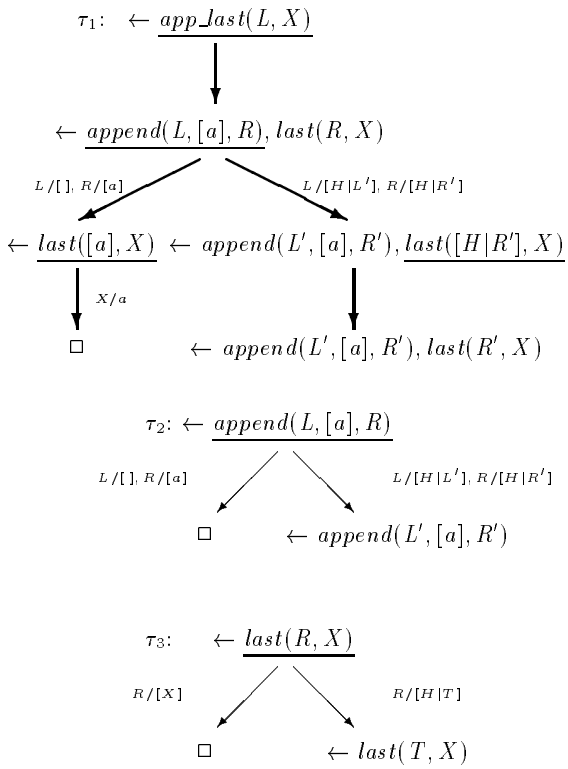


Figure 13.2: SLD-trees for Example 13.2.4

Unfortunately, in this case, even the combination with the more specific program transformation is insufficient to obtain the desired result. We get:

$T_P \uparrow 1 = U_P \uparrow 1 =$
$\{\ app\_last([\,], a),\ app_a([\,], [a], [a]),\ last([X], X)\ \}$
$T_P(U_P \uparrow 1) = T_P \uparrow 2 =$
$\{\ app\_last([\,], a),\ app\_last([H], a)$
$app_a([\,], [a], [a]),\ app_a([H], [a], [H, a]),$
$last([X], X), last([H, X], X)\ \}$

after which most specific generalisation yields

$U_P \uparrow 2 = \{\ app\_last(L, a),\ app_a(X, [a], [Y|Z]),\ last([X|Y], Z)\ \}$

At this stage, all information concerning the last elements of the lists is lost and we reach the fix-point in the next iteration:

$U_P \uparrow 3 = \{\ app\_last(L, Z),\ app_a(X, [a], [Y|Z]),\ last([X|Y], Z)\ \}$

One could argue that the failure is not due to the more specific programs transformation itself, but to a weakness of the most specific generalisation operator: it's inability to retain information at the end of a data-structure. Note however that even if we use other abstractions and their corresponding abstract operation proposed in the literature, such as *type-graphs* [134], *regular types* [102] or refined types for compile-time garbage collection of [208], the information still gets lost.

The heart of the problem is that in all these methods the abstract operator is applied to atoms of each predicate symbol *separately*. In this program (as well as in *many, much more relevant others*, as we will discuss later in this chapter), we are interested in analysing the conjunction $append(L, [a], R), last(R, X)$ with a linking intermediate variable (whose structure is too complex for the particular abstract domain). If we could consider this conjunction as a *basic unit* in the analysis, and therefore not perform abstraction on the separate atoms, but only on conjunctions of the involved atoms, we would retain a precise side-ways information passing analysis.

In Chapter 10 we have developed foundations for *conjunctive partial deduction*, which extends the standard partial deduction approach by considering a set of *conjunctions of atoms* instead of individual atoms. Although this extension of standard partial deduction was motivated by totally different concerns than the ones in the current chapter (the aim was to achieve a large class of unfold/fold transformations [222] within a simple extension of the partial deduction framework), experiments with conjunctive partial deduction on standard partial deduction examples (cf. Chapter 12) also showed significant improvements. These somewhat surprising optimisations are actually due to a considerably improved side-ways information-propagation.

Let us illustrate how conjunctive partial deduction combined with the more specific program transformation *does* solve Example 13.2.4. Starting

from the goal $app\_last(X)$ and using an analysis scheme similar to Algorithm 3.3.11, but with the role of atoms replaced by conjunctions of atoms, we can obtain $\mathcal{A} = \{\ app\_last(X),\ append(L,[a],R) \wedge last(R,X)\ \}$ and the corresponding SLD-trees, which are sub-trees of τ_1 of Figure 13.2. Here, "$\wedge$" is used to denote conjunction in those cases where "," is ambiguous.

The main difference with the standard partial deduction analysis above is that the goal $append(L',[a],R'),last(R',X)$ in the leaf of τ_1 is now considered as an undecomposed conjunction. This conjunction is already an instance of an element in $\mathcal{A}$, so that no separate analysis for $append(L',[a],R')$ and $last(R',X)$ are required.

Based on an atomic renaming α (cf. Chapter 10) such that

$$\alpha(append(x,y,z) \wedge last(z,u)) = al(x,y,z,u)$$

the resulting transformed program is:

$$app\_last(L,X) \leftarrow al(L,[a],R,X)$$
$$al([\,],[a],[a],a) \leftarrow$$
$$al([H|L'],[a],[H|R'],X) \leftarrow al(L',[a],R',X)$$

Applying the non-ground T_P-operator and more specific generalisation abstractions produces the sets:

$$U_P \uparrow 1 = \{\ al([\,],[a],[a],a)\}$$
$$U_P \uparrow 2 = \{\ al(X,[a],Y,a),\ app\_last(X,a)\}$$

which is a fix-point. Unifying the success-information with the body-atoms in the above program and performing (ordinary) filtering (cf. Chapter 3) and then redundant argument filtering (cf. Chapter 11) produces the desired more specific program:

$$app\_last(L,a) \leftarrow al(L)$$
$$al([\,]) \leftarrow$$
$$al([H|L']) \leftarrow al(L')$$

The abstract interpretation framework of [33] extends OLDT [270, 145] by performing tabling operations not on atoms but on conjunctions. This makes [33] powerful enough, given a proper way to construct the set of conjunctions (one of the things that conjunctive partial deduction can do), to also solve Example 13.2.4. The exact relationship between [33] and the combination of conjunctive partial deduction and more specific program construction is a subject for further study.

13.3 Some motivating examples

In this section we illustrate the relevance of the introduced techniques by more realistic, practical examples.

13.3.1 Storing values in an environment

The following piece of code P stores values of variables in an association list and is taken from a interpreter for imperative languages (see the imperative.power benchmark in Appendix C). Note that we use typewriter font for those programs containing built-ins.

```
store([],Key,Value,[Key/Value]).
store([Key/Value2|T],Key,Value,[Key/Value|T]).
store([K2/V2|T],Key,Value,[K2/V2|BT]) :-
    Key \= K2,store(T,Key,Value,BT).
lookup(Key,[Key/Value|T],Value).
lookup(Key,[K2/V2|T],Value) :-
    Key \= K2,lookup(Key,T,Value).
```

During specialisation it may happen that a known (static) value is stored in an unknown environment.[3] When we later on retrieve this value from the environment it is vital for good specialisation to be able to recover this static value. This is a problem quite similar to the *append-last* problem of Example 13.2.4. So again, calculating $msv(P)$ (even if we perform a magic-set transformation on P) does not give us any new information for a query like $\leftarrow$ `store(E,k,2,E1),lookup(E1,k,Val)`. To be able to solve this problem one needs again to combine abstract interpretation with conjunctive partial deduction (the latter will "deforest" [281] the intermediate environment E_1). The specialised program P' for the query $\leftarrow$ `store(E,k,2,E1),lookup(E1,k,Val)` using the ECCE (cf. Chapters 6 and 12) system with determinate unfolding is the following (a duplicate `k\=X1` has been removed in the third clause; the associated incomplete SLDNF-tree can be found in Figure 13.3):

```
store_lookup__1([],[k/2],2).
store_lookup__1([k/X1|X2],[key/2|X2],2).
store_lookup__1([X1/X2|X3],[X1/X2|X4],X5) :-
    k \= X1,store_lookup__1(X3,X4,X5).
```

If we now calculate $msv(P')$, we are able to derive that `Val` must have the value 2:

```
store_lookup__1([],[k/2],2).
store_lookup__1([k/X1|X2],[k/2|X2],2).
store_lookup__1([X1/X2|X3],[X1/X2,X4/X5|X6],2) :-
    k \= X1,store_lookup__1(X3,[X4/X5|X6],2).
```

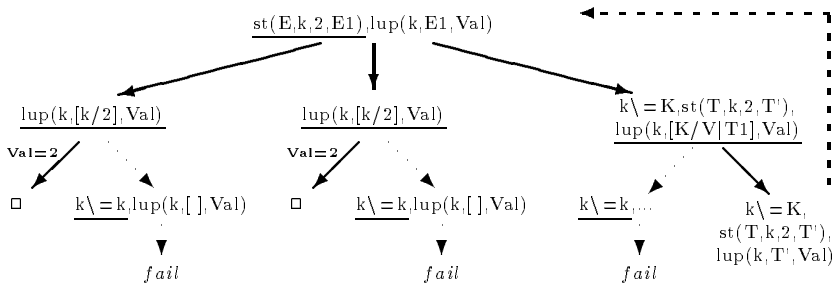


Figure 13.3: Unfolding part of an interpreter for an imperative language

Such information is of course even more relevant when one can continue specialisation with it. For instance, in an interpreter for an imperative language there might be multiple static values which are stored and then examined again. These values might control tests or loops, and for good specialisation to take place, the above information is crucial.

13.3.2 Proving functionality

The following is a generalisation of the standard definition of functionality (see e.g. [231] or [72]).

Definition 13.3.1 We say that a predicate p defined in a program P is *functional wrt the terms* $t_1, \ldots, t_h$ iff for every pair of atoms $A = p(t_1, \ldots, t_h, a_1, \ldots, a_k)$ and $B = p(t_1, \ldots, t_h, b_1, \ldots, b_k)$ we have:

- $\leftarrow A, B$ has a correct answer θ iff $\leftarrow A, A = B$ has
- $\leftarrow A, B$ finitely fails iff $\leftarrow A, A = B$ finitely fails

Note that in the above definition we allow A, B to be used as atoms as well as terms (as arguments to the predicate $= /2$). Also note that, for simplicity of the presentation, we restrict ourselves to correct answers (see Definition 2.3.3). Therefore it can be easily seen that,[4] if the goal $\leftarrow A', A'$ is a more specific version of $\leftarrow A, B$ then p is functional wrt $t_1, \ldots, t_h$ (because $msv(.)$ preserves computed answers and removing syntactically identical calls preserves the correct answers for definite logic programs).

Functionality is useful for many transformations, and is often vital to get super-linear speedups. For instance, it is needed to transform the naive

[3] The environment might be unknown for many reasons, e.g. due to abstraction or because part of the imperative program is unknown.

[4] The reasoning for computed answers is not so obvious.

(exponential) Fibonacci program into a linear one (see e.g. [231]). It can also be used to produce more efficient code (see e.g. [72]). Another example arises naturally from the `store-lookup` code of the previous section. For instance, often specialisation can be greatly improved if functionality of `lookup(Key,Env,Val)` wrt a given key `Key` and a given environment `Env` can be proven (in other words if we lookup the same variable in the same environment we get the same value). For instance, this would allow to replace, during specialisation, `lookup(Key,Env,V1),lookup(Key,Env,V2),p(V2)` by `lookup(Key,Env,V1),p(V1)`.

To prove functionality of `lookup(Key,Env,Val)` we simply add the following definition:[5]

```
ll(K,E,V1,V2) :- lookup(K,E,V1),lookup(K,E,V2).
```

By specialising the query `ll(Key,Env,V1,V2)` using the ECCE system with determinate unfolding and then calculating $msv(.)$ for the resulting program, we are able to derive that `V1` must be equal to `V2`:

```
ll(K,E,V,V) :- lookup_lookup__1(K,E,V,V).
lookup_lookup__1(X1,[X1/X2|X3],X2,X2).
lookup_lookup__1(X1,[X2/X3,X4/X5|X6],X7,X7) :-
    X1 \= X2, lookup_lookup__1(X1,[X4/X5|X6],X7,X7).
```

In addition to obtaining a more efficient program the above implies (because conjunctive partial deduction preserves the computed answers) that the conjunction `lookup(K,E,V),lookup(K,E,V)` is a more specific version of `lookup(K,E,V1),lookup(K,E,V2)`, and we have proven functionality of `lookup` wrt the first two arguments `Key,Env`.

In the same vein we can for example prove functionality of $plus/3$ wrt the first two arguments X and Y. Starting out from the initial program

$pp(X,Y,Z1,Z2) \leftarrow plus(X,Y,Z1), plus(X,Y,Z2)$
$plus(0,Z,Z) \leftarrow$
$plus(s(X),Y,s(Z)) \leftarrow plus(X,Y,Z)$

we obtain after specialising the following program:

$pp(X,Y,V,V) \leftarrow plus\_plus(X,Y,V,V)$
$plus\_plus(0,X1,X1,X1) \leftarrow$
$plus\_plus(s(X1),X2,s(X3),s(X3)) \leftarrow plus\_plus(X1,X2,X3,X3)$

13.3.3 The need for a more refined integration

So far we have always completely separated the conjunctive partial deduction phase and the bottom-up abstract interpretation phase. The following example shows that this is not always sufficient.

[5] This is not strictly necessary but it simplifies spotting functionality.

Take a look at the following excerpt from a unification algorithm for the ground representation, which takes care of extracting variable bindings out of (uncomposed) substitutions. The full code can be found in Appendix H.2. Notice that the algorithm does not not use accumulating parameters and delays composition as well as application of substitutions as long as possible. Also for simplicity we have not added an occurs check (adding an occur check will only increase the precision of our analysis).

```
get_binding(V,empty,var(V)).
get_binding(V,sub(V,S),S).
get_binding(V,sub(W,S),var(V)) :- V \= W.
get_binding(V,comp(L,R),S) :-
    get_binding(V,L,VL), apply(VL,R,S).

apply(var(V),Sub,VS) :- get_binding(V,Sub,VS).
apply(struct(F,A),Sub,struct(F,AA)) :- l_apply(A,Sub,AA).

l_apply([],Sub,[]).
l_apply([H|T],Sub,[AH|AT]) :-
    apply(H,Sub,AH),l_apply(T,Sub,AT).
```

At first sight this piece of code looks very similar to the example of the previous section and one would think that we could easily prove functionality of `get_binding(VarIdx,Sub,Bind)` wrt a particular variable index `VarIdx` and a particular substitution `Sub`. Exactly this kind of information is required for the applications in Section 13.5 (related to the precompilation of integrity checking for recursive databases).

Unfortunately this kind of information *cannot* be obtained by fully separated out phases, even if we systematically apply the $msv(.)$ once a new SLD-tree has been constructed. For simplicity we assume that the variable index `VarIdx` is known to be 1. As in the previous section, we add the definition:

```
gg(Sub,V1,V2) :- get_binding(1,Sub,V1),get_binding(1,Sub,V2).
```

If we construct one SLD-tree for the conjunction `get_binding(1,Sub,V1)`, `get_binding(1,Sub,V2)` and then apply renaming followed by $msv(.)$, we obtain:

```
gg(Sub,V1,V2) :- get_binding_get_binding__1(Sub,V1,V2).
get_binding_get_binding__1(empty,var(1),var(1)).
get_binding_get_binding__1(sub(1,X1),X1,X1).
get_binding_get_binding__1(sub(X1,X2),var(1),var(1)) :- 1\=X1.
```

```
get_binding_get_binding__1(comp(X1,X2),X3,X4) :-
    get_binding_get_binding__1(1,X1,X5,X6),
    apply(X5,X2,X3),apply(X6,X2,X4).
```

Observe that we have not yet established the desired functionality; the problem lies with the fourth clause for `get_binding_get_binding__1` (for the other three clauses the second and third arguments are already identical). Unfortunately, by applying conjunctive partial deduction to the body `apply(X5,X2,X3),apply(X6,X2,X4)` of this clause we cannot derive that `X3` must be equal to `X4`. Indeed, the variables indexes `X5` and `X6` are different and applying the same substitution on different terms can of course lead to differing results. It would only be possible to prove that `X3=X4` if we assume that `X5=X6`. So, we seem to be trapped in a dilemma: to be able to prove functionality we must assume that it holds.

However, the problem simply stems from the fact that we apply conjunctive partial deduction too late, namely only *after* the fixpoint of U_P has been reached. Indeed, after the first application of U_P we obtain $\mathcal{A} = U_P(\emptyset) = \{$`get_binding_get_binding__1(S,V,V)`$\}$, So $\mathcal{A}$ actually contains the assumption that functionality holds, and we have that $msv_{\mathcal{A}}(.)$ of the problematic clause looks like:

```
get_binding_get_binding__1(comp(X1,X2),X3,X4) :-
    get_binding_get_binding__1(1,X1,V,V),
    apply(V,X2,X3),apply(V,X2,X4).
```

If we *now* re-apply conjunctive partial deduction to `apply(V,X2,X3)`, `apply(V,X2,X4)` and then similarly in a next iteration to the conjunction `l_apply(V,X2,X3),l_apply(V,X2,X4)` we can derive functionality of `get_binding`. In summary, it is vital to provide for a finer integration of conjunctive partial deduction and more specific program transformation, which re-applies conjunctive partial deduction *before* the fixpoint of U_P has been reached. The details of this more refined integration are elaborated in the next section.

13.4 A more refined algorithm

We now present an algorithm which interleaves the least fixpoint construction of $msv(.)$ with conjunctive partial deduction unfolding steps. For that we have to adapt the more specific program transformation to work on possibly incomplete SLD-trees obtained by conjunctive partial deduction instead of for completely constructed programs.[6]

[6] This has the advantage that we do not actually have to apply a renaming transformation (and we might get more precision because several conjunctions might match).

We first introduce a special conjunction $\bot$ which is an instance of every conjunction, as well as the only instance of itself, and extend the *msg* such that $msg(S \cup \{\bot\}) = msg(S)$ and $msg(\{\bot\}) = \bot$. We also use the convention that if unification fails it returns a special substitution $fail$. Applying $fail$ to any conjunction in turn yields $\bot$. Finally, by $\uplus$ we denote the concatenation of tuples (e.g. $\langle a \rangle \uplus \langle b, c \rangle = \langle a, b, c \rangle$).

In the following definition we associate conjunctions with resultants:

Definition 13.4.1 (resultant tuple) Let $\mathcal{Q} = \{Q_1, \ldots, Q_s\}$ be a set of conjunctions of atoms, and $T = \{\tau_1, \ldots, \tau_s\}$ a set of finite, non-trivial SLD-trees for $P \cup \{\leftarrow Q_1\}, \ldots, P \cup \{\leftarrow Q_s\}$, with associated sets of resultants $R_1, \ldots, R_s$, respectively.
Then the tuple of pairs $RS = \langle (Q_1, R_1), \ldots, (Q_s, R_s) \rangle$ is called a *resultant tuple* for P. An *interpretation* of RS is a tuple $\langle Q'_1, \ldots, Q'_s \rangle$ of conjunctions such that each Q'_i is an instance of Q_i.

The following defines how interpretations of resultant tuples can be used to create more specific resultants:

Definition 13.4.2 (refinement) Let $R = H \leftarrow Body$ be a resultant and $I = \langle Q'_1, \ldots, Q'_s \rangle$ be an interpretation of a resultant tuple $RS = \langle (Q_1, R_1), \ldots, (Q_s, R_s) \rangle$. Let Q be a sub-conjunction of $Body$ such that Q is an instance of Q_i and such that $mgu^*(Q, Q'_i) = \theta$. Then $R\theta$ is called a *refinement* of R under RS and I. R itself, as well as any refinement of $R\theta$, is also called a refinement of R under RS and I.

Below we denote by $ref_{RS,I}(R)$, a particular refinement (e.g. the least one) of R under RS and I.

Note that a least refinement always exists.[7] Indeed, once we have unified a particular sub-conjunction Q of a resultant R with a particular Q_i, thus obtaining the refinement $R\theta$, it is of no use to unify $Q\theta$ (or an instance of it) again with Q_i (as it will result in no further refinement of $R\theta$). So, as there are only finitely many sub-conjunction and only finitely many conjunctions Q_i, the least refinement must exist.

Note that in [191, 192], it is not allowed to further refine refinements. As we found out through several examples however, (notably the ones of Section 13.3.3 and Section 13.5) this approach turns out to be too restrictive in general. In a lot of cases, applying a first refinement might instantiate R in such a way that a previously inapplicable element of RS can now be used for further instantiation.

We can now extend the U_P operator of Definition 13.2.1 to work on interpretations of resultant tuples:

[7] In [181] it is wrongly claimed that this is not the case.

Definition 13.4.3 ($\mathbf{U}_{P,RS}$) Let $I = \langle Q'_1, \ldots, Q'_s \rangle$ be an interpretation of a resultant tuple $RS = \langle (Q_1, R_1), \ldots, (Q_s, R_s) \rangle$. Then $U_{P,RS}$ is defined by $U_{P,RS}(I) = \langle M_1, \ldots, M_s \rangle$, where $M_i = msg(\{H \mid C \in R_i \wedge ref_{RS,I}(C) = H \leftarrow B\})$.

We refer the reader to Example 13.4.6 and Table 13.1 below for illustrations of the above concepts.

We can now present a generic algorithm which fully integrates the abstract interpretation $msv(.)$ with conjunctive partial deduction. Recall that $=_r$ denotes identity, up to reordering. We also suppose that we have an abstraction operator (see Definition 12.1.4) *abstract* at our disposal (cf. Section 12.1.1).

We also need the following definition:

Definition 13.4.4 (covered) Let $RS = \langle (Q_1, R_1), \ldots, (Q_s, R_s) \rangle$ be a resultant tuple. We say that a conjunction Q is *covered* by RS iff there exists an abstraction $\{Q'_1, \ldots, Q'_k\}$ of Q such that each Q'_i is an instance of some conjunction Q_j.

We can now present the promised algorithm.

Algorithm 13.4.5 (Conjunctive Msv)

Input: a program P, an initial query Q, an unfolding rule *Unf* for P mapping conjunctions to resultants.
Output: A specialised and more specific program P' for Q.
Initialisation: $i := 0$; $I_0 = \langle \bot \rangle$; $RS_0 = \langle (Q, Unf(Q)) \rangle$;
repeat
  for every resultant R in RS_i such that the body B of $ref_{RS_i,I_i}(R)$ is not covered:
    /* perform conjunctive partial deduction: */
      calculate $abstract(B) = B_1 \wedge \ldots \wedge B_q$
      let $\{C_1, \ldots, C_k\}$ be the B_j's which are not instances[8] of conjunctions in RS_i;
      $RS_{i+1} = RS_i \uplus \langle (C_1, Unf(C_1)), \ldots (C_k, Unf(C_k)) \rangle$;
      $I_{i+1} = I_i \uplus \langle \underbrace{\bot \ldots \bot}_{k} \rangle$; $i := i + 1$;
  /* perform one bottom-up propagation step: */
  $I_{i+1} = U_{P,RS_i}(I_i)$; $RS_{i+1} = RS_i$; $i := i + 1$;
until $I_i = I_{i-1}$
return a renaming of $\{ref_{RS_i,I_i}(C) \mid (Q, R) \in RS_i \wedge C \in R\}$

[8] Or *variants* to make the algorithm more precise.

Note that the above algorithm ensures coveredness. Also note that the above algorithm performs abstraction only when adding new conjunctions, the existing conjunctions are not abstracted (it is of course trivial to adapt this). This is like in Algorithm 6.2.36 of Chapter 6 but unlike e.g. Algorithm 3.3.11.

Example 13.4.6 We now illustrate Algorithm 13.4.5 by proving functionality of $mul(X, Y, Z1)$, $mul(X, Y, Z2)$ for the following program. Note that the general picture is very similar to showing functionality of `get_binding`, but leading to a shorter and simpler presentation.

$mul(0, X, 0) \leftarrow$
$mul(s(X), Y, Z) \leftarrow mul(X, Y, XY), plus(XY, Y, Z)$
$plus(0, X, X) \leftarrow$
$plus(s(X), Y, s(Z)) \leftarrow plus(X, Y, Z)$

R_1	$mul(0, Y, 0), mul(0, Y, 0).$
R_2	$mul(s(X), Y, Z1), mul(s(X), Y, Z2) \leftarrow$ $\quad mul(X, Y, XY1), plus(XY1, Y, Z1),$ $\quad mul(X, Y, XY2), plus(XY2, Y, Z2)$
R_2'	$mul(s(0), Y, Z1), mul(s(0), Y, Z2) \leftarrow$ $\quad mul(0, Y, 0), plus(0, Y, Z1), mul(0, Y, 0), plus(0, Y, Z2)$
R_2''	$mul(s(0), Y, Y), mul(s(0), Y, Y) \leftarrow$ $\quad mul(0, Y, 0), plus(0, Y, Y), mul(0, Y, 0), plus(0, Y, Y)$
R_2'''	$mul(s(X), Y, Z1), mul(s(X), Y, Z2) \leftarrow$ $\quad mul(X, Y, Z), plus(Z, Y, Z1), mul(X, Y, Z), plus(Z, Y, Z2)$
R_2''''	$mul(s(X), Y, V), mul(s(X), Y, V) \leftarrow$ $\quad mul(X, Y, Z), plus(Z, Y, V), mul(X, Y, Z), plus(Z, Y, V)$
R_3	$plus(0, Y, Y), plus(0, Y, Y).$
R_4	$plus(s(X), Y, Z1), plus(s(X), Y, Z2) \leftarrow$ $\quad plus(X, Y, Z1), plus(X, Y, Z2)$

Table 13.1: Resultants and refinements

The resultants R_1, R_2, R_3, R_4 and their refinements for this example can be found in Table 13.1. Using an abstraction based on [110] and a determinate unfolding rule (see e.g. [100, 97, 172]) we obtain the following behaviour of Algorithm 13.4.5.

1. Initialisation:
   $I_0 = \langle \bot \rangle$, $RS_0 = \langle (mul(X, Y, Z1) \wedge mul(X, Y, Z2), \{R_1, R_2\}) \rangle$
2. $ref_{RS_0, I_0}(R_2) = \bot$ and therefore all bodies are covered
3. We perform a bottom-up propagation step:
   $RS_1 = RS_0$, $I_1 = U_{P, RS_0}(I_0) = \langle mul(0, Y, 0) \wedge mul(0, Y, 0) \rangle \neq I_0$

4. Now $ref_{RS_1,I_1}(R_2) = R_2'$ and *abstract* of the body of R_2' yields:
   $\{mul(0, Y, 0) \wedge mul(0, Y, 0),\ plus(0, Y, Z1) \wedge plus(0, Y, Z2)\}$
   and we obtain:
   $I_2 = I_1 \uplus \langle \bot \rangle$, $RS_2 = RS_1 \uplus \langle\ (plus(0, Y, Z1) \wedge plus(0, Y, Z2),\ \{R_3\})\rangle$
5. We now go on with the bottom-up propagation:
   $RS_3 = RS_2$, $I_3 = U_{P,RS_2}(I_2) =$
   $\langle mul(0, Y, 0) \wedge mul(0, Y, 0),\ plus(0, Y, Y) \wedge plus(0, Y, Y)\rangle$
6. The body of $ref_{RS_3,I_3}(R_2) = R_2''$ is covered and we go on with the bottom-up propagation: $RS_4 = RS_3$, $I_4 = U_{P,RS_3}(I_3) =$
   $\langle mul(X, Y, Z) \wedge mul(X, Y, Z),\ plus(0, Y, Y) \wedge plus(0, Y, Y)\rangle$
7. Now $ref_{RS_4,I_4}(R_2) = R_2'''$ is no longer covered.
   Applying *abstract* to the body of R_2''' yields:
   $\{mul(X, Y, Z) \wedge mul(X, Y, Z),\ plus(Z, Y, Z1) \wedge plus(Z, Y, Z2)\}$
   and $I_5 = I_4 \uplus \langle \bot \rangle$,
   $RS_5 = RS_4 \uplus \langle\ (plus(Z, Y, Z1) \wedge plus(Z, Y, Z2),\ \{R_3, R_4\})\rangle$
8. We do a bottom-up propagation step:
   $RS_6 = RS_5$, $I_6 = U_{P,RS_5}(I_5) =$
   $\langle mul(X, Y, Z) \wedge mul(X, Y, Z),\ plus(0, Y, Y) \wedge plus(0, Y, Y),$
   $plus(Z, Y, V) \wedge plus(Z, Y, V)\rangle$
9. The bodies of $ref_{RS_6,I_6}(R_2) = R_2''''$ and $ref_{RS_6,I_6}(R_4)$ are covered and we have reached the fixpoint: $I_7 = U_{P,RS_6}(I_6) = I_6$.

The final specialised program is as follows (one unreachable conjunction has been removed, $mul(X, Y, Z1)$, $mul(X, Y, Z2)$ has been renamed to $mul\_mul(X, Y, Z1, Z2)$ and $plus(X, Y, Z1)$, $plus(X, Y, Z2)$ has been renamed to $plus\_plus(X, Y, Z1, Z2)$; the program could be further improved by a better renaming) and functionality is obvious:

$$
\begin{array}{l}
mul\_mul(0, Y, 0, 0) \leftarrow \\
mul\_mul(s(X), Y, Z, Z) \leftarrow \\
\quad mul\_mul(X, Y, XY, XY), plus\_plus(XY, Y, Z, Z) \\
plus\_plus(0, X, X, X) \leftarrow \\
plus\_plus(s(X), Y, s(Z), s(Z)) \leftarrow plus\_plus(X, Y, Z, Z)
\end{array}
$$

Note that when using the definitions of [191, 192] the least refinement of R_2 wrt RS_6 and I_6 is not R_2''' (because $plus(Z, Y, Z') \wedge plus(Z, Y, Z')$ cannot be applied) but R_2''. Hence the fixpoint is not reached and in the next iteration the vital functionality information would be lost!

Correctness of Algorithm 13.4.5 for preserving the least Herbrand model or even the computed answers, follows from correctness of conjunctive partial deduction (Theorem 10.3.16) and of the more specific program versions for suitably chosen conjunctions (because [191, 192] only allows one unfolding step, a lot of intermediate conjunctions have to be introduced) and extended for the more powerful refinements of Definition 13.4.2. Termination,

for a suitable abstraction operator (see [110]), follows from termination of conjunctive partial deduction (for the **for** loop) and termination of $msv(.)$ (for the **repeat** loop).

Note that in contrast to conjunctive partial deduction, $msv(.)$ can replace infinite failure by finite failure, and hence Algorithm 13.4.5 does not preserve finite failure. However, if the specialised program fails infinitely, then so does the original one (see [191, 192]).

The above algorithm can be extended to work for normal logic programs. But, because finite failure is not preserved, neither are the SLDNF computed answers. One may have to look at SLS [236] for a suitable procedural semantics which is preserved.

13.5 Specialising the ground representation

> Weil Etwas für uns durchsichtig geworden ist, meinen wir, es könne uns nunmehr keinen Widerstand leisten — und sind dann erstaunt, dass wir hindurchsehen und doch nicht hindurch können! Es ist diess die selbe Thorheit und das selbe Erstaunen, in welches die Fliege vor jedem Glasfenster geräth.
>
> *Friedrich Nietzsche* in *Morgenröthe, Nr. 444;3,270*

In this section we pursue the idea of pre-compiling integrity checking for deductive databases (as well as abductive or inductive logic programs) in a principled and non ad-hoc way, by writing the integrity checking as a meta-interpreter and then specialising it for given update patterns.

In Chapter 9 we developed this idea for hierarchical databases and obtained very promising results. However, when going to recursive databases, this meta-interpreter must contain a loop check (or one has to delegate the loop check to the underlying system, see the discussions in Section 9.4). As we pointed out in Chapter 8, this can only be done declaratively within the ground representation. But in [177] it was shown that, contrary to what one might expect, partial deduction is then unable to perform interesting specialisation and no pre-compilation of integrity checks can be obtained.[9] Similar problems related to specialising the ground representation were also reported in [67]. This lack of specialisation of the ground representation is precisely due to the limitations of partial deduction we have pointed out in this chapter.

[9] So, to paraphrase the above quote by Nietzsche, it is not because we have formalised something in a purely declarative manner that it no longer gives any resistance.

The crucial problem identified in [177] boils down to a lack of information propagation and specialisation at the object level. As already mentioned in Chapter 8, the ground representation has to make use of an explicit unification algorithm (see e.g. Appendix H.1 or Appendix H.2). Now, the incapability of partial deduction techniques to e.g. prove functionality of parts of such an explicit unification algorithm translates to a serious lack of specialisation. Take a meta-interpreter which implements specialised integrity checking as outlined in Section 8.2. To calculate the set of positive potential updates $pos(U) = \bigcup_{i \geq 0} pos^i(U)$ for an update $U = \langle Db^+, Db^=, Db^- \rangle$, the meta-interpreter will (among others) select an atom $C \in pos^i(U)$, unify it with an atom B in the body of a clause $A \leftarrow \ldots, B, \ldots \in Db^=$ and then apply the unifier to the head A to obtain an induced, potential update of $pos^{i+1}(U)$. At partial deduction time, the atoms A, B and C are in general not fully known. If we want to obtain effective specialisation, it is vital that the information we do possess about C (and B) is propagated "through" unification towards A. If this knowledge is not carried along no substantial compilation will occur and it will be impossible to obtain efficient specialised update procedures.

In other words, we are interested in deriving properties of the result `Res` of calculating `unify(A,B,S),apply(H,S,Res)`. In a concrete example we might have `A = status(X,student,Age)`, `B = status(ID,E,A)`, `H = category(ID,E)` and we would like to derive that the resulting term `Res` must be an instance of `category(ID',student)`. It turns out that, when using an explicit unification algorithm, the substitutions have a much more complex structure than e.g. the intermediate list of the *append-last* Example 13.2.4. Therefore current abstract interpretation systems, as well as current partial deduction methods alone, fail to derive the desired information.

This problem was solved in [177] via a new implementation of the ground representation combined with a custom specialisation technique. Fortunately Algorithm 13.4.5 can solve this information propagation problem in a more general and sometimes more precise manner and therefore contribute to improved specialisation of the ground representation as well as to produce highly specialised and efficient pre-compiled integrity checks.

Some experiments, conducted with a prototype implementation of Algorithm 13.4.5 based on the ECCE system (cf. Chapters 6 and 12), are summarised in Table 13.2. The unification algorithm of Appendix H.2 has been used, which encodes variables as `var(VarIndex)` and predicates/functors as `struct(p,Args)`. Notice that all the examples were successfully solved by the prototype.[10] The main ingredient of the success lay with proving

[10] Notice that for the fourth example it would be incorrect to derive `Res` =

functionality of `get_binding`.

Also note that the information propagations of Table 13.2 could neither be solved by regular approximations [102], nor by the abstract interpretation method of [191, 192] alone, nor by set-based analysis [118] nor even by current implementations of the type graphs of [134]. In summary, Algorithm 13.4.5 also provides for a powerful abstract interpretation scheme as well as a full replacement of the custom specialisation technique in [177].[11]

unify(A,B,S),apply(H,S,Res)			
A	B	H	Res
struct(p,[var(1),X])	struct(p,[struct(a,[]),Y])	var(1)	struct(a,[])
struct(p,[X,var(1)])	struct(p,[Y,struct(a,[])])	var(1)	struct(a,[])
struct(p,[X,X])	struct(p,[struct(a,[]),Y])	X	struct(a,[])
struct(F,[var(I)])	X	X	struct(F,[A])
struct(p,[X,var(1),X'])	struct(p,[Y,struct(a,[]),Y'])	var(1)	struct(a,[])

Table 13.2: Specialising the ground representation

We conclude this section by briefly mentioning another potential benefit of the improved specialisation of the ground representation, namely to perform the first *specialiser projection* as defined in [108]. Indeed, whereas the first Futamura projection specialises a meta-interpreter for a known meta-program but unknown object level parameters, the first specialiser projection specialises a meta-interpreter for a known meta-program and *partially known* object level parameters. Therefore, propagating the partial knowledge one possesses at the object level seems to be vital for this projection and Algorithm 13.4.5 might thus contribute to make the first specialiser projection feasible for meta-interpreters written in the ground representation.

13.6 Discussion

The approach presented in this chapter can be seen as a practical realisation of a combined backwards and forwards analysis as outlined in [57], but using the sophisticated control techniques of (conjunctive) partial deduction to guide the analysis. Of course, in addition to analysis, our approach also constructs a specialised, more efficient program.

`struct(F,[var(I)])`. For example, if we have that `B = X = struct(f,struct(a,[]))` then `Res = X = struct(f,struct(a,[]))`.

[11] It is sometimes even able to provide better results because it can handle structures with unknown functors or unknown number of arguments with no loss of precision.

The method of [191, 192] is not directly based on the T_P operator, but uses an operator on goal tuples which can handle conjunctions and which is sufficiently precise if deforestation can be obtained by 1-step unfolding without abstraction. For a lot of practical examples this will of course not be the case. Also, apart from a simple pragmatic approach, no way to obtain these conjunctions is provided (this is exactly one of the things which conjunctive partial deduction can do). We also already mentioned a drawback in the calculation of refinements, which makes [191, 192] unsuitable to e.g. derive functionality of `get_binding` or *mul*.

In Algorithm 13.4.5 a conflict between efficiency and precision might arise. Indeed, as we have seen in Chapter 12, some deforestation can only be obtained at the cost of possible slowdowns. But Algorithm 13.4.5 can be easily extended to allow different trees for the same conjunction (e.g. use determinate unfolding for efficient code and a more liberal unfolding for a precise analysis). A similar point was raised in Section 5.4.1 for the ecological partial deduction algorithm.

When using the unification algorithm from Appendix H.1, instead of the one in Appendix H.2, Algorithm 13.4.5 cannot yet handle all the examples of Table 13.2. The reason is that the substitutions in Appendix H.1 are actually *accumulating* parameters which are first fully generated before they can be consumed! Deforestation of accumulators is still an open research problem (for functional languages, first, not yet automatic, approaches can be found in [275], see also [163] for a discussion of accumulators in the context of partial evaluation). Let us adapt Example 13.2.4 into the *reverse-last* example:

Example 13.6.1 (reverse-last)

$rev\_last(L, X) \leftarrow reverse(L, [a], R), last(R, X)$
$reverse([\,], L, L) \leftarrow$
$reverse([H\,|\,T], Acc, Res) \leftarrow reverse(T, [H|Acc], Res)$
$last([X], X) \leftarrow$
$last([H\,|\,T], X) \leftarrow last(T, X)$

In the above program *reverse* is written using an accumulating parameter, and in that case neither conjunctive partial deduction nor any unfold/fold method we know of can deforest the intermediate variable `R`. Unfolding the goal $reverse(L, [a], R), last(R, X)$ is depicted in Figure 13.4. Notice that no matter how we unfold we cannot obtain a recursive definition. Conjunctive partial deduction would detect the growing of the accumulator and produce the abstraction $reverse(L, A, R), last(R, X)$. Unfolding can now produce a recursive definition, as can be seen in Figure 13.5. However, the partial input a has been abstracted away, and we are not able to deduce that $X = a$.

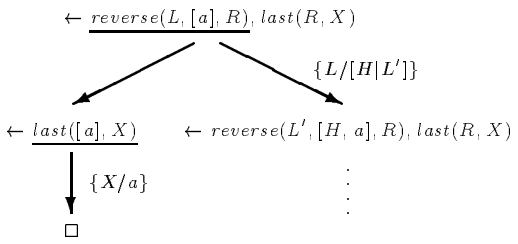


Figure 13.4: Unfolding of Example 13.6.1

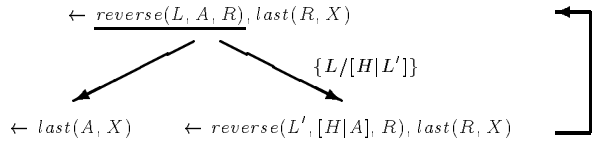


Figure 13.5: Unfolding of a generalisation of Example 13.6.1

On the other hand, if we use the naive reverse predicate without an accumulator, defined by:

$$nrev([\,],[\,]) \leftarrow$$
$$nrev([H\,|\,T], Res) \leftarrow nrev(\,T,\,TR), append(\,TR, [H], Res)$$

then, after unfolding $nrev([a|L], R)$, $last(R, X)$ once, we obtain the conjunction $nrev(L, TR)$, $append(\,TR, [a], R)$, $last(R, X)$. We are now in the situation of the *append-last* Example 13.2.4 and $X = a$ can be easily obtained via Algorithm 13.4.5.

In future work we want to explore solutions to the *reverse-last* problem, based on adding constraints to Algorithm 13.4.5 (in the similar way to constrained partial deduction [172], briefly discussed in Section 5.4.2). These constraints might provide an accurate description of the growth of the accumulator and allow generalisation without loosing the information that $X = a$. Similar refinments of extended OLDT [33] might also yield satisfactory solutions.

In conclusion, in this chapter we have illustrated limitations of both partial deduction and abstract interpretation on their own. We have argued for a tighter integration of these methods and presented a refined algorithm, interleaving a least fixpoint construction with conjunctive partial deduction. The practical relevance of this approach has been illustrated by several examples and we have shown its usefulness in proving functionality. Finally, a prototype implementation of the algorithm was able to achieve sophisticated specialisation *and* analysis for meta-interpreters written in the ground representation, outside the reach of current specialisation or abstract interpretation techniques.

Chapter 14

Conclusion and Outlook

The overall motivation of this thesis has been to promote (logic) program specialisation as a viable, automatic tool for software development and optimisation. The pursuit of this goal has taken on different forms, reflected in the division of the thesis into several parts.

- Part II investigated the problematic question of when is it sensible to generate different specialised versions for a particular predicate and when is it sensible to perform abstraction instead. For this recurring, difficult problem, termed the control of polyvariance problem, we presented the advantages of characteristic trees over a purely syntactic approach. We thereafter illustrated and solved the problems with existing approaches in terms of precision and termination. A framework, called ecological partial deduction, was developed whose correctness was formally proven. Termination was established as well, albeit at the expense of a depth bound on characteristic trees. Chapter 6 attended to the intricate problem of getting rid of this ad-hoc depth bound while keeping the termination and precision properties. A concrete algorithm has been developed, whose correctness and termination have been proven. An implementation was used to conduct extensive experiments and validate the practical usefulness of the method.

  Outlook: The control of polyvariance problem occurs in different disguises in many areas of program analysis, manipulation and optimisation. Similarly, depth bounds have to be imposed on a lot of these techniques to ensure termination. This is for instance the case for neighbourhoods in supercompilation. It is therefore our hope that our techniques can be adapted for other (declarative) programming

paradigms and that they might prove equally useful in the context of e.g. abstract interpretation systems or optimising compilers.

- Self-application is a very elegant concept with many applications. It allows e.g. to automatically build compilers and compiler generators from interpreters. In theory all one needs is a specialiser which is able to optimise itself. Despite its promises however, self-application has up to now not been very successful for logic programming languages. In Part III we overcame that situation, using the so called "cogen approach": instead of writing a self-applicable specialiser we simply wrote the compiler generator directly. In that way one gets the benefits of self-application without having to devise a self-applicable specialiser. In the context of logic programming this has, for instance, the advantage that the non-ground representation can be used. This resulted in a very efficient compiler generator which in turn produced very efficient compilers which, for some applications at least, perform very good optimisation.

  Outlook: The methods developed in Part III should be valuable whenever the speed of specialisation is of primary concern, for instance when the program to be specialised is frequently adapted. Also, run-time specialisation has recently attracted a lot of attention, and our approach might prove to be very useful and efficient in that context too.

- Part IV described a "success story" of program specialisation, of which there are not yet many in the context of logic programming. The central idea was to optimise integrity checking upon updates in deductive databases by program specialisation. To that end the integrity checking procedure was written as a meta-program which was then specialised for certain transaction patterns. We were able to automatically obtain very efficient specialised update procedures, executing up to 2 orders of magnitude faster than the unspecialised checker as well as substantially faster than other integrity checking procedures proposed in the literature.

  Outlook: Apart from being an example of the systematic use of a partial evaluator as a programming tool, the approach in this part of the thesis opens other possibilities by itself. Indeed, integrity checking is important for abductive and inductive logic programs as well. Using the described techniques one might thus automatically generate specialised update procedures for abducible (or inducible) predicates, hopefully leading to similar improvements in performance.

- In Part V we succeeded in augmenting the power of partial deduction. Indeed, partial deduction was heretofore incapable of performing certain useful unfold/fold transformations, like tupling or deforestation. We developed the framework of conjunctive partial deduction which, by specialising conjunctions instead of individual atoms, is able to accommodate these optimisations. We have presented concrete algorithms and showed that a lot of the techniques developed for "classical" partial deduction carry over to the context of conjunctive partial deduction. An implementation was used to perform extensive experiments on a large and challenging set of benchmarks. Although some control problems remain, we were able to demonstrate the practical viability and potential of conjunctive partial deduction.

  So we were able to consolidate partial deduction with unfold/fold program transformation, incorporating the power of the latter while keeping the automatic control and efficiency considerations of the former.

  Outlook: Deforestation and tupling like transformation are useful even in the absence of partial input. This warrants the integration of our techniques into a compiler, as their systematic use might prove to be highly beneficial and allow users to more easily decompose and combine procedures and programs without having to worry about the ensuing inefficiencies of intermediate data structures.

- Partial deduction, as well as conjunctive partial deduction, concentrate on a top-down propagation of information. Almost no information is propagated upwards. Abstract interpretation, on the other hand, only has (if at all) top-down propagation of information which is limited in several important aspects, and usually propagates information in a bottom-up manner. In Part VI we illustrate that this unnecessarily limits the power of both these program manipulation methods and that a combination of these techniques might therefore be extremely useful in practice. We instantiate that claim by developing a fine-grained algorithm, which interleaves top-down (conjunctive) partial deduction steps with bottom-up abstract interpretation steps, and by showing that the algorithm is able to obtain specialisation and analysis outside the reach of either method alone.

To put this thesis in a larger perspective, I feel that the time has come to lift program specialisation towards more widespread practical use and realise its potential as a tool for systematic program development. I hope that this thesis has contributed in that direction, by providing sound theoretical underpinnings for a wide variety of specialisation tasks, complemented with

concrete algorithms and methods whose practical significance was gauged by empirical evaluations.

This endeavour goes hand in hand with making the promises of declarative programming languages come true in practice. Indeed, most program specialisation and analysis methods flourish in the context of declarative programming languages and they can in turn help declarative programming languages to become a platform for the development of reliable, correct and efficient programs. The fulfilment of these goals will require practical work, e.g. in the form of programming environments which allow the user to take advantage of the analysis and specialisation methods, as well as theoretical work, e.g. providing a clear methodology for constructing correct and reliable programs.

Appendix A

Notations for Some Basic Mathematical Constructs

In this appendix we present some notations and definitions which we do not explicitly recall in the thesis.

Following standard conventions we use *iff* to denote "if and only if" and *wrt* to denote " with respect to". In this section we present notations for elementary mathematics that will be used throughout the thesis.

A.1 Sets and relations

Sets can be represented by enumeration, like e.g. $\{a, b, c\}$, or by a description like $\{i \mid i \in I\!N \wedge i > 2\}$. The latter set can also be denoted by the infinite enumeration $\{2, 3, \ldots\}$.

We denote by $\#(S)$ the *cardinality* of a set. The difference of two sets S_1 and S_2 will be denoted by $S_1 \setminus S_2$. The *cartesian product* of two sets A and B will be denoted by $A \times B$. For instance, $\{1,2\} \times \{3,4\} = \{(1,3),(1,4),(2,3),(2,4)\}$. The *powerset* over some domain D will be denoted by 2^D.

The following class of mappings from sets to sets often arises:

Definition A.1.1 A mapping $h : 2^A \mapsto 2^B$ is a *homomorphism* iff $h(\emptyset) = \emptyset$ and $h(S \cup S') = h(S) \cup h(S')$.

Definition A.1.2 Given a function $f : A \mapsto B$ the *natural extension* of f to sets, $f^* : 2^A \mapsto 2^B$, is defined by $f^*(S) = \{f(s) \mid s \in S\}$.
Similarly, given a function $f : A \mapsto 2^B$ we also define the function $f_\cup : 2^A \mapsto 2^B$, by $f_\cup(S) = \cup_{s \in S} f(s)$.

It can be easily seen that both f^* and $f_\cup$ are homomorphisms.

Given some domain/set D, we denote by D^n the following cartesian product: $\underbrace{D \times \ldots \times D}_{n}$. We will denote elements of D^n by $(d_1, \ldots, d_n)$ instead of the more cumbersome $(d_1, (d_2, \ldots, d_n) \ldots)$.

An n-ary *tuple* over some domain D is an element of D^n while an n-ary *relation* is a subset of D^n.

$\mathcal{M}(A)$ denotes all multisets composed of elements of a set A.

A.2 Sequences

We allow two notations for sequences of elements and their concatenation.

One is the standard notation used in formal language theory [1, 128]: abc stands for the sequence of the 3 elements a, b and c. The concatenation of two sequences α and β is represented by $\alpha\beta$.

However, as this notation can be inconvenient and confusing in places, we will usually prefer the notation $\langle a, b, c \rangle$ for sequences. On some occasions we also use $\alpha \uplus \beta$ for the concatenation of sequences.

The empty sequence will be denoted by either ϵ or $\langle\rangle$.

A.3 Graphs and trees

Definition A.3.1 A *graph* is a couple $G = (N, A)$ consisting of a set of *nodes* N and a set of *arcs* $A \in N \times N$. If $(a, b) \in A$ we denote this by $a \rightarrow b$ and say that a is a *predecessor* of b (in G) and b is a *successor* of a (in G).

Definition A.3.2 A *tree* T is a graph such that

- there is one node without predecessor, called the *root* of T, and
- every node except the root has exactly one predecessor.

A node without successor will be called a *leaf*. A successor node s of a node n in a tree is called a *child* of n while n is called the *parent* of s. Trees and graphs can be *labelled*, i.e. we have a labelling function l which maps nodes and/or arcs to a domain of labels L. Sometimes the children of nodes are also *ordered*, and we then talk about ordered trees and graphs.

Appendix B

Counterexample

In this appendix we present a counterexample to Lemma 4.11 on page 326 of [100]. Note that the definitions and notations differ from the ones in [97] and from the ones adopted in our thesis (for instance what is called a *chpath* in [100] corresponds more closely to the concept of a characteristic tree than to the notion of a characteristic path).

We take the following program P (similar to Example 4.3.3, the actual definitions of $r(X)$ and $s(X)$ are of no importance):

(c_1) $p(X) \leftarrow q(X)$
(c_2) $p(c) \leftarrow$
(c_3) $q(X) \leftarrow r(X)$
(c_4) $q(X) \leftarrow s(X)$
(c_5) $r(X) \leftarrow \ldots$
(c_6) $s(X) \leftarrow \ldots$

Now let the atom A be $p(b)$. Then according to Definition 4.5 of [100] we have that $chpath(A) = (\langle c_1 \rangle, \{c_3, c_4\})$. According to definition 4.10 of [100] we obtain: $chpaths(A) = \{\langle c_1, c_3 \rangle, \langle c_1, c_4 \rangle\}$.

The most general resultants (Definition 4.6 of [100]) of the paths in $chpaths(A)$ are $\{p(Z) \leftarrow r(Z), p(Z) \leftarrow s(Z)\}$.

By Definition 4.10 of [100] we obtain the *characteristic call* of A:

$$chcall(A) = msg\{p(Z), p(Z)\} = p(Z)$$

In Lemma 4.11 of [100] it is claimed that $chpath(chcall(A)) = chpath(A)$ and that $chpath(msg\{A, chcall(A)\}) = chpath(A)$. As $msg\{A, chcall(A)\}$ is more general than A this corresponds to asserting that $msg\{A, chcall(A)\}$ abstracts A while preserving the characteristic path structure. However in our example we have that:

$chpath(chcall(A)) = chpath(msg\{A, chcall(A)\}) = chpath(p(Z)) = (\langle\rangle, \{c_1, c_2\}) \neq chpath(A)$ and thus Lemma 4.11 is false.

Appendix C

Benchmark Programs

The benchmark programs were carefully selected and/or designed in such a way that they cover a wide range of different application areas, including: pattern matching, databases, expert systems, meta-interpreters (non-ground vanilla, mixed, ground), and more involved particular ones: a model-elimination theorem prover, the missionaries-cannibals problem, a meta-interpreter for a simple imperative language. The benchmarks marked with a star (*) can be fully unfolded. The size of the compiled code (under Prolog by BIM 4.0.12) is given in parentheses. Full descriptions can be found in [170].

advisor* (6810 bytes)
A very simple expert system which can be fully unfolded. A benchmark by Thomas Horváth [129].

applast (1077 bytes)
The *append − last* program of Chapter 13. In order to obtain an optimal solution deforestation has to be combined with a bottom-up inference of success-information.

contains.kmp (2570 bytes)
A benchmark based on the "contains" Lam & Kusalik benchmark [159], but with improved run-time queries. The program is a rather involved, but still inefficient (because highly non-deterministic), pattern matcher.

depth.lam* (4415 bytes)
A simple meta-interpreter which keeps track of the maximum length of refutations. It has to be specialised for a simple, fully unfoldable object program. A Lam & Kusalik benchmark [159].

doubleapp (653 bytes)
The double append example (see Chapters 10 and 11) in which three lists are appended by reusing the ordinary *append* program. Tests whether deforestation can be done.

ex_depth (4741 bytes)
A variation of *depth.lam* with a more sophisticated object program (which cannot be fully unfolded).

flip (1057 bytes)
A simple deforestation example from Wadler [281] in which a tree is flipped twice. The goal is to obtain a program which just copies the tree.

grammar.lam (9490 bytes)
A DCG (Definite Clause Grammar) parser which has to be specialised for a particular grammar. It is one of the Lam & Kusalik benchmarks [159].

groundunify.complex (10106 bytes)
The task consists in specialising an explicit unification algorithm for the ground representation. The full code can be found in Appendix H.1, where it is adapted from [67].

groundunify.simple* (10106 bytes)
The same unification algorithm as for groundunify.complex, but with a simpler specialisation query.

imperative.power (9368 bytes)
An interpreter for a simple imperative language which stores values of variables in an environment (see Section 13.3.1). It has to be specialised for a *power* sub-procedure, calculating $Base^{Exp}$, for a known exponent Exp and base $Base$ but an unknown environment.

liftsolve.app (5194 bytes)
A meta-interpreter for the ground representation which "lifts" the program to the non-ground representation for resolution. In Chapter 8 this is called the mixed representation. A description along with the code can be found in Section 8.4.2. The goal is to specialise this meta-interpreter for *append* as the object program.

liftsolve.db1* (5194 bytes)
The same meta-interpreter as liftsolve.app with a simple, fully unfoldable object program.

liftsolve.db2 (5194 bytes)
Again the same meta-interpreter as liftsolve.app, but this time with a *partially* specified object program.

liftsolve.lmkng (5194 bytes)
The goal here consists in specialising part of the above "lifting" meta-interpreter. The specialisation task is such that it may give rise to an ∞ number of characteristic trees.

map.reduce (2868 bytes)
Specialising the higher-order map/3 (using the built-ins `call/1` and `=../2`, see Section 2.3.3) for the higher-order reduce/4 in turn applied to add/3.

map.rev (2868 bytes)
Specialising the higher-order map for the reverse program.

match-append (669 bytes)
A very naive pattern matcher, written using 2 appends. Same queries as match.kmp. A similar matcher has recently been used in [224, 225].

match.kmp (975 bytes)
A semi-naive pattern matcher; the goal is to obtain a Knuth-Morris-Pratt (KMP) [148] pattern matcher by specialisation for the pattern "*aab*". The benchmark is based on the "match" Lam & Kusalik benchmark [159], but uses improved run-time queries (in order to detect whether a KMP-like matcher has been obtained).

maxlength (1632 bytes)
A program which calculates the maximum element and the length of a list by calling two separate predicates *max* and *length* (and thereby traversing the list twice). The goal is to obtain a program which traverses this list only once (i.e. the benchmark tests whether tupling can be done).

memo-solve (5251 bytes)
A variation of ex_depth with a simple loop prevention mechanism based on keeping a call stack.

missionaries (9221 bytes)
A program for the missionaries and cannibals problem.

model_elim.app (7948 bytes)
Specialise the Poole & Goebel [226] model elimination prover (also used by de Waal & Gallagher [68]) for the *append* program as the object level theory.

regexp.r1 (1489 bytes)
A naive regular expression matcher which has to be specialised for the regular expression (a+b)*aab.

regexp.r2 (1489 bytes)
Same program as regexp.r1 for ((a+b)(c+d)(e+f)(g+h))*.

regexp.r3 (1489 bytes)
Same program as regexp.r1 and regexp.r2 for the regular expression ((a+b)(a+b)(a+b)(a+b)(a+b)(a+b))*.

relative.lam* (3056 bytes)
A Lam & Kusalik benchmark [159] consisting of a fully unfoldable family database.

remove (1506 bytes)
A sophisticated deforestation example.

remove2 (1644 bytes)
An even more sophisticated deforestation example. Adapted from Turchin [274].

rev_acc_type (828 bytes)
The benchmark program consists of the "reverse with accumulating parameter" program to which type checking on the accumulator has been added. Without abstraction, the benchmark will give rise to an ∞ number of different characteristic trees. See Chapter 6 for details (and the code).

rev_acc_type.inffail (828 bytes)
The same benchmark program as rev_acc_type, but this time the specialisation task will give rise to infinite determinate failure at partial deduction time.

rotateprune (2958 bytes)
A more sophisticated deforestation example from [231]. The program rotates and prunes a binary tree by calling two distinct predicates $rotate(Tree, RTree)$ and $prune(RTree, PRTree)$. The goal is to deforest the unnecessary intermediate tree $RTree$.

ssuply.lam* (8335 bytes)
A Lam & Kusalik benchmark [159].

transpose.lam* (1599 bytes)
A Lam & Kusalik benchmark program [159] for transposing matrices. Also in [97].

upto.sum1 (3966 bytes)
Calculates the squares for 1 up to n and then sums them up. The specialisation goal is to get rid (i.e. deforest) of the intermediate list of squares. Adapted from Wadler [281].

upto.sum2 (3966 bytes)
Calculates the square of integers in nodes of a tree and sums these up. The goal is again to deforest the intermediate list of squares. Adapted from Wadler [281].

Appendix D

Extending the Cogen

It is straightforward to extend the cogen to handle primitives, i.e. built-ins (=/2, not/1, =../2, call/1,...) or externally defined user predicates. The code of these predicates will not be available and therefore no predicates to unfold them can be generated. The generating extension can either contain code that completely evaluates calls to primitives, in which case the call will then be marked reducible, or code that produces residual calls to such predicates, in which case the call is marked non-reducible. So we extend the transformation of Definition 7.3.1 with the following two rules:

3. $\mathcal{S}_i = A_i$ and $\mathcal{R}_i = []$ if A_i is a reducible built-in
4. $\mathcal{S}_i = true$ and $\mathcal{R}_i = A_i$ if A_i is a non-reducible built-in

As a last example of how to extend the method we will show how to handle the Prolog version of the conditional: $A_{\text{if}} \rightarrow A_{\text{then}}; A_{\text{else}}$. For this we will introduce the notation $G^{\mathcal{R}}$ where $G = A_1, ..., A_k$ to mean the following:

$$G^{\mathcal{R}} = \mathcal{S}_1, ..., \mathcal{S}_k$$

where $\mathcal{S}_i, \mathcal{R}_i$ are defined as in Definition 7.3.1 and $\mathcal{R} = [\mathcal{R}_1, ..., \mathcal{R}_k]$ (i.e. this allows us perform the transformations recursively on the sub-components of a conditional).

If the test of a conditional is marked as reducible then the generating extension will simply contain a conditional with the test unchanged and where the two "branches" contain code for unfolding the two branches (similar to the body of a function indexed by "u"), i.e. Definition 7.3.1 is extended with the following rule:

5. $\mathcal{S}_i = (G_1 \rightarrow (G_2^{\mathcal{R}}, eq(\mathcal{R}_i, \mathcal{R})) \ ; (G_3^{\mathcal{R}'}, eq(\mathcal{R}_i, \mathcal{R}')))$ and $\mathcal{R}_i$ is a fresh variable, if $A_i = (G_1 \rightarrow G_2 \ ; G_3)$ is reducible.

If the test goal of the conditional is non-reducible then we assume that the three subgoals are either a call to a non-reducible predicate, a call to a non-reducible (dynamic) primitive or another dynamic conditional. This restriction is not severe, since if a program contains conditionals that get classified as dynamic by the BTA and these contain arbitrary subgoals then the program may by a simple source language transformation be transformed into a program which satisfies the restriction. Definition 7.3.1 is extended with the following rule:

6. $\mathcal{S}_i = (A'_1, A'_2, A'_3)^{[\mathcal{R}, \mathcal{R}', \mathcal{R}'']}$ and $\mathcal{R}_i = (\mathcal{R} \rightarrow \mathcal{R}'; \mathcal{R}'')$, if $A_i = (A'_1 \rightarrow A'_2 ; A'_3)$ is non-reducible.

where A'_1, A'_2 and A'_3 are goals that satisfy the restriction above. This restriction ensures that the three goals $\{A'_i \mid i = 1, 2, 3\}$ compute their residual code independently of each other and the residual code for the conditional is then a conditional composed from this code.

Appendix E

A Prolog Cogen: Source Code

This appendix contains the listing of the main part of the Prolog cogen presented in Chapter 7 (and called LOGEN) .

```
/* ----------- */
/*  C O G E N  */
/* ----------- */

/* the file .ann contains:
   ann_clause(Head,Body),
   delta(Call,StaticVars,DynamicVars),
   residual(P) */

cogen :-
  findall(C,predicate(C),Clauses1),
  findall(C,clause(C),Clauses2),
  pp(Clauses1),
  pp(Clauses2).

flush_cogen :-
  print_header,
  flush_pp.

predicate(clause(Head,[if([find_pattern(Call,V)],
                          [true],
                          [insert_pattern(GCall,H),
                           findall(NClause,
                            (RCall,treat_clause(H,Body,NClause)),
```

```
                                    NClauses),
                            pp(NClauses),
                            find_pattern(Call,V)])])) :-
  generalise(Call,GCall),
  add_extra_argument("_u",GCall,Body,RCall),
  add_extra_argument("_m",Call,V,Head).


clause(clause(ResCall,ResBody)) :-
  ann_clause(Call,Body),
  add_extra_argument("_u",Call,Vars,ResCall),
  bodys(Body,ResBody,Vars).

bodys([],[],[]).
bodys([G|GS],GRes,VRes) :-
  body(G,G1,V),
  filter_cons(G1,GS1,GRes,true),
  filter_cons(V,VS,VRes,[]),
  bodys(GS,GS1,VS).

filter_cons(H,T,HT,FVal) :-
        ((nonvar(H),H = FVal) -> (HT = T) ; (HT = [H|T])).

body(unfold(Call),ResCall,V) :-
  add_extra_argument("_u",Call,V,ResCall).
body(memo(Call),true,memo(Call)).
body(call(Call),Call,[]).
body(rescall(Call),true,rescall(Call)).
body(if(G1,G2,G3),    /* Static if: */
     if(RG1,[RG2,(V=VS2)],[RG3,(V=VS3)]),V) :-
  bodys(G1,RG1,VS1),
  bodys(G2,RG2,VS2),
  bodys(G3,RG3,VS3).
body(resif(G1,G2,G3), /* Dynamic if: */
    [RG1,RG2,RG3],if(VS1,VS2,VS3)) :-
  body(G1,RG1,VS1),
  body(G2,RG2,VS2),
  body(G3,RG3,VS3).

generalise(Call,GCall) :-
  delta(Call,STerms,_),
  Call =.. [Pred|_],
  delta(GCall,STerms,_),
  GCall =.. [Pred|_].
```

```
add_extra_argument(T,Call,V,ResCall) :-
  Call =.. [Pred|Args],res_name(T,Pred,ResPred),
  append(Args,[V],NewArgs),ResCall =.. [ResPred|NewArgs].

res_name(T,Pred,ResPred) :-
  name(PE_Sep,T),string_concatenate(Pred,PE_Sep,ResPred).

print_header :-
  print('/'),print('*  --------------------  *'),print('/'),nl,
  print('/'),print('*  GENERATING EXTENSION  *'),print('/'),nl,
  print('/'),print('*  --------------------  *'),print('/'),nl,
  print(':'),print('- reconsult(memo).'),nl,
  print(':'),print('- reconsult(pp).'),nl,
  (static_consult(List) -> pp_consults(List) ; true),nl.
```

Appendix F

A Prolog Cogen: Some Examples

F.1 The parser example

The original program is as follows:

```
/* file: parser.pro */

nont(X,T,R) :- t(a,T,V),nont(X,V,R1).
nont(X,T,R) :- t(X,T,R).

t(X,[X|Es],Es).
```

The annotated program looks like:

```
/* file: parser.ann */


delta(nont(X,T,R),[X],[T,R]).

residual(nont(_,_,_)).

ann_clause(nont(X,T,R),[unfold(t(a,T,V)),memo(nont(X,V,R))]).
ann_clause(nont(X,T,R),[unfold(t(X,T,R))]).

ann_clause(t(X,[X|Es],Es),[]).
```

This supplies LOGEN with all the necessary information about the parser program, this is, the code of the program (with annotations) and the result

of the binding-time analysis. The predicate `delta` implements the division for the program and the predicate `residual` represents the set $\mathcal{L}$ in the following way. If `residual(`A`)` succeeds for a call A then the predicate symbol p of A is in $Pred(P)\backslash\mathcal{L}$ and p is therefore one of the predicates for which a m-predicate is going to be generated. The annotations `unfold` and `memo` is used by LOGEN to determine whether or not to unfold a call.
The generating extension produced by the LOGEN system for the annotation $nont(s,d,d)$ is:

```
/* file: parser.gx */

/*  --------------------  */
/*  GENERATING EXTENSION  */
/*  --------------------  */
:- reconsult(memo).
:- reconsult(pp).

nont_m(B,C,D,E) :-
  ((
    find_pattern(nont(B,C,D),E)
   ) -> (
    true
   ) ; (
    insert_pattern(nont(B,F,G),H),
    findall(I, (
      ','(nont_u(B,F,G,J),treat_clause(H,J,I))),K),
    pp(K),
    find_pattern(nont(B,C,D),E)
  )).
ta_m(L,M,N,O) :-
  ((
    find_pattern(ta(L,M,N),O)
   ) -> (
    true
   ) ; (
    insert_pattern(ta(L,P,Q),R),
    findall(S, (
      ','(ta_u(L,P,Q,T),treat_clause(R,T,S))),U),
    pp(U),
    find_pattern(ta(L,M,N),O)
  )).
nont_u(B,C,D,[E,memo(nont(B,F,D))]) :- t_u(a,C,F,E).
nont_u(G,H,I,[J]) :- t_u(G,H,I,J).
t_u(K,[K|L],L,[]).
```

Running the generating extension for

```
nont(c,T,R)
```

yields the following residual program:

```
nont__0([a|B],C) :-
  nont__0(B,C).
nont__0([c|D],D).
```

F.2 The solve example

The original program is as follows:

```
/* file: solve.pro */

go(Prog,Atom) :- solve(Prog,[Atom]).

solve(Prog,[]).
solve(Prog,[H|T]) :-
  non_ground_member(struct(clause,[H|Body]),Prog),
  solve(Prog,Body),
  solve(Prog,T).

non_ground_member(NgX,[GrH|GrT]) :-
  make_non_ground(GrH,NgX).
non_ground_member(NgX,[GrH|GrT]) :-
  non_ground_member(NgX,GrT).

make_non_ground(G,NG) :- mng(G,NG,[],Sub).

mng(var(N),X,[],[sub(N,X)]).
mng(var(N),X,[sub(N,X)|T],[sub(N,X)|T]).
mng(var(N),X,[sub(M,Y)|T],[sub(M,Y)|T1]) :-
  not(N=M),
  mng(var(N),X,T,T1).
mng(struct(F,Args),struct(F,IArgs),InSub,OutSub) :-
  l_mng(Args,IArgs,InSub,OutSub).

l_mng([],[],Sub,Sub).
l_mng([H|T],[IH|IT],InSub,OutSub) :-
  mng(H,IH,InSub,IntSub),
  l_mng(T,IT,IntSub,OutSub).
```

The annotated program looks like:

```
/* file: solve.ann */
```

```
delta(go(P,A),[P],[A]).
delta(solve(P,Q),[P],[Q]).

residual(go(_,_)).
residual(solve(_,_)).

ann_clause(go(Prog,A),[memo(solve(Prog,[A]))]).

ann_clause(solve(Prog,[]),[]).
ann_clause(solve(Prog,[H|T]),
        [unfold(non_ground_member(struct(clause,[H|Body]),Prog)),
         memo(solve(Prog,Body)),
         memo(solve(Prog,T))]).

ann_clause(non_ground_member(NgX,[GrH|GrT]),
        [unfold(make_non_ground(GrH,NgX))]).
ann_clause(non_ground_member(NgX,[GrH|GrT]),
        [unfold(non_ground_member(NgX,GrT))]).

ann_clause(make_non_ground(G,NG),
        [unfold(mng(G,NG,[],Sub))]).

ann_clause(mng(var(N),X,[],[sub(N,X)]),[]).
ann_clause(mng(var(N),X,[sub(N,X)|T],[sub(N,X)|T]),[]).
ann_clause(mng(var(N),X,[sub(M,Y)|T],[sub(M,Y)|T1]),
        [call(not(N=M)),
         unfold(mng(var(N),X,T,T1))]).
ann_clause(mng(struct(F,Args),struct(F,IArgs),InSub,OutSub),
        [unfold(l_mng(Args,IArgs,InSub,OutSub))]).

ann_clause(l_mng([],[],Sub,Sub),[]).
ann_clause(l_mng([H|T],[IH|IT],InSub,OutSub),
        [unfold(mng(H,IH,InSub,IntSub)),
         unfold(l_mng(T,IT,IntSub,OutSub))]).
```

The generating extension produced by LOGEN for the annotation $go(s, d)$ is:

```
/* file: solve.gx */

/*  --------------------  */
/*  GENERATING EXTENSION  */
/*  --------------------  */
:- reconsult(memo).
:- reconsult(pp).
```

```
go_m(B,C,D) :-
  ((
    find_pattern(go(B,C),D)
   ) -> (
    true
   ) ; (
    insert_pattern(go(B,E),F),
    findall(G, (
      ','(go_u(B,E,H),treat_clause(F,H,G))),I),
    pp(I),
    find_pattern(go(B,C),D)
  )).
solve_m(J,K,L) :-
  ((
    find_pattern(solve(J,K),L)
   ) -> (
    true
   ) ; (
    insert_pattern(solve(J,M),N),
    findall(O, (
      ','(solve_u(J,M,P),treat_clause(N,P,O))),Q),
    pp(Q),
    find_pattern(solve(J,K),L)
  )).
go_u(B,C,[memo(solve(B,[C]))]).
solve_u(D,[],[]).
solve_u(E,[F|G],[H,memo(solve(E,I)),memo(solve(E,G))]) :-
  non_ground_member_u(struct(clause,[F|I]),E,H).
non_ground_member_u(J,[K|L],[M]) :-
  make_non_ground_u(K,J,M).
non_ground_member_u(N,[O|P],[Q]) :-
  non_ground_member_u(N,P,Q).
make_non_ground_u(R,S,[T]) :-
  mng_u(R,S,[],U,T).
mng_u(var(V),W,[],[sub(V,W)],[]).
mng_u(var(X),Y,[sub(X,Y)|Z],[sub(X,Y)|Z],[]).
mng_u(var(A_1),B_1,[sub(C_1,D_1)|E_1],[sub(C_1,D_1)|F_1],[G_1]) :-
  not((A_1) = (C_1)),
  mng_u(var(A_1),B_1,E_1,F_1,G_1).
mng_u(struct(H_1,I_1),struct(H_1,J_1),K_1,L_1,[M_1]) :-
  l_mng_u(I_1,J_1,K_1,L_1,M_1).
l_mng_u([],[],N_1,N_1,[]).
l_mng_u([O_1|P_1],[Q_1|R_1],S_1,T_1,[U_1,V_1]) :-
  mng_u(O_1,Q_1,S_1,W_1,U_1), l_mng_u(P_1,R_1,W_1,T_1,V_1).
```

Running the generating extension for

```
go([struct(clause,[struct(q,[var(1)]), struct(p,[var(1)])]),
    struct(clause,[struct(p,[struct(a,[])])])],G)
```

yields the following residual program:

```
solve__1([]).
solve__1([struct(q,[B])|C]) :-
  solve__1([struct(p,[B])]),
  solve__1(C).
solve__1([struct(p,[struct(a,[])])|D]) :-
  solve__1([]),
  solve__1(D).
go__0(B) :-
  solve__1([B]).
```

F.3 The regular expression example

```
/* file: regexp.pro */

:- ensure_consulted('regexp.calls').

dgenerate(RegExp,[]) :-
  nullable(RegExp).
dgenerate(RegExp,[C|T]) :-
  first(RegExp,C2),
  dnext(RegExp,C2,NextRegExp),
  C2=C,
  dgenerate(NextRegExp,T).
```

The annotated program looks like:

```
static_consult(['regexp.calls']).

delta(dgenerate(RX,S),[RX],[S]).

residual(dgenerate(_,_)).

ann_clause(dgenerate(RegExp,[]),
        [call(nullable(RegExp))]).

ann_clause(dgenerate(RegExp,[C|T]),
        [call(first(RegExp,C2)),
         call(dnext(RegExp,C2,NextRegExp)),
         call(C2=C),
         memo(dgenerate(NextRegExp,T))]).
```

The `static_consult` primitive tells LOGEN that some auxiliary predicates are defined in the file `regexp.calls`. This will translate to a `consult` being inserted into the generating extension. The file `regexp.calls` contains the definitions of `first, dnext` and `nullable`.
For the annotation $dgenerate(s, d)$ LOGEN produces the following generating extension:

```
/* file: regexp.gx */

/*  --------------------  */
/*  GENERATING EXTENSION  */
/*  --------------------  */
:- reconsult(memo).
:- reconsult(pp).
:- consult('regexp.calls').

dgenerate_m(B,C,D) :-
  ((
    find_pattern(dgenerate(B,C),D)
   ) -> (
    true
   ) ; (
    insert_pattern(dgenerate(B,E),F),
    findall(G, (
      ','(dgenerate_u(B,E,H),treat_clause(F,H,G))),I),
    pp(I),
    find_pattern(dgenerate(B,C),D)
  )).
dgenerate_u(B,[],[]) :- nullable(B).
dgenerate_u(C,[D|E],[memo(dgenerate(F,E))]) :-
  first(C,G),
  dnext(C,G,F),
  (G) = (D).
```

Running the generating extension for

```
dgenerate(cat(star(or(a,b)),cat(a,cat(a,b))),String)
```

yields the following program corresponding to a deterministic automaton for the regular expression $(a + b)^*aab$:

```
dgenerate__3([]).
dgenerate__3([a|B]) :-
  dgenerate__1(B).
dgenerate__3([b|C]) :-
  dgenerate__0(C).
```

```
dgenerate__2([a|B]) :-
  dgenerate__2(B).
dgenerate__2([b|C]) :-
  dgenerate__3(C).
dgenerate__1([a|B]) :-
  dgenerate__2(B).
dgenerate__1([b|C]) :-
  dgenerate__0(C).
dgenerate__0([a|B]) :-
  dgenerate__1(B).
dgenerate__0([b|C]) :-
  dgenerate__0(C).
```

Appendix G

Meta-Interpreters and Databases for Integrity Checking

G.1 The *ic-solve* meta-interpreter

This appendix contains the full Prolog code of a meta-interpreter performing incremental integrity checking based upon Theorem 8.2.10. It is used for the experiments in Chapter 9.

In order to make the experiments more realistic, the facts of the database are stored, via the `fact/2` relation, in the Prolog clausal database. Updating the facts — which is of no relevance to the integrity checking phases — occurs via the `assert`, `retract` or `reconsult` primitives.

```
/* ------------------------------------------------ */
/* normal_solve(GrXtraFacts,GrDelFacts,GrRules,NgGoal) */
/* ------------------------------------------------ */

/* This normal_solve makes no assumptions about the Facts and the Rules.
   For instance the predicates defined in Rules can also be present
   in Facts and vice versa */

normal_solve(GrXtraFacts,GrDelFacts,GrRules,[]).
normal_solve(GrXtraFacts,GrDelFacts,GrRules,[neg(NgG)|NgT]) :-
    (normal_solve(GrXtraFacts,GrDelFacts,GrRules,[pos(NgG)])
     -> fail
     ;  (normal_solve(GrXtraFacts,GrDelFacts,GrRules,NgT))
    ).
```

```
normal_solve(GrXtraFacts,GrDelFacts,GrRules,[pos(NgH)|NgT]) :-
    db_fact_lookup(NgH),
    not(non_ground_member(NgH,GrDelFacts)),
    normal_solve(GrXtraFacts,GrDelFacts,GrRules,NgT).
normal_solve(GrXtraFacts,GrDelFacts,GrRules,[pos(NgH)|NgT]) :-
    non_ground_member(NgH,GrXtraFacts),
    normal_solve(GrXtraFacts,GrDelFacts,GrRules,NgT).
normal_solve(GrXtraFacts,GrDelFacts,GrRules,[pos(NgH)|NgT]) :-
    non_ground_member(term(clause,[pos(NgH)|NgBody]),GrRules),
    normal_solve(GrXtraFacts,GrDelFacts,GrRules,NgBody),
    normal_solve(GrXtraFacts,GrDelFacts,GrRules,NgT).

/* ---------------------------- */
/*    INCREMENTAL IC CHECKER    */
/* ---------------------------- */

incremental_solve(GoalList,DB) :-
    verify_one_potentially_added(GoalList,DB),
    inc_resolve(GoalList,DB).

inc_resolve([pos(NgH)|NgT],DB) :-
    DB = db(AddedFacts,DeletedFacts,
    ValidOldRules,AddedRules,DeletedRules),
    db_fact_lookup(NgH),
    not(non_ground_member(NgH,DeletedFacts)),
    incremental_solve(NgT,DB).
inc_resolve([pos(NgH)|NgT],DB) :-
    DB = db(AddedFacts,DeletedFacts,
    ValidOldRules,AddedRules,DeletedRules),
    non_ground_member(NgH,AddedFacts),
    /* print(found_added_fact(NgH)),nl, */
    append(AddedRules,ValidOldRules,NewRules),
    normal_solve(AddedFacts,DeletedFacts,NewRules,NgT).
inc_resolve([pos(NgH)|NgT],DB) :-
    DB = db(AddedFacts,DeletedFacts,
    ValidOldRules,AddedRules,DeletedRules),
    non_ground_member(term(clause,[pos(NgH)|NgBody]),ValidOldRules),
    append(NgBody,NgT,NewGoal),
    incremental_solve(NewGoal,DB).
inc_resolve([pos(NgH)|NgT],DB) :-
    DB = db(AddedFacts,DeletedFacts,
    ValidOldRules,AddedRules,DeletedRules),
    non_ground_member(term(clause,[pos(NgH)|NgBody]),AddedRules),
    append(AddedRules,ValidOldRules,NewRules),
    normal_solve(AddedFacts,DeletedFacts,NewRules,NgBody),
    normal_solve(AddedFacts,DeletedFacts,NewRules,NgT).
inc_resolve([neg(NgH)|NgT],DB) :-
    DB = db(AddedFacts,DeletedFacts,
                ValidOldRules,AddedRules,DeletedRules),
    append(AddedRules,ValidOldRules,NewRules),
    (normal_solve(AddedFacts,DeletedFacts,NewRules,[pos(NgH)])
     ->  (fail)
```

```
     ;   (verify_potentially_added(neg(NgH),DB)
           -> (normal_solve(AddedFacts,DeletedFacts,NewRules,NgT))
           ;  (incremental_solve(NgT,DB))
         )
    ).

verify_one_potentially_added(GoalList,DB) :-
    ( (one_potentially_added(GoalList,DB) -> fail ; true)
    -> fail
    ;  true
    ).
one_potentially_added(GoalList,DB) :-
    member(Literal,GoalList),
    potentially_added(Literal,DB).

/* ------------------------------------------------ */
/* Determining the literals that are potentially added */
/* ------------------------------------------------ */

/* verify if a literal is potentially added -
   without making any bindings and succeeding only once */
verify_potentially_added(Literal,DB) :-
    ( (potentially_added(Literal,DB) -> fail ; true)
    -> fail
    ;  true
    ).

potentially_added(neg(Atom),DB) :-
    potentially_deleted(pos(Atom),DB).
potentially_added(pos(Atom),DB) :-
    DB = db(AddedFacts,DeletedFacts,
    ValidOldRules,AddedRules,DeletedRules),
    non_ground_member(Atom,AddedFacts).
potentially_added(pos(Atom),DB) :-
    DB = db(AddedFacts,DeletedFacts,
    ValidOldRules,AddedRules,DeletedRules),
    non_ground_member(term(clause,[pos(Atom)|NgBody]),AddedRules).
potentially_added(pos(Atom),DB) :-
    DB = db(AddedFacts,DeletedFacts,
    ValidOldRules,AddedRules,DeletedRules),
    non_ground_member(term(clause,[pos(Atom)|NgBody]),ValidOldRules),
    member(BodyLiteral,NgBody),
    potentially_added(BodyLiteral,DB).

potentially_deleted(neg(Atom),DB) :-
    potentially_added(pos(Atom),DB).
potentially_deleted(pos(Atom),DB) :-
    DB = db(AddedFacts,DeletedFacts,
    ValidOldRules,AddedRules,DeletedRules),
    non_ground_member(Atom,DeletedFacts).
potentially_deleted(pos(Atom),DB) :-
    DB = db(AddedFacts,DeletedFacts,
```

```
    ValidOldRules,AddedRules,DeletedRules),
    non_ground_member(term(clause,[pos(Atom)|NgBody]),DeletedRules).
potentially_deleted(pos(Atom),DB) :-
    DB = db(AddedFacts,DeletedFacts,
    ValidOldRules,AddedRules,DeletedRules),
    non_ground_member(term(clause,[pos(Atom)|NgBody]),ValidOldRules),
    member(BodyLiteral,NgBody),
    potentially_deleted(BodyLiteral,DB).

/* ------------------------------------ */
/* non_ground_member(NgExpr,GrListOfExpr) */
/* ------------------------------------ */

non_ground_member(NgX,[GrH|GrT]) :-
    make_non_ground(GrH,NgX).
non_ground_member(NgX,[GrH|GrT]) :-
    non_ground_member(NgX,GrT).

/* ------------------------------------------------------ */
/*    make_non_ground(GroundRepOfExpr,NonGroundRepOfExpr)    */
/* ------------------------------------------------------ */
/* ex. ?-make_non_ground(pos(term(f,[var(1),var(2),var(1)])),X). */

make_non_ground(G,NG) :-
    mng(G,NG,[],Sub).

mng(var(N),X,[],[sub(N,X)]).
mng(var(N),X,[sub(M,Y)|T],[sub(M,Y)|T1]) :-
    ((N=M)
    -> (T1=T, X=Y)
    ;  (mng(var(N),X,T,T1))
    ).
mng(term(F,Args),term(F,IArgs),InSub,OutSub) :-
    l_mng(Args,IArgs,InSub,OutSub).
mng(neg(G),neg(NG),InSub,OutSub) :-
    mng(G,NG,InSub,OutSub).
mng(pos(G),pos(NG),InSub,OutSub) :-
    mng(G,NG,InSub,OutSub).

l_mng([],[],Sub,Sub).
l_mng([H|T],[IH|IT],InSub,OutSub) :-
    mng(H,IH,InSub,IntSub),
    l_mng(T,IT,IntSub,OutSub).

/* ------------------------------------------ */
/* SIMULATING THE DEDUCTIVE DATABASE FACT LOOKUP */
/* ------------------------------------------ */

db_fact_lookup(term(Pred,Args)) :-
    fact(Pred,Args).

fact(female,[term(mary,[])]).
```

```
fact(male,[term(peter,[])]).
fact(male,[term(paul,[])]).
...
```

G.2 The *ic-lst* meta-interpreter

This appendix contains the code of an implementation of the method by Lloyd, Sonenberg and Topor [186] for specialised integrity checking in deductive databases. It is used for the experiments in Chapter 9.

```
/* ================================================================ */
/* Bottom-Up Propagation of updates according to Lloyd et al's Method */
/* ================================================================ */

:- dynamic lts_rules/1.

construct_lts_rules :-
    retract(lts_rules(R)),fail.
construct_lts_rules :-
    findall(clause(Head,Body),rule(Head,Body),Rules),
    assert(lts_rules(Rules)).

lts_check(Nr,Update) :-
    lts_rules(Rules),
    check_ic(Nr,Update,Rules).

check_ic(Nr,Update,Rules) :-
    bup(Rules,Update,AllPos),!,
    member(false(Nr),AllPos),
    member(clause(false(Nr),Body),Rules),
    member(Atom,Body),
    member(Atom,AllPos),
    normal_solve(Body,Update).

/* This is the main Predicate */
/* Rules is the intensional part of the database */
/* Update are the added facts to the extensional database */
/* Pos is the set of (most general) atoms potentially */
/*     affected by the update */
bup(Rules,Update,Pos) :-
    bup(Rules,Update,Update,Pos).

bup(Rules,Update,InPos,OutPos) :-
    bup_step(Rules,Update,[],NewPos,InPos,IntPos),
    ((NewPos=[])
    -> (OutPos=IntPos)
    ;  (bup(Rules,NewPos,IntPos,OutPos))
    ).

bup_step([],_Pos,NewPos,NewPos,AllPos,AllPos).
```

```
bup_step([Clause1|Rest],Pos,InNewPos,ResNewPos,InAllPos,ResAllPos) :-
    Clause1 = clause(Head,Body),
    bup_treat_clause(Head,Body,Pos,InNewPos,InNewPos1,InAllPos,InAllPos1),
    bup_step(Rest,Pos,InNewPos1,ResNewPos,InAllPos1,ResAllPos).

bup_treat_clause(Head,[],Pos,NewPos,NewPos,AllPos,AllPos).
bup_treat_clause(Head,[BodyAtom|Rest],Pos,InNewPos,OutNewPos,
                    InAllPos,OutAllPos) :-
    bup_treat_body_atom(Head,BodyAtom,Pos,InNewPos,InNewPos1,
                        InAllPos,InAllPos1),
    bup_treat_clause(Head,Rest,Pos,InNewPos1,OutNewPos,InAllPos1,OutAllPos).

bup_treat_body_atom(Head,BodyAtom,[],NewPos,NewPos,AllPos,AllPos).
bup_treat_body_atom(Head,BodyAtom,[Pos1|Rest],InNewPos,OutNewPos,
      InAllPos,OutAllPos) :-
    copy(Pos1,Pos1C),
    copy(g(Head,BodyAtom),g(CHead,CBodyAtom)),
    (propagate_atom(CHead,CBodyAtom,Pos1C,NewHead)
    -> (add_atom(NewHead,InAllPos,InAllPos1,Answer),
        ((Answer=dont_add)
          -> (InNewPos2=InNewPos,InAllPos2=InAllPos1)
          ;  (add_atom(NewHead,InNewPos,InNewPos1,Answer2),
          ((Answer2=dont_add)
           -> (InNewPos2=InNewPos1,InAllPos2=InAllPos1)
           ;  (InNewPos2=[NewHead|InNewPos1],
               InAllPos2=[NewHead|InAllPos1]
              )
          )
         )
        )
       )
    ;  (InNewPos2=InNewPos,InAllPos2=InAllPos)
    ),
    bup_treat_body_atom(Head,BodyAtom,Rest,InNewPos2,OutNewPos,
                        InAllPos2,OutAllPos).

propagate_atom(Head,BodyAtom,Pos,NewAtom) :-
    BodyAtom = Pos, !,
    NewAtom = Head.
propagate_atom(Head,BodyAtom,neg(Pos),NewAtom) :- !,
    BodyAtom = Pos,
    NewAtom = neg(Head).
propagate_atom(Head,neg(BodyAtom),Pos,NewAtom) :- !,
    BodyAtom = Pos,
    NewAtom = neg(Head).

add_atom(NewAtom,[],[],add).
add_atom(NewAtom,[Pos1|Rest],OutPos,Answer) :-
    (covered(NewAtom,Pos1)
    -> (OutPos = [Pos1|Rest],
        Answer=dont_add
       )
```

```
;  (covered(Pos1,NewAtom)
    -> (OutPos=OutRest,
    add_atom(NewAtom,Rest,OutRest,Answer)
       )
    ;  (OutPos=[Pos1|OutRest],
    add_atom(NewAtom,Rest,OutRest,Answer)
       )
   )
).
```

G.3 A more sophisticated database

The following is the intensional part of a database adapted from [251] (where it is the most complicated database) and transformed into rule format (using Lloyd-Topor transformations [187] done by hand) required by [169].

```
parent(B,C) <- father(B,C)
parent(B,C) <- mother(B,C)

mother(B,C) <- father(D,C) & husband(D,B)

age(Id,Age) <- civil_status(Id,Age,D,E)

sex(Id,C) <- civil_status(Id,Age,C,E)

dependent(B,C) <- parent(C,B) & occupation(C,service)
                              & occupation(B,student)

occupation(Id,C) <- civil_status(Id,D,E,C)

eq(B,B) <-

aux_male_female(male) <-
aux_male_female(female) <-
aux_status(student) <-
aux_status(retired) <-
aux_status(business) <-
aux_status(service) <-
aux_limit(Id,Age) <- greater_than(Id,0) & less_than(Id,100000) &
                     greater_than(Age,0) & less_than(Age,125)

false(a1) <- civil_status(Id,Age,D,E) &
             civil_status(Id,F,G,H) & ~eq(Age,F)
false(a2) <- civil_status(Id,Age,D,E) &
```

```
            civil_status(Id,F,G,H) & ~eq(D,G)
false(a3) <- civil_status(Id,Age,D,E) &
            civil_status(Id,F,G,H) & ~eq(E,H)
false(2) <- father(B,C) & father(D,C) & ~eq(B,D)
false(3) <- husband(B,C) & husband(D,C) & ~eq(B,D)
false(4) <- husband(B,C) & husband(B,D) & ~eq(C,D)
false(5) <- civil_status(Id,Age,D,E) & aux_male_female(D) &
            aux_status(E) & ~aux_limit(Id,Age)
false(6) <- civil_status(Id,Age,D,student) & ~less_than(Age,25)
false(7) <- civil_status(Id,Age,D,retired) & ~greater_than(Age,60)
false(8) <- father(B,C) & ~sex(B,male)
false(9a) <- husband(B,C) & ~sex(B,male)
false(9b) <- husband(B,C) & ~sex(C,female)
false(10a) <- husband(B,C) & age(B,D) & ~greater_than(D,19)
false(10b) <- husband(B,C) & age(C,D) & ~greater_than(D,19)
false(11)  <- civil_status(Id,Age,D,E) &
      less_than(Age,20) & ~eq(E,student)
false(12) <- dependent(X,Y) & ~tax(Y,X)
```

Appendix H

Explicit Unification Algorithms

H.1 A unification algorithm with accumulators

Below, we include an explicit, ground representation *unify* slightly adapted from [67] (which uses \== instead of `not(eq(.))`).

Note that unifiers are not calculated in idempotent form, meaning that new bindings do not have to be explicitly composed with the incoming substitution inside the unification algorithm. Partial deduction would be even more complicated if this was the case.

```
unify(X,Y,S) :-
       unify(X,Y,[],S).

unify(var(N),T,S,S1) :-
       bound(var(N),S,B,V),
       unify(var(N),T,S,S1,B,V).
unify(struct(F,Args),var(N),S,S1) :-
       unify(var(N),struct(F,Args),S,S1).
unify(struct(F,Args1),struct(F,Args2),S,S2) :-
       unifyargs(Args1,Args2,S,S2).

unify(var(_),T,S,S1,B,true) :-
       unify(B,T,S,S1).
unify(var(N),T,S,S1,_,false) :-
       unify1(T,var(N),S,S1).
```

```
unifyargs([],[],S,S).
unifyargs([T|Ts],[R|Rs],S,S2) :-
       unify(T,R,S,S1),
       unifyargs(Ts,Rs,S1,S2).

unify1(struct(F,Args),var(N),S,[var(N)/struct(F,Args)|S]) :-
       not(occur_args(var(N),Args,S)).
unify1(var(N),var(N),S,S).
unify1(var(M),var(N),S,S1) :-
       diff(M,N),
       bound(var(M),S,B,V),
       unify1(var(M),var(N),S,S1,B,V).
unify1(var(_),var(N),S,S1,B,true) :-
       unify1(B,var(N),S,S1).
unify1(var(M),var(N),S,[var(N)/var(M)|S],_,false).

bound(var(N),[var(N)/T|_],T,true) :-
       diff(T,var(N)).
bound(var(N),[B/_|S],T,F) :-
       diff(B,var(N)),
       bound(var(N),S,T,F).
bound(var(_),[],_,false).

dereference(var(N),[var(N)/T|_],T) :-
       diff(T,var(N)).
dereference(var(N),[B/_|S],T) :-
       diff(B,var(N)),
       dereference(var(N),S,T).

occur(var(N),var(M),S) :-
       dereference(var(M),S,T),
       occur(var(N),T,S).
occur(var(N),var(N),_).
occur(var(N),struct(_,Args),S) :-
       occur_args(var(N),Args,S).

occur_args(var(N),[A|_],S) :-
       occur(var(N),A,S).
occur_args(var(N),[_|As],S) :-
       occur_args(var(N),As,S).

diff(X,Y) :-
       not(eq(X,Y)).
eq(X,X).
```

H.2 A unification algorithm without accumulators

Below, we present an explicit unification algorithm for the ground representation which does not use accumulating parameters (and does not perform the occurs check to avoid using negation).

```
unify(T1,T2,MGU) :- unify(T1,empty,T2,empty,MGU).

unify(struct(F,A1),S1,struct(F,A2),S2,MGU) :-
   l_unify(A1,S1,A2,S2,MGU).
unify(var(V),S1,struct(F,A2),S2,MGU) :-
   get_binding(V,S1,VS),
   unify2(VS,S1,struct(F,A2),S2,MGU).
unify(struct(F,A1),S1,var(V),S2,MGU) :-
   get_binding(V,S2,VS),
   unify2(struct(F,A1),S1,VS,S2,MGU).
unify(var(V),S1,var(W),S2,MGU) :-
   get_binding(V,S1,VS),
   get_binding(W,S2,WS),
   unify2(VS,S1,WS,S2,MGU).

unify2(struct(F,A1),S1,struct(F,A2),S2,MGU) :-
   l_unify(A1,S1,A2,S2,MGU).
unify2(var(V),S1,struct(F,A2),S2,sub(V,struct(F,A2))).
unify2(struct(F,A1),S1,var(V),S2,sub(V,struct(F,A1))).
unify2(var(V),S1,var(V),S2,empty).
unify2(var(V),S1,var(W),S2,sub(V,var(W))) :- V\=W.

l_unify([],S1,[],S2,[]).
l_unify([H|T],S1,[H2|T2],S2,comp(HMGU,TMGU)) :-
   unify(H,S1,H2,S2,HMGU),
   l_unify(T,comp(S1,HMGU),T2,comp(S2,HMGU),TMGU).

apply(var(V),Sub,VS) :-
   get_binding(V,Sub,VS).
apply(struct(F,A),Sub,struct(F,AA)) :-
   l_apply(A,Sub,AA).

l_apply([],Sub,[]).
l_apply([H|T],Sub,[AH|AT]) :-
   apply(H,Sub,AH),l_apply(T,Sub,AT).

get_binding(V,empty,var(V)).
get_binding(V,sub(V,S),S).
```

```
get_binding(V,sub(W,S),var(V)) :- V \= W.
get_binding(V,comp(L,R),S) :-
   get_binding(V,L,VL),apply(VL,R,S).
```

Bibliography

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers — Principles, Techniques and Tools*. Addison-Wesley, 1986.

[2] M. Alpuente, M. Falaschi, and G. Vidal. Narrowing-driven partial evaluation of functional logic programs. In H. Riis Nielson, editor, *Proceedings of the 6th European Symposium on Programming, ESOP'96*, LNCS 1058, pages 45–61. Springer-Verlag, 1996.

[3] L. O. Andersen. Partial evaluation of C and automatic compiler generation. In U. Kastens and P. Pfahler, editors, *4th International Conference on Compiler Construction*, LNCS 641, pages 251–257, Paderborn, Germany, 1992. Springer-Verlag.

[4] L. O. Andersen. Binding-time analysis and the taming of c pointers. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 47–58, Copenhagen, Denmark, 1993. ACM Press.

[5] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994. (DIKU report 94/19).

[6] P. H. Andersen. Partial evaluation applied to ray tracing. In W. Mackens and S. Rump, editors, *Software Engineering im Scientific Computing*, pages 78–85. Vieweg, 1996.

[7] K. R. Apt. Introduction to logic programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 10, pages 495–574. North-Holland Amsterdam, 1990.

[8] K. R. Apt. *From Logic Programming to Prolog*. Prentice Hall, 1997.

[9] K. R. Apt and R. N. Bol. Logic programming and negation: A survey. *The Journal of Logic Programming*, 19 & 20:9–72, May 1994.

[10] K. R. Apt and H. Doets. A new definition of SLDNF-resolution. *The Journal of Logic Programming*, 8:177–190, 1994.

[11] K. R. Apt and E. Marchiori. Reasoning about Prolog programs: from modes through types to assertions. *Formal Aspects of Computing*, 6(6A):743–765, 1994.

[12] K. R. Apt and M. H. van Emden. Contributions to the theory of logic programming. *Journal of the ACM*, 29(3):841–862, 1982.

[13] C. Aravindan and P. M. Dung. On the correctness of unfold/fold transformation of normal and extended logic programs. *The Journal of Logic Programming*, 24(3):201–217, 1995.

[14] J. Barklund. Metaprogramming in logic. In A. Kent and J. Williams, editors, *Encyclopedia of Computer Science and Technology*. Marcell Dekker, Inc., New York. To Appear.

[15] M. Baudinet. Proving termination of Prolog programs: A semantic approach. *The Journal of Logic Programming*, 14(1 & 2):1–29, 1992.

[16] L. Beckman, A. Haraldson, Ö. Oskarsson, and E. Sandewall. A partial evaluator and its use as a programming tool. *Artificial Intelligence*, 7:319–357, 1976.

[17] K. Benkerimi and P. M. Hill. Supporting transformations for the partial evaluation of logic programs. *Journal of Logic and Computation*, 3(5):469–486, October 1993.

[18] K. Benkerimi and J. W. Lloyd. A partial evaluation procedure for logic programs. In S. Debray and M. Hermenegildo, editors, *Proceedings of the North American Conference on Logic Programming*, pages 343–358. MIT Press, 1990.

[19] K. Benkerimi and J. C. Shepherdson. Partial deduction of updateable definite logic programs. *The Journal of Logic Programming*, 18(1):1–27, January 1994.

[20] L. Birkedal and M. Welinder. Hand-writing program generator generators. In M. Hermenegildo and J. Penjam, editors, *Programming Language Implementation and Logic Programming. Proceedings, Proceedings of PLILP'91*, LNCS 844, pages 198–214, Madrid, Spain, 1994. Springer-Verlag.

[21] R. Bol. Loop checking in partial deduction. *The Journal of Logic Programming*, 16(1&2):25–46, 1993.

[22] A. Bondorf. Towards a self-applicable partial evaluator for term rewriting systems. In D. Bjørner, A. P. Ershov, and N. D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 27–50. North-Holland, 1988.

[23] A. Bondorf. A self-applicable partial evaluator for term rewriting systems. In J. Diaz and F. Orejas, editors, *TAPSOFT'89, Proceedings of the International Joint Conference on Theory and Practice of Software Development*, LNCS 352, pages 81–96, Barcelona, Spain, March 1989. Springer-Verlag.

[24] A. Bondorf and O. Danvy. Automatic autoprojection of recursive equations with global variables and abstract data types. *Science of Computer Programming*, 16:151–195, 1991.

[25] A. Bondorf and D. Dussart. Improving cps-based partial evaluation: writing cogen by hand. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 1–9, Orlando, Florida, 1994.

[26] A. Bondorf, F. Frauendorf, and M. Richter. An experiment in automatic self-applicable partial evaluation of Prolog. Technical Report 335, Lehrstuhl Informatik V, University of Dortmund, 1990.

[27] A. Bossi and N. Cocco. Preserving universal termination through unfold/fold. In G. Levi and M. Rodriguez-Artalejo, editors, *Proceedings of the Fourth International Conference on Algebraic and Logic Programming*, LNCS 850, pages 269–286, Madrid, Spain, 1994. Springer-Verlag.

[28] A. Bossi and N. Cocco. Replacement can preserve termination. In J. Gallagher, editor, *Pre-Proceedings of the International Workshop on Logic Program Synthesis and Transformation (LOPSTR'96)*, pages 78–91, Stockholm, Sweden, August 1996.

[29] A. Bossi, N. Cocco, and S. Dulli. A method for specialising logic programs. *ACM Transactions on Programming Languages and Systems*, 12(2):253–302, 1990.

[30] A. Bossi, N. Cocco, and S. Etalle. Transformation of left terminating programs: The reordering problem. In M. Proietti, editor, Logic Program Synthesis and Transformation. *Proceedings of LOPSTR'95*, LNCS 1048, pages 33–45, Utrecht, The Netherlands, September 1995. Springer-Verlag.

[31] A. Bossi and S. Etalle. More on unfold/fold transformations of normal programs: Preservation of fitting's semantics. In L. Fribourg and F. Turini, editors, Logic Program Synthesis and Transformation — Meta-Programming in Logic. *Proceedings of LOPSTR'94 and META'94*, LNCS 883, pages 311–331, Pisa, Italy, June 1994. Springer-Verlag.

[32] A. Bossi and S. Etalle. Transforming acyclic programs. *ACM Transactions on Programming Languages and Systems*, 16(4):1081–1096, 1994.

[33] D. Boulanger and M. Bruynooghe. Deriving fold/unfold transformations of logic programs using extended OLDT-based abstract interpretation. *Journal of Symbolic Computation*, 15(5&6):495–521, 1993.

[34] A. F. Bowers. Representing Gödel object programs in Gödel. Technical Report CSTR-92-31, University of Bristol, November 1992.

[35] A. F. Bowers and C. A. Gurr. Towards fast and declarative meta-programming. In K. R. Apt and F. Turini, editors, *Meta-logics and Logic Programming*, pages 137–166. MIT Press, 1995.

[36] M. Bruynooghe. A practical framework for the abstract interpretation of logic programs. *The Journal of Logic Programming*, 10:91–124, 1991.

[37] 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.

[38] F. Bry. Query evaluation in recursive databases: bottom-up and top-down reconciled. *Data and Knowledge Engineering*, 5:289–312, 1990.

[39] F. Bry, H. Decker, and R. Manthey. A uniform approach to constraint satisfaction and constraint satisfiability in deductive databases. In J. Schmidt, S. Ceri, and M. Missikoff, editors, *Proceedings of the International Conference on Extending Database Technology*, LNCS, pages 488–505, Venice, Italy, 1988. Springer-Verlag.

[40] F. Bry and R. Manthey. Tutorial on deductive databases. In *Logic Programming Summer School*, 1990.

[41] F. Bry, R. Manthey, and B. Martens. Integrity verification in knowledge bases. In A. Voronkov, editor, Logic Programming. *Proceedings of the First and Second Russian Conference on Logic Programming*, LNCS 592, pages 114–139. Springer-Verlag, 1991.

[42] M. Bugliesi and F. Russo. Partial evaluation in Prolog: Some improvements about cut. In E. L. Lusk and R. A. Overbeek, editors, *Logic Programming: Proceedings of the North American Conference*, pages 645–660. MIT Press, 1989.

[43] R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, 1977.

[44] L. Cavedon. Acyclic logic programs and the completeness of SLDNF-resolution. *Theoretical Computer Science*, 86:81–92, 1991.

[45] L. Cavedon and J. W. Lloyd. A completeness theorem for SLDNF resolution. *The Journal of Logic Programming*, 7(3):177–192, November 1989.

[46] M. Celma and H. Decker. Integrity checking in deductive databases — the ultimate method ? In *Proceedings of the 5th Australasian Database Conference*, January 1994.

[47] M. Celma, C. Garcí, L. Mota, and H. Decker. Comparing and synthesizing integrity checking methods for deductive databases. In *Proceedings of the 10th IEEE Conference on Data Engineering*, 1994.

[48] D. Chan. Constructive negation based on the completed database. In *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 111–125, Seattle, 1988. IEEE, MIT Press.

[49] D. Chan and M. Wallace. A treatment of negation during partial evaluation. In H. Abramson and M. Rogers, editors, *Meta-Programming in Logic Programming, Proceedings of the Meta88 Workshop, June 1988*, pages 299–318. MIT Press, 1989.

[50] W. Chen and D. S. Warren. Tabled evaluation with delaying for general logic programs. *Journal of the ACM*, 43(1):20–74, January 1996.

[51] W.-N. Chin. *Automatic Methods for Program Transformation.* PhD thesis, Imperial College, University of London, 1990.

[52] K. L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, 1978.

[53] W. Clocksin and C. Mellish. *Programming in Prolog (Third Edition).* Springer-Verlag, 1987.

[54] M. Codish and B. Demoen. Analyzing logic programs using "prop"-ositional logic programs and a magic wand. *The Journal of Logic Programming*, 25(3):249–274, December 1995.

[55] C. Consel. A tour of Schism: A partial evaluation system for higher-order applicative languages. In *Proceedings of PEPM'93, the ACM Sigplan Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 145–154. ACM Press, 1993.

[56] C. Consel and O. Danvy. Tutorial notes on partial evaluation. In *Proceedings of ACM Symposium on Principles of Programming Languages (POPL'93)*, Charleston, South Carolina, January 1993. ACM Press.

[57] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *The Journal of Logic Programming*, 13(2 & 3):103–179, 1992.

[58] H. B. Curry. *Foundations of Mathematical Logic*. Dover Publications, 1976.

[59] S. Das and M. Williams. A path finding method for constraint checking in deductive databases. *Data & Knowledge Engineering*, 4:223–244, 1989.

[60] D. De Schreye and M. Bruynooghe. The compilation of forward checking regimes through meta-interpretation and transformation. In H. Abramson and M. Rogers, editors, *Meta-Programming in Logic Programming, Proceedings of the Meta88 Workshop, June 1988*, pages 217–232. MIT Press, 1989.

[61] D. De Schreye and S. Decorte. Termination of logic programs: The never ending story. *The Journal of Logic Programming*, 19 & 20:199–260, May 1994.

[62] D. De Schreye, R. Glück, J. Jørgensen, M. Leuschel, B. Martens, and M. H. Sørensen. Conjunctive partial deduction: Foundations, control, algorithms and experiments. Submitted for Publication, January 1997.

[63] D. De Schreye, M. Leuschel, and B. Martens. Tutorial on program specialisation (abstract). In J. W. Lloyd, editor, *Proceedings of ILPS'95, the International Logic Programming Symposium*, Portland, USA, December 1995. MIT Press.

[64] D. De Schreye and B. Martens. A sensible least Herbrand semantics for untyped vanilla meta-programming. In A. Pettorossi, editor, *Proceedings Meta'92*, LNCS 649, pages 192–204. Springer-Verlag, 1992.

[65] 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.

[66] D. A. de Waal. *Analysis and Transformation of Proof Procedures*. PhD thesis, University of Bristol, October 1994.

[67] D. A. de Waal and J. Gallagher. Specialisation of a unification algorithm. In T. Clement and K.-K. Lau, editors, Logic Program Synthesis and Transformation. *Proceedings of LOPSTR'91*, pages 205–220, Manchester, UK, 1991.

[68] D. A. de Waal and J. Gallagher. The applicability of logic program analysis and transformation to theorem proving. In A. Bundy, editor, *Automated Deduction—CADE-12*, pages 207–221. Springer-Verlag, 1994.

[69] S. Debray. Resource-bounded partial evaluation. In *Proceedings of PEPM'97, the ACM Sigplan Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, 1997. To appear.

[70] S. Debray and N.-W. Lin. Cost analysis of logic programs. *ACM Transactions on Programming Languages and Systems*, 15(5):826–875, November 1993.

[71] S. Debray, P. López García, M. Hermenegildo, and N.-W. Lin. Estimating the computational cost of logic programs. In B. Le Charlier, editor, *Proceedings of SAS'94*, LNCS 864, pages 255–265, Namur, Belgium, September 1994. Springer-Verlag.

[72] S. Debray and D. S. Warren. Functional computations in logic programs. *ACM Transactions on Programming Languages and Systems*, 11(3):451–481, 1989.

[73] H. Decker. Integrity enforcement on deductive databases. In L. Kerschberg, editor, *Proceedings of the 1st International Conference on Expert Database Systems*, pages 381–395, Charleston, South Carolina, 1986. The Benjamin/Cummings Publishing Company, Inc.

[74] H. Decker. Personal communication. January 1997.

[75] H. Decker and M. Celma. A slick procedure for integrity checking in deductive databases. In P. Van Hentenryck, editor, *Proceedings of ICLP'94*, pages 456–469. MIT Press, June 1994.

[76] S. Decorte, D. De Schreye, M. Leuschel, B. Martens, and K. Sagonas. Termination analysis for tabled logic programming. April 1997. Accepted for the Pre-Proceedings of LOPSTR'97.

[77] B. Demoen. On the transformation of a prolog program to a more efficient binary program. In K.-K. Lau and T. Clement, editors, Logic Program Synthesis and Transformation. *Proceedings of LOPSTR'92*, pages 242–252, Manchester, UK, 1992.

[78] M. Denecker. *Knowledge Representation and Reasoning in Incomplete Logic Programming.* PhD thesis, Department of Computer Science, K.U.Leuven, 1993.

[79] M. Denecker and D. De Schreye. SLDNFA; an abductive procedure for normal abductive programs. In K. Apt, editor, *Proceedings of the International Joint Conference and Symposium on Logic Programming, Washington*, 1992.

[80] P. Derensart, A. Ed-Dbali, and L. Cervoni. *Prolog: The Standard, Reference Manual.* Springer-Verlag, 1996.

[81] N. Dershowitz. Termination of rewriting. *Journal of Symbolic Computation*, 3:69–116, 1987.

[82] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. B*, pages 243–320. Elsevier, MIT Press, 1990.

[83] N. Dershowitz and Z. Manna. Proving termination with multiset orderings. *Communications of the ACM*, 22(8):465–476, 1979.

[84] K. Doets. Levationis laus. *Journal of Logic and Computation*, 3(5):487–516, 1993.

[85] K. Doets. *From Logic to Logic Programming.* MIT Press, 1994.

[86] W. Drabent. What is failure ? An apporach to constructive negation. *Acta Informatica*, 32:27–59, 1995.

[87] W. Drabent. Completeness of SLDNF-resolution for nonfloundering queries. *The Journal of Logic Programming*, 27(2):89–106, 1996.

[88] D. Dussart, E. Bevers, and K. De Vlaminck. Polyvariant constructor specialisation. In *Proceedings of PEPM'95, the ACM Sigplan Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 54–65, La Jolla, California, June 1995. ACM Press.

[89] E. Eder. Properties of substitutions and unifications. *Journal of Symbolic Computation*, 1:31–46, 1985.

[90] A. Ershov. Mixed computation: Potential applications and problems for study. *Theoretical Computer Science*, 18:41–67, 1982.

[91] A. Ershov. On Futamura projections. *BIT (Japan)*, 12(14):4–5, 1982. In Japanese.

[92] S. Etalle and M. Gabbrielli. A transformation system for modular CLP programs. In L. Sterling, editor, *Proceedings of the 12th International Conference on Logic Programming*, pages 681–695. The MIT Press, 1995.

[93] M. S. Feather. A survey and classification of some program transformation techniques. In L. Meertens, editor, *Proceedings TC2 IFIP Working Conference on Program Specification and Transformation*, pages 165–195, Bad Tölz, Germany, 1986.

[94] M. Fitting. *First-Order Logic and Automated Theorem Proving.* Springer-Verlag, 1990.

[95] H. Fujita and K. Furukawa. A self-applicable partial evaluator and its use in incremental compilation. *New Generation Computing*, 6(2 & 3):91–118, 1988.

[96] Y. Futamura. Partial evaluation of a computation process — an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.

[97] J. Gallagher. A system for specialising logic programs. Technical Report TR-91-32, University of Bristol, November 1991.

[98] J. Gallagher. Tutorial on specialisation of logic programs. In *Proceedings of PEPM'93, the ACM Sigplan Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 88–98. ACM Press, 1993.

[99] J. Gallagher and M. Bruynooghe. Some low-level transformations for logic programs. In M. Bruynooghe, editor, *Proceedings of Meta90 Workshop on Meta Programming in Logic*, pages 229–244, Leuven, Belgium, 1990.

[100] J. Gallagher and M. Bruynooghe. The derivation of an algorithm for program specialisation. *New Generation Computing*, 9(3 & 4):305–333, 1991.

[101] J. Gallagher and D. A. de Waal. Deletion of redundant unary type predicates from logic programs. In K.-K. Lau and T. Clement, editors, Logic Program Synthesis and Transformation. *Proceedings of LOPSTR'92*, pages 151–167, Manchester, UK, 1992.

[102] J. Gallagher and D. A. de Waal. Fast and precise regular approximations of logic programs. In P. Van Hentenryck, editor, *Proceedings of the Eleventh International Conference on Logic Programming*, pages 599–613. The MIT Press, 1994.

[103] J. Gallagher and L. Lafave. Regular approximations of computation paths in logic and functional languages. In O. Danvy, R. Glück, and P. Thiemann, editors, *Proceedings of the 1996 Dagstuhl Seminar on Partial Evaluation*, LNCS 1110, pages 115–136, Schloß Dagstuhl, 1996. Springer-Verlag.

[104] P. A. Gardner and J. C. Shepherdson. Unfold/fold transformations in logic programs. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic, Essays in Honor of Alan Robinson*, pages 565–583. MIT Press, 1991.

[105] M. L. Ginsberg. Negative subgoals with free variables. *The Journal of Logic Programming*, 11(3 & 4):271–294, October/November 1991.

[106] A. J. Glenstrup and N. D. Jones. BTA algorithms to ensure termination of off-line partial evaluation. In *Perspectives of System Informatics: Proceedings of the Andrei Ershov Second International Memorial Conference*, LNCS. Springer-Verlag, June 25–28 1996.

[107] R. Glück. Towards multiple self-application. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 309–320, New Haven, Connecticut, 1991. ACM Press.

[108] R. Glück. On the generation of specialisers. *Journal of Functional Programming*, 4(4):499–514, 1994.

[109] R. Glück and J. Jørgensen. Efficient multi-level generating extensions for program specialization. In S. Swierstra and M. Hermenegildo, editors, *Programming Languages, Implementations, Logics and Programs (PLILP'95)*, LNCS 982, pages 259–278, Utrecht, The Netherlands, September 1995. Springer-Verlag.

[110] R. Glück, J. Jørgensen, B. Martens, and M. H. Sørensen. Controlling conjunctive partial deduction of definite logic programs. In H. Kuchen and S. Swierstra, editors, *Proceedings of the International Symposium on Programming Languages, Implementations, Logics and Programs (PLILP'96)*, LNCS 1140, pages 152–166, Aachen, Germany, September 1996. Springer-Verlag. Extended version as Technical Report CW 226, K.U. Leuven.

[111] R. Glück and A. V. Klimov. Occam's razor in metacomputation: the notion of a perfect process tree. In G. Filé, P. Cousot, M. Falaschi, and A. Rauzy, editors, *Proceedings of WSA'93*, LNCS 724, pages 112–123, Padova, Italy, September 1993. Springer-Verlag.

[112] R. Glück, R. Nakashige, and R. Zöchling. Binding-time analysis applied to mathematical algorithms. In J. Dolezal and J. Fidler, editors, *17th IFIP Conference on System Modelling and Optimization*, Prague, Czech Republic, 1995. Chapman & Hall.

[113] R. Glück and M. H. Sørensen. Partial deduction and driving are equivalent. In M. Hermenegildo and J. Penjam, editors, *Programming Language Implementation and Logic Programming. Proceedings, Proceedings of PLILP'91*, LNCS 844, pages 165–181, Madrid, Spain, 1994. Springer-Verlag.

[114] R. Glück and M. H. Sørensen. A roadmap to supercompilation. In O. Danvy, R. Glück, and P. Thiemann, editors, *Proceedings of the 1996 Dagstuhl Seminar on Partial Evaluation*, LNCS 1110, pages 137–160, Schloß Dagstuhl, 1996. Springer-Verlag.

[115] C. A. Gurr. *A Self-Applicable Partial Evaluator for the Logic Programming Language Gödel.* PhD thesis, Department of Computer Science, University of Bristol, January 1994.

[116] C. A. Gurr. Specialising the ground representation in the logic programming language Gödel. In Y. Deville, editor, Logic Program Synthesis and Transformation. *Proceedings of LOPSTR'93*, Workshops in Computing, pages 124–140, Louvain-La-Neuve, Belgium, 1994. Springer-Verlag.

[117] J. Gustedt. *Algorithmic Aspects of Ordered Structures.* PhD thesis, Technische Universität Berlin, 1992.

[118] N. Heintze. Practical aspects of set based analysis. In *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 765–779, Washington D.C., 1992. MIT Press.

[119] J. Herbrand. Investigations in proof theory. In J. van Heijenoort, editor, *From Frege to Gödel: A Source Book in Mathematical Logic, 1879-1931*, pages 525–581. Harvard University Press, 1967.

[120] M. Hermenegildo, R. Warren, and S. K. Debray. Global flow analysis as a practical compilation tool. *The Journal of Logic Programming*, 13(4):349–366, 1992.

[121] G. Higman. Ordering by divisibility in abstract algebras. *Proceedings of the London Mathematical Society*, 2:326–336, 1952.

[122] P. Hill and J. Gallagher. Meta-programming in logic programming. Technical Report 94.22, School of Computer Studies, University of Leeds, 1994. To be published in *Handbook of Logic in Artificial Intelligence and Logic Programming, Vol. 5.* Oxford Science Publications, Oxford University Press.

[123] P. Hill and J. W. Lloyd. *The Gödel Programming Language*. MIT Press, 1994.

[124] K. Hinkelmann. *Transformationen von Hornklausel-Wissensbasen: Verarbeitung gleichen Wissens durch verschiedene Inferenzen*. PhD thesis, Universität Kaiserslautern, 1995.

[125] C. K. Holst. Syntactic currying: yet another approach to partial evaluation. Technical report, DIKU, Department of Computer Science, University of Copenhagen, 1989.

[126] C. K. Holst. Finiteness analysis. In J. Hughes, editor, *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture (FPCA)*, LNCS 523, pages 473–495. Springer-Verlag, August 1991.

[127] C. K. Holst and J. Launchbury. Handwriting cogen to avoid problems with static typing. Working paper, 1992.

[128] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[129] T. Horváth. Experiments in partial deduction. Master's thesis, Departement Computerwetenschappen, K.U. Leuven, Belgium, 1993.

[130] J. Hughes. Backwards analysis of functional programs. In D. Bjørner, A. P. Ershov, and N. D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 187–208. North-Holland, 1988.

[131] D. Jacobs and A. Langen. Static analysis of logic programs for independent AND-parallelism. *The Journal of Logic Programming*, 13(2 &3):291–314, May/July 1992.

[132] J.-M. Jacquet. *Constructing Logic Programs*. Wiley, Chichester, 1993.

[133] J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *The Journal of Logic Programming*, 19 & 20:503–581, 1994.

[134] G. Janssens and M. Bruynooghe. Deriving descriptions of possible values of program variables by means of abstract interpretation. *The Journal of Logic Programming*, 13(2 & 3):205–258, 1992.

[135] N. D. Jones. The essence of program transformation by partial evaluation and driving. In M. S. Neil D. Jones, Masami Hagiya, editor, *Logic, Language and Computation*, LNCS 792, pages 206–224. Springer-Verlag, 1994.

[136] N. D. Jones. An introduction to partial evaluation. *ACM Computing Surveys*, 28(3):480–503, September 1996.

[137] N. D. Jones. What not to do when writing an interpreter for specialisation. In O. Danvy, R. Glück, and P. Thiemann, editors, *Proceedings of the 1996 Dagstuhl Seminar on Partial Evaluation*, LNCS 1110, pages 216–237, Schloß Dagstuhl, 1996. Springer-Verlag.

[138] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.

[139] N. D. Jones and A. Mycroft. Stepwise development of operational and denotational semantics for Prolog. In *Proceedings of the 1984 International Symposium on Logic Programming*, pages 289–298, Atlantic City, New Jersey, 1984. IEEE Computer Society Press.

[140] N. D. Jones, P. Sestoft, and H. Søndergaard. An experiment in partial evaluation: The generation of a compiler generator. In J.-P. Jouannaud, editor, *Rewriting Techniques and Applications*, LNCS 202, pages 124–140, Dijon, France, 1985. Springer-Verlag.

[141] 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.

[142] J. Jørgensen and M. Leuschel. Efficiently generating efficient generating extensions in Prolog. In O. Danvy, R. Glück, and P. Thiemann,

editors, *Proceedings of the 1996 Dagstuhl Seminar on Partial Evaluation*, LNCS 1110, pages 238–262, Schloß Dagstuhl, 1996. Springer-Verlag. Extended version as Technical Report CW 221, K.U. Leuven.

[143] J. Jørgensen, M. Leuschel, and B. Martens. Conjunctive partial deduction in practice. In J. Gallagher, editor, *Proceedings of the International Workshop on Logic Program Synthesis and Transformation (LOPSTR'96)*, LNCS 1207, pages 59–82, Stockholm, Sweden, August 1996. Springer-Verlag. Also in the Proceedings of BENELOG'96. Extended version as Technical Report CW 242, K.U. Leuven.

[144] T. Kanamori and K. Horiuichi. Construction of logic programs based on generalized unfold/fold rules. In J.-L. Lassez, editor, *Proceedings of the 4th International Conference on Logic Programming*, pages 744–768. The MIT Press, 1987.

[145] T. Kanamori and T. Kawamura. Oldt-based abstract interpretation. *The Journal of Logic Programming*, 15:1–30, 1993.

[146] S. Kleene. *Introduction to Metamathematics*. van Nostrand, Princeton, New Jersey, 1952.

[147] K. Knight. Unification: A multidisciplinary survey. *ACM Computing Surveys*, 21(1):93–124, March 1989.

[148] D. E. Knuth, J. H. Morris, and V. R. Pratt. Fast pattern matching in strings. *Siam Journal on Computing*, 6(2):323–350, 1977.

[149] H.-P. Ko and M. E. Nadel. Substitution and refutation revisited. In K. Furukawa, editor, *Logic Programming: Proceedings of the Eighth International Conference*, pages 679–692. MIT Press, 1991.

[150] J. Komorowksi. *A Specification of an Abstract Prolog Machine and its Application to Partial Evaluation*. PhD thesis, Linköping University, Sweden, 1981. Linköping Studies in Science and Technology Dissertations 69.

[151] J. Komorowski. Partial evaluation as a means for inferencing data structures in an applicative language: a theory and implementation in the case of prolog. In *Ninth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. Albuquerque, New Mexico*, pages 255–267, 1982.

[152] J. Komorowski. An introduction to partial deduction. In A. Pettorossi, editor, *Proceedings Meta'92*, LNCS 649, pages 49–69. Springer-Verlag, 1992.

[153] R. Kowalski. Predicate logic as a programming language. In *Proceedings IFIP Congress*, pages 569–574. IEEE, 1974.

[154] R. Kowalski and D. Kühner. Linear resolution with selection function. *Artificial Intelligence*, 2:227–260, 1971.

[155] J. B. Kruskal. Well-quasi ordering, the tree theorem, and Vazsonyi's conjecture. *Tansactions of the American Mathematical Society*, 95:210–225, 1960.

[156] V. Küchenhoff. On the efficient computation of the difference between consecutive database states. In C. Delobel, M. Kifer, and Y. Masunaga, editors, *Deductive and Object-Oriented Databases, Second International Conference*, pages 478–502, Munich, Germany, 1991. Springer-Verlag.

[157] A. Lakhotia. To PE or not to PE. In M. Bruynooghe, editor, *Proceedings of Meta90 Workshop on Meta Programming in Logic*, pages 218–228, Leuven, Belgium, 1990.

[158] A. Lakhotia and L. Sterling. How to control unfolding when specializing interpreters. *New Generation Computing*, 8:61–70, 1990.

[159] J. Lam and A. Kusalik. A comparative analysis of partial deductors for pure Prolog. Technical report, Department of Computational Science, University of Saskatchewan, Canada, May 1990. Revised April 1991.

[160] J.-L. Lassez, M. Maher, and K. Marriott. Unification revisited. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 587–625. Morgan-Kaufmann, 1988.

[161] K.-K. Lau, M. Ornaghi, A. Pettorossi, and M. Proietti. Correctness of logic program transformations based on existential termination. In J. W. Lloyd, editor, *Proceedings of ILPS'95, the International Logic Programming Symposium*, pages 480–494, Portland, USA, December 1995. MIT Press.

[162] J. Launchbury. *Projection Factorisations in Partial Evaluation*. Distinguished Dissertations in Computer Science. Cambridge University Press, 1991.

[163] J. L. Lawall. *Continuation Introduction and Elimination in Higher-Order Programming Langauges*. PhD thesis, Department of Computer Science, Indiana University, USA, October 1994.

[164] S. Y. Lee and T. W. Ling. Improving integrity constraint checking for stratified deductive databases. In *Proceedings of DEXA'94*, 1994.

[165] S. Y. Lee and T. W. Ling. Further improvement on integrity constraint checking for stratifiable deductive databases. In *Proceedings of the 22nd VLDB Conference*, Mumbai (Bombay), India, 1996.

[166] M. Leuschel. Self-applicable partial evaluation in Prolog. Master's thesis, Departement Computerwetenschappen, K.U. Leuven, Belgium, 1993.

[167] M. Leuschel. Partial evaluation of the "real thing". In L. Fribourg and F. Turini, editors, Logic Program Synthesis and Transformation — Meta-Programming in Logic. *Proceedings of LOPSTR'94 and META'94*, LNCS 883, pages 122–137, Pisa, Italy, June 1994. Springer-Verlag.

[168] M. Leuschel. Ecological partial deduction: Preserving characteristic trees without constraints. In M. Proietti, editor, Logic Program Synthesis and Transformation. *Proceedings of LOPSTR'95*, LNCS 1048, pages 1–16, Utrecht, The Netherlands, September 1995. Springer-Verlag.

[169] M. Leuschel. Prototype partial evaluation system to obtain specialised integrity checks by specialising meta-interpreters. Prototype Compulog II, D 8.3.3, Departement Computerwetenschappen, K.U. Leuven, Belgium, September 1995. Obtainable at `ftp://ftp.cs.kuleuven.ac.be/pub/compulog/ICLeupel/`.

[170] M. Leuschel. The ECCE partial deduction system and the DPPD library of benchmarks.
Obtainable via `http://www.cs.kuleuven.ac.be/~lpai`, 1996.

[171] M. Leuschel and D. De Schreye. An almost perfect abstraction operator for partial deduction. Technical Report CW 199, Departement Computerwetenschappen, K.U. Leuven, Belgium, December 1994.

[172] M. Leuschel and D. De Schreye. An almost perfect abstraction operation for partial deduction using characteristic trees. Technical Report CW 215, Departement Computerwetenschappen, K.U. Leuven, Belgium, October 1995. Accepted for Publication in New Generation Computing. Revised in December 1996. Accessible via `http://www.cs.kuleuven.ac.be/~lpai`.

[173] M. Leuschel and D. De Schreye. Towards creating specialised integrity checks through partial evaluation of meta-interpreters. In *Proceedings of PEPM'95, the ACM Sigplan Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 253–263, La Jolla, California, June 1995. ACM Press.

[174] M. Leuschel and D. De Schreye. Creating specialised integrity checks through partial evaluation of meta-interpreters. Technical Report CW 237, Departement Computerwetenschappen, K.U. Leuven, Belgium, July 1996. Submitted for Publication.

[175] M. Leuschel, D. De Schreye, and A. de Waal. A conceptual embedding of folding into partial deduction: Towards a maximal integration. In M. Maher, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming JICSLP'96*, pages 319–332, Bonn, Germany, September 1996. MIT Press.

[176] M. Leuschel and B. Martens. Obtaining specialised update procedures through partial deduction of the ground representation. In H. Decker, U. Geske, T. Kakas, C. Sakama, D. Seipel, and T. Urpi, editors, *Proceedings of the ICLP'95 Joint Workshop on Deductive Databases and Logic Programming and Abduction in Deductive Databases and Knowledge Based Systems*, GMD-Studien Nr. 266, pages 81–95, Kanagawa, Japan, June 1995.

[177] M. Leuschel and B. Martens. Partial deduction of the ground representation and its application to integrity checking. In J. W. Lloyd, editor, *Proceedings of ILPS'95, the International Logic Programming Symposium*, pages 495–509, Portland, USA, December 1995. MIT Press. Extended version as Technical Report CW 210, K.U. Leuven. Accessible via `http://www.cs.kuleuven.ac.be/~lpai`.

[178] M. Leuschel and B. Martens. Global control for partial deduction through characteristic atoms and global trees. In O. Danvy, R. Glück, and P. Thiemann, editors, *Proceedings of the 1996 Dagstuhl Seminar on Partial Evaluation*, LNCS 1110, pages 263–283, Schloß Dagstuhl, 1996. Springer-Verlag.

[179] M. Leuschel, B. Martens, and D. De Schreye. Controlling generalisation and polyvariance in partial deduction of normal logic programs. Technical Report CW 248, Departement Computerwetenschappen, K.U. Leuven, Belgium, February 1996. Submitted for Publication.

[180] M. Leuschel, B. Martens, and K. Sagonas. Preserving termination of tabled logic programs while unfolding. April 1997. Accepted for the Pre-Proceedings of LOPSTR'97.

[181] M. Leuschel and D. Schreye. Logic program specialisation: How to be more specific. In H. Kuchen and S. Swierstra, editors, *Proceedings of the International Symposium on Programming Languages, Implementations, Logics and Programs (PLILP'96)*, LNCS 1140, pages 137–151, Aachen, Germany, September 1996. Springer-Verlag.

[182] M. Leuschel and M. H. Sørensen. Redundant argument filtering of logic programs. In J. Gallagher, editor, *Proceedings of the International Workshop on Logic Program Synthesis and Transformation (LOPSTR'96)*, LNCS 1207, pages 83–103, Stockholm, Sweden, August 1996. Springer-Verlag.

[183] H. Levesque, R. Reiter, Y. Lespérance, F. Lin, and R. Scherl. Golog: a logic programming language for dynamic domains. *The Journal of Logic Programming*, 1997. To appear.

[184] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1987.

[185] J. W. Lloyd and J. C. Shepherdson. Partial evaluation in logic programming. *The Journal of Logic Programming*, 11(3& 4):217–242, 1991.

[186] J. W. Lloyd, E. A. Sonenberg, and R. W. Topor. Integrity checking in stratified databases. *The Journal of Logic Programming*, 4(4):331–343, 1987.

[187] J. W. Lloyd and R. W. Topor. Making PROLOG more expressive. *Journal of Logic Programming*, 1(3):225–240, 1984.

[188] J. W. Lloyd and R. W. Topor. A basis for deductive database systems. *The Journal of Logic Programming*, 2:93–109, 1985.

[189] M. Maher. A logic programming view of CLP. In D. S. Warren, editor, *Proceedings of the 10th International Conference on Logic Programming*, pages 737–753. The MIT Press, 1993.

[190] R. Marlet. *Vers une Formalisation de l'Évaluation Partielle.* PhD thesis, Université de Nice - Sophia Antipolis, December 1994.

[191] K. Marriott, L. Naish, and J.-L. Lassez. Most specific logic programs. In *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 909–923, Seattle, 1988. IEEE, MIT Press.

[192] K. Marriott, L. Naish, and J.-L. Lassez. Most specific logic programs. *Annals of Mathematics and Artificial Intelligence*, 1:303–338, 1990.

[193] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, April 1982.

[194] M. Martelli and C. Tricomi. A new SLDNF-tree. *Information Processing Letters*, 43(2):57–62, 1992.

[195] B. Martens. Finite unfolding revisited (part II): Focusing on subterms. Technical Report Compulog II, D 8.2.2.b, Departement Computerwetenschappen, K.U. Leuven, Belgium, 1994.

[196] B. Martens. *On the Semantics of Meta-Programming and the Control of Partial Deduction in Logic Programming*. PhD thesis, K.U. Leuven, February 1994.

[197] B. Martens and D. De Schreye. Two semantics for definite meta-programs, using the non-ground representation. In K. R. Apt and F. Turini, editors, *Meta-logics and Logic Programming*, pages 57–82. MIT Press, 1995.

[198] B. Martens and D. De Schreye. Why untyped non-ground meta-programming is not (much of) a problem. *The Journal of Logic Programming*, 22(1):47–99, 1995.

[199] B. Martens and D. De Schreye. Automatic finite unfolding using well-founded measures. *The Journal of Logic Programming*, 28(2):89–146, August 1996. Abridged and revised version of Technical Report CW180, Departement Computerwetenschappen, K.U.Leuven, October 1993, accessible via `http://www.cs.kuleuven.ac.be/~lpai`.

[200] 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.

[201] B. Martens and J. Gallagher. Ensuring global termination of partial deduction while allowing flexible polyvariance. In L. Sterling, editor, *Proceedings ICLP'95*, pages 597–613, Kanagawa, Japan, June 1995.

MIT Press. Extended version as Technical Report CSTR-94-16, University of Bristol.

[202] G. Metakides and A. Nerode. *Principles of Logic and Logic Programming*. North-Holland, 1996.

[203] D. Meulemans. Partiële deductie: Een substantiële vergelijkende studie. Master's thesis, Departement Computerwetenschappen, K.U. Leuven, Belgium, 1995.

[204] A. Miniuissi and D. J. Sherman. Squeezing intermediate construction in equational programs. In O. Danvy, R. Glück, and P. Thiemann, editors, *Proceedings of the 1996 Dagstuhl Seminar on Partial Evaluation*, LNCS 1110, pages 284–302, Schloß Dagstuhl, 1996. Springer-Verlag.

[205] T. Mogensen. The application of partial evaluation to ray-tracing. Master's thesis, DIKU, University of Copenhagen, Denmark, 1986.

[206] T. Mogensen. Constructor specialization. In *Proceedings of PEPM'93, the ACM Sigplan Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 22–32. ACM Press, June 1993.

[207] T. Mogensen and A. Bondorf. Logimix: A self-applicable partial evaluator for Prolog. In K.-K. Lau and T. Clement, editors, Logic Program Synthesis and Transformation. *Proceedings of LOPSTR'92*, pages 214–227. Springer-Verlag, 1992.

[208] A. Mulkers, W. Winsborough, and M. Bruynooghe. Live-structure data-flow analysis for prolog. *ACM Transactions on Programming Languages and Systems*, 16(2):205–258, 1994.

[209] K. Muthukumar and M. Hermenegildo. Deriving A Fixpoint Computation Algorithm for Top-Down Abstract Interpretation of Logic Programs. Technical Report ACT-DC-153-90, Microelectronics and Computer Technology Corporation (MCC), Austin, TX 78759, Apr. 1990.

[210] K. Muthukumar and M. Hermenegildo. Combined determination of sharing and freeness of program variables through abstract interpretation. In K. Furukawa, editor, *Proceedings of the Eighth International Conference on Logic Programming*, pages 49–63, Paris, 1991. MIT Press, Cambridge.

[211] K. Muthukumar and M. Hermenegildo. Compile-time Derivation of Variable Dependency Using Abstract Interpretation. *The Journal of Logic Programming*, 13(2&3):315–347, July 1992.

[212] L. Naish. Higher-order logic programming in Prolog. Technical Report 96/2, Department of Computer Science, University of Melbourne, 1995.

[213] A. P. Nemytykh, V. A. Pinchuk, and V. F. Turchin. A self-applicable supercompiler. In O. Danvy, R. Glück, and P. Thiemann, editors, *Proceedings of the 1996 Dagstuhl Seminar on Partial Evaluation*, LNCS 1110, pages 322–337, Schloß Dagstuhl, 1996. Springer-Verlag.

[214] G. Neumann. Transforming interpreters into compilers by goal classification. In M. Bruynooghe, editor, *Proceedings of Meta90 Workshop on Meta Programming in Logic*, pages 205–217, Leuven, Belgium, 1990.

[215] G. Neumann. A simple transformation from Prolog-written metalevel interpreters into compilers and its implementation. In A. Voronkov, editor, Logic Programming. *Proceedings of the First and Second Russian Conference on Logic Programming*, LNCS 592, pages 349–360. Springer-Verlag, 1991.

[216] M. H. A. Newman. On theories with a combinatorial definition of 'equivalence'. *Annals of Mathematics*, 43(2):223–243, 1942.

[217] J.-M. Nicolas. Logic for improving integrity checking in relational databases. *Acta Informatica*, 18(3):227–253, 1982.

[218] U. Nilsson and J. Małuszyński. *Logic, Programming and Prolog*. Wiley, Chichester, 1990.

[219] R. O'Keefe. On the treatment of cuts in Prolog source-level tools. In *Proceedings of the Symposium on Logic Programming*, pages 68–72. IEEE, 1985.

[220] S. Owen. Issues in the partial evaluation of meta-interpreters. In H. Abramson and M. Rogers, editors, *Meta-Programming in Logic Programming, Proceedings of the Meta88 Workshop, June 1988*, pages 319–339. MIT Press, 1989.

[221] M. Paterson and M. Wegman. Linear unification. *Journal of Computer and System Sciences*, 16(2):158–167, 1978.

[222] A. Pettorossi and M. Proietti. Transformation of logic programs: Foundations and techniques. *The Journal of Logic Programming*, 19& 20:261–320, May 1994.

[223] A. Pettorossi and M. Proietti. Rules and strategies for transforming functional and logic programs. *ACM Computing Surveys*, 28(2):360–414, June 1996.

[224] A. Pettorossi, M. Proietti, and S. Renault. Enhancing partial deduction by unfold/fold rules. In J. Gallagher, editor, *Proceedings of the International Workshop on Logic Program Synthesis and Transformation (LOPSTR'96)*, LNCS 1207, Stockholm, Sweden, August 1996. Springer-Verlag. To appear.

[225] A. Pettorossi, M. Proietti, and S. Renault. Reducing nondeterminism while specializing logic programs. In N. D. Jones, editor, *Proceedings of ACM Symposium on Principles of Programming Languages (POPL'97)*, pages 414–427, Paris, France, January 1997.

[226] D. Poole and R. Goebel. Gracefully adding negation and disjunction to Prolog. In E. Shapiro, editor, *Proceedings of the Third International Conference on Logic Programming*, pages 635–641. Springer-Verlag, 1986.

[227] S. Prestwich. The PADDY partial deduction system. Technical Report ECRC-92-6, ECRC, Munich, Germany, 1992.

[228] S. Prestwich. An unfold rule for full Prolog. In K.-K. Lau and T. Clement, editors, Logic Program Synthesis and Transformation. *Proceedings of LOPSTR'92*, Workshops in Computing, University of Manchester, 1992. Springer-Verlag.

[229] S. Prestwich. Online partial deduction of large programs. In *Proceedings of PEPM'93, the ACM Sigplan Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 111–118. ACM Press, 1993.

[230] M. Proietti and A. Pettorossi. Semantics preserving transformation rules for Prolog. In *Proceedings of the ACM Symposium on Partial Evaluation and Semantics based Program Manipulation, PEPM'91*, Sigplan Notices, Vol. 26, N. 9, pages 274–284, Yale University, New Haven, U.S.A., 1991.

[231] M. Proietti and A. Pettorossi. Unfolding-definition-folding, in this order, for avoiding unnecessary variables in logic programs. In J. Małuszyński and M. Wirsing, editors, *Proceedings of the 3rd International Symposium on Programming Language Implementation and Logic Programming, PLILP'91*, LNCS 528, pages 347–358. Springer-Verlag, 1991.

[232] M. Proietti and A. Pettorossi. The loop absorption and the generalization strategies for the development of logic programs and partial deduction. *The Journal of Logic Programming*, 16(1 & 2):123–162, May 1993.

[233] M. Proietti and A. Pettorossi. Completeness of some transformation strategies for avoiding unnecessary logical variables. In P. Van Hentenryck, editor, *Proceedings of International Conference on Logic Programming, ICLP'94*, MIT Press, pages 714–729, 1994.

[234] M. Proietti and A. Pettorossi. Synthesis of programs from unfold/fold proofs. In Y. Deville, editor, Logic Program Synthesis and Transformation. *Proceedings of LOPSTR'93*, Workshops in Computing, pages 141–158, Louvain-La-Neuve, Belgium, 1994. Springer-Verlag.

[235] *Prolog by BIM 4.0*, October 1993.

[236] T. C. Przymusinksi. On the declarative and procedural semantics of logic programs. *Journal of Automated Reasoning*, 5(2):167–205, 1989.

[237] G. Puebla and M. Hermenegildo. Implementation of multiple specialization in logic programs. In *Proceedings of PEPM'95, the ACM Sigplan Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 77–87, La Jolla, California, June 1995. ACM Press.

[238] R. Reiter. On closed world data bases. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 55–76. Plenum Press, 1978.

[239] T. Reps. Program specialization via program slicing. In O. Danvy, R. Glück, and P. Thiemann, editors, *Proceedings of the 1996 Dagstuhl Seminar on Partial Evaluation*, LNCS 1110, pages 409–429, Schloß Dagstuhl, 1996. Springer-Verlag.

[240] A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

[241] S. A. Romanenko. A compiler generator produced by a self-applicable specializer can have a surprisingly natural and understandable structure. In D. Bjørner, A. P. Ershov, and N. D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 445–463. North-Holland, 1988.

[242] F. Sadri and R. Kowalski. A theorem-proving approach to database integrity. In J. Minker, editor, *Foundations of Deductive Databases*

and Logic Programming, chapter 9, pages 313–362. Morgan Kaufmann Publishers, Inc., Los Altos, California, 1988.

[243] K. Sagonas, T. Swift, and D. S. Warren. XSB as an efficient deductive database engine. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pages 442–453, Minneapolis, Minnesota, May 1994. ACM.

[244] D. Sahlin. *An Automatic Partial Evaluator for Full Prolog.* PhD thesis, Swedish Institute of Computer Science, March 1991.

[245] D. Sahlin. Mixtus: An automatic partial evaluator for full Prolog. *New Generation Computing*, 12(1):7–51, 1993.

[246] T. Sato. Equivalence-preserving first-order unfold/fold transformation systems. *Theoretical Computer Science*, 105:57–84, 1992.

[247] S. Schoenig and M. Ducassé. A backward slicing algorithm for prolog. In R. Cousot and D. A. Schmidt, editors, *Static Analysis, Proceedings of 3rd International Symposium, SAS'96*, LNCS, pages 317–331, Aachen, Germany, September 1996. Springer-Verlag.

[248] H. Seidl. Parameter-reduction of higher-level grammars. *Theoretical Computer Science*, 55:47–85, 1987.

[249] H. Seki. Unfold/fold transformation of stratified programs. *Theoretical Computer Science*, 86:107–139, 1991.

[250] H. Seki. Unfold/fold transformation of general programs for the well-founded semantics. *The Journal of Logic Programming*, 16:5–23, 1993.

[251] R. Seljée. A new method for integrity constraint checking in deductive databases. *Data & Knowledge Engineering*, 15:63–102, 1995.

[252] J. C. Shepherdson. Language and equality theory in logic programming. Technical Report PM-91-02, University of Bristol, 1991.

[253] J. C. Shepherdson. Unsolvable problems for SLDNF resolution. *The Journal of Logic Programming*, 10(1,2,3 & 4):19–22, 1991.

[254] D. A. Smith. Constraint operations for CLP($\mathcal{FT}$). In K. Furukawa, editor, *Logic Programming: Proceedings of the Eighth International Conference*, pages 760–774. MIT Press, 1991.

[255] D. A. Smith. Partial evaluation of pattern matching in constraint logic programming languages. In N. D. Jones and P. Hudak, editors, *ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 62–71. ACM Press Sigplan Notices 26(9), 1991.

[256] D. A. Smith and T. Hickey. Partial evaluation of a CLP language. In S. Debray and M. Hermenegildo, editors, *Proceedings of the North American Conference on Logic Programming*, pages 119–138. MIT Press, 1990.

[257] Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury: An efficient purely declarative logic programming language. *The Journal of Logic Programming*, 1996. To Appear.

[258] M. H. Sørensen and R. Glück. An algorithm of generalization in positive supercompilation. In J. W. Lloyd, editor, *Proceedings of ILPS'95, the International Logic Programming Symposium*, pages 465–479, Portland, USA, December 1995. MIT Press.

[259] M. H. Sørensen, R. Glück, and N. D. Jones. Towards unifying partial evaluation, deforestation, supercompilation, and GPC. In D. Sannella, editor, *Programming Languages and Systems — ESOP '94. Proceedings*, LNCS 788, pages 485–500, Edinburgh, Scotland, 1994. Springer-Verlag.

[260] M. Sperber. Self-applicable online partial evaluation. In O. Danvy, R. Glück, and P. Thiemann, editors, *Proceedings of the 1996 Dagstuhl Seminar on Partial Evaluation*, LNCS 1110, pages 465–480, Schloß Dagstuhl, 1996. Springer-Verlag.

[261] L. Sterling and R. D. Beer. Metainterpreters for expert system construction. *The Journal of Logic Programming*, 6(1 & 2):163–178, 1989.

[262] L. Sterling and E. Shapiro. *The Art of Prolog.* MIT Press, 1986.

[263] M. Stickel. Upside-down meta-interpretation for the model-elimination theorem-proving procedure for deduction and abduction. April 1993.

[264] J. Stillman. *Computational Problems in Equational Theorem Proving.* PhD thesis, State University of New York at Albany, 1988.

[265] K. Stroetmann. A completeness result for SLDNF-resolution. *The Journal of Logic Programming*, 15(4):337–35, April 1993.

[266] P. J. Stuckey. Constructive negation for constraint logic programming. In *Proceedings, Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 328–339, Amsterdam, The Netherlands, July 1991. IEEE Computer Society Press.

[267] P. J. Stuckey. Negation and constraint logic programming. *Information and Computation*, 118(1):12–33, April 1995.

[268] A. Takeuchi and K. Furukawa. Partial evaluation of Prolog programs and its application to meta programming. In H.-J. Kugler, editor, *Information Processing 86*, pages 415–420, 1986.

[269] H. Tamaki and T. Sato. Unfold/fold transformations of logic programs. In S.-Å. Tärnlund, editor, *Proceedings of the Second International Conference on Logic Programming*, pages 127–138, Uppsala, Sweden, 1984.

[270] H. Tamaki and T. Sato. OLD Resolution with Tabulation. In E. Shapiro, editor, *Proceedings of the Third International Conference on Logic Programming*, number 225 in LNCS, pages 84–98, London, July 1986. Springer-Verlag.

[271] P. Tarau and K. De Bosschere. Memoing techniques for logic programs. In Y. Deville, editor, Logic Program Synthesis and Transformation. *Proceedings of LOPSTR'93*, Workshops in Computing, pages 196–209, Louvain-La-Neuve, Belgium, 1994. Springer-Verlag.

[272] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3:121–181, 1995.

[273] V. F. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):292–325, 1986.

[274] V. F. Turchin. The algorithm of generalization in the supercompiler. In D. Bjørner, A. P. Ershov, and N. D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 531–549. North-Holland, 1988.

[275] V. F. Turchin. Program transformation with metasystem transitions. *Journal of Functional Programming*, 3(3):283–313, 1993.

[276] V. F. Turchin. Metacomputation: Metasystem transitions plus supercompilation. In O. Danvy, R. Glück, and P. Thiemann, editors, *Proceedings of the 1996 Dagstuhl Seminar on Partial Evaluation*, LNCS 1110, pages 482–509, Schloß Dagstuhl, 1996. Springer-Verlag.

[277] M. van Emden and R. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM*, 23(4):733–742, 1976.

[278] F. van Harmelen. The limitations of partial evaluation. In P. Jackson and F. van Harmelen, editors, *Logic-Based Knowledge Representation*, pages 87–111. MIT Press, 1989.

[279] R. Venken. A Prolog meta interpreter for partial evaluation and its application to source to source transformation and query optimization. In *ECAI-84: Advances in Artificial Intelligence*, pages 91–100, Pisa, Italy, 1984. North-Holland.

[280] R. Venken and B. Demoen. A partial evaluation system for Prolog: Theoretical and practical considerations. *New Generation Computing*, 6(2 & 3):279–290, 1988.

[281] P. Wadler. Deforestation: Transforming programs to eliminate intermediate trees. *Theoretical Computer Science*, 73:231–248, 1990. Preliminary version in ESOP'88, LNCS 300.

[282] M. Wallace. Compiling integrity checking into update procedures. In J. Mylopoulos and R. Reiter, editors, *Proceedings of IJCAI*, Sydney, Australia, 1991.

[283] D. H. D. Warren. Higher-order extensions to Prolog: Are they needed? In J. E. Hayes, D. Michie, and Y.-H. Pao, editors, *Machine Intelligence 10*, pages 441–454. Ellis Horwood Ltd., Chicester, England, 1982.

[284] D. Weise, R. Conybeare, E. Ruf, and S. Seligman. Automatic online partial evaluation. In *Proceedings of the Conference on Functional Programming Languages and Computer Architectures*, LNCS 523, pages 165–191, Harvard University, 1991. Springer-Verlag.

[285] W. Winsborough. Multiple specialization using minimal-function graph semantics. *The Journal of Logic Programming*, 13(2 & 3):259–290, 1992.

Index

EIN WORT

Ein Wort, ein Satz —: aus Chiffern steigen
erkanntes Leben, jäher Sinn,
die Sonne steht, die Sphären schweigen
und alles ballt sich zu ihm hin.

Ein Wort —, ein Glanz, ein Flug, ein Feuer,
ein Flammenwurf, ein Sternenstrich —,
und wieder Dunkel, ungeheuer,
im leeren Raum um Welt und Ich.

Gottfried Benn, 1941.