University of Southampton

Judgement Day: Terminating Logic Programs

by

Jonathan Charles Martin

A thesis submitted for the degree of
Doctor of Philosophy

in the
Faculty of Engineering and Applied Science
Department of Electronics and Computer Science

June 2000

University of Southampton

ABSTRACT

Faculty of Engineering and Applied Science

Department of Electronics and Computer Science
Doctor of Philosophy

Judgement Day: Terminating Logic Programs

by Jonathan Charles Martin

Program specialisation is a source-to-source program transformation technique which can be used to improve the efficiency of programs. It includes traditional compiler optimisations and also incorporates more aggressive transformations which offer greater potential for improvements in performance.

Termination is a key issue in the construction of fully automatic tools, such as compilers and program specialisers, which are used in program development. For any such tool to be effectively usable by a non-specialist user, a minimal requirement is that it should terminate for all input.

This thesis studies termination of logic programs and termination of program specialisation in particular. Two approaches to the latter are traditionally recognised. The *offline* approach divides the specialisation process into two phases; the first is an analysis phase which gathers termination information which is used to guide the specialisation proper in the second phase. This separation of components provides an identifiable termination component within a tool and is good software engineering practice. It also offers a number of other advantages over the *online* approach where the two phases are intertwined. In logic programming, however, the focus of attention has been on online techniques since they have generally offered better potential for optimisation.

This thesis proposes the first solution to automatic, offline specialisation of logic programs which compares favourably with current online techniques with regard to its optimisation capability. Specifically, it is the first offline technique in logic programming to pass the, so called, KMP test which has become the acid test for program specialisation techniques; the automatic generation of a fast pattern matcher from a naive one.

To this end, a number of techniques for termination analysis are developed culminating in the identification of a useful termination criterion for coroutining logic programs. Such programs are notoriously difficult to prove terminating, yet they provide an extremely useful model for an essential part of the program specialisation process. Tackling this problem in turn leads to the establishment of a solid link between the fields of program specialisation and termination analysis, laying the foundation for the proposed offline approach.

Acknowledgements

First and foremost, I would like to express my deepest gratitude to Andy King for the interest he has always shown in my work and his indispensable support. He has contributed to this thesis in so many ways: from the intensive brainstorming sessions during my visits to Kent, which made the most out of my ideas and helped me to formulate them in writing; through collaboration on two papers which formed a basis for two of the chapters; to the reading and correcting of parts of the final text. The opportunity to work with him and learn from him has been a fruitful and rewarding experience and above all highly enjoyable.

I am extremely grateful also to Paul Soper, my supervisor, whose guidance and encouragement have kept me going to the last. His relaxed approach combined with his uncanny ability to always find the silver lining in my clouds of doom, provided me with the free hand I needed to develop my own ideas and the safety net to catch me when I fell. I would never have believed that anyone could make me feel positive about a complete lack of progress, but Paul even managed that too.

Many thanks to Corin Gurr who proved invaluable in the early stages in helping me to understand the partial evaluation literature and get to grips with the problems involved. His patience in explaining complex issues in detail via email was very much appreciated.

Thanks to Hugh Glaser and Manuel Hermenegildo who between them arranged my visit to the CLIP group in Madrid under the Erasmus scheme, and to all the members of the group: Francisco Bueno, María García de la Banda, Daniel Cabeza, Manuel Carro, Pedro López and Germán Puebla who, together with Manuel, went out of their way to make me feel welcome and ultimately made my visit a pleasant and enjoyable one. I am also grateful to Hugh Glaser, and Pieter Hartel too, for their help and advice on a number of different research and departmental matters.

Also on the home front I would like to thank among others Jon Hallet, Stuart Maclean, Wouter von Oortmerssen, Chris Pratten and the staff of the Hedgehog; Bob Kemp for solving all my UNIX problems and a whole host of others; the inimitable Mark Longley for having a genuinely useful answer to just about everything and, most of all, just for being Mark Longley; Charlie Boardman – a constant source of entertainment; and all the other members of the DSSE group, both staff and researchers, past and present, who have contributed to making my time at Southampton enjoyable and for providing a stimulating environment in which to work.

In no particular order, I would like to thank Florence Benoy for discussions, her careful reading of parts of the text, helpful comments, and her unique style of encouragement; Michael Leuschel for his interest in the work and his most welcome contribution, which has provided a very satisfying conclusion; Elena Marchiori for patient discussions on delay recurrency; Danny de Schreye for several useful discussions, clarifying my understanding of the termination literature and for providing helpful feedback; Bern Martens whose words of encouragement and interest for my work were the first to make me feel that it was actually worthwhile; Kostis Sagonas for his encouragement and interest; Peter Holst Andersen, Maurice Bruynooghe, Stefann Decorte, John Gallagher, Laura Lafave, Fred Mesnard and Jan Smaus for useful discussions and comments; for their constructive comments, the anonymous referees who reviewed the papers forming the basis of this thesis; and all those whom I had useful discussions with, at one time or another over the years, either in person or via

email.

Finally, on the personal side, I am grateful to all those who have been supportive throughout, but in particular to Benjamin Alberti, Cressida Fforde, Andy Winter and my mother.

This work was supported by EPSRC studentship ref. no. 93315269 and, also, in part, by the Nuffield grant ref. no. SCI/180/94/417/G.

Contents

1 Introduction

Imperative languages, such as *C*, *C++* and *Java*, are founded on the idea of giving commands which express which actions must be taken to perform a computational task. They force a programmer to think algorithmically, focusing on the details of the individual steps necessary to solve the problem.

Consider the problem of string searching; trying to find a sequence of characters, or *pattern*, in a piece of text. For example, the problem might be to find the pattern "gorith" in the text of this chapter. An imperative approach would be to form a sequence of instructions which might begin as follows:

1. Compare the first letter of the pattern to the first letter of the text;
2. If they are the same, then compare the next letter of the pattern with the next letter of the text; otherwise compare the first letter of the pattern with the next letter of the text...

The imperative programmer quickly becomes engrossed in the details of specific comparisons between characters, keeping track of which characters have already been compared and which characters should be compared next.

An alternative is to describe the problem *declaratively*. That is, to state *what* the problem is rather than *how* it should be solved. A simple declarative specification of the string searching problem is the following:

pattern is a substring of text if there exist strings a and b such that a + pattern + b = text

where "+" represents string concatenation. From this specification it can be determined, for example, that "gorith" is a substring of "algorithm" because "al" and "m" are strings such that "al" + "gorith" + "m" = "algorithm". This specification is simple and readily understood, but it does not in itself constitute an algorithm and cannot be executed directly on a machine.

The role of declarative programming languages is to bridge this gap between specification and algorithm. Fundamentally, a declarative language must provide support for synthesising from the specification, a correct and efficient algorithm which can be executed. Ideally, the synthesis would be completely automatic direct from the specification, though in practice this ideal has yet to be achieved. The benefits of declarative programming are numerous: smaller, cleaner programs which are easier to understand, to write and to maintain.

1.1 Logic programming

Logic programming is an approach to declarative programming based on first order logic, where specifications are logical formulae. For example, the above string search specification can be translated directly into the following first order logical formula:

$$\forall \text{ pattern } \forall \text{ text } (\text{Substring(pattern, text)} \leftarrow \exists \text{ a } \exists \text{ b } (\text{Concat(a, pattern, prefix)} \wedge \text{Concat(prefix, b, text)}))$$

Substring and Concat are called *relation symbols* or *predicate symbols*. Substring(x, y) is read as "x is a substring of y" and Concat(x, y, z) is read as "z is the concatenation of x and y", or "$z = x + y$". The symbols $\leftarrow$, $\wedge$, $\forall$ and $\exists$ have their usual meaning in first order logic, namely "if", "and", "for all" and "there exists" respectively. The close correspondence between this formula and the original specification should now be apparent.

The equation "*Algorithm = Logic + Control*" due to Kowalski 1979, captures the main idea behind logic programming. The "*Logic*" part of the equation represents the purely declarative component; the logical specification provided by a number of logical formulae such as the one above. The "*Control*" part represents the *procedural interpretation* of the logic and determines how the problem should be solved. The procedural interpretation of the above logic for the string searching problem might be something like:

> Choose a string a and calculate a + pattern = prefix. If prefix + b = text for some string b then stop, otherwise repeat with a different string a.

An alternative strategy is the following:

> Divide text into two substrings, prefix and b. Divide prefix into two substrings a and c. If c = pattern then stop, otherwise repeat with a different division of text and/or prefix.

Different algorithms for the same problem can be obtained by varying the control component, as in the example, or alternatively by replacing the logic component with another one that is (logically) equivalent. Hence, the automatic synthesis of programs from specifications focuses on these two techniques. The *control generation problem* deals with the automatic derivation of a suitable control component for a given logical specification. The aim of *program transformation* is to improve the efficiency of an algorithm through transformation of the logic component. In this case, the control component is usually kept the same.

Modern logic programming systems provide a default control component so that logical specifications can be written directly as programs. A system will usually provide a number of mechanisms for the programmer to refine the control in order to tune the efficiency of the program. It is often the case, however, that simple, high-level specifications lead to inefficient algorithms regardless of the control. For example, expressing the logical formula above directly in a logic programming system would result in a program which is very inefficient. Most likely, it would implement a (rather poor) brute force search algorithm, attempting to match the pattern with the text at all possible positions until finding a match. Modifying the control component will do little, if anything, to improve the underlying complexity of this program (though it could make it worse!). Any fundamental improvement can only be achieved through modification of the logic.

1.2 Specialisation of logic programs

Most of the efficient string searching algorithms that have been devised rely on some form of preprocessing on the pattern to be searched for. They exploit knowledge about

the given pattern in order to search for it more quickly. This knowledge can also be exploited to improve the performance of the brute force search program encountered above. In fact, for any given pattern, the program can be transformed into another which mimics the efficient Knuth-Morris-Pratt string matching algorithm (Pettorossi *et al.* 1996, Knuth *et al.* 1977). This kind of transformation, which is directed by part of a program's input, is called *program specialisation*.

The example provides a useful illustration of the basic principles of program specialisation. The search program takes two inputs, a pattern p and some text t. When the pattern is known the program may be specialised with respect to this input, resulting in a program which takes t as its only argument. The pattern p which was an input to the original program is incorporated into the specialised version. This new version may be used to efficiently search for p in any number of different texts, but cannot of course be used to search for other patterns distinct from p in those texts. To search for a new pattern q, the original program must be used or alternatively it can be specialised as before with respect to the pattern q. In program specialisation parlance, any input known at compile time such as p is *static*, while an input like t, which only becomes available at runtime is *dynamic*. In general, a program may have more than two inputs, but each of these can be classified as static or dynamic. The program would then be specialised with respect to its static inputs.

Observe that the specialisation process in the example corresponds to the preprocessing that would ordinarily occur in a hand-coded version of the Knuth-Morris-Pratt algorithm. But as Sedgewick 1990 points out, "the Knuth-Morris-Pratt algorithm requires some complicated preprocessing on the pattern that is difficult to understand and has limited the extent to which it is used.". One advantage of the naive specification/transformation approach is that, with problems of this kind, the programmer neither needs to devise any complicated preprocessing, understand it, program it nor even be aware of it. Program specialisation offers the possibility of *automatically* deriving correct, efficient algorithms from simple specifications for a variety of application areas.

1.3 Termination and correctness

Reconsider the two example control strategies in Section 1.1. Both involve some repetition, essentially of the form "If pattern not found then repeat". Arising out of this repetition is the question as to whether or not this process of searching for the pattern will ever end. In the second strategy, the process is repeated "with a different division of text and/or prefix". Since there is only a finite number of distinct ways to divide a string into two, the search process must end as either the pattern will be found or the possible divisions of the string will be exhausted without finding a match. With the first strategy, however, a similar reasoning cannot be applied. The search process is repeated each time "with a different string a" without any constraint on the form or length of a. An infinite number of strings could be tried and the pattern might never be found (even if it occurs in text).

Formally, a process or computation which ends is said to terminate or be terminating and one which never ends is said not to terminate or be non-terminating. In contrast to imperative languages which contain several looping constructs, e.g. *while, for, goto, repeat* the only possible cause of non-termination in logic programs is recursion (i.e. predicates defined in terms of themselves). For example, consider the

following

Brother(x, y) ← Brother(y, x).

This rule for the Brother relation is recursively defined and its intended interpretation is that "x is the brother of y if y is the brother of x". Thus given the fact Brother(Alan, Deryk), one may deduce Brother(Deryk, Alan) from the above rule. But applying the rule to this last fact one may further deduce the original fact Brother(Alan, Deryk). There is no limit to the number of times that this rule may be applied

An issue closely related to termination, which has already been mentioned in passing is *program correctness*. Loosely speaking, a program is said to be *partially correct* if for any input, whenever it terminates, it produces the right output. A program is *totally correct* if it is partially correct and terminates for all inputs. Partial correctness is relatively easy to achieve in logic programming given the close correlation between logical specifications and logic programs. Thus the primary concern of control generation, with regard to correctness, is termination.

1.4 Partial evaluation

One form of program specialisation which has attracted a considerable amount of interest is partial evaluation. In its simplest form, partial evaluation consists of the evaluation or *reduction* of expressions combined with *unfolding*.

The evaluation of an expression during the specialisation process is performed exactly as it would be at runtime. Reduction of expressions occurs when full evaluation is not possible. As an example, consider the expression $(x + y) * z$ occurring in a program P. During the specialisation of P, this expression can be evaluated only if the values of x, y and z are known (static). Now suppose that $x = 3$ and $y = 4$, but z is dynamic. Then the expression cannot be evaluated but it may be reduced to $7 * z$. This *residual* expression must form part of the specialised program and may be evaluated at runtime when the value of z becomes known.

An *unfolding* step conceptually replaces a procedure, function or predicate call with the body of the called procedure, function or predicate. Unfolding then, is similar in spirit to the idea of *inlining* used in imperative languages. Since predicates, etc., may be defined recursively it is immediately apparent that unfolding steps may be followed one after another *ad infinitum*. Infinite unfolding will, of course, lead to non-termination of the specialisation process and must be avoided. Unfolding, itself, cannot be avoided altogether, since very little specialisation can be achieved without it. Thus, means are required to curb its occasional tendency to get carried away.

1.5 Aims and outline of the thesis

This thesis is divided into two parts. The second part comprises the original contribution while the first provides the necessary technical background. The main thread follows the development of an approach to the control of unfolding of logic programs. The key idea is to use static termination analysis to derive sufficient conditions for finite unfolding. Since unfolding effectively models the computation process, this problem is closely related to the control generation problem which is also examined.

Chapters 2, 3 and 4 respectively introduce the basics of logic programming, termination of logic programs and partial evaluation of logic programs.

Norms, which measure the size of data structures, play an important role in modern termination analyses for logic programs. Chapter 5 examines how norms can be automatically derived from the types of a logic program. An earlier version of this chapter appeared in Martin *et al.* 1996.

Much of the practical static termination analysis work that has been developed builds on a few theoretical characterisations of termination. These characterisations, however, are not cast in terms of the recursive structure of programs which in itself forms an intuitive and practical basis for reasoning about termination. Alternative characterisations based on the recursive structure of programs are proposed in Chapter 6, which potentially provide a more useful foundation for practical termination analyses.

Chapter 7 studies the control generation problem and shows how it can be tackled using a transformation approach. The emphasis is on termination but the framework developed is sufficiently flexible to allow a range of search strategies to be incorporated within it. This offers the opportunity to tune the efficiency of a program once its total correctness has been established. This chapter is a revised and extended version of Martin & King 1997.

Chapter 8 extends the results of the previous chapter to obtain a refined strategy for unfolding. As before, finiteness of the unfolding process is ensured without unduly restricting the search strategy which is applied. Within this framework there is not only scope for improving the efficiency of the unfolding, but also for introducing *determinacy control* in an independent way. Determinacy control amounts to deciding whether certain choices in the search space should be explored at specialisation time or at runtime. Making the wrong decision can lead to a specialised program which is *less* efficient than the original program. Clearly, the ability to incorporate determinacy control in a specialiser is essential, and this is catered for within the framework. The main results of this chapter can also be found in Martin & Leuschel 1999.

Pre-requisites Though not essential, it would certainly be helpful if the reader had some understanding of the basic results of computability theory (see, e.g, Harel 1989), including Turing machines, the halting problem and undecidability. Some experience with a logic programming language such as Prolog would also be useful. A basic grounding in computer science and a rudimentary knowledge of set theory is assumed throughout.

Part I

Technical Background

2 Logic Programming

Logic programming is based on the fundamental idea, mainly due to Kowalski 1974 and Colmerauer *et al.* 1973, that a subset of first order logic may be given a procedural interpretation and hence used as a programming language. In this chapter, the basic concepts of typed logic programming are reviewed beginning with the syntax and semantics of typed first order logic. This is then used as a basis for the development of the syntax and semantics of typed logic programs. The presentation closely follows Lloyd 1987 and Hill & Lloyd 1994 and the reader is encouraged to consult these texts for further details.

2.1 Syntax of polymorphic many-sorted languages

Definition 2.1 (alphabet) An *alphabet* of a first order, many-sorted, polymorphic language is composed of the following classes of symbols

1. a countably infinite set U of *parameters* (type variables);
2. a finite set $\Sigma_\tau = \Sigma_{base} \cup \Sigma_{constructor}$ of type constructor symbols where
   (a) Σ_{base} is a non-empty set of symbols of arity zero called *bases* and
   (b) $\Sigma_{constructor}$ is a set of symbols of arity $n > 0$ called *constructors*;
3. a countably infinite set V of variables;
4. a finite set $\Sigma_{fun} = \Sigma_{constant} \cup \Sigma_{functor}$ of function symbols where
   (a) $\Sigma_{constant}$ is a non-empty set of symbols of arity zero called *constants* and
   (b) $\Sigma_{functor}$ is a set of symbols of arity $n > 0$ called *functors*;
5. a non-empty finite set $\Sigma_{pred} = \Sigma_{proposition} \cup \Sigma_{predicate}$ of predicate symbols where
   (a) $\Sigma_{proposition}$ is a set of symbols of arity zero called *propositions* and
   (b) $\Sigma_{predicate}$ is a set of symbols of arity $n > 0$ called *predicates*;
6. the connectives $\neg$ (negation), $\wedge$ (conjunction), $\vee$ (disjunction), $\leftarrow$ (implication) and $\leftrightarrow$ (equivalence);
7. the universal quantifier $\forall$ and the existential quantifier $\exists$;
8. the punctuation symbols "(", ")", "{", "}", "." and ",".

Classes 1–5 distinguish different alphabets whilst classes 6–8 are the same for all alphabets. The syntactic objects of a language are the strings that can be built from the symbols of its alphabet. In order to define the semantics of a language, that is to

associate meanings with its syntactic objects, attention is usually restricted to a subset of objects which are *well-formed* in some sense. Objects which are not well-formed have no meaning. Furthermore, in a typed language, the property of being well-formed also includes that of being *well-typed*.

In the following sections these notions are made more precise, and the well-formed objects of a first order, many-sorted, polymorphic language are defined.

2.1.1 Types

Definition 2.2 (type) A *type* is defined inductively as follows

1. A parameter is a type.
2. A base is a type.
3. If c is a constructor of arity n and $\tau_1, \ldots, \tau_n$ are types then $c(\tau_1, \ldots, \tau_n)$ is a type.

A *ground type* or *monotype* is a type not containing parameters. □

The set of parameters occuring in a syntactic object o is denoted by $pars(o)$. A syntactic object which contains no parameters is said to be *type-ground*.

Example 2.1 Let $U = \{u, u_1, u_2, \ldots\}$, $\Sigma_{base} = \{int\}$ and $\Sigma_{constructor} = \{list/1\}$ (i.e. *list* is a constructor of arity one). Then *int*, $list(int)$, u, and $list(list(u))$ are examples of types, the first two of which are type-ground since they contain no parameters. Furthermore, $pars(u) = pars(list(list(u))) = \{u\}$. □

Given a type τ containing parameters, a new type may be obtained from τ by replacing certain parameters of τ with other types. For example, the type $list(int)$ may be obtained from the type $list(u)$ by substituting the type *int* for the parameter u. In this case, the type $list(int)$ is said to be a *type instance* of $list(u)$ by the *type substitution* which *binds* u to *int*. These concepts are formally defined below.

Definition 2.3 (type substitution) A *type substitution* ψ is a finite set $\{u_1/\tau_1, \ldots u_n/\tau_n\}$ of *type bindings* where $u_1, \ldots, u_n$ are distinct parameters and $\tau_1, \ldots, \tau_n$ are types such that $u_i \neq \tau_i$ for all $i \in [1, n]$. □

Definition 2.4 (parameter renaming) A type substitution $\psi = \{u_1/\tau_1, \ldots u_n/\tau_n\}$ is a *parameter renaming* iff τ_i is a parameter for all $i \in [1, n]$. □

Definition 2.5 (type instance) Let $\psi = \{u_1/\tau_1, \ldots, u_n/\tau_n\}$ be a type substitution and o a syntactic object. Then the *type instance* of o by ψ, denoted $\psi(o)$, is the syntactic object obtained from o by simultaneously replacing each occurence of the parameter u_i in o by the type τ_i for all $i \in [1, n]$. If $\psi(o)$ is type ground then $\psi(o)$ is a *type-ground instance* of o, and ψ is called a *type-grounding substitution* for o. □

Any two type substitutions ψ and ϑ can be *composed* to produce a third type substitution κ such that for any type τ the type instances $\vartheta(\psi(\tau))$ and $\kappa(\tau)$ are equivalent.

Definition 2.6 (composition of type substitutions) Let $\psi = \{a_1/\tau_1, \ldots, a_m/\tau_m\}$ and $\vartheta = \{b_1/\sigma_1, \ldots, b_n/\sigma_n\}$ be type substitutions. Then the *composition* of ψ and ϑ, denoted $\psi\vartheta$, is the set $\{a_i/\tau_i\vartheta \mid a_i \neq \tau_i\vartheta \wedge i \in [1, m]\} \cup \{b_i/\sigma_i \mid b_i \notin \{a_1, \ldots, a_m\} \wedge i \in [1, n]\}$. □

The above definition is based on Definition 2.3.1 in Leuschel 1997 which defines how term substitutions are composed. It is easy to see that the composition of any two type substitutions according to Definition 2.6 is also a type substitution. Moreover, it can be shown that, given type substitutions ψ and ϑ, for any type τ, the type instances $\vartheta(\psi(\tau))$ and $(\psi\vartheta)(\tau)$ are equivalent.

If, following the application of a type substitution ψ to two types τ and σ, the two resulting type instances $\psi(\tau)$ and $\psi(\sigma)$ are equal to each other, then ψ is called a *type unifier* of τ and σ. If τ and σ are unifiable in this way, then there may be many type unifiers for the two types. A *most general type unifier* of τ and σ is intuitively one of the simplest of these unifiers.

Definition 2.7 (mgtu) Let S be a finite set of type equations $\{\sigma_1 = \tau_1, \ldots, \sigma_n = \tau_n\}$. A type substitution ψ is a *type unifier* of S iff $\psi(\sigma_i) \equiv \psi(\tau_i)$ for all $i \in [1, n]$. A type unifier ψ of S is a *most general type unifier*, or *mgtu*, of S iff for each type unifier ψ' of S there exists a type substitution ϑ such that $\psi' = \psi\vartheta$. □

The notion of a most general type unifier will play an important role in the construction of terms, atoms and formulae (see Section 2.1.2) and also in the application of term substitutions (see Section 2.4.2).

In a many-sorted, polymorphic language each function symbol is assigned a *function type* and each predicate symbol is assigned a *predicate type*. The assignment is such that the types of function and predicate symbols are unique modulo parameter renaming[1].

Definition 2.8 (function and predicate type) A *function type* (resp. *predicate type*) takes the form $\langle \tau_1 \ldots \tau_n, \tau \rangle$ (resp. $\tau_1 \ldots \tau_n$) where $\tau_1 \ldots \tau_n$ is a (possibly empty) string of types and τ is a type which is not a parameter. □

In function and predicate types, the empty string will be denoted by ϵ. Note that all constants have a function type of the form $\langle \epsilon, \tau \rangle$ and all propositions have predicate type ϵ.

Definition 2.9 (transparency) A function type $\sigma = \langle \tau_1 \ldots \tau_n, \tau \rangle$ is *transparent* iff every parameter occuring in σ also occurs in τ. That is, $pars(\sigma) \subseteq pars(\tau)$. □

Definition 2.10 (type assignment) Let Σ_τ, Σ_{fun}, Σ_{pred} be respectively the sets of type constructor symbols, function symbols and predicate symbols of a first order, many-sorted, polymorphic language. A *type assignment* is a mapping from function symbols to transparent function types and from predicate symbols to predicate types, such that each function symbol of arity n maps to a type of the form $\langle \tau_1 \ldots \tau_n, \tau \rangle$ and each predicate symbol of arity n maps to a type of the form $\tau_1 \ldots \tau_n$. □

It will usually be more convenient to work with sets of typed function and predicate symbols rather than the untyped ones defined by an alphabet. These shall be denoted by Σ_f and Σ_p respectively and are defined as follows

- $\Sigma_f = \{f_\sigma \mid f \in \Sigma_{fun} \wedge \sigma \text{ is the type assigned to } f\}$

[1] For overloaded symbols, for example +, it is assumed that the symbol is uniquely renamed for each of its types.

- $\Sigma_p = \{p_\sigma \mid p \in \Sigma_{pred} \wedge \sigma$ is the type assigned to $p\}$

In addition, use will often be made of the set of all type-ground instances of Σ_f, and the set of all type-ground instances of Σ_p defined by

- $\Sigma_f^* = \{f_\delta \mid f_\sigma \in \Sigma_f \wedge \delta$ is a type-ground instance of $\sigma\}$
- $\Sigma_p^* = \{p_\delta \mid p_\sigma \in \Sigma_p \wedge \delta$ is a type-ground instance of $\sigma\}$

Finally, whenever $f_{\langle\tau_1 \ldots \tau_n, \tau\rangle} \in \Sigma_f$ and $n > 0$, the type τ is called the *range type* of $f_{\langle\tau_1 \ldots \tau_n, \tau\rangle}$.

2.1.2 Terms, Atoms and Formulae

Typed terms, atoms and formulae can be defined in much the same way as in an untyped language, though there is of course the additional complication of ensuring that each construct is well-typed. In particular, it is necessary to ensure that every subterm in a term, atom or formula has a type and that multiple occurences of a variable all have the same type. In each case, the well-typing is guaranteed by finding a most general unifier of a set of type equations. If such a unifier does not exist then the construct cannot be well-typed and the construction fails. The following definitions formalise these notions.

Definition 2.11 (typed variable) If $v \in V$ is a variable and τ is a type, then v_τ denotes a *typed variable*. If o is a syntactic object, then $vars(o)$ is the set of typed variables which occur in o. □

Definition 2.12 (term, subterm) *Terms* and *subterms* are defined inductively as follows

1. If $v \in V$ and $u \in U$ then v_u is a term of type u. The only subterm of v_u is v_u itself.
2. If $c_{\langle\epsilon,\tau\rangle} \in \Sigma_f$ then $c_{\langle\epsilon,\tau\rangle}$ is a term of type τ. The only subterm of $c_{\langle\epsilon,\tau\rangle}$ is $c_{\langle\epsilon,\tau\rangle}$ itself.
3. Let $f_{\langle\sigma_1 \ldots \sigma_n, \sigma\rangle} \in \Sigma_f$ and for all $i \in [1, n]$ let t_i be a term of type τ_i such that
   - $(pars(\sigma_1) \cup \ldots \cup pars(\sigma_n) \cup pars(\sigma)) \cap pars(\tau_i) = \emptyset$, and
   - $pars(\tau_i) \cap pars(\tau_j) = \emptyset$ for all $j \in [1, n]$, $i \neq j$.

   If, and only if, there exists a most general unifier ψ of the set of type equations

   $$\{\sigma_i = \tau_i \mid i \in [1, n]\} \cup \{\rho_i = \rho_j \mid v_{\rho_i} \in vars(t_i) \wedge v_{\rho_j} \in vars(t_j) \wedge i, j \in [1, n]\}$$

   then $t = \psi(f_{\langle\sigma_1 \ldots \sigma_n, \sigma\rangle}(t_1, \ldots, t_n))$ is a term of type $\psi(\sigma)$. Furthermore, t is a subterm of t and for all $i \in [1, n]$, if s is a subterm of t_i then $\psi(s)$ is a (strict) subterm of t. □

Recall that every function symbol $f \in \Sigma_{fun}$ has a function type, $\langle\tau_1 \ldots \tau_n, \tau\rangle$ say, such that $f_{\langle\tau_1 \ldots \tau_n, \tau\rangle} \in \Sigma_f$. Each occurence of f in a term t, however, has a *relative type* in t which is an instance of its function type $\langle\tau_1 \ldots \tau_n, \tau\rangle$. It is this relative type which appears in the term as a subscript to the function symbol. It should be noted that a relative type is unique up to parameter renaming.

Although variables are not assigned types in the same way as function symbols, each occurence of a variable in a term t also has a unique relative type in t. Again it

is this relative type which appears in the term as a subscript to the variable. Where a variable is defined to be a term, its relative type is defined to be a parameter. This ensures that the relative type is most general in the sense that the variable could stand for any term regardless of its type. The relative type of a variable when it appears as a subterm of a term t depends upon the context, i.e. on the function types in t. Here again the relative type is most general but now in the sense that the variable could stand for any term of an appropriate type determined by the context.

Observe that multiple occurences of a variable in a term t all have the same relative type in t. This is guaranteed by the construction of the term. Whenever a variable v occurs in terms t_i and t_j, and v has relative types ρ_i and ρ_j in t_i and t_j respectively, ρ_i and ρ_j must be unifiable if a new term is to be constructed which has (type instances of) t_i and t_j as strict subterms.

A consequence of explicitly tagging all occurences of function symbols and variables in a term with their relative types is that a subterm s of a term may not necessarily be a term itself. Instead, for every subterm s of a term there exists a term s' and a type substitution ψ such that $s = \psi(s')$.

Terms may alternatively be written such that each function symbol and variable appears without its relative type. This will usually be done when the types are clear from the context or irrelevant. Note that, when all terms are written in this form, every subterm of a term is also a term. Furthermore, for a term t in this form, $vars(t)$ denotes the set of variables in t rather than the set of typed variables.

Definition 2.13 (atom) An *atom* is defined as follows

1. A proposition $p_\epsilon \in \Sigma_p$ is an atom.
2. Let $p_{\sigma_1 \ldots \sigma_n} \in \Sigma_p$ and for all $i \in [1, n]$ let t_i be a term of type τ_i such that
   - $(pars(\sigma_1) \cup \ldots \cup pars(\sigma_n)) \cap pars(\tau_i) = \emptyset$, and
   - $pars(\tau_i) \cap pars(\tau_j) = \emptyset$ for all $j \in [1, n]$, $i \neq j$.

   If, and only if, there exists a most general unifier ψ of the set of type equations

   $$\{\sigma_i = \tau_i \mid i \in [1, n]\} \cup \{\rho_i = \rho_j \mid v_{\rho_i} \in vars(t_i) \wedge v_{\rho_j} \in vars(t_j) \wedge i, j \in [1, n]\}$$

   then $A = \psi(p_{\sigma_1 \ldots \sigma_n}(t_1, \ldots, t_n))$ is an atom. For all $i \in [1, n]$, if s is a subterm of t_i then $\psi(s)$ is a subterm of A. □

The predicate symbol of an atom A is denoted by $rel(A)$. Formulae are formed by combining atoms using the connectives and quantifiers of the underlying language. If F is a formula, then every subterm of an atom in F is also a subterm of F.

Definition 2.14 (formula and free variables) The set of *free variables* of a *(polymorphic, many-sorted) formula* F is denoted by $freevars(F)$ where a formula and its free variables are defined inductively as follows

1. An atom A is a formula and $freevars(A) = vars(A)$.
2. If F is a formula then so is $\neg F$ and $freevars(\neg F) = freevars(F)$.
3. Let F be a formula with $v \in freevars(F)$. Then $\forall v F$ and $\exists v F$ are formulae whose free variables are given by $freevars(F) \setminus \{v\}$.

4. Let F and G be formulae whose common variables are free in both F and G. Suppose further that

   - $\mathit{pars}(F) \cap \mathit{pars}(G) = \emptyset$, and
   - there exists a most general unifier ψ of the set of type equations

   $$\{\rho_F = \rho_G \mid v_{\rho_F} \in \mathit{freevars}(F) \wedge v_{\rho_G} \in \mathit{freevars}(G)\}$$

   Then $\psi(F \wedge G)$, $\psi(F \vee G)$, $\psi(F \leftarrow G)$ and $\psi(F \leftrightarrow G)$ are all formulae whose free variables are given by $\{\psi(v) \mid v \in \mathit{freevars}(F) \cup \mathit{freevars}(G)\}$. □

Definition 2.15 (type of a subterm) Given a syntactic object o, and a subterm t of o, the type of t in o is defined by

$$\mathit{type}(t, o) = \begin{cases} \tau & \text{if } t = v_\tau \text{ and } v \in V \\ \tau & \text{if } t = f_{\langle \tau_1 \ldots \tau_n, \tau \rangle}(t_1, \ldots, t_n) \end{cases}$$

Observe that multiple occurences of a variable in a formula are all of the same type in the formula. Also the types of subterms of a formula are unique up to parameter renaming. A term, atom or formula is said to be *ground* if it does not contain any variables.

Definition 2.16 (literal) If A is an atom, then the formulae A and $\neg A$ are called literals.

Definition 2.17 (conjunction, disjunction) Let $A_1, \ldots, A_n$ be atoms (resp. literals). Then $A_1 \wedge \ldots \wedge A_n$ (which may also be written as $A_1, \ldots, A_n$) is a *conjunction* of atoms (resp. literals) and $A_1 \vee \ldots \vee A_n$ is a *disjunction* of atoms (resp. literals). □

Definition 2.18 (closed formula) A formula is *closed* iff it has no free variables. □

Definition 2.19 (universal and existential closure) Let F be a formula whose free variables are $v_1, \ldots, v_n$. Then

- $\forall(F) = \forall v_1, \ldots, v_n F$ denotes the *universal closure* of F.
- $\exists(F) = \exists v_1, \ldots, v_n F$ denotes the *existential closure* of F. □

Definition 2.20 (language) The *polymorphic many-sorted language* given by an alphabet consists of the set of polymorphic many-sorted formulae that can be constructed from the symbols of the alphabet. □

Since certain symbols of an alphabet appear in all alphabets, and the types of interest are closely associated with the function and predicate symbols, a polymorphic, many-sorted, first-order language can be defined by a triple $\mathcal{L} = \langle \Sigma_p, \Sigma_f, V \rangle$.

Definition 2.21 (theory) A *polymorphic many-sorted theory* consists of a polymorphic many-sorted language and a set of axioms which is a designated subset of closed formulae in the language of the theory. □

2.2 Semantics of polymorphic many-sorted languages

The sentences of a language may be thought of as statements about a "world" of objects and the relations among those objects. The objects in the world are represented in the language by terms, formed from the constant and function symbols of the language. The precise association between terms and objects is given by a *pre-interpretation*. A pre-interpretation for a language defines a world and uniquely identifies each term in the language with an object in the world. This "world" is formally called a *domain*, though in a typed language it is split into a family of domains, one domain for each ground type in the language.

Definition 2.22 (pre-interpretation) A *pre-interpretation* of a first order polymorphic many-sorted language is a pair $J = \langle D_J, \Lambda_J \rangle$ where

1. $D_J = \{D_{J,\delta} \mid \delta \text{ is a ground type}\}$, is a family of *domains* where each $D_{J,\delta}$ is a non-empty set called the *domain of type* δ in the pre-interpretation J;
2. Λ_J is an assignment defined such that
   (a) Each $f_{\langle \epsilon,\delta \rangle} \in \Sigma_f^*$ is assigned an element in $D_{J,\delta}$;
   (b) Each $f_{\langle \delta_1 \ldots \delta_n,\delta \rangle} \in \Sigma_f^*$ is assigned a mapping from $D_{J,\delta_1} \times \ldots \times D_{J,\delta_n}$ to $D_{J,\delta}$.

□

The ground, monomorphic, atomic formulae (ground, monomorphic atoms) of a language, express simple statements about the objects of the world (the elements of the domains of the pre-interpretation). An *interpretation* directly determines the truth or falsity of each of these statements.

Definition 2.23 (interpretation) An *interpretation* I_J for a polymorphic many-sorted language consists of a pre-interpretation $J = \langle D_J, \Lambda_J \rangle$ and an assignment of truth functions to elements of Σ_p^* such that

1. Each $p_\epsilon \in \Sigma_p^*$ is assigned a value *true* or *false*;
2. Each $p_{\delta_1 \ldots \delta_n} \in \Sigma_p^*$ is assigned a mapping from $D_{J,\delta_1} \times \ldots \times D_{J,\delta_n}$ to $\{$ *true*, *false* $\}$.

□

To determine truth values for non-ground (monomorphic) atoms, the notions of variable assignment and term assignment are used.

Definition 2.24 (variable assignment) Let $J = \langle D_J, \Lambda_J \rangle$ be a pre-interpretation. A *variable assignment* V_J maps each variable of type δ to an element of $D_{J,\delta}$, where δ is a ground type. □

Definition 2.25 (term assignment) Given a pre-interpretation $J = \langle D, \Lambda \rangle$ and a variable assignment V_J, the *term assignment wrt* J *and* V_J, denoted $\phi_{J,V}$, is defined as follows:

1. $\phi_{J,V}(v_\delta) = V_J(v_\delta)$ for all $v \in V$ and for every ground type δ.

2. $\phi_{J,V}(f_{\langle\epsilon,\delta\rangle}) = \Lambda_J(f_{\langle\epsilon,\delta\rangle})$

3. $\phi_{J,V}(f_{\langle\delta_1\ldots\delta_n,\delta\rangle}(t_1,\ldots,t_n)) = \Lambda_J(f_{\langle\delta_1\ldots\delta_n,\delta\rangle})(\phi_{J,V}(t_1),\ldots,\phi_{J,V}(t_n))$

for all $f_{\langle\epsilon,\delta\rangle} \in \Sigma_f^*$ and $f_{\langle\delta_1\ldots\delta_n,\delta\rangle} \in \Sigma_f^*$. □

Definition 2.26 (truth value of a monomorphic atom) Let I_J be an interpretation, V_J a variable assignment and $a = p_{\delta_1\ldots\delta_n}(t_1,\ldots,t_n)$ a monomorphic atom. The *truth value* of a (wrt I_J and V_J) is *true* iff $p_{\delta_1\ldots\delta_n}(\phi_{J,V}(t_1),\ldots,\phi_{J,V}(t_n))$ maps to true in I_J, where $\phi_{J,V}$ is the term assignment wrt J and V, and *false* otherwise. □

Clearly, the truth value of a ground monomorphic atom depends solely on the intepretation I_J. The convention is followed here of overloading I_J to additionally represent the set of ground atoms whose truth values are *true* wrt I_J. That is $a \in I_J$ iff a maps to *true* in I_J. The truth values of monomorphic formulae can easily be defined in an analogous manner to the untyped case.

Definition 2.27 (truth value of a monomorphic formula) Let I_J be an interpretation, V_J a variable assignment and w a monomorphic formula. The *truth value* of w (wrt I_J and V_J) is determined as follows.

1. If w is an atom, then its truth value is determined by Definition 2.26.

2. If w has the form $\neg F$, $F \wedge G$, $F \vee G$, $F \leftarrow G$ or $F \leftrightarrow G$, then the truth value of w is given by the following table

$\mathcal{T}(F)$	$\mathcal{T}(G)$	$\mathcal{T}(\neg F)$	$\mathcal{T}(F \wedge G)$	$\mathcal{T}(F \vee G)$	$\mathcal{T}(F \leftarrow G)$	$\mathcal{T}(F \leftrightarrow G)$
true	*true*	*false*	*true*	*true*	*true*	*true*
true	*false*	*false*	*false*	*true*	*true*	*false*
false	*true*	*true*	*false*	*true*	*false*	*false*
false	*false*	*true*	*false*	*false*	*true*	*true*

   where $\mathcal{T}(F)$ is the truth value of F.

3. If w has the form $\exists v_\delta F$, then the truth value of w is *true* iff there exists $d \in D_{J,\delta}$ such that F has truth value *true* wrt I_J and $V_J(v_\delta/d)$ where $V_J(v_\delta/d)$ is V_J except that v_δ is assigned d; otherwise, its truth value is false.

4. If w has the form $\forall v_\delta F$, then the truth value of w is *true* iff F has truth value *true* wrt I_J and $V_J(v_\delta/d)$, for all $d \in D_{J,\delta}$; otherwise, its truth value is false.

□

Note that the truth value of a *closed* formula depends only on the interpretation and not on the variable assignment. Thus a closed formula F is said to be true (resp. false) wrt an interpretation I iff the truth value of F is true (resp. false) wrt I. Truth values of polymorphic formulae can now be defined in terms of truth values for monomorphic formulae.

Definition 2.28 (truth value of a polymorphic formula) Let I be an interpretation of a polymorphic many-sorted language L and let w be a closed polymorphic formula of L. Then w is *true* wrt I iff for every grounding type substitution ψ for w, $\psi(w)$ is *true* wrt I. On the other hand, w is *false* wrt I iff there exists a grounding type substitution ψ for w such that $\psi(w)$ is *false* wrt I. □

From the above, it can be seen that the truth or falsity of any formula in a language is determined by an interpretation for the language. In general, only the formulae of a given theory are of interest at any one time. Furthermore, the interpretations of interest will be those for which all the formulae of the theory are true. Such interpretations are called models.

Definition 2.29 (model) Let I be an interpretation of a polymorphic many-sorted language L and let w be a closed formula of L. Then I is a *model* for w, denoted $I \models w$, iff w is *true* wrt I. If S is a set of closed polymorphic formulae in L then $I \models S$ iff $I \models w$ for all $w \in S$. □

Definition 2.30 (logical consequence) Given a closed polymorphic formula w and a set S of closed polymorphic formulae, w is a *logical consequence* of S, denoted $S \models w$, iff for every interpretation I, the premise $I \models S$ implies the conclusion $I \models w$. □

2.3 Syntax of polymorphic many-sorted programs

Definition 2.31 (statement) A *statement* s is of the form

$$H \leftarrow B$$

where H is an atom, called the *head*, and B is either absent or a polymorphic many-sorted formula, called the *body*. The free variables of s are assumed to be universally quantified at the front of s. The set of atoms appearing in B is denoted by $body(s)$. □

Definition 2.32 (program) A *polymorphic many-sorted logic program* is a pair $\langle \Delta, S \rangle$ where Δ is a triple $\langle \Delta_\tau, \Delta_f, \Delta_p \rangle$ of type declarations and S is a finite set of statements. The type declarations Δ_τ, Δ_f and Δ_p define respectively Σ_τ, Σ_f and Σ_p in the following way

1. Each constant declaration $c : \tau \in \Delta_f$ implies $c_{\langle \epsilon, \tau \rangle} \in \Sigma_f$.
2. Each function declaration $f : \tau_1 \times \ldots \times \tau_n \rightarrow \tau \in \Delta_f$ implies $f_{\langle \tau_1 \ldots \tau_n, \tau \rangle} \in \Sigma_f$.
3. Each proposition declaration $p \in \Delta_p$ implies $p_\epsilon \in \Sigma_p$.
4. Each predicate declaration $p : \tau_1 \times \ldots \times \tau_n \in \Delta_p$ implies $p_{\tau_1 \ldots \tau_n} \in \Sigma_p$

Furthermore, the following four conditions must be satisfied

1. Each statement is a polymorphic many-sorted formula in the language defined by the type declarations.
2. Each function declaration is *transparent* (see below).
3. Each statement satisfies the *head condition* (see below).
4. Δ_f and Δ_p are universal. That is each symbol has exactly one declaration in Δ_f (resp. Δ_p) so that Σ_f (resp. Σ_p) is well-defined.

Definition 2.33 (transparency) A declaration for a function $f : \tau_1 \times \ldots \times \tau_n \rightarrow \tau$ is *transparent* iff every parameter appearing in the declaration also appears in the range type τ. □

Definition 2.34 (head condition) Let $p_{\sigma_1 \ldots \sigma_n}$ be the typed predicate symbol occuring in the head of a statement s and let $p : \tau_1 \times \ldots \times \tau_n$ be the type declaration for p. Then s satisfies the *head condition* iff $\sigma_1 \ldots \sigma_n$ is a variant of $\tau_1 \ldots \tau_n$. □

Definition 2.35 (proposition and predicate definition) The *definition* of a proposition or predicate p in a program P is the set of all statements in P which have p in their head. □

Definition 2.36 (goal) A *goal* is of the form

$$\leftarrow F$$

where F, called the *body*, is a polymorphic many-sorted formula. The free variables of F are assumed to be universally quantified at the front of the goal. □

Definition 2.37 (definite program and goal) A *definite clause* (resp. *definite goal*) is a statement (resp. goal) whose body is a (possibly empty) conjunction of atoms. A definite program is a program whose statements are all definite clauses. □

In the sequel, attention will be restricted to definite programs and goals. Moreover, it will often be convenient to ignore the type declarations and consider a program $P = \langle \Delta, S \rangle$ as being equivalent to the set of clauses S.

2.4 Semantics of polymorphic many-sorted programs

2.4.1 Declarative semantics

Definition 2.38 (Herbrand pre-interpretation) Let P be a definite program and let $\mathcal{L}_P = \langle \Sigma_p, \Sigma_f, V \rangle$ be the first order polymorphic many-sorted language underlying P. The *Herbrand pre-interpretation* of $\mathcal{L}_P$ is the pair $Herb = \langle D_{Herb}, \Lambda_{Herb} \rangle$ where

1. $D_{Herb} = \{D_{Herb,\delta} \mid \delta \text{ is a ground type}\}$, and for every ground type δ, $D_{Herb,\delta}$ is the least set such that if $f_{\langle \delta_1 \ldots \delta_n, \delta \rangle} \in \Sigma_f^*$ and $t_i \in D_{Herb,\delta_i}$ for all $i \in [1, n]$, then $f_{\langle \delta_1 \ldots \delta_n, \delta \rangle}(t_1, \ldots, t_n) \in D_{Herb,\delta}$.
2. Λ_{Herb} is defined such that if $f_{\langle \delta_1 \ldots \delta_n, \delta \rangle} \in \Sigma_f^*$ and $t_i \in D_{Herb,\delta_i}$ for all $i \in [1, n]$, then $\Lambda_{Herb}(f_{\langle \delta_1 \ldots \delta_n, \delta \rangle})(t_1, \ldots, t_n) = f_{\langle \delta_1 \ldots \delta_n, \delta \rangle}(t_1, \ldots, t_n)$. □

Observe that the domain equal to the union of the domains in D_{Herb} contains precisely all of the ground terms that can be constructed in the language $\mathcal{L}_P$. This domain is usually referred to as the *Herbrand universe* and is denoted by U_P. Another important domain, consisting of all the ground atoms in the language $\mathcal{L}_P$ is known as the *Herbrand base* and is denoted by B_P.

Definition 2.39 (Herbrand base) Let $\mathcal{L}_P = \langle \Sigma_p, \Sigma_f, V \rangle$ be a first order polymorphic many-sorted language defined by a program P. The *Herbrand base* of $\mathcal{L}_P$ is the least set B_P such that

1. If $p_\epsilon \in \Sigma_p^*$ then $p_\epsilon \in B_P$.
2. If $p_{\delta_1 \ldots \delta_n} \in \Sigma_p^*$ then $p_{\delta_1 \ldots \delta_n}(d_1, \ldots, d_n) \in B_P$ where $d_i \in D_{Herb,\delta_i}$ for all $i \in [1, n]$.

An *Herbrand interpretation* for a language $\mathcal{L}_P$ is any interpretation based on the Herbrand pre-interpretation for $\mathcal{L}_P$. By abuse of terminology, an Herbrand interpretation for a program P is any Herbrand interpretation for the language underlying P. Note then that any Herbrand interpretation can be defined as a subset of the Herbrand base.

Definition 2.40 (Herbrand model) Let $\mathcal{L}_P$ be a first order polymorphic many-sorted language defined by a program $P = \langle \Delta, S \rangle$. An *Herbrand model* $I \subseteq B_P$ for S is an Herbrand interpretation for $\mathcal{L}_P$ which is a model for S. An Herbrand model I for S is *minimal* iff there exists no other Herbrand model I' for S such that $I' \subset I$. □

For every definite program $P = \langle \Delta, S \rangle$, there exists a unique *minimal Herbrand model* for S. This minimal model is equivalent to the set $\{a \mid a \in B_P \wedge S \models a\}$ of ground atoms which are logical consequences of S. This set is also known as the *success set* of P and it defines the declarative semantics of P.

2.4.2 Procedural semantics

Definition 2.41 An *expression* is either a term, an atom, a conjunction of atoms, or a definite clause. A *simple expression* is either a term or an atom. □

Definition 2.42 (term substitution) A *substitution* θ is a finite set $\{v_1/t_1, \ldots v_n/t_n\}$ of *bindings* where $v_1, \ldots, v_n$ are distinct variables and $t_1, \ldots, t_n$ are terms such that $v_i \neq t_i$ for all $i \in [1, n]$. The set $\{v_1, \ldots, v_n\}$ is called the *domain* of the substitution and is denoted by $dom(\theta)$. □

Definition 2.43 (term instance) Let $\theta = \{v_1/t_1, \ldots, v_n/t_n\}$ be a term substitution and o a syntactic object such that $pars(o) \cap pars(v_1/t_1) \cap \ldots \cap pars(v_n/t_n) = \emptyset$. If there exists a most general unifier ψ of the set of type equations

$$\{type(v_i, o) = type(t_i, t_i) \mid i \in [1, n]\}$$

then the *instance* of o by θ, denoted $o\theta$, is the syntactic object obtained from o by simultaneously replacing each occurence of the variable v_i in o by the term t_i for all $i \in [1, n]$ and applying the type substitution ψ. If $o\theta$ is ground then θ is called a *grounding substitution* for o. □

An expression d is an instance of another expression e if $d = e\theta$ for some substitution θ. If e is also an instance of d then d and e are said to be *variants* of each other and θ is known as a *renaming* substitution for e. If $S = \{E_1, \ldots, E_n\}$ is a finite set of expressions and θ is a substitution, then $S\theta$ denotes the set $\{E_1\theta, \ldots, E_n\theta\}$.

Definition 2.44 (composition of term substitutions) Let $\theta = \{u_1/s_1, \ldots, u_m/s_m\}$ and $\phi = \{v_1/t_1, \ldots, v_n/t_n\}$ be substitutions. Then the *composition* of θ and ϕ, denoted $\theta\phi$, is the set $\{u_i/s_i\phi \mid u_i \neq s_i\phi \wedge i \in [1, m]\} \cup \{v_i/t_i \mid v_i \notin \{u_1, \ldots, u_m\} \wedge i \in [1, n]\}$. □

Definition 2.45 (mgu) Let S be a finite set of simple expressions. A substitution θ is a *unifier* for S iff $S\theta$ is a singleton. A unifier θ for S is a *most general unifier*, or *mgu*, for S iff for each unifier θ' of S there exists a substitution ϑ such that $\theta' = \theta\vartheta$. □

The set of most general unifiers of $\{e_1, e_2\}$ where e_1 and e_2 are arbitrary simple expressions is denoted $mgu(e_1, e_2)$.

Definition 2.46 (substitution restriction) Let θ be a substitution and V a set of variables. The *restriction* of θ to V, denoted $\theta|_V$, is the substitution obtained from θ by deleting any binding v/t for which $v \notin V$. □

Definition 2.47 (computation rule) A *computation rule* is a function from a set of goals to a set of atoms such that the value of the function for a goal G is an atom, called the *selected atom*, in G. □

Definition 2.48 (SLD-resolution) Let $G = \leftarrow A_1, \ldots, A_s, \ldots, A_m$ be a goal with $m \geq 1$ and $1 \leq s \leq m$, and let $c : H \leftarrow B_1, \ldots, B_n$ be a clause. Then G' is *derived* from G and c using θ iff the following conditions hold:

- A_s is the selected atom in G;
- $\theta \in mgu(A_s, H)$;
- G' is the goal $\leftarrow (A_1, \ldots, A_{s-1}, B_1, \ldots, B_n, A_{s+1}, \ldots, A_m)\theta$.

The goal G' is called a *resolvent* of G and c. □

Definition 2.49 (SLD-derivation) Let P be a program and G a goal. An *SLD-derivation* of $P \cup \{G\}$ consists of a (possibly infinite) sequence $G_0, G_1, G_2, \ldots$ of goals, a sequence $c_1, c_2, \ldots$ of variants of program clauses of P and a sequence $\theta_1, \theta_2, \ldots$ of substitutions such that $G_0 = G$ and each G_{i+1} is derived from G_i and c_{i+1} using θ_{i+1} for all $i \geq 0$. □

SLD-derivations may be charaterised as follows. An SLD-derivation is

- *finite* iff it consists of a finite sequence of goals; otherwise it is *infinite*.
- *successful* iff it is finite and the last goal is the empty goal.
- *failed* iff it is finite and it ends in a non-empty goal such that the selected atom in this goal does not unify with the head of any program clause.
- *complete* iff it is either successful, failed or infinite.
- *incomplete* iff it is finite and neither successful nor failed; in other words, not complete.

Definition 2.50 (SLD-refutation) An *SLD-refutation* is a successful SLD-derivation. □

Definition 2.51 (SLD-tree) Let P be a program and G a goal. An SLD-tree for $P \cup \{G\}$ is a tree satisfying the following:

1. Each node of the tree is a (possibly empty) goal.
2. The root node is G.
3. Let $G' = \leftarrow A_1, \ldots, A_s, \ldots, A_m$ be a node in the tree. Then exactly one of the following hold:

(a) no atom is selected in G', and the node has no children;

(b) A_s is the selected atom in G', and for each input clause $H \leftarrow B_1, \ldots, B_n$ such that a substitution $\theta \in mgu(H, A_m)$ exists, the goal

$$\leftarrow (A_1, \ldots, A_{s-1}, B_1, \ldots, B_n, A_{s+1}, \ldots, A_m)\theta$$

is a child of the node.

4. Nodes which are the empty goal have no children.

□

Each branch of an SLD-tree is an SLD-derivation. Hence branches may be called infinite or finite, successful or failed, complete or incomplete, according to the characterisation of the corresponding derivation. Observe that the leaf node of an incomplete branch is a non-empty goal where no atom has been selected. An SLD-tree is *complete* iff all its branches are complete, and *incomplete* otherwise. Note then, that both complete *and* incomplete trees may contain infinite branches.

An incomplete SLD-tree may be further expanded by *unfolding* the goal at a leaf node. This involves selecting an atom in the goal and adding as children to the node the goals described in 3(b) Definition 2.51. Thus a complete SLD-tree may be obtained from one that is incomplete by performing a (possibly infinite) number of unfolding steps.

An SLD-tree is said to be *trivial* if the root node is the only node of the tree, and *non-trivial* otherwise. SLD-trees can be depicted graphically. In figures, selected atoms are underlined and the empty goal is denoted by "□". Failed derivations end in "■".

Definition 2.52 (search rule) A *search rule* is a strategy for searching SLD-trees to find success branches. □

Search rules are often defined as *clause selection rules* which given a set of clauses forming a predicate definition define a fixed order in which the clauses should be used to form resolvents. For example, the clause selection rule of Prolog selects clauses in the order in which they appear in the program. An *SLD-refutation procedure* is specified by a computation rule together with a search rule.

The following notions tie together the declarative and procedural semantics.

Definition 2.53 (answer) Let P be a program and G a goal. An *answer* for $P \cup \{G\}$ is a substitution θ such that $dom(\theta) \subseteq vars(G)$. □

Definition 2.54 (correct answer) Let P be a program and $\leftarrow Q$ a goal. An answer θ for $P \cup \{\leftarrow Q\}$ is a *correct answer* for $P \cup \{\leftarrow Q\}$ iff $P \models \forall(Q\theta)$. □

Definition 2.55 (computed answer) Let P be a program, G_0 a goal, and $G_0, \ldots, G_n$ an SLD-refutation of $P \cup \{G_0\}$, where the sequence of substitutions is $\theta_1, \ldots, \theta_n$. Then the substitution $(\theta_1 \ldots \theta_n)|_{vars(G_0)}$ is a *computed answer* for $P \cup \{G_0\}$. □

Theorem 2.56 (soundness of SLD-resolution) Let P be a program and G a goal. Every computed answer for $P \cup \{G\}$ is a correct answer for $P \cup \{G\}$. □

Theorem 2.57 (completeness of SLD-resolution) Let P be a program and G a 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 $G\sigma = G\theta\gamma$. □

3 Introduction to Termination

This chapter introduces the fundamental notions of termination that will be used in the remainder of the thesis. Section 3.1 introduces the halting problem for logic programs, outlining the different areas of research which have evolved, and places the current work in context. It turns out that the logic programming paradigm admits a number of notions of termination and some of the more important ones will be reviewed here.

There are several concepts which are common to a large number of works on termination and which will play a prominent role in this thesis. These include level mappings, the notion of boundedness and interargument relationships. They are defined in Sections 3.2.1 and 3.2.4. Norms and rigidity, two other important notions are only touched on here; these are the subject of Chapter 5.

Also to be found in this chapter are the definitions of the classes of recurrent and acceptable programs. They are, arguably, the two most significant classes of program to have been defined in the literature on termination of definite logic programs. An understanding of them and the surrounding concepts is crucial to the assimilation of Chapter 6, which builds extensively on them.

Much of the material of this chapter, and a great deal more besides, can be found in the survey by De Schreye & Decorte 1994.

3.1 The Halting Problem in Logic Programming

For an arbitrary logic program P and arbitrary goal G, the halting problem is to determine whether or not G terminates wrt P. Exactly what it means for G to terminate wrt P is examined in Section 3.1.1. If G is simply regarded as a particular input to the program P, however, then the problem essentially is to determine whether or not the execution of P with this input requires a finite amount of time. The problem inherits the undecidability of the halting problem for Turing machines, meaning that no algorithm can be encoded which will determine the correct answer in a finite amount of time. This is a direct consequence of the fact that every computable function can be encoded as an appropriate logic program (see, e.g., Lloyd 1987).

The undecidable nature of termination has led to three main directions of research as identified in De Schreye & Decorte 1994. Firstly, on the subject of decidability itself a number of works have sought to establish the boundary between minimal subclasses of programs which are computationally complete, and maximal classes for which the halting problem is decidable. The current work makes no contribution in this direction. The interested reader is referred to the references contained in De Schreye & Decorte 1994.

A second line of work has been to investigate necessary and sufficient conditions for termination. Such conditions are, of course, undecidable, but, nonetheless, can be used as a theoretical basis for constructing practical termination analyses. Some works

in this area have led to a classification of programs according to their termination behaviour. For example, *acceptable* programs are precisely those which, for ground input, terminate under the left-to-right computation rule of Prolog (Apt & Pedreschi 1990). Programs which, for ground input, terminate under any selection rule are classified as *recurrent* (Bezem 1993). The original definitions of these classes, introduced in Sections 3.2.2 and 3.2.3, do not, in fact, form an ideal basis for automatic termination analyses. Some difficulties arise in formulating termination proofs in terms of the definitions. Chapter 6 examines the technicalities involved and concludes with alternative characterisations of the two classes which help to alleviate the problems.

The third category encompasses the development and use of *sufficient* conditions for proving termination. These techniques may be used, for example, to provide support for program development or program transformation tools such as partial evaluators. Chapter 7 introduces a class of programs which are terminating under a dynamic selection rule. A sufficient condition for termination, which can be checked at compile-time, is that a given program lies within the class. Chapter 8 develops sufficient conditions for ensuring unfolding during partial deduction. The work of Chapter 5, which is designed to facilitate the construction of termination proofs, may also be considered to fall in this category.

3.1.1 Some Definitions of Termination

Whether or not a goal G terminates wrt a program P is obviously dependent on the operational behaviour which in a logic program is defined by the control component. Thus termination is in fact sensitive to the following four components.

1. The program
2. The goal
3. The computation rule
4. The search rule

To complicate the issue a little further, different notions of termination have been defined in the literature. Firstly, due to the inherent non-determinism present in the logic programming paradigm the following distinction, put forward by Vasak & Potter 1986, can be made.

Definition 3.1 (existential termination Vasak & Potter 1986) Let P be a program, G a goal and s a search rule. Then G *existentially terminates* wrt P (under s) iff either all derivations for $P \cup \{G\}$ are finitely failed or the search rule s finds a successful derivation for $P \cup \{G\}$ in a finite number of steps. □

Definition 3.2 (universal termination Vasak & Potter 1986) Let P be a program and G a goal. Then G *universally terminates* wrt P iff all derivations for $P \cup \{G\}$ are finite. □

Intuitively, existential termination and universal termination correspond respectively to the notions of finding one and all solutions for a given goal and program. Observe that existential termination is sensitive to the search rule whereas universal termination is not. Thus, where the computation rule is fixed, universal termination implies existential termination.

Example 3.1 Let P be the program

```
P(A).
P(x) ← P(x).
```

and G the goal ← P(x). Then G existentially terminates wrt P under the Prolog search rule, but not under a search rule which selects clauses in the reverse order. G does not universally terminate wrt P in any case. □

Vasak and Potter also characterised goals as strongly or weakly terminating. The definitions are provided here for completeness, though, the terms are seldom encountered in the literature.

Definition 3.3 (strong termination Vasak & Potter 1986) Let P be a program and G a goal. Then G is *strongly terminating* wrt P iff G terminates wrt P for all computation rules. □

Definition 3.4 (weak termination Vasak & Potter 1986) Let P be a program and G a goal. Then G is *weakly terminating* wrt P iff G terminates wrt P for some computation rule. □

In the above definitions, "G terminates" may be taken to mean either "G existentially terminates" or "G universally terminates". In the remainder of the thesis, as in the majority of works, attention will be restricted to universal termination. Thus the expression "G terminates" will mean "G universally terminates". Moreover, with reference to Definition 3.2, three types of finite derivation will be permitted: successful, finitely failed and incomplete.

The next two definitions introduce the two most frequently used notions of termination.

Definition 3.5 (termination Bezem 1989) Let P be a program and G a goal. Then G is *terminating* wrt P iff every SLD-derivation for $P \cup \{G\}$ is finite. P is terminating iff every variable-free goal is terminating wrt P. □

Definition 3.6 (left termination Apt & Pedreschi 1990) Let P be a program and G a goal. Then G is *left terminating* wrt P iff every LD-derivation for $P \cup \{G\}$ is finite. P is left terminating iff every variable-free goal is left terminating wrt P. □

Observe that "termination" as defined by Bezem is equivalent to "strong, universal termination" as defined by Vasak and Potter. On the other hand, "left termination" is an example of "weak, universal termination". Since strong termination implies weak termination, any goal which is terminating is also left terminating.

Example 3.2 Consider the Permute program below

```
perm1  Perm([], []).
perm2  Perm([h|t], [a|p]) ←
           Delete(a, [h|t], l) ∧
           Perm(l, p).

del1   Delete(x, [x|y], y).
del2   Delete(x, [y|z], [y|w]) ←
           Delete(x, z, w).
```

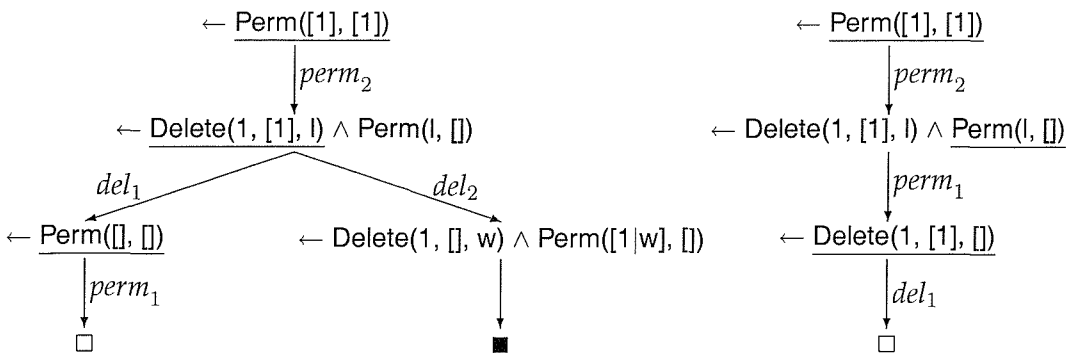


Figure 3.1: SLD-derivations for Permute ∪{← Perm([1], [1])}. Note that the left SLD-tree actually represents two derivations; the goal ← Delete(1, [], w) ∧ Perm([1|w], []) fails immediately regardless of which atom is selected.

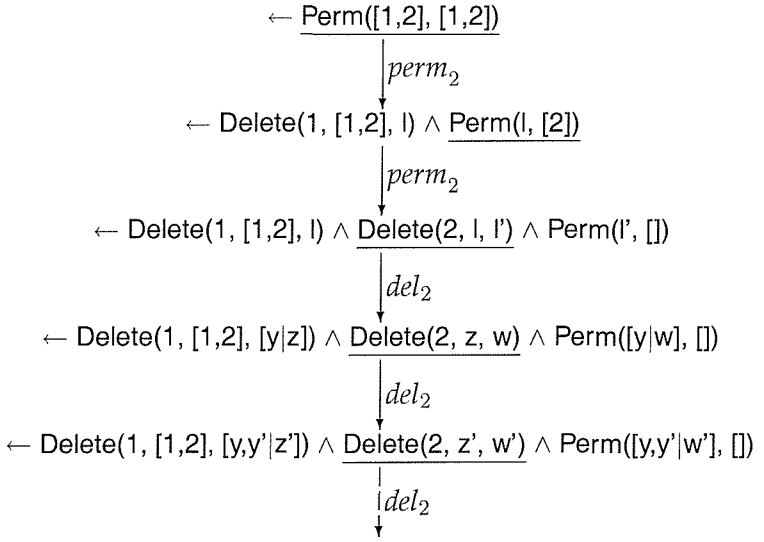


Figure 3.2: Infinite SLD-derivation for Permute ∪{← Perm([1,2], [1,2])}. Here, variants of the atom Delete(2, z, w) recur infinitely in subsequent goals if the second atom is always selected and the clause del_2 is always used.

The goal ← Perm([1], [1]) is terminating wrt Permute (see Figure 3.1) and as a consequence is left terminating also. The goal ← Perm([1, 2], [1, 2]) is left terminating but *not* terminating, since there exists a computation rule which selects non-ground Delete/3 goals resulting in an infinite derivation (see Figure 3.2). It follows that the program is not terminating though it can be proven to be left terminating (see Section 3.2.3). □

3.2 The Nuts and Bolts of Termination Proofs

The fundamental idea underlying all termination proofs is to define an order on the set of all goals that can occur in a derivation.

Definition 3.7 (order) A *partial order* on a set S is a binary relation $\sqsubseteq$ that is

1. *reflexive* $(\forall x \in S : x \sqsubseteq x)$,
2. *antisymmetric* $(\forall x, y \in S : (x \sqsubseteq y \wedge y \sqsubseteq x) \Rightarrow x = y)$ and
3. *transitive* $(\forall x, y, z \in S : (x \sqsubseteq y \wedge y \sqsubseteq z) \Rightarrow x \sqsubseteq z)$.

A *linear* or *total order* on a set S is a partial order such that any two elements of S are comparable $(\forall x, y \in S : x \sqsubseteq y \vee y \sqsubseteq x)$. □

An *ordered set* $S(\sqsubseteq)$ is a set S together with an order $\sqsubseteq$ on S. A partial order $\sqsubseteq$ on a set S induces a *strict order* $\sqsubset$ on S $(\forall x, y \in S : (x \sqsubset y \Leftrightarrow (x \sqsubseteq y \wedge x \neq y))$. A strict order $\sqsubset$ on a set S is *well founded* if there exists no infinite descending chain $e_1 \sqsupset e_2 \sqsupset \ldots$ of elements of S. The set $S(\sqsubset)$ is well-founded if $\sqsubset$ is *well founded* on S.

In this thesis, attention will be mostly restricted to the use of well-founded orders. Under this restriction, the basis of a termination proof then reduces to the following. Given a program P and goal G_0, assume that $>$ is a well-founded order on the set of goals that can occur in any derivation of $P \cup \{G_0\}$, and let $G_0, G_1, G_2, \ldots$ be such a derivation. Quite simply, if $G_i > G_{i+1}$ for all $i \geq 0$, one deduces that the sequence $G_0, G_1, G_2, \ldots$ is finite by the well-foundedness of $>$.

Two issues become apparent at this point. The most obvious is the problem of defining a suitable order on goals which can be used to prove termination. To simplify the problem, it is sometimes convenient to define the order on abstractions of goals rather than on the goals themselves. Thus the order $>$ is defined such that $G > G'$ holds iff $\mathcal{A}(G) > \mathcal{A}(G')$ holds where $\mathcal{A}$ is an abstraction function. For example, $\mathcal{A}$ might be defined to map each goal G to a multiset of natural numbers, where each atom in G maps to a single number in the multiset. This particular abstraction will be formalised in the next section. The idea of mapping atoms to natural numbers occurs frequently and forms the basis of many goal abstractions used in the literature.

The other issue which arises is how to verify for each derivation $G_0, G_1, G_2, \ldots$, that $G_i > G_{i+1}$ for all $i \geq 0$. The derivations cannot of course be explicitly constructed in a finite amount of time and so it is necessary to use a finite approximation to the set of all derivations. This can be achieved by considering individual SLD-resolution steps together with the abstraction function mentioned above. More precisely, the aim is to prove that if G is a goal, A is the selected atom in G and A unifies with the head of a clause c, then the resolvent G' of G is such that $G > G'$. Since G and A may be arbitrary, the decrease from G to G' is usually obtained by requiring the clause c to satisfy certain conditions. By verifying these conditions for all clauses in the program, it can be asserted that all derivations are finite. The abstraction function simplifies the proof by allowing irrelevant details to be ignored such as the individual syntactic structure of goals and the details of unifications. In reality, it is unlikely that the decrease from G to G' will hold for all goals, though it may hold for a specific subset of them. In this case, it is also necessary to show that any resolvent G' of G is also a member of the subset. Sections 3.2.2 and 3.2.3 present two conditions on clauses which can be used to ensure finiteness of derivations in the above manner.

Four well-founded orders will occur frequently throughout: the usual order on the natural numbers, the lexicographical ordering, the multiset ordering, and the ordering between the predicates of a program. These last three orders are defined below.

The lexicographical ordering allows tuples to be compared.

Definition 3.8 (lexicographical ordering) Let $\sqsubset_1$ and $\sqsubset_2$ be strict orders on the sets S_1 and S_2 respectively. The *lexicographical ordering* $\ll$ on $S_1(\sqsubset_1) \times S_2(\sqsubset_2)$ is defined by

$$(s_1, s_2) \ll (s'_1, s'_2)$$

iff $s_1 \sqsubset_1 s'_1$ or $s_1 = s'_1$ and $s_2 \sqsubset_2 s'_2$. □

If $S_1(\sqsubset_1)$ and $S_2(\sqsubset_2)$ are well-founded sets then the set $(S_1(\sqsubset_1) \times S_2(\sqsubset_2))(\ll)$ is well-founded also (Van Leeuwen 1990). Clearly then, well-founded lexicographical orderings can be defined over tuples of any fixed length.

Another mechanism for comparing collections of elements is the multiset ordering. A multiset is a collection of elements where the number of occurrences of each element is significant. Formally, a multiset is a function from a set S to the natural numbers which returns the multiplicity of each element in S. It will be convenient to consider multisets as being sets with duplicate elements. Thus if $s_1 = \{3\}$ and $s_2 = \{3\}$ are multisets then the multiset $s_1 \cup s_2 = \{3, 3\} \neq \{3\}$.

Informally, the multiset ordering is defined as follows. Given two multisets s_1 and s_2, s_2 is smaller than s_1 in the multiset ordering if s_2 can be obtained from s_1 by replacing an element e of s_1 with zero or more elements each of which is strictly smaller than e (wrt the ordering over the elements of the multisets).

Definition 3.9 (multiset ordering) Let $\sqsubset$ be a strict order on the set S. The *multiset ordering* $\sqsubset_{mul}$ on multisets of elements of $S(\sqsubset)$ is defined by

$$s_2 \sqsubset_{mul} s_1$$

iff there exists $e \in s_1$ and $e_1, \ldots, e_n \in s_2$ such that $s_2 = s_1/\{e\} \cup \{e_1, \ldots, e_n\}$ and $e_i \sqsubset e$ for all $i \in [1, n]$. □

This ordering is particularly useful for defining an ordering on goals in a derivation. For example, a goal G and its resolvent G' could be abstracted by multisets s_1 and s_2 of natural numbers, where each natural number represents the abstraction of one atom in G or G'. The resolvent G', by definition, is obtained from G by replacement of one atom in G with zero or more atoms from the body of a clause (combined with a substitution application). Hence the abstraction s_2 would also be obtainable from s_1 by replacement of one element e of s_1 with zero or more natural numbers (i.e. the abstractions of the body atoms of the resolving clause). In the case that each of these numbers is smaller than e then s_2 is smaller than s_1 in the multiset ordering and G' may be seen to be smaller than G in the goal ordering.

Finally, an ordering exists among the predicates of a program based on its recursive structure.

Definition 3.10 (predicate dependency) Let $\mathcal{L}_P = \langle \Sigma_p, \Sigma_f, V \rangle$ be a language defined by a program P and let $p, q \in \Sigma_p$. Then p *directly depends on* q iff

$$p(t_1, \ldots, t_{n_p}) \leftarrow B_1, \ldots, B_n \in P \text{ and } B_i = q(s_1, \ldots, s_{n_q}), \text{ for some } i \in [1, n].$$

The *depends on* relation is defined as the reflexive, transitive closure of the directly depends on relation. If p depends on q and q depends on p then p and q are *mutually dependent* and this is denoted by $p \simeq q$. □

The well-founded ordering among the predicates of a program is induced by the depends on relation: $p \sqsupset q$ whenever p depends on q but q does not depend on p, i.e. p calls q as a subprogram. By abuse of terminology, two atoms are mutually dependent (with each other) if they have mutually dependent predicate symbols. Furthermore, a body atom in a clause is said to be recursive if it is mutually dependent with the head of the clause.

3.2.1 Level Mappings, Norms and Boundedness

The idea of mapping atoms to natural numbers to construct termination proofs was originally proposed in Cavedon 1989 and Bezem 1989.

Definition 3.11 (level mapping Cavedon 1989) Let P be a program. A *level mapping* for P is a function $|.| : B_P \mapsto \mathbf{N}$ from the Herbrand base to the natural numbers. For an atom $A \in B_P$, $|A|$ denotes the *level* of A. □

Example 3.3 Let P be the program

P(A, x) ← P(B, x).
P(B, A).
P(B, B).

The function $|.| : \{\mathsf{P(A, A)}, \mathsf{P(A, B)}, \mathsf{P(B, A)}, \mathsf{P(B, B)}\} \mapsto \mathbf{N}$ defined by $|\mathsf{P(A, A)}| = 34$, $|\mathsf{P(A, B)}| = 12$, $|\mathsf{P(B, A)}| = 0$ and $|\mathsf{P(B, B)}| = 27$ is a level mapping for P. □

A level mapping is only defined for ground atoms. The lifting of the mapping to non-ground atoms was proposed in Bezem 1989.

Definition 3.12 (bounded atom Bezem 1989) An atom A is *bounded* wrt a level mapping $|.|$ if $|.|$ is bounded on the set $[A]$ of variable free instances of A. If A is bounded then $|[A]|$ denotes the maximum that $|.|$ takes on $[A]$. □

The importance of the notion of boundedness cannot be over stressed. Since goals which are ground cannot be used to compute values, they are the exception rather than the norm in logic programming. Thus practical termination proofs must be able to deal with non-ground goals and boundedness provides the basis for this. It has shaped much of the work on termination and plays a prominent role in this thesis.

Example 3.4 Let P be the program and $|.|$ the level mapping of Example 3.3. The atom P(A, x) is bounded since $|.|$ is bounded on the set $[\mathsf{P(A, x)}] = \{\mathsf{P(A, A)}, \mathsf{P(A, B)}\}$. Moreover, $|[\mathsf{P(A, x)}]| = max(\{|\mathsf{P(A, A)}|, |\mathsf{P(A, B)}|\}) = max(\{34, 12\}) = 34$. □

Level mappings are usually defined in terms of norms. Basically, a norm is a mapping from terms to natural numbers which provides some measure of the size of a term. Norms will be examined in some detail in Chapter 5. For now, it will be sufficient to consider a single norm which can be used to construct some interesting level mappings.

Example 3.5 The list-length norm $|.|_{\textit{list-length}} : U_P \mapsto \mathbf{N}$ from the Herbrand universe to the natural numbers can be defined by

$$|t|_{\textit{list-length}} = \begin{cases} 1 + |t_2|_{\textit{list-length}} & \text{whenever } t = [t_1|t_2] \\ 0 & \text{otherwise} \end{cases}$$

Then, for example, $|[\mathsf{x}, \mathsf{y}, \mathsf{z}]|_{\textit{list-length}} = 3$. □

Example 3.6 Let P be the program

```
OneList([]).
OneList([1|y]) ←
    OneList(y).
```

Let $|.|$ be the level mapping defined by $|\mathsf{OneList}(x)| = |x|_{\textit{list-length}}$. Then the atom OneList([1, 1, z]) is bounded and $|[\mathsf{OneList}([1, 1, z])]| = 3$. The atom OneList([1|x]) is unbounded and $|[\mathsf{OneList}([1|x])]|$ is not defined since $|.|$ is not bounded on the set $[\mathsf{OneList}([1|x])] = \{\mathsf{OneList}([1]), \ldots, \mathsf{OneList}([1, 1]), \ldots, \mathsf{OneList}([1, 1, 1]), \ldots\}$. □

The next two lemmas, which promote reasoning directly with bounded atoms (rather than sets of ground instances of bounded atoms), follow easily from Definition 3.12.

Lemma 3.13 Let $|.|$ be a level mapping and A a bounded atom. Then for every substitution θ, the atom $A\theta$ is also bounded and moreover $|[A]| \geq |[A\theta]|$. □

Proof 1 *Recall that* $[A] = \{A\phi \mid \phi$ *is a grounding substitution for* $A\}$. *Then* $[A] \supseteq [A\theta]$, *so* $|[A]| \geq |[A\theta]|$. □

Lemma 3.14 Let H be a bounded atom, B an atom and $|.|$ a level mapping. If for every grounding substitution θ for H and B, $|H\theta| > |B\theta|$, then B is also bounded and moreover $|[H]| > |[B]|$. □

Proof 2 *Recall that* $[B] = \{B\theta \mid \theta$ *is a grounding substitution for* $B\}$. *But* $|H\theta| > |B\theta|$ *for every grounding substitution* θ *for* H *and* B, *so* $|.|$ *is bounded on* $[B]$, *since* $|.|$ *is bounded on* $[H]$. *Let* θ *be any grounding substitution for* H *and* B *such that* $|B\theta| = |[B]|$. *Then, by Lemma 3.13,* $|[H]| \geq |[H\theta]| = |H\theta| > |B\theta| = |[B]|$. □

3.2.2 Recurrency

In Bezem 1989, level mappings were used to define a class of terminating programs.

Definition 3.15 (recurrency Bezem 1993) Let P be a definite logic program and $|.|$ a level mapping for P. A clause $c : H \leftarrow B_1, \ldots, B_n$ is recurrent (wrt $|.|$) if for every grounding substitution θ for c, $|H\theta| > |B_i\theta|$ for all $i \in [1, n]$. P is recurrent (wrt $|.|$) if every clause in P is recurrent (wrt $|.|$). □

Example 3.7 Consider the Append program below

```
app1  Append([], x, x).
app2  Append([u|x], y, [u|z]) ←
          Append(x, y, z).
```

and the level mappings $|.|_1$, $|.|_2$, $|.|_3$ and $|.|_4$ defined by

$$|\mathsf{Append}(t_1, t_2, t_3)|_1 = |t_1|_{\textit{list-length}}$$
$$|\mathsf{Append}(t_1, t_2, t_3)|_2 = 3 \times |t_1|_{\textit{list-length}} + 1$$
$$|\mathsf{Append}(t_1, t_2, t_3)|_3 = |t_3|_{\textit{list-length}}$$
$$|\mathsf{Append}(t_1, t_2, t_3)|_4 = \textit{min}(|t_1|_{\textit{list-length}}, |t_3|_{\textit{list-length}})$$

The clause app_1 is trivially recurrent wrt any level mapping. Now for every grounding substitution θ for app_2,

$$\begin{aligned} |\mathsf{Append([u|x], y, [u|z])}\theta|_1 &= |\mathsf{[u|x]}\theta|_{\textit{list-length}} \\ &= 1 + |\mathsf{x}\theta|_{\textit{list-length}} \\ &> |\mathsf{x}\theta|_{\textit{list-length}} \\ &= |\mathsf{Append(x, y, z)}\theta|_1 \end{aligned}$$

Hence the program is recurrent wrt $|.|_1$. Similarly, it can be shown that the program is recurrent wrt $|.|_i$ for all $i \in [1, 4]$. □

Bezem proved the following result.

Theorem 3.16 (recurrency Bezem 1989) Every recurrent program is terminating.

The same result was also obtained independently by Cavedon 1989 in the more general context of recurrent programs with negation (called *locally ω-hierarchical programs* in Cavedon 1989 and later renamed *acyclic programs* in Apt & Bezem 1990). The proof in Bezem 1989 relies on the following definition.

Definition 3.17 (bounded goal Bezem 1989) A goal $G = \leftarrow A_1, \ldots, A_n$ is *bounded* wrt a level mapping $|.|$ if every A_i is bounded wrt $|.|$. If G is bounded then $||[G]||$ denotes the finite multiset consisting of the natural numbers $||[A_1]||, \ldots, ||[A_n]||$. □

The proof follows the basic outline of Section 3.2. In particular the abstraction function $\mathcal{A} = ||[.]||$ and as a result a well-founded order $>$ is defined over the set of bounded goals by taking $G > G'$ iff $||[G]|| >_{mul} ||[G']||$, where $>_{mul}$ is the multiset ordering over the natural numbers. The proof is completed by showing for every SLD-resolvent G' of a bounded goal G, that G' is bounded and $G > G'$. In fact, this proof suggests a stronger corollary.

Corollary 3.18 (recurrency Bezem 1989) Let P be a program, G a goal and $|.|$ a level mapping. If P is recurrent wrt $|.|$ and G is bounded wrt $|.|$ then G is terminating wrt P.

The strength of this corollary lies in the fact that Theorem 3.16 applies only to ground goals (by virtue of Definition 3.5) whereas Corollary 3.18 applies also to non-ground goals.

Example 3.8 Reconsider the Append program and the level mappings of Example 3.7. Then

$\leftarrow$ Append([u, v, w], y, z) is bounded wrt $|.|_1$, $|.|_2$ and $|.|_4$,
$\leftarrow$ Append(x, y, [u, v, w]) is bounded wrt $|.|_3$ and $|.|_4$

Hence these goals are terminating wrt Append. Also, for a goal G observe that

$$G \text{ is bounded wrt } |.|_1 \leftrightarrow G \text{ is bounded wrt } |.|_2$$
$$(G \text{ is bounded wrt } |.|_1 \vee G \text{ is bounded wrt } |.|_3) \rightarrow G \text{ is bounded wrt } |.|_4$$

Thus by proving recurrency of Append wrt $|.|_4$ a larger class of goals can be proven terminating than by proving recurrency wrt $|.|_1$, $|.|_2$ or $|.|_3$. □

The above example demonstrates that the choice of the level mapping for proving recurrency is important with regard to the set of goals which can be proven terminating. As a final remark, Bezem also proved the converse of Theorem 3.16.

Theorem 3.19 (recurrency Bezem 1989) A program is recurrent iff it is terminating.

3.2.3 Acceptability

The notion of recurrency is a theoretical one and is not of much use in proving termination of Prolog programs. Most Prolog programs are intended to terminate under a left-to-right selection rule, and are not recurrent.

Example 3.9 Reconsider the Permute program of Example 3.2. By Theorem 3.19, the program is not recurrent since it does not terminate for all ground goals (Figure 3.2). The program is left terminating. □

The class of recurrent programs was extended in Apt & Pedreschi 1990 to the class of acceptable programs in order to provide a theoretical basis for proving termination of left terminating programs.

Definition 3.20 (acceptability Apt & Pedreschi 1990) Let $|.|$ be a level mapping and I an interpretation for a program P. A clause $c : H \leftarrow B_1, \ldots, B_n$ is *acceptable* wrt $|.|$ and I iff

1. I is a model for c and
2. for all $i \in [1, n]$ and for every grounding substitution θ for c such that $I \models \{B_1, \ldots, B_{i-1}\}\theta$, we have that $|H\theta| > |B_i\theta|$.

P is acceptable wrt $|.|$ and I if every clause in P is acceptable wrt $|.|$ and I. □

Note the role that the model I plays in this definition. In condition 2, $|H\theta|$ is only required to be greater than $|B_i\theta|$ when $I \models \{B_1, \ldots, B_{i-1}\}\theta$. This captures the fact that during a computation which uses a left-to-right computation rule instances of the body atoms to the left of B_i (where they appear in a goal) must be successfully resolved before the corresponding instance of B_i (as it appears in the derived goal) can be selected. Any ground instance of the conjunction of the successfully resolved atoms (with answer substitution applied) is modelled by I and it is only in such cases where (an instance of) B_i is selected as part of a successful derivation that the level decrease is required to ensure termination.

Analogous results to those for recurrent programs (Theorem 3.16, Corollary 3.18 and Theorem 3.19) have been proven for acceptable programs. The proofs again follow the same basic outline of Section 3.2. The abstraction function used is rather more complicated than that used in the proof of recurrency. First, observe that, if a goal $G = \leftarrow A_1, \ldots, A_n$ terminates under a left-to-right computation rule then each atom A_i is not necessarily bounded, but should be once the atoms to its left have been resolved. This idea forms the basis of the following definitions.

Definition 3.21 (maximum function Apt & Pedreschi 1994) The maximum function $max : \wp(\mathbf{N}) \mapsto \mathbf{N} \cup \{\infty\}$, where $\wp(\mathbf{N})$ denotes the powerset of $\mathbf{N}$, is defined as

$$max\ S = \begin{cases} 0 & \text{if } S = \emptyset \\ n & \text{if } S \text{ is finite and non-empty, and } n \text{ is the maximum of } S \\ \infty & \text{if } S \text{ is infinite} \end{cases}$$

Then $max\ S < \infty$ iff the set S is finite. □

Definition 3.22 (left bounded goal Apt & Pedreschi 1994)[1] Let $|.|$ be a level mapping, I an interpretation and $G = \leftarrow A_1, \ldots, A_n$ a goal. Then G is *left bounded* wrt $|.|$ and I iff for all $i \in [1, n]$, the set

$$|[G]_I^i| = \left\{ |A_i\theta| \;\middle|\; \begin{array}{l} \theta \text{ is a grounding substitution for } G \\ I \models \{A_1, \ldots, A_{i-1}\}\theta \end{array} \right\}$$

is finite. If G is left bounded wrt $|.|$ and I then $|[G]_I|$ denotes the finite multiset $\{max|[G]_I^1|, \ldots, max|[G]_I^n|\}$. □

Using the abstraction function $\mathcal{A} = |[.]_I|$ allows one to prove that for a goal G which is left bounded wrt $|.|$, any SLD-resolvent G' of G is left bounded and furthermore $|[G]_I| >_{mul} |[G']_I|$. The result is the analogue of Corollary 3.18.

Corollary 3.23 (acceptability Apt & Pedreschi 1990) Let P be a program, G a goal, $|.|$ a level mapping and I an interpretation for P. If P is acceptable wrt $|.|$ and I and G is left bounded wrt $|.|$ and I then G is left terminating wrt P. □

Sufficient and necessary conditions for left termination are characterised by the following theorem.

Theorem 3.24 (acceptability) A program is acceptable iff it is left terminating.

Example 3.10 Considering the Permute program from Example 3.2 again, let $|.|$ be the level mapping defined by

$$\begin{aligned} |\mathsf{Permute}(t_1, t_2)| &= |t_1|_{\textit{list-length}} + 1 \\ |\mathsf{Delete}(t_1, t_2, t_3)| &= |t_2|_{\textit{list-length}} \end{aligned}$$

and I be the interpretation

$$\begin{array}{c} \{\mathsf{Delete}(t_1, t_2, t_3) \mid |t_2|_{\textit{list-length}} = |t_3|_{\textit{list-length}} + 1\} \cup \\ \{\mathsf{Permute}(t_1, t_2) \mid |t_1|_{\textit{list-length}} = |t_2|_{\textit{list-length}}\} \end{array}$$

Now I is a model for the program and, in particular, for the clause $perm_2$, and for every grounding substitution θ for $perm_2$,

$$\begin{aligned} |\mathsf{Permute([h|t], [a|p])}\theta| &= |\mathsf{[h|t]}\theta|_{\textit{list-length}} + 1 \\ &> |\mathsf{[h|t]}\theta|_{\textit{list-length}} \\ &= |\mathsf{Delete(a, [h|t], l)}\theta| \end{aligned}$$

and for every grounding substitution θ for $perm_2$ such that $I \models \mathsf{Delete(a, [h|t], l)}\theta$,

$$\begin{aligned} |\mathsf{Permute([h|t], [a|p])}\theta| &= |\mathsf{[h|t]}\theta|_{\textit{list-length}} + 1 \\ &= (|\mathsf{l}\theta|_{\textit{list-length}} + 1) + 1 \\ &> |\mathsf{l}\theta|_{\textit{list-length}} + 1 \\ &= |\mathsf{Permute(l, p)}\theta| \end{aligned}$$

Hence $perm_2$ is acceptable wrt $|.|$ and I. The clauses $perm_1$ and del_1 are trivially acceptable wrt $|.|$ and I since I is a model for them, while the clause del_2 can easily be shown to be acceptable wrt $|.|$ and I in the same way as for $perm_2$. This proves the program Permute is left terminating. □

[1] The term left bounded is introduced here to avoid confusion with Definition 3.17.

3.2.4 Interargument Relationships

The intuition behind the proof of left termination in Example 3.10 is rather simple. One key step is to show that the size of the first argument in the head of the clause $perm_2$ is strictly greater than the size of the first argument in the recursive body atom, that is $|[\mathsf{h}|\mathsf{t}]|_{list\text{-}length} > |\mathsf{l}|_{list\text{-}length}$, *whenever this body atom is selected.* Since the computation proceeds under a left-to-right computation rule, this body atom will only be selected following the refutation of the Delete(a, [h|t], l) call. The model I of the program is then used to infer that the equation $|[\mathsf{h}|\mathsf{t}]\theta|_{list\text{-}length} = |\mathsf{l}\theta|_{list\text{-}length} + 1$ holds proving that the required inequality holds also. The equation $|t_2|_{list\text{-}length} = |t_3|_{list\text{-}length} + 1$ appearing in the definition of the interpretation I constitutes an *interargument relationship.* It expresses the relation between the sizes of arguments of any Delete/3 atom occurring in the success set of the program. This notion is formalised in the following definition.

Definition 3.25 (interargument relationship) Let p/n be a predicate defined in a program P whose minimal model is M, and let $|.|$ be a norm. An *interargument relationship* for p/n (wrt $|.|$) is a relation $I \subseteq \mathbf{N}^n$, such that if $M \models p(t_1, \ldots, t_n)$ then $(|t_1|, \ldots, |t_n|) \in I$. □

It will usually be convenient to write interargument relationships in the form $p(t_1, \ldots, t_n) : Expr$ where $Expr$ is an expression over the terms $t_1, \ldots, t_n$. For example, $\mathsf{P}(t_1, t_2) : |t_1| = |t_2| + 1$ defines an interargument relationship for the predicate P/2 in a program P such that if $M \models \mathsf{P}(t_1, t_2)$ then $|t_1| = |t_2| + 1$ where M is the minimal model for P.

Interargument relationships play an essential role in proving termination of a large class of programs. They were first identified by Ullman & Van Gelder 1988 who used *interargument inequalities* of the form $p_i + c \geq p_j$ where p_k denotes the list length of the kth argument of the predicate p, for $k = i, j$, and c is an integer constant. These were generalised in Plümer 1990a, Plümer 1990b, Plümer 1991 and Gröger & Plümer 1992 to *linear predicate inequalities* of the form $\sum_{i \in I} p_i + c \geq \sum_{j \in J} p_j$ where I and J are sets of input and output positions for p. Verschaetse & De Schreye 1991 considers linear equations of the form $c_0 + \sum_{i=1}^{n} c_i p_i = 0$ where c_i for all $i \in [0, n]$ are integers. In Verschaetse *et al.* 1992 this is extended to conjunctions of linear equations. Finally, Debray *et al.* 1990 describe the derivation of non-linear interargument relationships using difference equations.

4 Introduction to Partial Deduction

This chapter introduces partial evaluation in the context of logic programming. To recapitulate, partial evaluation is an example of a program specialisation technique. Classic partial evaluation techniques divide a program's input into a *static* part and a *dynamic* part, and specialise the program with respect to the static input. The effect is that a computation becomes staged; the first stage being the specialisation process, and the second, the execution of the specialised program, also called the *residual program*. Hence, instead of consuming all of the input at once, the static part is consumed in the first stage, and the dynamic part in the second (Figure 4.1).

By staging the computation in this way, those parts of it which rely exclusively on the static data, can be performed once and for all at specialisation time. The remaining parts of the computation, which depend, either in whole or in part, on the dynamic input, are performed during the second stage. Given then, that the residual program has less work to do than the original, theoretically, at least, it should be more efficient.

In the context of logic programming, the objective of program specialisation can be stated as follows. Given a program P and a goal G, specialise P wrt G to obtain the residual program P', such that for any substitution θ, the following properties hold:

- computations of $P \cup \{G\theta\}$ and $P' \cup \{G\theta\}$ give identical results;
- computations of $P' \cup \{G\theta\}$ are more efficient than $P \cup \{G\theta\}$.

Here, the static data is input to the specialisation process through the partially instantiated goal G; the dynamic data is supplied through the substitution θ. The second property above captures the real motivation for performing specialisation, while the first expresses a necessary correctness criterion.

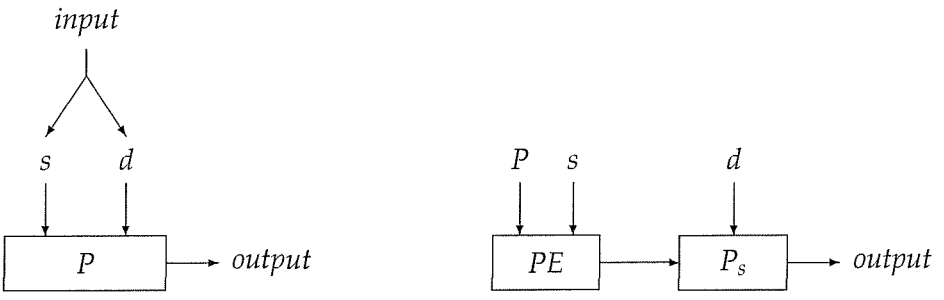


Figure 4.1: Staging a computation by partial evaluation: *Stage 1:* The partial evaluator *PE*, specialises the program *P*, wrt the static data *s*, resulting in the program P_s. *Stage 2:* P_s takes the dynamic data *d* as input, producing the same *output* obtained by supplying both *s* and *d* simultaneously as input to *P*.

4.1 Partial deduction

Partial evaluation was first introduced into logic programming by Komorowksi 1981. It is a mixture of execution and code generation and for this reason was called *mixed computation* by Ershov 1982. The term *partial deduction* was coined in Komorowski 1992, meaning partial evaluation of pure logic programs. Since only pure logic programs are considered in this thesis, this term will be used throughout.

Partial deduction was placed on a firm theoretical foundation in Lloyd & Shepherdson 1991. This section reviews the key notions and the correctness results of that paper and presents some simple examples. The results of Lloyd & Shepherdson 1991 are cast in the context of normal logic programs, i.e. programs containing negative literals in the bodies of clauses. Since only definite logic programs are considered in this thesis the following definitions and results have been simplified accordingly.

Definition 4.1 (resultant Lloyd & Shepherdson 1991) A *resultant* is a first order formula of the form $\forall(Q_1 \leftarrow Q_2)$, where for all $i \in [1,2]$, Q_i is either absent or a conjunction of atoms. □

Definition 4.2 (resultant of a derivation) Let P be a program, $\leftarrow Q_0$ a goal, and $d = \langle \leftarrow Q_0, \ldots, \leftarrow Q_n \rangle$ a finite SLD-derivation of $P \cup \{\leftarrow Q_0\}$, where the sequence of substitutions is $\theta_1, \ldots, \theta_n$. Let $\theta = \theta_1 \ldots \theta_n|_{vars(Q_0)}$. Then the derivation has length n with computed answer θ and the *resultant* of d, denoted *resultant*(d), is $Q_0\theta \leftarrow Q_n$. In the case when $n = 0$, the resultant is $Q_0 \leftarrow Q_0$. □

Definition 4.3 Let P be a program, G a goal and let τ be a finite SLD-tree for $P \cup \{G\}$. Let D be the set of non-failing SLD-derivations associated with the branches of τ. Then $resultants(\tau) = \{resultant(d) \mid d \in D\}$ is the set of resultants of τ. □

Example 4.1 Consider the Append program below.

app_1 Append([], x, x).
app_2 Append([u|x], y, [u|z]) ←
Append(x, y, z).

Let τ_1 be the finite, incomplete SLD-tree for Append $\cup\{\leftarrow$ Append([1,2|x], y, z)$\}$ depicted in Figure 4.2. Then $resultants(\tau_1)$ contains the following two resultants:

Append([1,2], y, [1,2|y]) ←
Append([1,2,u|x'], y, [1,2,u|z'"]) ← Append(x', y, z'")

Let τ_2 be the subtree of τ_1 rooted at the goal ← Append(x, y, z"). Observe that τ_2 is a finite SLD-tree for Append $\cup\{\leftarrow$ Append(x, y, z")$\}$. Then $resultants(\tau_2)$ contains the following two resultants:

Append([], y, y) ←
Append([u|x'], y, [u|z'"]) ← Append(x', y, z'") □

Definition 4.4 (partial deduction) Let P be a program and A an atom. Let τ be a finite non-trivial SLD-tree for $P \cup \{\leftarrow A\}$. Then the set of clauses $resultants(\tau)$ is called a *partial deduction of A in P*. □

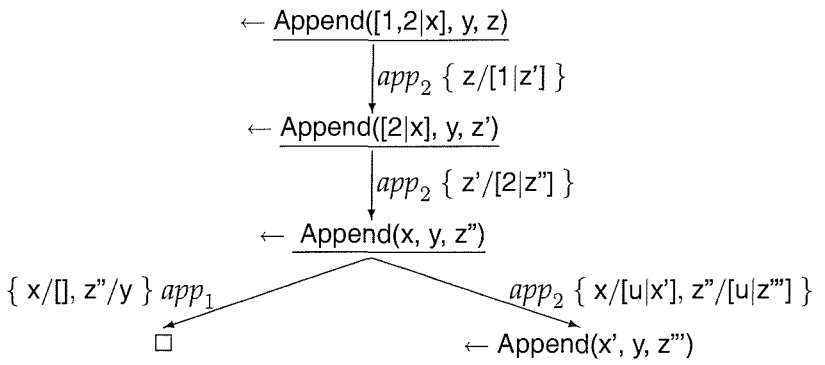


Figure 4.2: Finite, incomplete SLD-tree for Append $\cup\{\leftarrow$ Append([1,2|x], y, z)$\}$

If $\mathbf{A} = \{A_1, \ldots, A_n\}$ is a finite set of atoms, then a partial deduction of $\mathbf{A}$ in P is the union of partial deductions of $A_1, \ldots, A_n$ in P. A partial deduction of P wrt $\mathbf{A}$ is a program obtained from P by replacing the set of clauses in P, whose head contains one of the predicate symbols appearing in $\mathbf{A}$ (called the partially deduced predicates), with a partial deduction of $\mathbf{A}$ in P.

Example 4.2 Let $\mathbf{A}$ = {Append([1,2|x], y, z), Append(x, y, z)}. Then the following program Append_1 is both a partial deduction of $\mathbf{A}$ in Append and also a partial deduction of Append wrt $\mathbf{A}$:

```
Append([1,2], y, [1,2|y]).
Append([1,2,u|x], y, [1,2,u|z]) ←
    Append(x, y, z).

Append([], y, y).
Append([u|x], y, [u|z]) ←
    Append(x, y, z).
```
□

Observe that the program Append_1 above admits two (equivalent) solutions to the goal ← Append([1,2], [], z) whereas the Append program only admits one. Hence this program is not strictly a specialised version of the original. The problem arises because Append([1,2], [], z) is an instance of both atoms in the set $\mathbf{A}$. The solution is to impose a condition on the set $\mathbf{A}$.

Definition 4.5 (common instance) Let A and B be atoms. Then A and B have a *common instance* iff there exists an atom C such that C is an instance of both A and B, i.e. there exist substitutions θ and ϕ such that $A\theta = C = B\phi$. □

Observe that if two atoms A and B have a common instance, then A and B are unifiable *after renaming apart*. Hence, it is possible for two non-unifiable atoms to have a common instance.

Definition 4.6 (independence) Let $\mathbf{A}$ be a finite set of atoms. Then $\mathbf{A}$ is *independent* iff no pair of atoms in $\mathbf{A}$ have a common instance. □

The set $\mathbf{A}$ = {Append([1,2|x], y, z), Append(x, y, z)} of Example 4.2 is not independent. The two atoms in $\mathbf{A}$ are not unifiable, but can be unified after renaming apart.

Example 4.3 Let **A** = {Append([1,2|x], y, z)} be a set of atoms. Then the following program is a partial deduction of Append wrt **A**:

```
Append([1,2], y, [1,2|y]).
Append([1,2,u|x], y, [1,2,u|z]) ←
    Append(x, y, z).
```

Note that **A** is an independent set. □

The problem with the above program is that a goal such as ← Append([1,2,3], y, z) will fail whereas it succeeds in the original program. The cause here is that the goal gives rise to a call Append([], y, z') which is not an instance of any of the atoms in **A**. Hence a further condition needs to be imposed on the program to ensure equivalence with the original.

Definition 4.7 (closedness) Let S be a set of first order formulae and **A** a finite set of atoms. Then S is **A**-*closed* iff each atom in S containing a predicate symbol in an atom in **A** is an instance of an atom in **A**. □

Definition 4.8 (coveredness) Let P be a program, G a goal, **A** a finite set of atoms, P' a partial evaluation of P wrt **A**, and P^* the subprogram of P' consisting of the definitions of predicates in P' upon which G depends. Then $P' \cup \{G\}$ is **A**-*covered* if $P^* \cup \{G\}$ is **A**-closed. □

The independence and coveredness conditions are together sufficient to ensure correctness of partial deduction.

Theorem 4.9 (Lloyd & Shepherdson 1991) Let P be a program, G a goal, **A** a finite, independent set of atoms, and P' a partial deduction of P wrt **A** such that $P' \cup \{G\}$ is **A**-covered. Then the following hold:

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

Example 4.4 Let G = ← Append([1,2|x], y, z) and **A** = {Append(x, y, z")}. Then the following program Append' is a partial deduction of Append wrt **A**:

```
Append([], y, y).
Append([u|x'], y, [u|z"']) ←
    Append(x', y, z"').
```

Observe that **A** is independent and Append' ∪{G} is **A**-covered. Hence the premises of Theorem 4.9 hold and correctness is ensured. This is not surprising! □

In fact, if P' is a partial deduction of Append wrt some set of atoms **A** such that P' ∪ {← Append([1,2|x], y, z)} is **A**-covered then the atom Append(x, y, z) (modulo renaming) must be contained in **A**. It follows, that if **A** is independent, Append(x, y, z) is the only Append/3 atom in **A**, and as a consequence Append' is the only partial deduction of Append that can be used to refute the goal ← Append([1,2|x], y, z). Of course, given that Append' is equivalent to Append, this is not a very useful specialisation. To achieve, a better specialised program a little bit of cheating is required.

Example 4.5 Reconsider the program Append_1 of Example 4.2, the partial deduction of Append wrt the set $\mathbf{A} = \{\text{Append}([1,2|x], y, z), \text{Append}(x, y, z)\}$. As noted earlier, the set $\mathbf{A}$ is not independent, but observe that $\text{Append}_1 \cup \{\leftarrow \text{Append}([1,2|x], y, z)\}$ is $\mathbf{A}$-covered. Independence of $\mathbf{A}$ can be achieved by renaming the atom Append(x, y, z) as Append_1(x, y, z) and defining the two atoms to be equivalent.

Let $\mathbf{B} = \{\text{Append}([1,2|x], y, z), \text{Append\_1}(x, y, z)\}$. Then the program Append_2 below can be obtained as a partial deduction of $\text{Append} \cup \{\text{Append\_1}(x, y, z) \leftarrow \text{Append}(x, y, z)\}$ wrt $\mathbf{B}$, followed by replacing each body atom Append(x, y, z) occurring in the body of a clause by Append_1(x, y, z). This last step is known as *folding* since it is a reverse of the unfolding process (Burstall & Darlington 1977, Tamaki & Sato 1984).

```
Append([1,2], y, [1,2|y]).
Append([1,2,u|x], y, [1,2,u|z]) ←
    Append_1(x, y, z).

Append_1([], x, x).
Append_1([u|x], y, [u|z]) ←
    Append_1(x, y, z).
```

Note that the set $\mathbf{B}$ is independent and that $\text{Append}_2 \cup \{\leftarrow \text{Append}([1,2|x], y, z)\}$ is $\mathbf{B}$-covered. Although, Append_2 is not strictly a partial deduction of Append wrt $\mathbf{B}$, it follows by the correctness of the folding process (Kawamura & Kanamori 1988, Seki 1989) that the conclusions of Theorem 4.9 still hold. □

One interesting property of this last specialisation it that the original Append/3 predicate has given rise to two versions of the predicate in the residual program, i.e. Append/3 and Append_1/3. This is known as *polyvariant* specialisation.

4.2 Control of partial deduction

The above examples illustrate several key features of the partial deduction process. Firstly, a partial deduction of a program is derived from a number of SLD-trees. For example, the Append_2 of Example 4.5 is derived from two SLD-trees τ_1 and τ_2 of Example 4.1. Note that the predicate Append/3 of the specialised program corresponds to the tree τ_1 and the predicate Append_1/3 is derived from τ_2. In general, each SLD-tree generated during partial deduction gives rise to a specialised predicate in the residual program. The definition of a predicate is determined precisely by its corresponding SLD-tree.

Two levels of control in the partial deduction process can now be distinguished. The *global* control decides which trees should be generated. More precisely, since each tree is rooted at a single atom, the global control determines the set of atoms which should be used to construct SLD-trees from. This set is, in fact, the independent set $\mathbf{A}$ of Theorem 4.9 (or, rather, a set such as $\mathbf{B}$ in Example 4.5). Hence, the global control also determines the amount of polyvariance, i.e. the number of specialised versions generated for each predicate. The *local* control, on the other hand, determines the structure of each individual SLD-tree. Since construction of a tree proceeds by unfolding, the local control is often described by an *unfolding rule*.

Each level of control has associated with it, its own termination problem. For the global control, the problem is to ensure finiteness of the set $\mathbf{A}$, whereas the local control

must ensure that every SLD-tree generated is finite. This thesis focuses exclusively on local termination.

Of course, finiteness, in itself, is not hard to achieve. A simple depth bound approach, for example, will suffice. But the quality of the residual code depends very much on the construction of the SLD-trees and it is often the case, though not always, that more unfolding leads to better specialisation. Thus the problem restated, is to "*unfold finitely as much as possible*".

4.3 Online and offline control

There are two basic approaches to the control of partial evaluation. In the *online* approach, all control decisions, including both local and global, are made at specialisation time. In the opposing *offline* approach, control decisions are taken prior to the actual specialisation itself. This can be achieved by (statically) analysing the program to be specialised and producing an annotated version containing control information. This annotated program is submitted to the partial evaluator which picks up the control information and uses it to guide the subsequent specialisation process.

An offline analysis usually works with descriptions of values and can thus sometimes be too conservative in its control decisions. At specialisation time when the concrete data is available more refined decisions can be made. It is for this reason, that online methods usually offer better specialisation potential than offline ones.

The offline approach offers its own advantages, however. The separation of the specialisation process into components is good software engineering practice and permits the development of these components to occur independently. Of more use to the user, is that offline partial evaluation can be significantly faster than online partial evaluation, since, at specialisation time, no control decisions need to be made. The time taken to perform the offline analysis, does not necessarily need to be taken into account. The reason for this, is that when several different specialisations of a single program are required, only one analysis is ever necessary. Hence, the analysis time can often become insignificant overall.

The offline approach also has an advantage when it comes to *self-application*. A self-applicable partial evaluator, is one which is able to specialise itself. The experience of many researchers has been that it is much easier to build an offline partial evaluator, that is amenable to self-application, than it is to build an online one. The interest in self-application stems from the Futamura projections described in the next section.

4.3.1 The Futamura projections

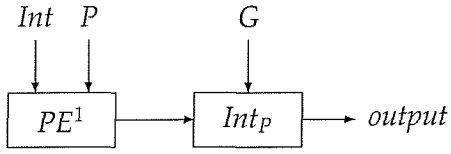
One application of partial evaluation which has attracted considerable attention is its use in the automatic derivation of compilers from interpreters. The main ideas in this area were formally captured in the Futamura projections (Futamura 1971).

Let *Int* be a meta-interpreter which takes two inputs; a program P (known as the object program) and an input G for P. The interpreter effectively executes P with input G and produces the same output that would be obtained from a "genuine" execution of the program with the same input:

It is often the case that one would like to run the program P on a variety of different inputs. The situation then is exactly that depicted in Figure 4.1 where the program P corresponds to the static input data s and G represents the dynamic part of the input. Hence, program specialisation techniques can be applied to good effect.

4.3.1.1 The first Futamura projection

The interpreter *Int* can be partially evaluated with respect to the object program P to produce a specialised version Int_P of *Int* dedicated to the "interpretation" of P. This new "interpreter" is dedicated in the sense that P is the only program that it can "interpret". As such Int_P, with input G, should execute more quickly than *Int*, with inputs P and G, and produce the same output. This, after all, is the whole point of the partial evaluation process. The quotes here are used with reference to interpretation, since very often the interpretation layer can be entirely removed through specialisation.



Observe the similarity between the program Int_P and the program P. Both take an input G and produce the same output. For this reason, Int_P is often referred to as a "compiled" version of the object program P. A consideration of the underlying languages involved adds weight to this idea of partial evaluation as a compilation process.

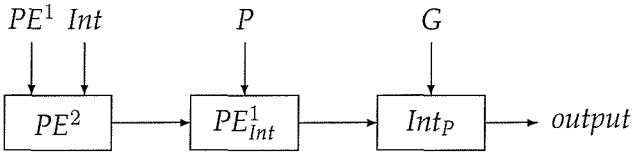
Suppose that the interpreter is written in a language $\mathcal{L}_{Int}$ and the language of the object program is $\mathcal{L}_P$. The program Int_P is just a specialised version of *Int* and hence is also in the language $\mathcal{L}_{Int}$. Thus the partial evaluation process is effectively a form of compilation from the language $\mathcal{L}_P$ to $\mathcal{L}_{Int}$. An interesting case arises when $\mathcal{L}_P$ and $\mathcal{L}_{Int}$ are equivalent. Then, ideally, the program Int_P should be the same as the original program P (or possibly an optimised version of it). This case clearly demonstrates the potential for completely removing the interpretation overhead through partial evaluation.

4.3.1.2 The second Futamura projection

Note that the partial evaluator PE^1 used in the first Futamura projection is not in itself a compiler as such, since it must take both an object program P and an interpreter *Int* as input, whereas a compiler would only require the object program P. The second Futamura projection details how such a compiler can be obtained from a partial evaluator and an interpreter.

It usual for an interpreter such as *Int* to be used to interpret a number of different programs. This gives rise to another static/dynamic classification and an opportunity

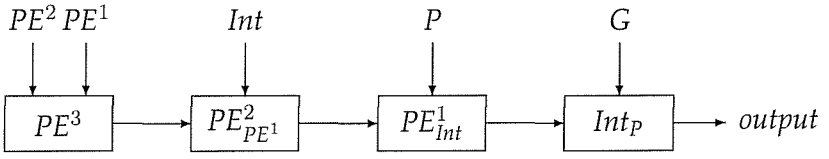
for specialisation. Specifically, the program PE^1 can be partially evaluated regarding the input *Int* as static and the input P as dynamic. This, of course, requires a partial evaluator PE^2 which is capable of specialising the partial evaluator PE^1.



The result of specialising PE^1 with respect to *Int* is the partial evaluator PE^1_{Int} which is dedicated to the partial evaluation of the interpreter *Int*, in the same sense as before. This program can now take an object program P and produce a compiled version Int_P of P. Consequently, a program such as PE^1_{Int} is called a compiler.

4.3.1.3 The third Futamura projection

It may well be desirable to generate compilers for a range of different interpreters. Specialisation of the compiler generation process of the second Futamura projection, permits more rapid generation of a compiler from an interpreter. This time, the partial evaluator PE^2 is the program to be specialised, the static input is the partial evaluator PE^1 and the unknown input is the interpreter *Int*. Again a partial evaluator PE^3 is required which is capable of specialising the partial evaluator PE^2.



The result of specialising PE^2 with respect to PE^1 is the partial evaluator $PE^2_{PE^1}$ which is dedicated to the partial evaluation of the partial evaluator PE^1. This program is referred to as a compiler generator since it takes an interpreter *Int* as its only input and produces a compiler PE^1_{Int} as output.

4.3.1.4 Self application

The interrelationship of the three Futamura projections is shown in Figure 4.3. Consider the case when the partial evaluator PE^1 above is self-applicable. Then, since PE^1 is capable of specialising itself, the partial evaluator PE^2 may be replaced by PE^1. Similarly, PE^3 may also be replaced by PE^1 and a compiler generator may be obtained through self-application of a single partial evaluator.

4.3.2 Perspective

This thesis focuses on developing offline, local control for partial deduction. In particular, it addresses the termination issue: how to ensure the construction of finite SLD-trees during partial deduction. Instead of developing an offline termination analysis for partial deduction from scratch, it explores the work which has been done on static termination analysis of logic programs (Chapters 5, 6 and 7), and examines how the existing analysis techniques can be adapted for partial deduction (Chapter 8).

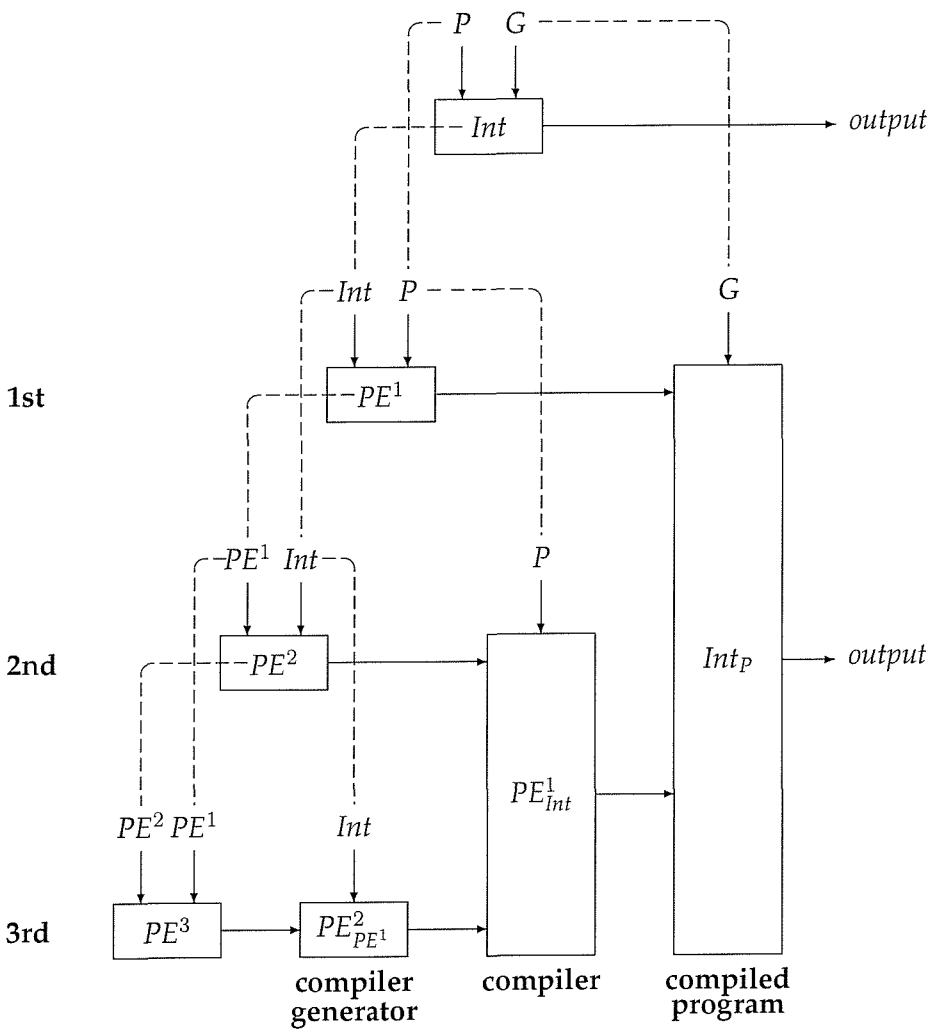


Figure 4.3: Interrelationship of the three Futamura projections.

Part II

Terminating Logic Programs

5 Typed Norms for Typed Logic Programs

5.1 Introduction

The usefulness of considering the sizes of arguments as a suitable abstraction for proving termination of logic programs was first illustrated by Ullman & Van Gelder 1988. This work was the starting point for the development of functions called *norms* which map terms to natural numbers.

Choosing the right set of norms is crucial for deducing termination and also for deriving useful interargument relationships. Early work on termination relied on the user to provide the necessary norms. As this had limited usefulness a method to automatically generate norms from a program was proposed in Decorte *et al.* 1993. The approach focuses on deriving norms from type graphs that have previously been inferred by an analysis of the program. The technique is effective in generating norms for proving termination of many of the programs found in the termination literature. However, a more direct approach can be adopted in the context of a typed language such as Gödel (Hill & Lloyd 1994), when the types are already known.

As typed logic programming becomes more mainstream, development tools like partial deduction systems will need to be mapped from untyped languages to typed ones. SAGE (Gurr 1994) is one example of a partial deduction system developed for the typed language Gödel. Although SAGE does well to demonstrate the effectiveness of self-application and how the overheads of the ground representation in meta-programs can be removed, there is much potential for improvement (Gurr 1995). One of its weaknesses is that it relies on a rather rudimentary termination analysis which could benefit considerably from the well developed techniques found in the termination literature. Revamping the analysis would require incorporating a number of techniques, including norm derivation, developed for untyped logic programs. It is important, however, when mapping techniques across from the untyped setting that the new techniques should exploit the underlying type system as much as possible. In the case of automatic norm derivation the approach in Decorte *et al.* 1993 clearly would not take advantage of the prescribed types. As a result of this and since "any state-of-the-art approach to termination analysis needs to take type information into account" (Decorte *et al.* 1994), new techniques are needed to derive norms directly from these types and avoid the overhead of type graph generation. This chapter lays a foundation for such techniques.

This chapter shows how norms can be generated from the prescribed types of a program written in a language, such as Gödel, which supports parametric polymorphism. Interestingly, the types highlight restrictions of earlier norms and suggest how these norms can be extended to obtain some very general and powerful notions of norm which can be used to measure any term in an almost arbitrary way.

The next section introduces *typed norms* and defines the classes of linear, semi-

linear and type-linear typed norms[1]. Some technical issues in the definition of typed norms are also addressed and the important notion of rigidity is defined. Section 5.3 describes how to infer the norms of Section 5.2 from the prescribed types of a program and relates the approach to that of Decorte *et al.* 1993. Related work is addressed in the penultimate section and the conclusion outlines some directions for future work.

5.2 Typed norms

Before proceeding with the main development, a short remark is in order regarding the range of norms. Originally, norms were defined as mappings from terms to natural numbers. Thus both ground *and* non-ground terms were mapped to natural numbers which was achieved by mapping variables to zero. The norm $|.|_{list\text{-}length}$ defined in Example 3.5 is an example of such a norm. It is often more useful, however, particularly when it comes to deriving norms, to map a non-ground term t to an arithmetic expression over the variables in t. This approach will be followed here. It has also been adopted by others, e.g. Benoy & King 1996, Lindenstrauss & Sagiv 1997, Codish & Talboch 1997, and is fast becoming the norm for practical analyses.

Let ED_τ denote the set of all (ground and non-ground) terms of type τ.

Definition 5.1 Let $\Sigma_\tau = \{Lin\}$ and $\Sigma_f = \{+_{\langle Lin.Lin,Lin\rangle}, 0_{\langle \epsilon,Lin\rangle}, 1_{\langle \epsilon,Lin\rangle}\}$ be alphabets of type and function symbols respectively and let V_{Lin} be a countably infinite set of variables. Then ED_{Lin} represents the class of linear expressions on V_{Lin} where a term such as $x_{Lin} + y_{Lin} + y_{Lin} + 1_{\langle \epsilon,Lin\rangle} + 1_{\langle \epsilon,Lin\rangle} + 1_{\langle \epsilon,Lin\rangle}$ is abbreviated by $x + 2y + 3$ (Note that associativity and commutativity are assumed because of the intended interpretation). □

Having established the range of a norm, the next step is to define the domain. The domain of untyped norms is simply the Herbrand universe. In a typed language, there is a natural division of this universe determined by the types in the language. This motivates the introduction, for each type τ in the language, of a typed norm $|.|_\tau$ which only measures terms of type τ.

Definition 5.2 (typed norm I) A *typed norm* for a polymorphic type τ is a mapping $|.|_\tau : \mathsf{ED}_\tau \rightarrow \mathsf{ED}_{Lin}$. □

Example 5.1 The typed norm $|.|_{\mathsf{List(Int)}} : \mathsf{ED}_{\mathsf{List(Int)}} \rightarrow \mathsf{ED}_{Lin}$ defined below measures the length of both open and closed lists of integers.

$$|v|_{\mathsf{List(Int)}} = v$$
$$|\mathsf{Nil}|_{\mathsf{List(Int)}} = 0$$
$$|\mathsf{Cons}(t_1, t_2)|_{\mathsf{List(Int)}} = 1 + |t_2|_{\mathsf{List(Int)}}$$

Then $|\mathsf{Cons(1, Cons(2, Nil))}| = 2$ and $|\mathsf{Cons(1, Cons(x, Cons(y, z)))}| = 3 + z$. □

It is appropriate at this point to review the important concept of rigidity which was originally introduced by Bossi *et al.* 1994 in order to prove termination for a class of goals with possibly non-ground terms. A rigid term is one whose size, as determined by a norm, is not affected by substitutions applied to the term. In the following, ϕ denotes the variable assignment which binds all variables in a term to the term 0_{Lin}.

[1] Originally called hierarchical typed norms in Martin *et al.* 1996.

Definition 5.3 (rigid term) Let $|.|_\tau$ be a typed norm for τ and t be a term of type τ. Then t is *rigid* with respect to $|.|_\tau$ iff for every substitution θ, $|t|_\tau\phi = |t\theta|_\tau\phi$. □

Example 5.2 The term Cons(x, Cons(y, Nil)) is rigid wrt the norm $|.|_{\mathsf{List(Int)}}$ of Example 5.1 since for every substitution $\{\mathsf{x} \mapsto t_1, \mathsf{y} \mapsto t_2\}$ where t_1 and t_2 are terms $|\mathsf{Cons}(t_1, \mathsf{Cons}(t_2, \mathsf{Nil}))| = 2$. □

By defining level mappings in terms of norms, it is possible to define a class of bounded goals in terms of rigidity. More precisely, an atom is bounded with respect to a level mapping if each argument of the atom whose size is measured in the level mapping is rigid. A problem arises, however, with the typed norms used in level mappings. In measuring the level of an atom, a norm $|.|_\tau$, which can only measure terms of type τ may be applied to a term of type σ, where $\sigma = \psi(\tau)$ for some type substitution ψ.

Example 5.3 Let P define the language $\langle \Sigma_p, \Sigma_f, V \rangle$, where

$$\begin{aligned} \Sigma_\tau &= \{\mathsf{Int}, \mathsf{List}\} \\ \Sigma_f &= \{\mathsf{Nil}_{\langle \epsilon, \mathsf{List(u)} \rangle}, \mathsf{Cons}_{\langle \mathsf{u.List(u)}, \mathsf{List(u)} \rangle}\}, \\ \Sigma_p &= \{\mathsf{Traverse}_{\mathsf{List(u)}}\} \end{aligned}$$

and $S = \{\mathsf{Traverse(Nil).}, \mathsf{Traverse(Cons(x, y))} \leftarrow \mathsf{Traverse(y).}\}$ then the norm $|.|_{\mathsf{List(u)}}$ defined by

$$\begin{aligned} |v|_{\mathsf{List(u)}} &= v \\ |\mathsf{Nil}|_{\mathsf{List(u)}} &= 0 \\ |\mathsf{Cons}(t_1, t_2)|_{\mathsf{List(u)}} &= 1 + |t_2|_{\mathsf{List(u)}} \end{aligned}$$

can be used to define a level mapping $|.|$ for the Traverse/1 predicate as follows

$$|\mathsf{Traverse}(t)| = |t|_{\mathsf{List(u)}}$$

The problem is that in trying to prove recurrency with respect to the level mapping $|.|$ for Traverse/1, the level mapping can be applied to atoms such as Traverse(Cons(1, Nil)), yet the type of the argument of Traverse/1 in this instance, List(Int), is not the type List(u) for which the mapping is defined. □

This problem arises due to the polymorphism in the typed language and is not difficult to remedy. The domain of the norm must be changed and a constraint imposed to ensure that the rigidity property still holds. To see why the constraint is required, suppose that the term t is rigid wrt the typed norm $|.|_\tau$. Then, by the definition of rigidity, for every substitution θ,

$$|t|_\tau\phi = |t\theta|_\tau\phi \tag{5.1}$$

Now applying a variable substitution to a term often has the effect of further instantiating the type of the term. For example the type of the term Cons(x, Nil) is List(u), but the type of Cons(x, Nil)$\{ \mathsf{x} \mapsto 1 \}$ = Cons(1, Nil) is List(Int). Hence the definition of $|.|_\tau$ needs to be constrained so that equation (5.1) holds. This leads to the following definition.

Definition 5.4 (typed norm II) A *typed norm* for a polymorphic type τ is a mapping $|.|_\tau : \cup_{\psi\in\Phi}\mathsf{ED}_{\psi(\tau)} \rightarrow \mathsf{ED}_{Lin}$ where Φ denotes the set of all type substitutions, and for every term t of type τ and for every type substitution ψ, $|t|_\tau = |\psi(t)|_\tau$. □

The definition of a type I norm naturally induces a norm of type II. Thus any type II norm may be unambiguously defined by means of a type I norm. This approach, which avoids unnecessary notation and does not cloud the intuitions involved, will be adopted in the sequel.

To prove rigidity of a term with respect to a norm it is infeasible to apply all possible substitutions to it to verify that its size with respect to the norm is invariant. Instead, a syntactic characterisation of rigid terms is needed. By imposing the following condition on the way norms are defined, a simple, syntactic check can be obtained to determine the rigidity of terms with respect to norms defined under these conditions.

Definition 5.5 (linearity property) A typed norm $|.|_\tau$ satisfies the linearity property iff for every variable v, $|v|_\tau \in V_{Lin}$ and for all $t \in \mathsf{ED}_\tau$, the following properties hold

1. $|t|_\tau$ is of the form $c_0 + c_1|v_1|_{\tau_1} + \ldots + c_n|v_n|_{\tau_n}$, where $c_0, n \geq 0$, $c_1, \ldots, c_n > 0$ and for all $i \in [1, n]$, $v_i \in vars(t)$;
2. if $|t|_\tau = c_0 + c_1|v_1|_{\tau_1} + \ldots + c_n|v_n|_{\tau_n}$ then $|t\theta|_\tau = c_0 + c_1|v_1\theta|_{\tau_1} + \ldots + c_n|v_n\theta|_{\tau_n}$, for every substitution θ.

□

Proposition 5.6 (rigid term) Let $|.|_\tau$ be a typed norm satisfying the linearity property and t be a term of type τ. Then t is *rigid* with respect to $|.|_\tau$ if $vars(|t|_\tau) = \emptyset$. □

Proof 3 *If $vars(|t|_\tau) = \emptyset$ then it follows by the linearity property that $|t|_\tau = |t\theta|_\tau$ for every substitution θ.* □

Although each norm is annotated with its type, the following example illustrates that several norms may exist for the same type.

Example 5.4 The typed norm $|.|^{len}_{\mathsf{List(List(Int))}}$ measures the length of a list whose elements are lists of integers. The typed norm $|.|^{sum}_{\mathsf{List(List(Int))}}$ sums the lengths of the elements of such a list.

$$
\begin{aligned}
|v|^{len}_{\mathsf{List(List(Int))}} &= v \\
|\mathsf{Nil}|^{len}_{\mathsf{List(List(Int))}} &= 0 \\
|\mathsf{Cons}(t_1, t_2)|^{len}_{\mathsf{List(List(Int))}} &= 1 + |t_2|^{len}_{\mathsf{List(List(Int))}}
\end{aligned}
$$

$$
\begin{aligned}
|v|^{sum}_{\mathsf{List(List(Int))}} &= v \\
|\mathsf{Nil}|^{sum}_{\mathsf{List(List(Int))}} &= 0 \\
|\mathsf{Cons}(t_1, t_2)|^{sum}_{\mathsf{List(List(Int))}} &= |t_1|^{sum}_{\mathsf{List(Int)}} + |t_2|^{sum}_{\mathsf{List(List(Int))}}
\end{aligned}
$$

where $|.|^{sum}_{\mathsf{List(Int)}}$ is equal to the norm $|.|_{\mathsf{List(Int)}}$ of Example 5.1. Note that the norm $|.|^{len}_{\mathsf{List(List(Int))}}$ is characterised by a weight of 1 in its recursive equation and the selection of the second argument position only, whereas the norm $|.|^{sum}_{\mathsf{List(List(Int))}}$ is characterised by a weight of 0 in its recursive equation and the selection of both argument positions. □

Let $\Sigma_f^\dagger = \{f_{\psi(\sigma)} \mid f_\sigma \in \Sigma_f \wedge \psi \text{ is a type substitution}\}$ denote the set of all instances of Σ_f. Then a norm can be uniquely characterised by a partial mapping $w : \Sigma_f^\dagger \times \mathbf{N} \mapsto \mathbf{N}$ which assigns weights to typed function symbols and argument positions. More specifically, given a function symbol $f_{\langle \tau_1 \ldots \tau_n, \tau \rangle} \in \Sigma_f^\dagger$, let $w(f_{\langle \tau_1 \ldots \tau_n, \tau \rangle}, 0)$ denote the weight assigned to $f_{\langle \tau_1 \ldots \tau_n, \tau \rangle}$ and for all $i \in [1, n]$, let $w(f_{\langle \tau_1 \ldots \tau_n, \tau \rangle}, i)$ denote the weight assigned to the ith argument position of $f_{\langle \tau_1 \ldots \tau_n, \tau \rangle}$. The definition of a norm for a type τ depends on w and therefore the norm is denoted by $|.|_\tau^w$.

Example 5.5 Let *len* and *sum* be partial mappings defined by

$$len(f_1, 0) = 0 \quad len(f_2, 0) = 1 \quad len(f_2, 1) = 0 \quad len(f_2, 2) = 1$$

$$\begin{array}{llll} sum(f_1, 0) = 0 & sum(f_2, 0) = 0 & sum(f_2, 1) = 1 & sum(f_2, 2) = 1 \\ sum(f_3, 0) = 0 & sum(f_4, 0) = 1 & sum(f_4, 1) = 0 & sum(f_4, 2) = 1 \end{array}$$

where $f_1 = \mathsf{Nil}_{\mathsf{List(List(Int))}}$, $f_2 = \mathsf{Cons}_{\langle \mathsf{List(Int).List(List(Int)),List(List(Int))} \rangle}$, $f_3 = \mathsf{Nil}_{\mathsf{List(Int)}}$ and $f_4 = \mathsf{Cons}_{\langle \mathsf{Int.List(int),List(Int)} \rangle}$. Then the norms $|.|^{len}_{\mathsf{List(List(Int))}}$ and $|.|^{sum}_{\mathsf{List(List(Int))}}$ of Example 5.4 may be defined as

$$\begin{aligned} |v|^{len}_{\mathsf{List(List(Int))}} &= v \\ |f_1|^{len}_{\mathsf{List(List(Int))}} &= len(f_1, 0) \\ |f_2(t_1, t_2)|^{len}_{\mathsf{List(List(Int))}} &= len(f_2, 0) + len(f_2, 1)|t_1|^{len}_{\mathsf{List(List(Int))}} + len(f_2, 2)|t_2|^{len}_{\mathsf{List(List(Int))}} \end{aligned}$$

$$\begin{aligned} |v|^{sum}_{\mathsf{List(List(Int))}} &= v \\ |f_1|^{sum}_{\mathsf{List(List(Int))}} &= sum(f_1, 0) \\ |f_2(t_1, t_2)|^{sum}_{\mathsf{List(List(Int))}} &= sum(f_2, 0) + sum(f_2, 1)|t_1|^{sum}_{\mathsf{List(Int)}} + sum(f_2, 2)|t_2|^{sum}_{\mathsf{List(List(Int))}} \end{aligned}$$

$$\begin{aligned} |v|^{sum}_{\mathsf{List(Int)}} &= v \\ |f_3|^{sum}_{\mathsf{List(Int)}} &= sum(f_3, 0) \\ |f_4(t_1, t_2)|^{sum}_{\mathsf{List(Int)}} &= sum(f_4, 0) + sum(f_4, 1)|t_1|^{sum}_{\mathsf{Int}} + sum(f_4, 2)|t_2|^{sum}_{\mathsf{List(Int)}} \end{aligned}$$

□

A notion of linear and semi-linear norms can now be defined for typed programs. These two classes of norms were originally introduced in the context of non-typed programs by Plümer 1990a and Bossi *et al.* 1992 respectively.

Definition 5.7 (linear typed norm) A typed norm $|.|_\tau^w$ is *linear* iff for all $v \in V$ and for all $f_{\langle \tau_1 \ldots \tau_n, \tau \rangle} \in \Sigma_f^\dagger$

$$\begin{aligned} |v|_\tau^w &= v \\ |f_{\langle \tau_1 \ldots \tau_n, \tau \rangle}(t_1, \ldots, t_n)|_\tau^w &= w(f_{\langle \tau_1 \ldots \tau_n, \tau \rangle}, 0) + \textstyle\sum_{i=1}^n w(f_{\langle \tau_1 \ldots \tau_n, \tau \rangle}, i)|t_i|_\tau^w \end{aligned}$$

where $w(f_{\langle \tau_1 \ldots \tau_n, \tau \rangle}, i) = 1$ for all $i \in [1, n]$. □

Note that the types highlight an inherent restriction of linear norms, that is, these norms are only defined when $\tau_i = \tau$ for all $i \in [1, n]$. Such norms have limited applicability.

Example 5.6 Given $\Sigma_\tau = \{\mathsf{Tree}\}$ and $\Sigma_f = \{\mathsf{Leaf}_{\langle\epsilon,\mathsf{Tree}\rangle}, \mathsf{Node}_{\langle\mathsf{Tree.Tree,Tree}\rangle}\}$, the linear typed norm for Tree that counts the number of function symbols in a term is defined by

$$|v|^{size}_{\mathsf{Tree}} = v$$
$$|\mathsf{Leaf}|^{size}_{\mathsf{Tree}} = 1$$
$$|\mathsf{Node}(t_1, t_2)|^{size}_{\mathsf{Tree}} = 1 + |t_1|^{size}_{\mathsf{Tree}} + |t_2|^{size}_{\mathsf{Tree}}$$

□

Semi-linear norms are a generalisation of linear norms where for all $i \in [1, n]$, $w(f_{\langle\tau_1 \ldots \tau_n,\tau\rangle}, i) \in \{0, 1\}$.

Definition 5.8 (semi-linear typed norm) A typed norm $|.|^w_\tau$ is *semi-linear* iff for all $v \in V$ and for all $f_{\langle\tau_1 \ldots \tau_n,\tau\rangle} \in \Sigma^\dagger_f$

$$|v|^w_\tau = v$$
$$|f_{\langle\tau_1 \ldots \tau_n,\tau\rangle}(t_1, \ldots, t_n)|^w_\tau = w(f_{\langle\tau_1 \ldots \tau_n,\tau\rangle}, 0) + \textstyle\sum_{i=1}^n w(f_{\langle\tau_1 \ldots \tau_n,\tau\rangle}, i)|t_i|^w_\tau$$

where $w(f_{\langle\tau_1 \ldots \tau_n,\tau\rangle}, i) \in \{0, 1\}$ for all $i \in [1, n]$. □

Example 5.7 If $\Sigma_\tau = \{\mathsf{Int}, \mathsf{List}\}$ and $\Sigma_f = \{\mathsf{Nil}_{\langle\epsilon,\mathsf{List(u)}\rangle}, \mathsf{Cons}_{\langle\mathsf{u.List(u),List(u)}\rangle}\}$, then the norm $|.|^{len}_{\mathsf{List(List(Int))}}$ defined in Example 5.4 is semi-linear. □

Semi-linear norms are not expressive enough to measure the sizes of terms that can be defined in a typed language such as Gödel. To quote Bossi *et al.* 1992, p. 72, paragraph 2 "The recursive structure of a semi-linear norm gets into the term structure by only one level. Moreover so far it is not defined how different semi-linear norms can be linked to work together. The definition of a semi-linear norm is recursively based only onto itself and it is easy to understand that this is a severe restriction." Again the types highlight where the essential problem lies: the norm applied to t_i is $|.|_\tau$ whereas the type of t_i is τ_i. The following definition overcomes this limitation of semi-linear norms. It also lifts the restriction of the definition of the weight function.

Definition 5.9 (type-linear typed norm) A typed norm $|.|^w_\tau$ is *type-linear* iff for all $v \in V$ and for all $f_{\langle\tau_1 \ldots \tau_n,\tau\rangle} \in \Sigma^\dagger_f$

$$|v|^w_\tau = v$$
$$|f_{\langle\tau_1 \ldots \tau_n,\tau\rangle}(t_1, \ldots, t_n)|^w_\tau = w(f_{\langle\tau_1 \ldots \tau_n,\tau\rangle}, 0) + \textstyle\sum_{i=1}^n w(f_{\langle\tau_1 \ldots \tau_n,\tau\rangle}, i)|t_i|^w_{\tau_i}$$

where $|t_i|^w_{\tau_i}$ are type-linear typed norms. □

Example 5.8 With Σ_τ and Σ_f as defined in Example 5.7, the norm $|.|^{sum}_{\mathsf{List(List(int))}}$ defined in Example 5.4 is type-linear and, in fact, cannot be expressed as a semi-linear norm. □

Note that Definition 5.9 is closely related to definition 4.5 of Decorte *et al.* 1993. Both generalise the definition of a type norm proposed in Plümer 1990a. In Decorte *et al.* 1993 the relationship between typed norms and semi-linear norms is not made explicit, but the presentation here makes the relationships between the various norms clear. In particular, it can be seen that every linear typed norm is semi-linear and every semi-linear typed norm is type-linear. The following proposition is needed to establish a syntactic characterisation of rigidity with respect to type-linear typed norms.

Proposition 5.10 Let $|.|_\tau$ be a type-linear typed norm. Then $|.|_\tau$ satisfies the linearity property. □

It follows from Proposition 5.10 that linear typed norms and semi-linear typed norms also satisfy the linearity property. The power of typed norms is illustrated in the following example.

Example 5.9 Consider the predicate Flatten/2 defined below which flattens a list of lists.

```
Flatten(Nil, Nil).
Flatten(Cons(e, x), r) ←
    Append(e, y, r) ∧
    Flatten(x, y).
```

Observe that for any atom Flatten(t_1, t_2) in the minimal model for this program, where t_1 and t_2 are ground terms, the sum of the lengths of the sublists of t_1 is equal to the length of the list t_2. This interargument relationship can be expressed as follows

$$\mathsf{Flatten}(t_1, t_2) : |t_1|^{sum}_{\mathsf{List(List(Int))}} = |t_2|^{len}_{\mathsf{List(Int)}}$$

where $|.|^{len}_{\mathsf{List(List(Int))}}$ and $|.|^{sum}_{\mathsf{List(Int)}}$ are the norms defined in Example 5.4. Note that this precise relationship can be expressed only using type-linear typed norms, or the typed norms of Decorte *et al.* 1993 and Bossi *et al.* 1992. □

5.3 Automatic generation of norms

To perform termination analysis or interargument relationship analysis on a program P, a finite set of norms is usually required which will enable the size of any term occurring in P to be measured. This section outlines how a set of type-linear typed norms suitable for this purpose can be derived directly from the prescribed types of a program. The actual norms needed will be determined by the types of the terms that can occur in P. In the following, two types are considered to be equivalent if one is a renaming of the other.

Definition 5.11 (argument types) Let $\langle \Sigma_p, \Sigma_f, V \rangle$ denote the underlying language of P. Then $P_{arg} = \{\tau_i \mid p_{\tau_1 \ldots \tau_n} \in \Sigma_p \wedge 1 \leq i \leq n\}$ is the set of *argument types* for P. □

The set P_{arg} represents the types of all terms occurring as arguments of atoms in P, in that if the type of an argument of some atom is τ, then either $\tau \in P_{arg}$, or there exists a type $\sigma \in P_{arg}$ and a type substitution ψ such that $\tau = \psi(\sigma)$. The following definition captures the types of subterms of arguments.

Definition 5.12 (argument subtypes) For each $\tau \in P_{arg}$, the set of *argument subtypes* of τ is the least set P^τ_{sub} such that $\tau \in P^\tau_{sub}$ and if $f_{\langle \rho_1 \ldots \rho_n, \rho \rangle} \in \Sigma_f$, $\sigma \in P^\tau_{sub}$ and $\sigma = \psi(\rho)$, then for all $i \in [1, n]$, $\psi(\rho_i) \in P^\tau_{sub}$. □

Example 5.10 Let P define the language $\langle \Sigma_p, \Sigma_f, V \rangle$, where

$$\Sigma_f = \{\mathsf{Nil}_{\langle \epsilon, \mathsf{List(u)} \rangle}, \mathsf{Cons}_{\langle \mathsf{u.List(u)}, \mathsf{List(u)} \rangle}\}$$
$$\Sigma_p = \{\mathsf{P}_{\mathsf{List(List(u))}}, \mathsf{Q}_{\mathsf{List(u)}}\}$$

Then

$$\begin{aligned} P_{arg} &= \{\mathsf{List(List(u))}, \mathsf{List(u)}\} \\ P_{sub}^{\mathsf{List(List(u))}} &= \{\mathsf{List(List(u))}, \mathsf{List(u)}, \mathsf{u}\} \\ P_{sub}^{\mathsf{List(u)}} &= \{\mathsf{List(u)}, \mathsf{u}\} \end{aligned}$$

□

By defining a norm $|.|_\tau$ for each $\tau \in P_{arg}$, the size of any argument occurring in the program can be measured. The sets $P_{sub}^{\tau_i}$ are used to facilitate the definitions of these norms. It will often be the case that some of the arguments in a program have the same type and different norms may be required to measure the sizes of such arguments. Thus for each $\tau \in P_{arg}$ a norm is defined which is parameterised by a weight function w as in the preceding section. Later, different w can be defined for individual arguments.

Before defining the induction process it is worth making an important observation which has an effect on the definition of the norms. First note that the type of a constant or the range type of a function must be either a base type or a type with a constructor in it (i.e. it cannot be a parameter). A consequence of this is that any term whose type is a parameter is a variable. The term structure of any term assigned to this variable cannot be accessed or altered in any way within the local computation, since if it could, the type of the term would be known and thus the variable would be of a more specific type. Thus the term (and its size measured wrt to any norm) never changes and hence has no effect on termination at the local level. This means that when defining the norm $|.|_u$ where $u \in U$, the value of $|t|_u$ for any term t should be constant. To simplify the definition it may be assumed that this constant value is zero. Furthermore, the norm $|.|_u$ can be removed from any definition which depends on it.

Definition 5.13 (induced typed norm) For each $\tau \in P_{arg}$ the type-linear typed norm $|.|_\tau^w : \mathsf{ED}_\tau \rightarrow \mathsf{ED}_{\mathsf{Lin}}$ is defined as the least set of equations E_τ^w as follows. If $\tau \in U$ then $E_\tau^w = \{|.|_\tau^w = 0\}$, else

$$\begin{aligned} E_\tau^w = &\left\{ |v|_\sigma^w = v \,|\, v \in V \wedge \sigma \in P_{sub}^\tau \right\} \cup \\ &\left\{ |f_{\langle\sigma_1 \ldots \sigma_n,\sigma\rangle}(t_1,\ldots,t_n)|_\sigma^w = w(f_{\langle\sigma_1\ldots\sigma_n,\sigma\rangle},0) + \sum_{i=1}^n w(f_{\langle\sigma_1\ldots\sigma_n,\sigma\rangle},i)|t_i|_{\sigma_i}^w \;\middle|\; \begin{array}{l} \sigma \in P_{sub}^\tau \wedge \\ f_{\langle\sigma_1\ldots\sigma_n,\sigma\rangle} \in \Sigma_f^\dagger \end{array} \right\} \end{aligned}$$

where w is a weight function partially defined for each $\tau \in P_{arg}$ such that for each $\sigma \in P_{sub}^\tau$ and $f_{\langle\sigma_1\ldots\sigma_n,\sigma\rangle} \in \Sigma_f$, and for all $i \in [1,n]$, $w(f_{\langle\sigma_1\ldots\sigma_n,\sigma\rangle},i) \in \mathbf{N}$ and for all $i \in [1,n]$ such that σ_i is a parameter then $w(f_{\langle\sigma_1\ldots\sigma_n,\sigma\rangle},i) = 0$. □

Note that due to the definition of P_{sub}^τ each $|.|_{\sigma_i}^w$ is defined in E_τ^w. Thus each E_τ^w is well defined pending a complete definition of the weight function w.

Example 5.11 Given P_{arg} as defined in Example 5.10, let p be a weight function for the type List(List(u)) and q be a weight function for the type List(u) partially defined as follows:

$$\begin{aligned} p(\mathsf{Nil}_{\langle\epsilon,\mathsf{List(List(u))}\rangle},0) &= w_1 & p(\mathsf{Nil}_{\langle\epsilon,\mathsf{List(u)}\rangle},0) &= w_5 \\ p(\mathsf{Cons}_{\langle\mathsf{List(u).List(List(u))},\,\mathsf{List(List(u))}\rangle},0) &= w_2 & p(\mathsf{Cons}_{\langle\mathsf{u.List(u)},\,\mathsf{List(u)}\rangle},0) &= w_6 \\ p(\mathsf{Cons}_{\langle\mathsf{List(u).List(List(u))},\,\mathsf{List(List(u))}\rangle},1) &= w_3 & p(\mathsf{Cons}_{\langle\mathsf{u.List(u)},\,\mathsf{List(u)}\rangle},1) &= 0 \\ p(\mathsf{Cons}_{\langle\mathsf{List(u).List(List(u))},\,\mathsf{List(List(u))}\rangle},2) &= w_4 & p(\mathsf{Cons}_{\langle\mathsf{u.List(u)},\,\mathsf{List(u)}\rangle},2) &= w_8 \end{aligned}$$

$$
\begin{aligned}
q(\mathsf{Nil}_{\langle \epsilon, \mathsf{List(u)} \rangle}, 0) &= w_9 \\
q(\mathsf{Cons}_{\langle \mathsf{u.List(u)},\ \mathsf{List(u)} \rangle}, 0) &= w_{10} \\
q(\mathsf{Cons}_{\langle \mathsf{u.List(u)},\ \mathsf{List(u)} \rangle}, 1) &= 0 \\
q(\mathsf{Cons}_{\langle \mathsf{u.List(u)},\ \mathsf{List(u)} \rangle}, 2) &= w_{12}
\end{aligned}
$$

where for all $i \in [1, 12]$, $w_i \in \mathbf{N}$. Choosing, for example, $w_1 = w_2 = w_5 = w_9 = 0$ and $w_3 = w_4 = w_6 = w_8 = w_{10} = w_{12} = 1$, the following equation sets may be derived

$$
E^p_{\mathsf{List(List(u))}} = \left\{ \begin{array}{ll}
|v|^p_{\mathsf{List(List(u))}} & = v, \\
|\mathsf{Nil}|^p_{\mathsf{List(List(u))}} & = 0, \\
|\mathsf{Cons}(t_1, t_2)|^p_{\mathsf{List(List(u))}} & = |t_1|^p_{\mathsf{List(u)}} + |t_2|^p_{\mathsf{List(List(u))}}, \\
|v|^p_{\mathsf{List}(u)} & = v, \\
|\mathsf{Nil}|^p_{\mathsf{List(u)}} & = 0, \\
|\mathsf{Cons}(t_1, t_2)|^p_{\mathsf{List(u)}} & = 1 + |t_2|^p_{\mathsf{List(u)}}
\end{array} \right\}
$$

$$
E^q_{\mathsf{List(u)}} = \left\{ \begin{array}{ll}
|v|^q_{\mathsf{List(u)}} & = v, \\
|\mathsf{Nil}|^q_{\mathsf{List(u)}} & = 0, \\
|\mathsf{Cons}(t_1, t_2)|^q_{\mathsf{List(u)}} & = 1 + |t_2|^q_{\mathsf{List(u)}}
\end{array} \right\}
$$

□

Note that the sets of terms for which the norms are defined are not disjoint. For example, the domain of the norm $|.|^p_{\mathsf{List(List(u))}}$ of Example 5.11 is a subset of the domain for the norm $|.|^q_{\mathsf{List(u)}}$. There is no confusion, however, when deciding which norm to use on a particular argument of an atom since the choice is determined by the atom's predicate symbol.

Example 5.12 Consider the atom $\mathsf{Q}_{\mathsf{List(List(Int))}}(\mathsf{Cons}(\mathsf{Cons}(1, \mathsf{Nil}), \mathsf{Nil}))$ which may appear as part of a goal for the predicate $\mathsf{Q}_{\mathsf{List(u)}}$. Although the type of the atom's argument is List(List(Int)), the correct norm to use would be $|.|^q_{\mathsf{List(u)}}$ and not $|.|^p_{\mathsf{List(List(u))}}$ since the type of the predicate is List(u). □

All that remains now to complete the definitions of the derived norms is to fully define a suitable weight function. This in itself is a non-trivial problem.

5.3.1 Defining the weight function

Most of the approaches to termination analysis based on norms essentially use a simple generate-and-test method for deducing termination. Norms are generated (either automatically or otherwise) and used to form level mappings which are then applied to the program for which a termination proof is sought. Inequalities are then derived whose solubility indicates the success or failure of the termination proof.

The main difficulty with this approach is the potentially infinite number of norms that can be generated. To reduce the complexity of this problem a number of heuristics can be used. Decorte et al. Decorte *et al.* 1993, for example, propose the following (adapted) heuristics for deriving typed norms.

1. A weight of one is assigned to all functors of arity $n > 0$.

2. A weight of zero is assigned to all constants.

3. Any argument position whose type is not a parameter is assigned a weight of one.

Applying these heuristics to the partially derived norms results in the same norms that would be derived by Decorte *et al.* 1993 given the same type information in the form of a type graph. Although this approach works well on a large number of examples, there are occasions when it will fail to generate norms that can be used in a termination proof. The naive reverse program with an accumulating parameter Decorte *et al.* 1993 is one example where an argument position needs to be assigned a weight of zero, effectively meaning that the size of the subterms occuring in that argument position are not counted by the norm. In that paper a solution to this problem is sketched using *symbolic norms* which effectively define an argument index function through an exhaustive search. The example below shows that the second heuristic is also not always effective.

Example 5.13 If each constant occurring in the program below is assigned a weight of zero then the interargument relation derived for Path(x, y) would be $|x| = |y| = 0$. With this relationship, termination cannot be proved since $|x| > |z|$ is required to hold in the recursive TransitiveClosure/2 clause. To prove termination each constant must take on a different value.

TransitiveClosure(x, y) ← Path(x, y).
TransitiveClosure(x, y) ← Path(x, z) ∧ TransitiveClosure(z, y).

Path(A, B).
Path(B, C).

□

This example seems to suggest that the determination of weights must take place as an integral part of a termination analysis – the variety of the weights occurring indicates the futility of a generate and test approach in this instance. Recently, such a demand-driven approach has been described in Decorte & De Schreye 1997, where the weights are determined so as to satisfy the various inequalities needed to prove termination. This approach relies on first generating norms which are parameterised in exactly the same way as the induced typed norms of Definition 5.13. Thus the technique described here could be integrated with the analysis of Decorte & De Schreye 1997 in a typed context.

In summary, there are several approaches to the problem of deriving the weight function. No particular method is advocated here since it is necessary to further investigate and compare suitable methods. The open-ended definitions of the derived norms should facilitate such a study.

5.4 Related work

One weakness of Decorte *et al.* 1993 is that its norms are derived from type graphs. Type graph analyses, however, have not always been renowned for their tractability. Even for small programs, the prototype analyser of Janssens & Bruynooghe 1992, used in Decorte *et al.* 1993, is typically 15 times slower than the optimising PLM compiler (Van Roy 1984). Recently, type graph analysis has been shown to be practical for *medium*-sized Prolog programs (Van Hentenryck *et al.* 1994) when augmented

with an improved widening and compacting procedure. In addition, Gallagher & de Waal 1994 have shown how type graphs can be efficiently represented as unary logic programs. Clearly, however, any approach which avoids the costs of inferring type graphs is preferable.

Bossi *et al.* 1992 define a very general concept of norm in terms of type schemata which describe structural properties of terms. Their typed norms for termination analysis are very similar to the ones presented in this chapter, though they are able to define some norms which cannot be inferred using the present framework.

Example 5.14 Consider the following program from Bossi *et al.* 1992

```
Check(Cons(x, xs)) ← Check(xs).
Check(Cons(x, Nil)) ← Nat(x).
Nat(Succ(x)) ← Nat(x).
Nat(0).
```

We would like to define a norm $|.|_{\mathsf{List(Nat)}}$ so that we can prove termination for goals $\leftarrow$ Check(x) where x is rigid wrt $|.|_{\mathsf{List(Nat)}}$. The following norm adapted from Bossi *et al.* 1992 satisfies this criterion.

$$
\begin{array}{lll}
|v|_{\mathsf{List(Nat)}} = v & |v|_{\mathsf{Nat}} = v & |v|_{\mathsf{Empty}} = v \\
|\mathsf{Cons}(t_1, t_2)|_{\mathsf{List(Nat)}} = 1 + |t_2|_{\mathsf{List(Nat)}} & |0|_{\mathsf{Nat}} = 0 & |\mathsf{Nil}|_{\mathsf{Empty}} = 0 \\
|\mathsf{Cons}(t_1, t_2)|_{\mathsf{List(Nat)}} = |t_1|_{\mathsf{Nat}} + |t_2|_{\mathsf{Empty}} & |\mathsf{Succ}(t)|_{\mathsf{Nat}} = 1 + |t|_{\mathsf{Nat}} &
\end{array}
$$

This norm cannot be inferred automatically using the proposed method (nor that of Decorte *et al.* 1993) since it is necessary for the functor Cons to have two distinct types, namely $\langle \mathsf{Nat.List(Nat)}, \mathsf{List(Nat)} \rangle$ and $\langle \mathsf{Nat.Empty}, \mathsf{List(Nat)} \rangle$, but this is forbidden in languages like Gödel where the declarations are universal. Note that this is not a limitation of the framework but rather a limitation of the type system on which it is based. Given a more flexible system it would be possible to infer such norms as the above directly from the prescribed types. □

Finally, note that the typed norms of Bossi *et al.* 1992 are not derived automatically. By contrast, typed-linear typed norms, are simple enough to be easily derived using only the type declarations of a program.

5.5 Conclusions and future work

This chapter has presented a flexible method for inferring a number of norms from the type declarations of a program which are sufficient to measure the size of any Herbrand term occurring in the program in an almost arbitrary way. The norms are intended for use in termination analysis and the derivation of inter-argument relationships, though their applicability is not restricted to these areas. The definition of each derived norm is parameterised by a weight function. This open-ended definition allows the norms to be incorporated into a wide range of analyses which define these functions in different ways. Defining the weight function in an efficient and intelligent way is a non-trivial problem in itself. The definitions of norms proposed here provides a useful framework in which to study this problem.

6 Termination and Left Termination

Norms, such as those discussed in the last chapter, are often used in termination analysis as a basis for constructing level mappings. Recall that a level mapping is a function which provides some measure of the size of an atom and it is natural to express such a function in terms of norms which measure the sizes of the subterms in the atom. The interest in level mappings, of course, lies in their use in the construction of termination proofs based on notions such as recurrency or acceptability as introduced in Chapter 3.

However, while the notions of recurrency and acceptability provide a sound theoretical basis for reasoning about termination, they do not provide much insight into the practicalities of actually deriving the level mappings which are needed to prove a program terminating or left terminating. Instead, intuition has served as the guide in the development of automatic techniques. In particular, there has been a desire to derive "natural" level mappings based on the recursive structure of the program at hand. For example, given the program

P([h|t]) ← P(t).

it is natural to define a level mapping $|.|$ to prove termination by $|\mathsf{P}(x)| = |x|_{\textit{list-length}}$ since the predicate is inductively defined over the structure of its argument, which is a list. Other definitions, such as $|\mathsf{P}(x)| = |x|_{\textit{list-length}} + 1$ and $|\mathsf{P}(x)| = 2 \times |x|_{\textit{list-length}}$ do not possess the same "natural" correspondence with the intution behind the program's terminating behaviour. The desire for natural level mappings is not just an aesthetic predilection. Such level mappings are also easier to derive.

This chapter examines the reasons why termination proofs based on recurrency and acceptability are often difficult to obtain. The observations are not new and have been made, among others, by Apt & Pedreschi 1994. Their solution was to define alternative characterisations of terminating and left terminating programs which they called *semi recurrency* and *semi acceptability* respectively. This solution is investigated in Sections 6.2 and 6.3 where it is demonstrated that it is not entirely satisfactory.

This leads to the main contribution of this chapter in Sections 6.4 and 6.5 where the notions of *bounded recurrency* and *bounded acceptability* are introduced. The classes of bounded recurrent and bounded acceptable programs are shown to be equivalent to the recurrent and acceptable classes respectively. Moreover, since these new notions are more aligned with the intuitions underlying termination proofs, they lend themselves more naturally to the automatic construction of such proofs.

6.1 The Recurrent Problem

The main problem with recurrency, as noted by De Schreye *et al.* 1992 and Apt & Pedreschi 1994, is that it does not intuitively relate to recursion, the principal cause of non-termination in a logic program. The definition requires that, for every ground instance of a clause, the level of its head atom is greater than the level of every body atom irrespective of the recursive relation between the two.

Definition 6.1 (recurrency Bezem 1989) Let P be a definite logic program and $|.|$ a level mapping for P. A clause $H \leftarrow B_1, \ldots, B_n$ is recurrent (wrt $|.|$) if for every grounding substitution θ, $|H\theta| > |B_i\theta|$ for all $i \in [1, n]$. P is recurrent (wrt $|.|$) if every clause in P is recurrent (wrt $|.|$). □

There is a temptation to "fix" this by using a modified definition of recurrency which only requires a decrease for mutually recursive body atoms. The following example, from De Schreye *et al.* 1992, shows that this requirement alone is too weak to prove termination.

Example 6.1 Using the weaker form of recurrency suggested above, the following program would be classed as recurrent.

P([h|t]) ← Append(x, y, z) ∧ P(t).

Append([u|x], y, [u|z]) ← Append(x, y, z).
Append([], x, x).

Using the left-to-right computation rule and the top-down search rule, however, the goal ← P([1,2]) admits an infinite computation. Of course, the clause defining the predicate P/1 should not be classified as recurrent. The reason is that, while Append/3 is recurrent (even by Bezem's definition), only bounded goals should be guaranteed to terminate and the predicate P/1 contains an unbounded call to Append/3. □

This example shows that the level mapping decrease between the head and the non-recursive atoms of a clause implied by Definition 6.1, is required to ensure that all subcomputations are initiated from a bounded goal. Enforcing boundedness in this way, however, complicates the derivation of level mappings. The following example, illustrating this, also comes from De Schreye *et al.* 1992.

Example 6.2 Consider the following program

p_1 P([]).
p_2 P([h|t]) ← Q([h|t]) ∧ P(t).

q_1 Q([]).
q_2 Q([h|t]) ← Q(t).

It is clear that this program is terminating for any goal ← P(x) where x is a rigid list. To construct an automatic proof of this one would like to use the "natural" level mapping $|.|$ defined by

$$|\mathsf{P}(x)| = |x|_{\textit{list-length}} \qquad |\mathsf{Q}(x)| = |x|_{\textit{list-length}}$$

The problem is that the clause p_2 is not recurrent wrt this level mapping since it is not the case that $|\mathsf{P}([h|t])\theta| > |\mathsf{Q}([h|t])\theta|$ for all grounding substitutions θ. For the inequality to hold, an "unnatural" offset must be included in the level mapping definition by taking for example $|\mathsf{P}(x)| = |x|_{\textit{list-length}} + 1$. □

The above examples show that the strict decrease in the level mapping between head and body atoms of a recurrent clause is required for two distinct purposes.

1. To ensure that the levels of mutually recursive calls are strictly decreasing.
2. To ensure that subcomputations are initiated from a bounded goal.

6.2 Semi Recurrency

Apt and Pedreschi observed that, for termination, while it is necessary for the level mapping to decrease between the head of a clause and each mutually recursive body atom, a strict decrease is not required for the non-recursive body atoms. They introduced the notion of semi recurrency which exploited this observation and showed the classes of recurrent and semi recurrent programs to be equivalent.

Definition 6.2 (semi recurrency Apt & Pedreschi 1994) Let P be a definite logic program and $|.|$ a level mapping for P. A clause $H \leftarrow B_1, \ldots, B_n$ is semi recurrent (wrt $|.|$) if for every grounding substitution θ, for all $i \in [1, n]$

1. $|H\theta| > |B_i\theta|$ if $rel(H) \simeq rel(B_i)$,
2. $|H\theta| + 1 > |B_i\theta|$ if $rel(H) \not\simeq rel(B_i)$.

P is semi recurrent (wrt $|.|$) if every clause in P is semi recurrent (wrt $|.|$). □

Whilst this definition now admits a simple termination proof of Example 6.2 using the original "natural" level mapping of that example, it is not hard to construct examples where it is inadequate.

Example 6.3 Consider the following program

c_1 P([]).
c_2 P([h|t]) $\leftarrow$ Q([h, h|t]) $\wedge$ P(t).

c_3 Q([]).
c_4 Q([h|t]) $\leftarrow$ Q(t).

To prove that the above program is semi recurrent requires the following unnatural level mapping.

$$|\mathsf{P}(x)| = |x|_{\textit{list-length}} + 1 \qquad |\mathsf{Q}(x)| = |x|_{\textit{list-length}}$$

□

It seems that very little has actually been gained from this revised definition of recurrency which still insists that there is a non-increasing relationship between the level of the head and the level of all body atoms. In fact, it does not matter if the level of a non-recursive atom is greater than the level of the head provided that such an atom is bounded whenever it is selected.

To be fair, the notion of semi recurrency was introduced to facilitate modular termination proofs and does indeed, in some cases, allow proofs to be based on simpler level mappings than those used in proofs of recurrency. In the above example, however, this is not the case.

Example 6.4 Reconsider the program of Example 6.3. According to the methodology of Apt & Pedreschi 1994 a modular termination proof can be contructed in a kind of bottom up fashion on the recursive cliques of the predicate dependency graph. The details of the methodology will not be explained here; only the steps involved

in proving termination of the above program will be worked through. First Q/1 is proven to be (semi) recurrent wrt $|.|_Q$ defined by

$$|\mathsf{Q}(x)|_Q = |x|_{\textit{list-length}}$$

Second, P/1 is proven to be (semi) recurrent wrt $|.|_P$ defined by

$$|\mathsf{P}(x)|_P = |x|_{\textit{list-length}} \qquad |\mathsf{Q}(x)|_P = 0$$

The final step in the proof requires the derivation of a level mapping $|.|'$ such that

$$|\mathsf{P}([t_1|t_2])|' \geq |\mathsf{Q}([t_1, t_1|t_2])|_Q \quad \text{and} \quad |\mathsf{P}([t_1|t_2])|' \geq |\mathsf{P}(t_2)|'$$

for all ground terms t_1 and t_2. Providing the level mapping $|.|'$ exists, theorem 4.9 of Apt & Pedreschi 1994 can be used to draw the conclusion that the program is semi recurrent and hence terminating. In terms of automation, this existence proof is achieved through defining $|.|'$ so that the above inequalities are satisfied. However, the most likely choice of a definition for $|.|'$ is

$$|\mathsf{P}(x)|' = |x|_{\textit{list-length}} + 1$$

Of course, this is no easier to derive than the original mapping $|.|$ of Example 6.3. □

What is most conspicuous about the definition of semi recurrency, is that the difference in levels between a non-recursive body atom and the head atom of a clause is limited to be at most zero, whereas it could be arbitrarily large, though still finite. Indeed, a simple termination proof for the program of Example 6.3 can be obtained using a "natural" level mapping if condition 2 of Definition 6.2 is replaced by $|H\theta| + k > |B_i\theta|$ if $rel(H) \not\simeq rel(B_i)$, where k is some large constant. It is easy to prove that the class of programs captured by this revised definition of semi recurrency is equivalent to the class of recurrent programs. In addition, theorems 4.6, 4.8 and 4.9 of Apt & Pedreschi 1994, which are used for constructing modular termination proofs, all still hold with this alternative definition. Moreover, the premises of those theorems may be weakened in an obvious manner to permit such proofs to be constructed more easily.

Note that the problem with the termination proofs above arises because the atoms in the body of a clause contain extra function symbols which raise the levels of those atoms to the level of the head. Since it is fairly unlikely that such a body atom will contain, say, a million function symbols or more, by taking $k = 1000000$ the vast majority of recurrent programs which occur in practice could be proven terminating by focusing solely on their recursive structure and employing the appropriate weakened forms of the theorems of Apt and Pedreschi.

6.3 Semi Acceptability

Similar remarks to those of Section 6.1 can be made about the definition of acceptability. The notion of semi acceptability was introduced as an analogous concept to semi recurrency for left terminating programs.

Definition 6.3 (semi acceptability Apt & Pedreschi 1994) Let $|.|$ be a level mapping and I an interpretation for a program P. A clause $c : H \leftarrow B_1, \ldots, B_n$ is *semi acceptable* wrt $|.|$ and I iff

1. I is a model for c and
2. for all $i \in [1, n]$ and for every grounding substitution θ for c such that $I \models \{B_1, \ldots, B_{i-1}\}\theta$
   (a) $|H\theta| > |B_i\theta|$ if $rel(H) \simeq rel(B_i)$,
   (b) $|H\theta| + 1 > |B_i\theta|$ if $rel(H) \not\simeq rel(B_i)$.

P is semi acceptable (wrt $|.|$ and I) iff every clause in P is semi acceptable (wrt $|.|$ and I). □

Not surprisingly, termination proofs based on semi acceptability suffer from similar problems to those encountered in Examples 6.3 and 6.4. The definition could be adjusted in the manner prescribed above for semi recurrency, but the result is not as satisfactory as the following example shows.

Example 6.5 Consider the following program

% DoubleSquare$(x, l) : l = [2(x-1)^2, 2(x-2)^2, \ldots, 0]$

```
DoubleSquare(0, []).
DoubleSquare(S(x), [d|ds]) ←
    Square(x, 0, y)∧
    DoublePlus(y, 0, d)∧
    DoubleSquare(x, ds).
```

% Square$(x, 0, y) : y = x^2$

```
Square(0, y, y).
Square(S(x), acc, y) ←
    DoublePlus(x, S(acc), acc1)∧
    Square(x, acc1, y).
```

% DoublePlus$(x, y, z) : z = 2x + y$

```
DoublePlus(0, x, x).
DoublePlus(S(x), y, S(S(z))) ←
    DoublePlus(x, y, z).
```

Let the level mapping $|.|$ be defined by

$$|\text{DoubleSquare}(x, y)| = |x|_s \qquad |\text{Square}(x, y, z)| = |x|_s \qquad |\text{DoublePlus}(x, y, z)| = |x|_s$$

where $|0|_s = 0$ and $|\text{S}(x)|_s = 1 + |x|_s$. The predicates Square/3 and DoublePlus/3 are both recurrent (and hence acceptable) wrt $|.|$, but there is no value of k for which the inequality $|\text{DoubleSquare(S(x), [d|ds])}\theta| + k > |\text{DoublePlus(y, 0, d)}\theta|$ holds for all grounding substitutions θ such that $I \models \text{Square(x, 0, y)}\theta$ where I is a model of the program. Hence the predicate DoubleSquare/2 is not semi acceptable wrt $|.|$ even under the revised definition suggested above. It is easy to prove (semi) acceptability of the program, however, wrt the level mapping $|.|'$ where $|.|'$ is defined exactly as for $|.|$ except that $|\text{DoubleSquare}(x, y)|' = |x|_s^2$. Note that a goal is bounded wrt $|.|$ if and only if it is bounded wrt $|.|'$ and all such goals are left terminating. It seems reasonable then to base a proof of termination on the former level mapping since it more closely relates to the recursion and as a result is easier to derive automatically. Indeed, no automatic termination analysis has yet been devised which can derive level mappings defined in terms of polynomial expressions such as $|.|'$. □

Observe that the k above acts as an upper bound on the difference between the level of any body atom and the level of the head atom. Of course, this ad hoc approach falls down when there is no upper bound as in Example 6.5.

In summary, although semi recurrency and semi acceptability are more flexible notions than their predecessors, they still enforce a dependence between the level of a head atom and the levels of non-recursive body atoms. This dependence is counter intuitive and forces one to use artificial level mappings to obtain termination proofs.

6.4 Bounded Recurrency

Recall from Section 6.1 that there are two conditions which must be fulfilled to ensure that a program is terminating.

1. The levels of mutually recursive calls are strictly decreasing.
2. All subcomputations are initiated from a bounded goal.

This section examines how the notions of recurrency and semi recurrency can be developed such that the above two conditions are cleanly separated. This leads to the definition of bounded recurrency, a characterisation of terminating programs that more closely matches the intuition underlying termination proofs and as a result facilitates the automatic construction of such proofs.

To fully motivate the definition of bounded recurrency in Definition 6.5 it is useful to consider an evolutionary step in the form of Definition 6.4. This may be viewed as an initial attempt at defining the notion of bounded recurrency. It will be seen that this definition does not fully achieve the desired separation in terms of the conditions above and as such it is further refined to obtain Definition 6.5.

Definition 6.4 (single bounded recurrency) Let $|.|$ be a level mapping for a program P. A clause $c : H \leftarrow B_1, \ldots, B_n$ is *single bounded recurrent* wrt $|.|$ iff for all $i \in [1, n]$ and for every substitution θ for c such that $H\theta$ is bounded wrt $|.|$

1. $B_i\theta$ is bounded wrt $|.|$, and
2. $|[H\theta]| > |[B_i\theta]|$ whenever $rel(H) \simeq rel(B_i)$.

P is single bounded recurrent wrt $|.|$ iff every clause in P is single bounded recurrent wrt $|.|$. □

Observe that, in this definition, no decrease, or indeed any fixed difference, is enforced between the level of the head of a clause and the levels of the non-recursive body atoms. All that is required is that each atom is bounded whenever the head is bounded. While this is more intuitively appealing, observe that boundedness of non-recursive atoms still influences the definition of the level mapping in a non-modular way.

Example 6.6 Consider the following program for Curry's type assignment taken from Apt & Pedreschi 1994.

```
typ1  Type(e, Var(x), t) ←
          In(e, x, t).
typ2  Type(e, Apply(m, n), t) ←
          Type(e, m, Arrow(s, t)) ∧
          Type(e, n, s).
typ3  Type(e, Lambda(x, m), Arrow(s, t)) ←
          Type([(x, s) | e], m, t).

in1   In([(x, t) | e], x, t).
in2   In([(y, t) | e], x, t) ←
          x ≠ y ∧
          In(e, x, t).
```

One may observe that the predicate In/3 is inductively defined over the length of its first argument which is a list. The predicate Type/3 is inductively defined on the size of its second argument which is a λ-term. As a result, one would hope to base a termination proof on the level mapping $|.|$ defined by

$$|\mathsf{In}(x, y, z)| = |x|_{\textit{list-length}} \qquad |\mathsf{Type}(x, y, z)| = |y|_{\textit{term-size}}$$

where $|y|_{\textit{term-size}}$ denotes the number of function symbols in the term y. The problem, of course, is that any call Type(e, Var(x), t) which is bounded wrt $|.|$ can give rise to a call In(e, x, t) which is not bounded wrt $|.|$. Clearly this can lead to non-termination. Definition 6.4, therefore, insists that for the clause typ_1 the body atom In(e, x, t) is bounded whenever the head is. Unfortunately this entails that the level mapping must now be modified to take the first argument of Type/3 into account. This in turn leads to problems with the clause typ_3 since the first argument is increasing in the recursive call. Eventually, one arrives at a level mapping definition such as

$$|\mathsf{In}(x, y, z)|_1 = |x|_{\textit{list-length}} \qquad |\mathsf{Type}(x, y, z)|_1 = |x|_{\textit{list-length}} + 2 \times |y|_{\textit{term-size}}$$

which bears no immediate relation to the program structure. As a result such a mapping can be hard to derive automatically. □

Clearly there is an interdependence between ensuring non-recursive atoms are bounded wrt $|.|$ and ensuring that the levels of recursive calls are decreasing wrt $|.|_.$. This plainly arises out of the use of the one level mapping. It seems therefore that the obvious way to break the dependence is to use *two* level mappings. One holds the responsibility for ensuring the recursive decrease in levels, while the other assures that non-recursive atoms are bounded. This idea is captured in the following definition.

Definition 6.5 (bounded recurrency) Let $|.|_1$ and $|.|_2$ be level mappings for a program P. A clause $c : H \leftarrow B_1, \ldots, B_n$ is *bounded recurrent* (wrt $|.|_1$ and $|.|_2$) iff for all $i \in [1, n]$ and for every substitution θ for c such that $H\theta$ is bounded wrt $|.|_1$ and $|.|_2$

1. $B_i\theta$ is bounded wrt $|.|_1$ and $|.|_2$, and
2. $|[H\theta]|_1 > |[B_i\theta]|_1$ whenever $\mathit{rel}(H) \simeq \mathit{rel}(B_i)$.

P is bounded recurrent (wrt $|.|_1$ and $|.|_2$) iff every clause in P is bounded recurrent (wrt $|.|_1$ and $|.|_2$). □

It is informally understood that a goal G is bounded wrt $|.|_1$ and $|.|_2$ iff G is bounded wrt $|.|_1$ and G is bounded wrt $|.|_2$. Note that, when the two level mappings coincide, that is when $|.|_1 \equiv |.|_2$, then Definition 6.5 is equivalent to Definition 6.4.

Example 6.7 Returning to the program of Example 6.6, recall that the stumbling block in the derivation of a natural level mapping arose because any call Type(e, Var(x), t) which is bounded wrt $|.|$ can give rise to a call In(e, x, t) which is not bounded wrt $|.|$. At this point, one intuitively reasons that if the first argument of a call to Type/3 is a rigid list then the first argument of all subsequent calls to Type/3 will also be a rigid list. So define a second level mapping $|.|'$ by

$$|\mathsf{In}(x,y,z)|' = |x|_{\textit{list-length}} \qquad |\mathsf{Type}(x,y,z)|' = |x|_{\textit{list-length}}$$

The program is bounded recurrent wrt $|.|$ and $|.|'$. Indeed, any call to Type/3 or In/3 which is bounded wrt $|.|$ and $|.|'$ only gives rise to calls which are bounded wrt $|.|$ and $|.|'$. Combine this with the fact that recursive calls are decreasing wrt $|.|$ and termination can be proven in a very intuitive manner. Furthermore the level mappings $|.|$ and $|.|'$ follow directly from the structure of the program, facilitating their automatic derivation. □

Lemma 6.6 and Corollary 6.7 below establish that bounded recurrent programs are indeed terminating. Proof of this relies on orderings which not only take into account the levels of atoms but also their relation to each other in the predicate dependency graph.

For a level mapping $|.|$ and goal $G = \leftarrow A_1, \ldots, A_n$, if G is bounded wrt $|.|$ then let $|[G]|$ denote the finite multiset of pairs $\{(rel(A_1), |[A_1]|), \ldots, (rel(A_n), |[A_n]|)\}$. Let $\prec$ be the lexicographical ordering on $\Sigma_p(\sqsubset) \times \mathbf{N}(<)$ and let $\prec_{mul}$ be the multiset ordering based on $\prec$. Observe that $\prec_{mul}$ is well founded.

Lemma 6.6 Let $|.|_1$ and $|.|_2$ be level mappings for a program P. Let P be bounded recurrent wrt $|.|_1$ and $|.|_2$ and let G be a goal which is bounded wrt $|.|_1$ and $|.|_2$. Let G' be an SLD-resolvent of G from P. Then

1. G' is bounded wrt $|.|_1$ and $|.|_2$,
2. $|[G']|_1 \prec_{mul} |[G]|_1$, and
3. every SLD-derivation of $P \cup \{\leftarrow G\}$ is finite.

Proof 4 *Assume A_j is the selected literal in $G = \leftarrow A_1, \ldots, A_m$ and $c : H \leftarrow B_1, \ldots, B_n$ ($n \geq 0$) the program clause used. Then $G' = \leftarrow (A_1, \ldots, A_{j-1}, B_1, \ldots, B_n, A_{j+1}, \ldots, A_m)\theta$ where $\theta \in mgu(A_j, H)$.*

1. *Since G is bounded wrt $|.|_1$ and $|.|_2$, it follows that A_k and $A_k\theta$ are bounded wrt $|.|_1$ and $|.|_2$ for all $k \in [1, m]$. In particular, $A_j\theta = H\theta$ is bounded wrt $|.|_1$ and $|.|_2$. It follows, by Definition 6.5, that $B_i\theta$ is bounded wrt $|.|_1$ and $|.|_2$ for all $i \in [1, n]$ and hence G' is bounded wrt $|.|_1$ and $|.|_2$.*
2. *Moreover, $|[A_k]|_1 \geq |[A_k\theta]|_1$ for all $k \in [1, m]$ by Lemma 3.13. Finally, for all $i \in [1, n]$*

   (a) $|[A_j\theta]|_1 > |[B_i\theta]|_1$ if $rel(A_j) = rel(H) \simeq rel(B_i)$, by Definition 6.5, and

(b) $rel(A_j) \sqsupset rel(B_i\theta)$ *otherwise.*

Hence

$$\begin{array}{rcll} (rel(B_i\theta), |[B_i\theta]|_1) & \prec & (rel(A_j), |[A_j]|_1) & \text{for all } i \in [1,n] \\ (rel(A_k\theta), |[A_k\theta]|_1) & \preceq & (rel(A_k), |[A_k]|_1) & \text{for all } k \in [1,m] \end{array}$$

proving $|[G']|_1 \prec_{mul} |[G]|_1$.

3. *Since* $\prec_{mul}$ *is well-founded the result follows immediately.*

Corollary 6.7 Every bounded recurrent program is terminating.

Theorem 6.8 Let $|.|_1$ and $|.|_2$ be level mappings for a program P. The following hold.

1. If P is recurrent wrt $|.|_1$ then P is bounded recurrent wrt $|.|_1$ and $|.|_1$.
2. If P is bounded recurrent wrt $|.|_1$ and $|.|_2$, then there exists a level mapping $|.|_3$ such that P is recurrent wrt $|.|_3$. Moreover, for any atom A, A is bounded wrt $|.|_3$ if A is bounded wrt $|.|_1$ and $|.|_2$.

Proof 5 *Let* $c : H \leftarrow B_1, \ldots, B_n$ *be a clause in* P. *Suppose* P *is recurrent wrt* $|.|_1$. *Let* θ *be a substitution such that* $H\theta$ *is bounded wrt* $|.|_1$. *Then* $B_i\theta$ *is bounded and* $|[H\theta]|_1 > |[B_i\theta]|_1$ *for all* $i \in [1,n]$ *by recurrency. The second part follows by Lemma 6.6 and theorem 2.2 and corollary 2.2 of Bezem 1993.*

6.5 Bounded Acceptability

The definition of bounded recurrency is easily adapted to obtain a characterisation of left terminating programs.

Definition 6.9 (bounded acceptability) Let $|.|_1$ and $|.|_2$ be level mappings and I an interpretation for a program P. A clause $c : H \leftarrow B_1, \ldots, B_n$. is *bounded acceptable* (wrt $|.|_1$, $|.|_2$ and I) iff I is a model for c and for all $i \in [1,n]$, for every substitution θ such that $H\theta$ is bounded wrt $|.|_1$ and $|.|_2$, $\{B_1, \ldots, B_{i-1}\}\theta$ is ground and $I \models \{B_1, \ldots, B_{i-1}\}\theta$

1. $B_i\theta$ is bounded wrt $|.|_1$ and $|.|_2$, and
2. $|[H\theta]|_1 > |[B_i\theta]|_1$ whenever $rel(H) \simeq rel(B_i)$.

P is bounded acceptable (wrt $|.|_1$, $|.|_2$ and I) iff every clause in P is bounded acceptable (wrt $|.|_1$, $|.|_2$ and I). □

Lemma 6.10 asserts that every bounded acceptable program is left terminating. The proof of this follows along the same lines as that for acceptable programs.

Lemma 6.10 Let $|.|_1$ and $|.|_2$ be level mappings and I an interpretation for a program P. Let P be bounded acceptable wrt $|.|_1$, $|.|_2$ and I, and let G be a goal which is left bounded wrt $|.|_1$ and I, and wrt $|.|_2$ and I. Let G' be an LD-resolvent of G from P. Then

1. G' is left bounded wrt $|.|_1$ and I, and wrt $|.|_2$ and I,

2. $|[G']_I|_1 \prec_{mul} |[G]_I|_1$, and
3. every LD-derivation of $P \cup \{\leftarrow G\}$ is finite.

Proof 6 *Let $G = \leftarrow A_0, A_1, \ldots, A_m$ $(m > 0)$ and assume $c : H \leftarrow B_1, \ldots, B_n$ $(n \geq 0)$ is the program clause used. Then $G' = \leftarrow (B_1, \ldots, B_n, A_1, \ldots, A_m)\theta$ where $\theta \in mgu(A_0, H)$.*

1. *Need to show for all $j \in [1,2]$, $i \in [1, n+m]$ that $|[G']_I^i|_j$ is finite. Firstly, for all $j \in [1,2]$, $i \in [1,n]$*

$$\begin{aligned}
|[G']_I^i|_j &= |[\leftarrow (B_1, \ldots, B_n, A_1, \ldots, A_m)\theta]_I^i|_j \\
&= \left\{ |B_i\theta\phi|_j \;\middle|\; \begin{array}{l} \phi \text{ is a grounding substitution for } G' \\ I \models \{B_1, \ldots, B_{i-1}\}\theta\phi \end{array} \right\} \\
&= \left\{ |B_i\theta\phi\sigma|_j \;\middle|\; \begin{array}{l} \phi \text{ is a grounding substitution for } \{B_1, \ldots, B_{i-1}\}\theta \\ I \models \{B_1, \ldots, B_{i-1}\}\theta\phi \\ \sigma \text{ is a grounding substitution for } B_i\theta\phi \end{array} \right\}
\end{aligned}$$

Now by Definition 6.9, for all $i \in [1,n]$, for every substitution ϕ such that $H\theta\phi$ is bounded wrt $|.|_1$ and $|.|_2$, $\{B_1, \ldots, B_{i-1}\}\theta\phi$ is ground and $I \models \{B_1, \ldots, B_{i-1}\}\theta\phi$

   (a) $B_i\theta\phi$ is bounded wrt $|.|_1$ and $|.|_2$, and

   (b) $|[H\theta\phi]|_1 > |[B_i\theta\phi]|_1$ whenever $rel(H) \simeq rel(B_i)$.

Hence, $|[G']_I^i|_j$ is finite for all $i \in [1,n]$, $j \in [1,2]$. Now for all $j \in [1,2]$, $k \in [1,m]$

$$\begin{aligned}
|[G']_I^{n+k}|_j &= |[\leftarrow (B_1, \ldots, B_n, A_1, \ldots, A_m)\theta]_I^{n+k}|_j \\
&= \left\{ |A_k\theta\phi|_j \;\middle|\; \begin{array}{l} \phi \text{ is a grounding substitution for } G' \\ I \models \{B_1, \ldots, B_n, A_1, \ldots, A_{k-1}\}\theta\phi \end{array} \right\} \\
&\subseteq \left\{ |A_k\theta\phi|_j \;\middle|\; \begin{array}{l} \phi \text{ is a grounding substitution for } \{H, A_1, \ldots, A_m\}\theta \\ I \models \{H, A_1, \ldots, A_{k-1}\}\theta\phi \end{array} \right\} \\
&= |[\leftarrow (A_0, A_1, \ldots, A_m)\theta]_I^{k+1}|_j \\
&\subseteq |[\leftarrow (A_0, A_1, \ldots, A_m)]_I^{k+1}|_j
\end{aligned}$$

Since G is left bounded wrt $|.|_1$ and I, and wrt $|.|_2$ and I, then $|[G']_I^{n+k}|_j$ is finite for all $k \in [1,m]$, $j \in [1,2]$.

2. *It follows directly that for all $k \in [1,m]$, $j \in [1,2]$, $max|[G']_I^{n+k}|_j \leq max|[G]_I^{k+1}|_j$ and for all $i \in [1,n]$, whenever $rel(A_0) = rel(H) \simeq rel(B_i)$*

$$\begin{aligned}
max|[G']_I^i|_1 &< max\{|H\theta\phi|_1 \mid \phi \text{ is a grounding substitution for } H\theta\} \\
&= max\{|A_0\theta\phi|_1 \mid \phi \text{ is a grounding substitution for } A_0\theta\} \\
&= max|[\leftarrow A_0\theta]_I^1|_1 \\
&\leq max|[\leftarrow A_0]_I^1|_1 \\
&= max|[G]_I^1|_1
\end{aligned}$$

Hence

$$\begin{aligned}
(rel(B_i\theta), max|[G']_I^i|_1) &\prec (rel(A_0), max|[G]_I^1|_1) \qquad \text{for all } i \in [1,n] \\
(rel(A_k\theta), max|[G']_I^{n+k}|_1) &\preceq (rel(A_k), max|[G]_I^{k+1}|_1) \qquad \text{for all } k \in [1,m]
\end{aligned}$$

proving $|[G']_I|_1 \prec_{mul} |[G]_I|_1$.

3. *Since $\prec_{mul}$ is well-founded the result follows immediately.*

Corollary 6.11 Every bounded acceptable program is left terminating.

Theorem 6.12 Let $|.|_1$ and $|.|_2$ be level mappings and I an interpretation for a program P. The following hold.

1. If P is acceptable wrt $|.|_1$ then P is bounded acceptable wrt $|.|_1$ and $|.|_1$.
2. If P is bounded acceptable wrt $|.|_1$ and $|.|_2$, then there exists a level mapping $|.|_3$ such that P is acceptable wrt $|.|_3$. Moreover, for any atom A, A is bounded wrt $|.|_3$ if A is bounded wrt $|.|_1$ and $|.|_2$. □

Proof 7 *Let $c : H \leftarrow B_1, \ldots, B_n$ be a clause in P. Suppose P is acceptable wrt $|.|_1$. Then I is a model for c. Let θ be a substitution such that $H\theta$ is bounded wrt $|.|_1$, $\{B_1, \ldots, B_{i-1}\}\theta$ is ground and $I \models \{B_1, \ldots, B_{i-1}\}\theta$. Then $B_i\theta$ is bounded wrt $|.|_1$ and $|[H\theta]|_1 > |[B_i\theta]|_1$ by acceptability and Lemma 3.14. The second part follows by Lemma 6.10 and theorem 2.2 and corollary 2.2 of Bezem 1993.* □

Example 6.8 Observe that the program of Example 6.5 is bounded acceptable wrt $|.|$, $|.|$ and I where $|.|$ is the original level mapping defined in that example and I is a model of the program. Hence a proof of left termination is obtained which is based solely on the recursive structure of the program. □

6.6 Discussion

The concept of bounded acceptability proposed here is quite similar to that of *rigid acceptability* defined by Decorte & De Schreye 1997. This latter notion forms the basis of a practical, demand-driven termination analysis. The analysis is essentially top-down, attempting to prove termination for a set of queries S. An important step in the analysis is the calculation of the call set $Call(P, S)$, the set of all calls which may occur during the derivation of an atom in S. The analysis focuses on the recursive components to derive a level mapping $|.|$, enforcing boundedness of sub-computations by imposing a rigidity constraint on the call set. That is, during the derivation of $|.|$, every atom in $Call(P, S)$ is required to be rigid wrt $|.|$.

For program specialisation, and partial deduction in particular, it is more useful to derive sufficient termination conditions for individual predicates rather than proving that a given top-level goal will terminate (Bruynooghe *et al.* 1998). The reason is that the overall computation is unlikely to be left-terminating but some sub-computations probably will be. The required conditions can be derived in a bottom-up manner on the strongly connected components of the predicate dependency graph. The notion of bounded acceptability lends itself naturally to this process.

In Decorte & De Schreye 1998, the analysis of Decorte & De Schreye 1997 is adapted to obtain the above mentioned conditions. It attempts to derive for each predicate a maximal set S of left-terminating queries. Essentially, this amounts to deriving a level mapping $|.|$ which defines S, in that an atom A is in S if and only if A is bounded wrt $|.|$. However, an important step is omitted from the paper De Schreye 1998, and the set S may contain queries which are not left-terminating. The level mapping $|.|$ is derived by only considering the recursive components of the program and thus corresponds

to the level mapping $|.|_1$ in the definition of bounded acceptability. Sub-computations are no longer guaranteed to start from bounded goals since no rigidity constraint is placed on the level mapping during its derivation as in Decorte & De Schreye 1997: specifically, this is because the set $Call(P, S)$ is unknown since S is unknown (the idea after all being to derive S), and as a result no rigidity constraint can be imposed on $Call(P, S)$. Hence, in relation to the current work, the missing step is the derivation of the second level mapping $|.|_2$. The maximal set $S' \subseteq S$ of left-terminating queries then, contains only those atoms which are bounded wrt $|.|_1$ and $|.|_2$. Note that $|.|_2$ can be derived entirely independently of $|.|_1$, in the sense that there is never any need to alter the definition of $|.|_1$ in order to obtain a definition of $|.|_2$ which can be used to prove bounded acceptability. Thus the notion of bounded acceptability allows the set S' to be easily constructed from S without requiring any change to the method of Decorte & De Schreye 1998.

In summary, the notions of bounded recurrency and bounded acceptability provide practical criteria for constructing modular termination proofs based purely on the recursive structure of a program.

7 Generating Efficient, Terminating Logic Programs

A logic program can be considered as consisting of a logic component and a control component (Kowalski 1979). Although the meaning of the program is largely defined by its logical specification, choosing the right control mechanism is crucial in obtaining a correct and efficient program. In recent years, one of the most popular ways of defining control is via suspension mechanisms which delay the selection of an atom in a goal until some condition is satisfied. Such mechanisms include the block declarations of SICStus Prolog (SICS 1995) and the DELAY declarations of Gödel (Hill & Lloyd 1994). These mechanisms are used to define dynamic selection rules with the two main aims of enhancing performance through coroutining and ensuring termination. In practice, however, these two aims are not complementary and it is often the case that termination, and hence program correctness, is sacrificed for efficiency.

The objective of control generation in logic programming then, is to automatically derive a computation rule for a program that is efficient and yet does not compromise program correctness. Progress in solving this important problem has been slow and, to date, only partial solutions have been proposed where the generated programs are either incorrect or inefficient. This chapter shows how the control generation problem can be tackled with a simple automatic transformation that relies on information about the depths of SLD-trees.

To prove termination of the transformed programs some theoretical development will be necessary. The main result of this will be the introduction of the new class of *semi delay recurrent* programs (Section 7.2). The intention is that any program lying within this class is terminating with respect to a dynamic selection rule. Furthermore, the notion of a semi delay recurrent program simplifies previous ideas in the termination literature for reasoning about logic programs with delay.

Precedent to this is a discussion in Section 7.1 of the problems that can arise in termination for logic programs with delay. Some of the solutions that have been proposed to resolve these problems, and their short-comings, will be explored.

Section 7.3 presents a formal development of the proposed transformation, including correctness results. In particular, transformed programs are (by construction) semi delay recurrent and hence termination is guaranteed.

7.1 The Problems of Dynamism

The presence of delayed goals in a computation significantly complicates a program's termination behaviour. This section reviews the kind of problems which can arise, the solutions which have been proposed in the past and suggests why there is still room for improvement.

7.1.1 Local Boundedness

Consider the Append program below with its typical DELAY declaration which delays the selection of an Append/3 atom until either the first or third argument is instantiated to a non-variable term.

app_1 Append([], x, x).
app_2 Append([u|x], y, [u|z]) ← Append(x, y, z).

DELAY Append(x, _, z) UNTIL Nonvar(x) ∨ Nonvar(z).

Interestingly, although it is intended to assist termination the delay declaration is not sufficient to ensure that *all* Append/3 goals terminate. The goal ← Append([x|xs], ys, xs), for example, satisfies the condition in the declaration and yet its derivation is an infinite one, where each resolvent is a variant of the previous goal (Naish 1993).

Termination can only be guaranteed for all goals by strengthening the condition in the delay declaration. This is where the trade off between efficiency, termination and deadlock freedom takes place. The stronger the condition, the more goals suspend. Although termination may eventually be assured, it may be at the expense of failing to resolve goals which have finite derivations. Also, the stronger the delay condition, the more time consuming it usually is to check. Thus one of the main problems in generating control of this form is finding suitable conditions which are inexpensive to check and guarantee termination and deadlock freedom. This will be referred to here as the local boundedness issue, since a delay declaration is used to ensure that an atom is bounded in some sense, and this property is dependent solely on the atom itself. This is in contrast to global boundedness where the search tree as a whole is considered.

There have been several attempts at solving the local boundedness problem. Each of these will be examined in the context of the Append program above, though each technique has wider applicability.

7.1.1.1 Linearity

Lüttringhaus-Kappel 1993 observed that, in the case of single literal goals, one additional condition sufficient for termination is that the goal is linear, that is, no variable occurs more than once in the goal. Although this restriction would prevent the looping Append/3 call above from proceeding, it would also unfortunately delay many other goals with finite derivations such as ← Append([x, x], ys, zs).

7.1.1.2 Rigidity and Boundedness

An alternative approach proposed by both Marchiori & Teusink 1995 and Mesnard 1995 delays Append/3 goals until the first or third argument is a list of determinate length (i.e. rigid wrt the list length norm[1]). Termination is obtained for a large class of goals, but at a price. Checking such a condition requires the complete traversal of the

[1]This is equivalent to delaying an Append/3 atom until it is bounded wrt the level mapping $|.|$ defined by $|\text{Append}(t_1, t_2, t_3)| = min(|t_1|_{list\text{-}length}, |t_3|_{list\text{-}length})$. Then, for example, the atom Append([1,2,3], y, z) is bounded since its first argument is rigid. The atom would not be described as rigid, however, since its level could decrease if, for example, z were instantiated to the term [1].

list and the condition must be checked on *every* call to the predicate[2]. Naish argues that this approach can be "... expensive to implement and ... can delay the detection of failure in a sequential system and restrict parallelism in a stream and-parallel system" (Naish 1993).

7.1.1.3 Modes

Naish goes on to solve the problem with the use of modes. Termination can be guaranteed with the above DELAY declaration if the modes of the Append/3 calls are *acyclic*, or more generally *cycle bounded* (Naish 1993). This restriction essentially stops the output feeding back into the input. Although modes form a good basis for solving this problem, they have not been shown to be satisfactory for reasoning about another termination problem, that of speculative output bindings.

7.1.2 Global Boundedness

Even when finite derivations exist, delay conditions alone are not, in general, sufficient to ensure termination. Infinite computations may arise as a result of *speculative output bindings* (Naish 1993), which can occur due to the dynamic selection of atoms. There are several problems associated with speculative output bindings (see Naish 1993 for a discussion of these). The effect that they have on termination is the focus of interest here and will be referred to as the global boundedness issue. To illustrate the problem caused by speculative output bindings consider the Quicksort program shown below. This is a well known program whose termination behaviour can be unsatisfactory. With the given delay declarations, the program can be shown to terminate in forward mode, that is for queries of the form ← Quicksort(x, y) where x is bound and y is uninstantiated. In reverse mode, however, where y is bound and x is uninstantiated, the program does not always terminate. More precisely, a goal such as ← Quicksort(x, [1,2,3]) will terminate *existentially*, i.e. produce a solution, but not *universally*, i.e. produce all solutions. In fact, experimentation with the Gödel and SICStus implementations indicates that when the elements of the list are not strictly increasing, for example in the goals ← Quicksort(x, [1,1]) and ← Quicksort(x, [2,1]), the program does not even existentially terminate! This is illustrative of the subtle problems that dynamic selection rules pose in reasoning about termination, and which suggest that control should ideally be automated to avoid them.

qs_1 Quicksort([], []).
qs_2 Quicksort([x|xs], ys) ←
    Partition(xs, x, l, b) ∧
    Quicksort(l, ls) ∧
    Quicksort(b, bs) ∧
    Append(ls, [x|bs], ys).

DELAY Quicksort(x, y) UNTIL Nonvar(x) ∨ Nonvar(y).

pt_1 Partition([], _, [], []).
pt_2 Partition([x|xs], y, [x|ls], bs) ←
    x ≤ y ∧
    Partition(xs, y, ls, bs).
pt_3 Partition([x|xs], y, ls, [x|bs]) ←

[2]In Mesnard 1995 the check is, in fact, only performed on the initial call, but there is no justification for this optimisation given in the paper. For non-structurally recursive predicates, e.g. Quicksort/2 of Section 7.1.2, such an optimisation would not usually be possible.

x > y ∧
Partition(xs, y, ls, bs).

DELAY Partition(x, _, y, z) UNTIL Nonvar(x) ∨ (Nonvar(y) ∧ Nonvar(z)).

To improve matters, the delay conditions can be strengthened in the manner prescribed by Marchiori and Teusink or by Naish (see Sections 7.1.1.2 and 7.1.1.3). In general, however, no matter how strong the delay conditions are, they are not always sufficient to ensure termination, even though a terminating computation exists. To see why, consider augmenting the Quicksort program with the clause

app_0 Append(x, [_|x], x) ← False.

In what follows, it is assumed that the control strategy tries to execute goals left-to-right by default. The declarative semantics of the program are completely unchanged by the addition of the app_0 clause and one would hope that the new program would produce exactly the same set of answers as the original. This will not be the case, however, if this clause is selected before all other Append/3 clauses. Consider the goal ← Quicksort(x, [1,2,3]). Following resolution with the second clause of Quicksort/2, the only atom which can be selected is Append(ls, [x|bs], [1,2,3]). When this unifies with the above clause, both ls and bs are immediately bound to the term [1,2,3]. As a result of these speculative output bindings the previously suspended calls Quicksort(l, ls) and Quicksort(b, bs) will be woken *before* the computation reaches the call to False. The net result is an infinite computation due to recurring goals of the form ← Quicksort(x, [1,2,3]).

The problem here is that the output bindings are made before it is known that the goal will fail and no matter how stringent the conditions are on the Quicksort/2 goals, loops of this kind cannot generally be avoided. The reason for this is that a delay condition only measures a local property of a goal without regard for the computation as a whole. The conditions can ensure that goals are bounded, but are unable to ensure that the bounds are decreasing.

7.1.2.1 Local Selection Rule

To remedy this, Marchiori & Teusink 1995 propose the use of a *local selection rule*. Such a rule only selects atoms from those that are most recently introduced in a derivation. This ensures that any atom selected from a goal, is completely resolved before any other atom in the goal is selected. The effect in the above example is that the call to False would be selected and the Append/3 goal fully resolved before the calls to Quicksort/2 are woken. This prevents an infinite loop. The main disadvantage of local selection rules is that they do not allow any form of coroutining. This is clearly a very severe restriction.

7.1.2.2 Delayed Output Unification

A similar solution proposed by Naish 1993 is that of delaying output unification. In the example above, assuming a left-to-right computation rule, the clause app_0 would be rewritten as

app_0' Append(x, y, z) ← False ∧ y = [_|x] ∧ z = x.

The intended effect of such a transformation is that no output bindings should be made until the computation is known to succeed. This has parallels with the local selection rule and also restricts coroutining.

7.1.2.3 Constraints

Mesnard uses interargument relationships compiled as constraints to guarantee that the bounds on goals decrease (Mesnard 1995). For example, solving the constraint $|\mathsf{ys}|_{\textit{list-length}} = |\mathsf{ls}|_{\textit{list-length}} + 1 + |\mathsf{bs}|_{\textit{list-length}}$ before selecting the atom Append(ls, [x|bs], ys) ensures that bs and ls are only bound to lists with lengths less than that of ys. This is enough to guarantee termination, but is expensive to check as it requires calculating the lengths of all three arguments of Append/3.

7.1.3 Summary and Contribution

The most promising approaches to control generation, while guaranteeing termination and completeness, produce programs which are inefficient, either directly due to expensive checks which must be performed at run-time or indirectly by restricting coroutining.

This thesis presents an elegant solution to the above problems. To solve the local boundedness problem, delay declarations in the spirit of Marchiori & Teusink 1995 will be used to ensure boundedness of selected atoms. This will require rigidity checks to be performed on arguments, but a novel program transformation will be introduced to overcome the inefficiencies of the Marchiori and Teusink approach which were discussed in Section 7.1.1.2. Simultaneously, the transformation inexpensively solves the global boundedness problem *without* grossly restricting coroutining. The transformation is simple and is easy to automate. Transformed programs are guaranteed to terminate and are also efficient.

The technique is based on the following idea. If the maximum depth of the SLD-tree needed to solve a given goal can be determined, then by only searching to that depth, the goal will be completely solved, i.e. *all* answers (if any) will be obtained, in a finite number of steps.

Section 7.2 develops the necessary theoretical foundations on which the transfomation will be based, while the transformation itself is described in Section 7.3. The following subsection illustrates the essential ideas behind the approach through a concrete example.

7.1.4 Example

The Quicksort program of Section 7.1.2 can be transformed into a version where termination is guaranteed for all goals. Furthermore for a goal of the form $\leftarrow$ Quicksort(x, y) where x or y is a ground list of integers, the computation does not flounder and if it succeeds then the set of answers produced is complete with respect to the declarative semantics. The transformed program is shown below.

```
Quicksort(x, y) ←
    SetDepth_Q(x, y, d) ∧
    Quicksort_1(x, y, d).

DELAY Quicksort_1(_, _, d) UNTIL Ground(d).

Quicksort_1([], [], d) ← d ≥ 0.
Quicksort_1([x|xs], ys, d) ← d ≥ 0 ∧
    Partition(xs, x, l, b) ∧
    Quicksort_1(l, ls, d − 1) ∧
    Quicksort_1(b, bs, d − 1) ∧
```

```
    Append(ls, [x|bs], ys).

Partition(xs, x, l, b) ←
    SetDepth_P(xs, l, b, d) ∧
    Partition_1(xs, x, l, b, d).

DELAY Partition_1(_, _, _, _, d) UNTIL Ground(d).

Partition_1([], _, [], [], d) ← d ≥ 0.
Partition_1([x|xs], y, [x|ls], bs, d) ← d ≥ 0 ∧
    x ≤ y ∧
    Partition_1(xs, y, ls, bs, d − 1).
Partition_1([x|xs], y, ls, [x|bs], d) ← d ≥ 0 ∧
    x > y ∧
    Partition_1(xs, y, ls, bs, d − 1).

Append(x, y, z) ←
    SetDepth_A(x, z, d) ∧
    Append_1(x, y, z, d).

DELAY Append_1(_, _, _, d) UNTIL Ground(d).

Append_1([], x, x, d) ← d ≥ 0.
Append_1([u|x], y, [u|z], d) ← d ≥ 0 ∧
    Append_1(x, y, z, d − 1).
```

The predicate SetDepth_Q(x, y, d) calculates the lengths of the lists x and y, delaying until one of the lists is found to be of determinate length, at which point the variable d is instantiated to this length. Only then can the call to Quicksort_1/3 proceed. The purpose of this last argument is to ensure finiteness of the subsequent computation. More precisely, d is an upper bound on the number of calls to the recursive clause of Quicksort_1/3 *in any successful derivation.* Thus by failing any derivation where the number of such calls has exceeded this bound (using the test $d \geq 0$), termination is guaranteed without losing completeness. The predicates SetDepth_P/4 and SetDepth_A/3 are defined in a similar way.

7.1.4.1 Local and Global Boundedness

The local boundedness problem is solved in the first instance with a rigidity check in the style of Marchiori & Teusink 1995. This ensures that the initial goal is bounded. Boundedness of subsequent goals, however, is enforced by the depth parameter and further rigidity checks on these depth bounded goals are redundant. This allows, for example, the call Quicksort_1(l, ls, d − 1) to proceed, without fear of an infinite computation, even if both l and ls are uninstantiated, providing d is ground. A huge improvement in performance is possible by eliminating these checks. The global boundedness problem is also neatly solved. By restricting the search space to be finite, even though speculative output bindings may still occur, they cannot lead to infinite derivations.

7.1.4.2 A Simple Optimisation

Even though many of the rigidity checks have now been removed, the efficiency of the program is still unsatisfactory. This is due to the rigidity checks which are performed on each call to Append/3 and Partition/4. It is easy to show that the depths of these subcomputations are bounded by the same depth parameter occurring in Quicksort_1/3. Hence, the atoms Partition(xs, x, l, b) and Append(ls, [x|bs], ys) in the body of Quicksort_1/3 can be replaced respectively by Partition_1(xs, x, l, b, d − 1) and Append_1(ls, [x|bs], ys, d − 1).

Another, more minor, optimisation can be performed to reduce the effect of the delays on the program. Observe that according to the delay declaration for the predicate Quicksort_1/3, the third argument is tested for groundness every time the predicate is called. However, this test is unnecessary for every call except the first, since once instantiated, the depth parameter will always be ground on each recursive call. The delay can be factored out of the loop by introducing an auxilliary predicate with the following definition.

```
DELAY Quicksort_2(_, _, d) UNTIL Ground(d).

Quicksort_2(x, y, d) ←
    Quicksort_1(x, y, d).
```

With the introduction of this predicate, the call to Quicksort_1/3 in the body of Quicksort/3 is then replaced by a call (with the same arguments) to Quicksort_2/3. The delay declaration for Quicksort_1/3 can then be removed avoiding redundant groundness checks. This same optimisation can also be applied to the Partition/4 and Append/3 predicates (although if the first optimisation described above is performed, this last step is unnecessary since the Partition/4 and Append/3 predicates will never be called).

The version of the program incorporating these optimisations is quite efficient. The only rigidity checks that are performed are those on the initial input, exactly at the point where they are needed to guarantee termination. Following the initial call to Quicksort_2/3 the program runs completely without delays and the only other overhead is the decrementation of the depth parameters and some trivial boundedness checks on them. The net result is that, with the Bristol Gödel implementation, the program actually runs faster on average than the original program with the Nonvar delay declarations!

7.1.4.3 Coroutining

Notice in particular how the global boundedness problem is overcome without reducing the potential for coroutining. Simply knowing the maximum depth of any potentially successful branch of the SLD-tree allows one to force any derivations along this branch which extend beyond this depth to fail without losing completeness. These forced failures keep the computation tree finite but do not restrict the way in which the tree is searched. The addition of the failing Append/3 clause app_0 from Section 7.1.2 (which would appear here as an Append_1/4 clause) cannot affect the termination of the algorithm, even if the same coroutining behaviour of the original program is used. Of course, the computation rule needs to be restrained such that

1. the test d ≥ 0 is always selected before any other atom in the body of the clause with a subterm d, and

2. the depth parameter is ground for each recursive call (or for any call with a subterm d in the optimised version)

but this is not nearly as restrictive as using the local computation rule. Indeed, using the default left-to-right selection rule (with delay) these conditions will clearly be satisfied in the above program.

7.1.4.4 Termination *and* Efficiency

With termination guaranteed, the programmer is now free to concentrate on the program's performance. Notice for the program above that the order of the goals in the body of Quicksort_1 is critical to the efficiency of the algorithm. For the best performance, they must be arranged so that the computation is data driven. In fact, by defining SetDepth_Q/3 by

```
SetDepth_Q(x, y, d) ←
    Length(x, d) ∧
    Length(y, d).
```

the computation will be data driven in both forward and reverse modes with the ordering of the goals as above. This dependence on the ordering can be reduced by introducing the typical delay declarations used for this program. These declarations do *not* effect the terminating nature of the algorithm, in that they will not cause the algorithm to loop, though they may possibly reduce previously successful or failing derivations to floundering ones. They are inserted solely to improve the performance through coroutining. Alternatively, one may seek to optimise the performance for different modes through multiple specialisation, for example. The important point is that with the general approach described here the trade-off between termination and performance is significantly reduced. In seeking an efficient algorithm, correctness does not have to be compromised.

7.2 Theoretical Foundations

To provide a sound theoretical basis for termination of delay logic programs it is natural to build on the preceding theoretical foundations established for conventional logic programs. This was initiated with the work of Marchiori & Teusink 1995 on which this section further builds.

The intention is to introduce a new program class which subsumes that of delay recurrent programs introduced in Marchiori & Teusink 1995. Its introduction is motivated by an overly restrictive condition imposed in the definition of delay recurrency. By removing this unnecessary condition the new class of *semi delay recurrent* programs will be obtained.

7.2.1 Atom Selection

In all of the level mapping based approaches to termination examined so far a fundamental requirement is that only bounded atoms are selected. The reason is that, in general, when unbounded atoms are selected for resolution, it is extremely difficult to reason about the termination of the subsequent computation. The principle can still be applied when considering flexible computation rules. Moreover, delay declarations provide a mechanism to control this directly by delaying atoms until they become

bounded. This idea was encountered in Sections 7.1.1.2 and 7.1.2, and is formally captured in the following definition.

Definition 7.1 (safe delay declaration Marchiori & Teusink 1995) A delay declaration for a predicate p is *safe* wrt a level mapping $|.|$ if for every atom A with predicate symbol p, if A satisfies its delay declaration, then A is bounded wrt $|.|$. □

7.2.2 Covers

To determine whether or not an atom is bounded when it is selected requires a consideration of the atoms that have been (partially) resolved before the selection of the atom. The following definitions proposed by Marchiori & Teusink 1995 try to capture this notion.

Definition 7.2 (direct cover Marchiori & Teusink 1995) Let $c : H \leftarrow B_1, \ldots, B_n$ be a clause and $|.|$ a level mapping. Let $A \in body(c)$ and $D \subset body(c)$ such that $A \notin D$. Then D is a *direct cover* for A wrt $|.|$ in c, if there exists a substitution θ such that

1. $A\theta$ is bounded wrt $|.|$, and
2. $dom(\theta) \subseteq vars(H) \cup vars(D)$.

A direct cover D for A is minimal if no proper subset of D is a direct cover for A. The set of minimal direct covers of A wrt $|.|$ in c is denoted by $mdcovers_{|.|,c}(A)$. □

Intuitively, a direct cover of an atom A in a clause c is a subset D of the body atoms of c such that for some instantiation θ of the variables in the head of c and in D, $A\theta$ is bounded. Note that a body atom may have zero, one or more (mininal) direct covers. In particular, an atom A will have no direct cover when, in order for A to become bounded, it is necessary to instantiate a variable of A which does not occur elsewhere in the clause. On the other hand, the atom A will have the empty set as its only minimal direct cover if A is bounded whenever the head of the clause is bounded.

Example 7.1 Consider the program Quicksort and the level mapping $|.|$ defined by

$$|\mathsf{Qsort}(x, y)| = y' + 1 \qquad |\mathsf{Partition}(w, x, y, z)| = y' + z' \qquad |\mathsf{Append}(x, y, z)| = z'$$

where $y' = |y|_{\textit{list-length}}$ and $z' = |z|_{\textit{list-length}}$. Then

$$\begin{aligned} mdcovers_{|.|,qs_2}(\mathsf{Partition(xs, x, l, b)}) &= \{\{\mathsf{Qsort(l, ls)}, \mathsf{Qsort(b, bs)}\}\} \\ mdcovers_{|.|,qs_2}(\mathsf{Qsort(l, ls)}) &= \{\{\mathsf{Append(ls, [x|bs], ys)}\}\} \\ mdcovers_{|.|,qs_2}(\mathsf{Qsort(b, bs)}) &= \{\{\mathsf{Append(ls, [x|bs], ys)}\}\} \\ mdcovers_{|.|,qs_2}(\mathsf{Append(ls, [x|bs], ys)}) &= \{\emptyset\} \end{aligned}$$

Note in this example that each atom has exactly one minimal direct cover. □

Definition 7.3 (cover Marchiori & Teusink 1995) Let $c : H \leftarrow B_1, \ldots, B_n$ be a clause and $|.|$ a level mapping. Let $A \in body(c)$ and $C \subset body(c)$ such that $A \notin C$. Then C is a *cover* for A wrt $|.|$ in c, if (A, C) is an element of the least set S such that

1. $(A, \emptyset) \in S$ whenever the empty set is the minimal direct cover for A wrt $|.|$ in c

2. $(A, C) \in S$ whenever $A \notin C$, and C is of the form

$$\{A_1, .., A_k\} \cup C_1 \cup \ldots \cup C_k$$

such that $\{A_1, \ldots, A_k\}$ is a minimal direct cover for A wrt $|.|$ in c and for all $i \in [1, k]$, we have $(A_i, C_i) \in S$.

The set of covers of A wrt $|.|$ in c is denoted by $covers_{|.|,c}(A)$. □

Intuitively, a cover of an atom A in a clause c is a subset of the body atoms which must be (partially) resolved in order for A to become bounded wrt some level mapping in c. The cover relation is a kind of closure of the direct cover relation but not a transitive one; a direct cover of an atom is not necessarily a cover of that atom. Observe that, if an atom has no minimal direct cover, then neither does it have a cover.

Example 7.2 Consider the program Quicksort and the level mapping $|.|$ of Example 7.1. Then

$$\begin{aligned}
covers_{|.|,qs_2}(\mathsf{Partition(xs, x, l, b)}) &= \{\{\mathsf{Qsort(l, ls), Qsort(b, bs), Append(ls, [x|bs], ys)}\}\} \\
covers_{|.|,qs_2}(\mathsf{Qsort(l, ls)}) &= \{\{\mathsf{Append(ls, [x|bs], ys)}\}\} \\
covers_{|.|,qs_2}(\mathsf{Qsort(b, bs)}) &= \{\{\mathsf{Append(ls, [x|bs], ys)}\}\} \\
covers_{|.|,qs_2}(\mathsf{Append(ls, [x|bs], ys)}) &= \{\emptyset\}
\end{aligned}$$

Observe then, that each body atom in qs_2 has exactly one cover wrt $|.|$. □

7.2.3 Delay Recurrency

Using the notion of cover, Marchiori & Teusink 1995 introduced the class of delay recurrent programs. It was intended that programs lying within this class would be terminating under a dynamic selection rule.

Definition 7.4 (delay recurrency Marchiori & Teusink 1995)[3] Let P be a program, $|.|$ a level mapping and I an interpretation for P. A clause $c : H \leftarrow B_1, \ldots, B_n$ is *delay recurrent* wrt $|.|$ and I iff

1. I is a model for c and
2. for all $i \in [1, n]$, for every cover C for B_i and for every grounding substitution θ for c such that $I \models C\theta$, we have that $|H\theta| > |B_i\theta|$.

A program P is delay recurrent wrt $|.|$ and I iff every clause of P is delay recurrent wrt $|.|$ and I. □

Example 7.3 Let $|.|$ be the level mapping of Example 7.1 and I the interpretation

$$\begin{aligned}
\{\mathsf{Qsort}(x, y) \quad &| \quad |x|_{\textit{list-length}} = |y|_{\textit{list-length}}\} \cup \\
\{\mathsf{Partition}(x, w, y, z) \quad &| \quad |x|_{\textit{list-length}} = |y|_{\textit{list-length}} + |z|_{\textit{list-length}}\} \cup \\
\{\mathsf{Append}(x, y, z) \quad &| \quad |z|_{\textit{list-length}} = |x|_{\textit{list-length}} + |y|_{\textit{list-length}}\}
\end{aligned}$$

[3]The definition of delay recurrency in Marchiori & Teusink 1995 contains some slight redundancy/ambiguity and as such its correct interpretation is unclear. This definition accurately reflects the intentions of the authors (Marchiori 1996).

Note that I is a model for the clause qs_2 of the Quicksort program. Consider the body atom Partition(xs, x, l, b). Recall that

$$covers_{|.|,qs_2}(\text{Partition(xs, x, l, b)}) = \{\{\text{Qsort(l, ls)}, \text{Qsort(b, bs)}, \text{Append(ls, [x|bs], ys)}\}\}$$

Let $\theta = \{\text{x}/t_1, \text{xs}/t_2, \text{ys}/t_3, \text{l}/t_4, \text{b}/t_5, \text{ls}/t_6, \text{bs}/t_7\}$ be a grounding substitution for qs_2 such that $I \models \{\text{Qsort}(t_4, t_6), \text{Qsort}(t_5, t_7), \text{Append}(t_6, [t_1|t_7], t_3)\}$. Then

$$\begin{aligned}
|\text{Qsort}([t_1|t_2], t_3)| &= |t_3|_{\textit{list-length}} + 1 \\
&= (|t_6|_{\textit{list-length}} + |t_7|_{\textit{list-length}} + 1) + 1 \\
&= (|t_4|_{\textit{list-length}} + |t_5|_{\textit{list-length}} + 1) + 1 \\
&> |t_4|_{\textit{list-length}} + |t_5|_{\textit{list-length}} \\
&= |\text{Partition}(t_2, t_1, t_4, t_5)|
\end{aligned}$$

It is easy to check that condition 2 of Definition 7.4 holds for every other body atom of qs_2. Hence qs_2 is delay recurrent wrt $|.|$ and I. □

The intention behind the definition of delay recurrency is that a delay recurrent program P, when augmented with a set of safe delay declarations for the predicates of P, only admits finite derivations. The delay declarations handle the local boundedness issue, but there is still the global boundedness problem to consider.

Suppose C is a cover for an atom B in a delay recurrent clause, and θ is an answer substitution for C such that $B\theta$ is bounded (note that θ may not necessarily be a *correct* answer substitution since the atoms in C have not yet been fully resolved). At this point θ speculatively binds the variables of B since it is not yet know whether or not there exists some substitution σ such that $I \models C\theta\sigma$. If $B\theta$ is selected at this point an infinite computation may arise since there is no guarantee that the level of the head is greater than the level of $B\theta$. Instead, by fully resolving each atom in C such that a correct answer substitution θ is obtained, $B\theta$ can be safely selected since $I \models C\theta\sigma$ for all σ, whence by condition 2 of delay recurrency, the level of $B\theta$ is less than the level of the head. Full resolution of C can be achieved by using a local selection rule as mentioned in Section 7.1.2.1. To reiterate, a local selection rule only selects the most recently introduced atoms in a derivation and thus completely resolves subcomputations before proceeding with the main computation. The notion is formally defined below.

Definition 7.5 (age of an atom) For a goal $G = \leftarrow A_1, \ldots, A_m$ the ith atom in G is A_i. Let $G_0, \ldots, G_n$ be a derivation. The *age* of the ith atom in G_k, denoted $age_{G_k}(i)$ is defined as follows.

1. If $G_0 = \leftarrow A_1, \ldots, A_m$, then $age_{G_0}(i) = 0$, for all $i \in [1, m]$.
2. If $G_k = \leftarrow A_1, \ldots, A_m$ and $G_{k+1} = \leftarrow (A_1, \ldots, A_{s-1}, B_1, \ldots, B_n, A_{s+1}, \ldots, A_m)\theta$, then

$$age_{G_{k+1}}(i) = \begin{cases} age_{G_k}(i) + 1, & \text{for all } i \in [1, s-1] \\ 0, & \text{for all } i \in [s, s+n-1] \\ age_{G_k}(i-n+1) + 1, & \text{for all } i \in [s+n, n+m-1] \end{cases}$$

For a goal $G = \leftarrow A_1, \ldots, A_m$, the atom A_i is *introduced* in G if $age_G(i) = 0$. The atom A_i is *youngest* (or *most recently introduced*) in G if $age_G(i) \leq age_G(j)$ for all $j \in [1, m]$. □

Definition 7.6 (local selection rule Vieille 1989) Let $G = \leftarrow A_1, \ldots, A_m$ be a goal. An atom A_i is selectable in G under a local selection rule iff A_i is most recently introduced in G. □

The main result regarding delay recurrent programs can now be stated.

Theorem 7.7 (delay recurrency Marchiori & Teusink 1995) Let P be a program with a delay declaration for each predicate in P. Let $|.|$ be a level mapping and I an interpretation. Suppose that

1. P is delay recurrent wrt $|.|$ and I, and
2. the delay declarations for P are safe wrt $|.|$

Then every delay SLD-derivation for a goal, using a local selection rule is finite. □

7.2.4 Semi Delay Recurrency

Marchiori & Teusink 1995 noticed that boundedness of atoms could be enforced by using safe delay declarations but did not fully exploit this fact combined with the observations of Chapter 6 in defining delay recurrency. Their definition requires a decrease in the level mapping from the head to the non-recursive body atoms when in fact boundedness of selected atoms is already guaranteed by the safe delay declarations. Their definition is generalised here by removing this restriction. The new definition will prove useful for defining a large class of terminating programs which permit coroutining.

Definition 7.8 (semi delay recurrency) Let $|.|$ be a level mapping and I an interpretation for a program P. A clause $c : H \leftarrow B_1, \ldots, B_n$ is *semi delay recurrent* wrt $|.|$ and I iff

1. I is a model for c and
2. for all $i \in [1, n]$ such that $rel(H) \simeq rel(B_i)$, for every cover C for B_i and for every grounding substitution θ for c such that $I \models C\theta$, we have that $|H\theta| > |B_i\theta|$.

A program P is semi delay recurrent wrt $|.|$ and I iff every clause of P is semi delay recurrent wrt $|.|$ and I. □

Observe in this definition that there are no restrictions placed on the relation between the level of the head of the clause and the level of the non-recursive body atoms.

Example 7.4 Let I be the interpretation of Example 7.3, and $|.|$ the level mapping defined by

$$|\mathsf{Qsort}(x, y)| = y' \qquad |\mathsf{Partition}(w, x, y, z)| = y' + z' \qquad |\mathsf{Append}(x, y, z)| = z'$$

where $y' = |y|_{\textit{list-length}}$ and $z' = |z|_{\textit{list-length}}$. As before, I is a model for the clause qs_2 of the Quicksort program. Consider the body atom Qsort(b, bs). Then

$$\begin{aligned} \textit{covers}_{|.|,qs_2}(\mathsf{Qsort(l, ls)}) &= \{\{\mathsf{Append(ls, [x|bs], ys)}\}\} \\ \textit{covers}_{|.|,qs_2}(\mathsf{Qsort(b, bs)}) &= \{\{\mathsf{Append(ls, [x|bs], ys)}\}\} \end{aligned}$$

Let $\theta = \{\mathsf{x}/t_1, \mathsf{xs}/t_2, \mathsf{ys}/t_3, \mathsf{l}/t_4, \mathsf{b}/t_5, \mathsf{ls}/t_6, \mathsf{bs}/t_7\}$ be a grounding substitution for qs_2 such that $I \models \{\mathsf{Append}(t_6, [t_1|t_7], t_3)\}$. Then

$$\begin{aligned} |\mathsf{Qsort}([t_1|t_2], t_3)| &= |t_3|_{\textit{list-length}} \\ &= (|t_6|_{\textit{list-length}} + |t_7|_{\textit{list-length}} + 1) \\ &> |t_7|_{\textit{list-length}} \\ &= |\mathsf{Qsort}(t_5, t_7)| \end{aligned}$$

$$\begin{aligned} |\mathsf{Qsort}([t_1|t_2], t_3)| &> |t_6|_{\textit{list-length}} \\ &= |\mathsf{Qsort}(t_4, t_6)| \end{aligned}$$

Hence qs_2 is semi delay recurrent wrt $|.|$ and I. □

The relationship between delay recurrency and semi delay recurrency is unclear. Obviously every delay recurrent program is semi delay recurrent, but the converse may or may not be true. In Martin & King 1997 the following program was said to be semi delay recurrent but not delay recurrent.

P([x|y]) ← Append(_, _, _) ∧ P(y).

The reasoning was based upon interpreting the definition of delay recurrency in Marchiori & Teusink 1995 such that, for a body atom which has no cover, the decrease in level from the head of the clause to the atom must hold for all ground instances. Since the Append(_, _, _) atom has no cover and the decrease does not hold, the clause cannot be delay recurrent (Marchiori 1996). However, with the interpretation of delay recurrency given by Definition 7.4 the second condition of the definition is vacuously satisfied for an atom with no cover. As such with this definition, the above program is considered to be delay recurrent. This does not endanger termination. Observe that if an atom has no cover then it can never become bounded. If all delay declarations are safe then such atoms will never be selected. Indeed one necessary condition for deadlock freedom of a program is that every atom has at least one cover (Marchiori & Teusink 1996). It would be an interesting result, theoretically, if delay recurrency and semi delay recurrency were shown to be equivalent, but this is not considered any further here.

It would be straightforward to prove that Theorem 7.7 still holds if the program is replaced by one which is semi delay recurrent, but a much more significant result may be obtained. Observe that a local selection rule is used to ensure that a cover of an atom is completely resolved before the atom itself is selected. Notice, however, that for semi delay recurrency, it is only necessary for the covers of the mutually recursive atoms to be resolved completely. This means that following the resolution of these covers, an arbitrary amount of coroutining may take place amongst the remaining atoms of the clause.

To formalise a selection rule based on this idea the notion of a *covering* is introduced. Intuitively, this is a lifting of the notion of cover from the clause level to the goal level. A covering of a recursive atom A in a goal G is the set of atoms in G which have yet to be resolved before A can be safely selected. An atom A may have more than one covering, though it is only necessary to fully resolve the atoms of one of them before the selection of A. Coverings of atoms will change during the course of a derivation as new atoms are introduced and others are fully resolved via resolution.

Definition 7.9 (covering) Let $G_0, G_1, G_2, \ldots$ be a derivation and $|.|$ a level mapping. A *covering* for an atom A in a goal G_k wrt $|.|$ is defined as follows.

1. Suppose $G_0 = \leftarrow A_1, \ldots, A_m$. Then for all $i \in [1, m]$, the empty set is a covering for A_i in G_0 wrt $|.|$.
2. Suppose $G_{l+1} = \leftarrow (A_1, \ldots, A_{s-1}, B_1, \ldots, B_n, A_{s+1}, \ldots, A_m)\theta$ is the resolvent derived from $G_l = \leftarrow A_1, \ldots, A_s, \ldots, A_m$ and $c : H \leftarrow B_1, \ldots, B_n$, where A_s is the selected atom in G_l and $\theta \in mgu(H, A_s)$. Then
   (a) for all $i \in [1, n]$,
   - if $rel(H) \simeq rel(B_i)$ and C is a cover for B_i in c wrt $|.|$, then $C\theta$ is a covering for $B_i\theta$ in G_{l+1} wrt $|.|$;
   - if $rel(H) \not\simeq rel(B_i)$, then the empty set is a covering for $B_i\theta$ in G_{l+1} wrt $|.|$;

   (b) for all $i \in [1, m]$, $i \neq s$, if $C \subseteq \{A_1, \ldots, A_{i-1}, A_{i+1}, \ldots, A_m\}$ is a covering for A_i in G wrt $|.|$ then
   - $C\theta \backslash \{A_s\theta\} \cup \{B_1, \ldots, B_n\}\theta$ is a covering for $A_i\theta$ in G_{l+1} wrt $|.|$, if $A_s \in C$;
   - $C\theta$ is a covering for $A_i\theta$ in G_{l+1} wrt $|.|$, if $A_s \notin C$.

An occurence of an atom A is *uncovered* in a goal G wrt $|.|$ iff the empty set is a covering for A in G wrt $|.|$. □

Definition 7.10 (semi local selection rule) Let $\delta = G_0, G_1, G_2, \ldots$ be a derivation and $|.|$ a level mapping. Let $G_k = \leftarrow A_1, \ldots, A_m$ be a goal in δ. An atom A_s $(1 \leq s \leq m)$ is selectable in G_k under a semi local selection rule (parameterised by $|.|$) iff A_s is uncovered wrt $|.|$ in G_k. □

It can be shown that if a bounded atom is selectable under a local selection rule, then it is selectable under a semi local selection rule.

The main result can now be stated.

Theorem 7.11 Let P be a program with a delay declaration for each predicate in P. Let $|.|$ be a level mapping and I an interpretation. Suppose that

1. P is semi delay recurrent wrt $|.|$ and I
2. The delay declarations for P are safe wrt $|.|$

Then every delay SLD-derivation for a goal, using a semi-local selection rule (parameterised by $|.|$) is finite. □

Proof 8 *Follows as a corollary of Lemma 7.13.*

Definition 7.12 Let $|.|$ be a level mapping, I an interpretation and $G = \leftarrow A_1, \ldots, A_n$ a goal. Define for all $i \in [1, n]$,

$$|[G]_I^i| = \left\{ |A_i\theta| + 1 \;\middle|\; \begin{array}{l} \theta \text{ is a grounding substitution for } G \\ C \text{ is a covering for } A_i \text{ in } G \text{ wrt } |.| \\ I \models C\theta \end{array} \right\}$$

Then the finite multiset of pairs $\{(rel(A_1), max|[G]_I^1|), \ldots, (rel(A_n), max|[G]_I^n|)\}$ is denoted by $|[G]_I|$. □

Observe that if the atom A_i has no covering in the goal $G = \leftarrow A_1, \ldots, A_n$ then $|[G]_I^i| = \emptyset$. Also, if for every covering C for A_i and for every grounding substitution θ for G, $I \not\models C\theta$, then $|[G]_I^i| = \emptyset$. Note that the expression $|A_i\theta| + 1$ in the definition of $|[G]_I^i|$ ensures that $max|[G]_I^i| > 0$ when $|[G]_I^i|$ is non-empty. That is, $max|[G]_I^i|$ can only be zero if $|[G]_I^i| = \emptyset$. This device is used purely to facilitate the proof of Lemma 7.13.

Let $\prec$ be the lexicographical ordering on $\Sigma_p(\sqsubset) \times \mathbf{N}(<)$ and let $\prec_{mul}$ be the multiset ordering based on $\prec$. Observe that $\prec_{mul}$ is well founded.

Lemma 7.13 Let $|.|$ be a level mapping and I an interpretation for a program P. Let P be semi delay recurrent wrt $|.|$ and I, and let G be a goal. Let G' be an SLD-resolvent of G from P. Then $|[G']_I| \prec_{mul} |[G]_I|$.

Proof 9 *Suppose* $G' = \leftarrow (A_1, \ldots, A_{s-1}, B_1, \ldots, B_n, A_{s+1}, \ldots, A_m)\theta$ *is the SLD-resolvent derived from* $G = \leftarrow A_1, \ldots, A_s, \ldots, A_m$ *and* $c : H \leftarrow B_1, \ldots, B_n$, *where* A_s *is the selected atom in* G *and* $\theta \in mgu(H, A_s)$. *First show that for all* $i \in [1, n]$

$$(rel(B_i\theta), max|[G']_I^{s+i-1}|) \prec (rel(A_s), max|[G]_I^s|)$$

Clearly this holds for all $i \in [1, n]$ *such that* $rel(B_i) \not\simeq rel(H) \simeq rel(A_s)$. *It remains to show that the inequality holds for all* $i \in [1, n]$ *such that* $rel(B_i) \simeq rel(H)$. *By Definition 7.10, the empty set is a covering for* A_s *in* G *wrt* $|.|$. *Then*

$$max|[G]_I^s| = max\{|A_s\phi| + 1 \mid \phi \text{ is a grounding substitution for } G\}$$

Since A_s *is bounded (only bounded atoms may be selected), it follows that* $max|[G]_I^s| = |[A_s]| + 1$. *By Lemma 3.13,* $|[A_s]| \geq |[A_s\theta]| = |[H\theta]|$. *Thus* $max|[G]_I^s| > |[H\theta]|$. *Now, for all* $i \in [1, n]$,

$$\begin{aligned} |[G']_I^{s+i-1}| &= |[\leftarrow (A_1, \ldots, A_{s-1}, B_1, \ldots, B_n, A_{s+1}, \ldots, A_m)\theta]_I^{s+i-1}| \\ &= \left\{ |B_i\theta\phi| + 1 \;\middle|\; \begin{array}{l} \phi \text{ is a grounding substitution for } G' \\ D \text{ is a covering for } B_i\theta \text{ in } G' \\ I \models D\phi \end{array} \right\} \end{aligned}$$

Let $i \in [1, n]$ *such that* $rel(B_i) \simeq rel(H)$ *and* $|[G']_I^{s+i-1}| \neq \emptyset$. *Then there exists*

1. *a grounding substitution* ϕ *for* G', *and*
2. *a covering* D *for* $B_i\theta$ *in* G' *wrt* $|.|$,

such that

3. $I \models D\phi$, *and*
4. $|B_i\theta\phi| + 1 = max|[G']_I^{s+i-1}|$.

From (2) above it follows that there exists a cover C *for* B_i *in* c *such that* $D = C\theta$, *and hence by (3),* $I \models C\theta\phi$. *By (1),* $\theta\phi$ *is a grounding substitution for* $\leftarrow B_1, \ldots, B_n$. *Let* σ *be a grounding substitution for* $H\theta\phi$. *Then* $\theta\phi\sigma$ *is a grounding substitution for* c, *such that* $I \models C\theta\phi\sigma$ *(since* $C\theta\phi\sigma = C\theta\phi$*). Hence, by semi delay recurrency of* c, $|H\theta\phi\sigma| > |B_i\theta\phi\sigma| = |B_i\theta\phi|$. *Thus* $max|[G']_I^{s+i-1}| \leq |H\theta\phi\sigma| = |[H\theta\phi\sigma]| \leq |[H\theta]|$, *by Lemma 3.13. It follows that for all* $i \in [1, n]$ *such that* $rel(B_i) \simeq rel(H)$ *and* $|[G']_I^{s+i-1}| \neq \emptyset$, $max|[G]_I^s| > max|[G']_I^{s+i-1}|$.

Let $i \in [1,n]$ such that $rel(B_i) \simeq rel(H)$ and $|[G']_I^{s+i-1}| = \emptyset$. Then $max|[G']_I^{s+i-1}| = 0$. But $max|[G]_I^s| > 0$ so $max|[G]_I^s| > max|[G']_I^{s+i-1}|$. It follows that for all $i \in [1,n]$ such that $rel(B_i) \simeq rel(H)$, $max|[G]_I^s| > max|[G']_I^{s+i-1}|$. This completes the first part of the proof.

Next show that for all $i \in [1,m]$, $i \neq s$,

$$(rel(A_i\theta), max|[G']_I^i|) \preceq (rel(A_i), max|[G]_I^i|)$$

Let $i \in [1,m]$, $i \neq s$, such that $|[G']_I^i| \neq \emptyset$. Then there exists

1. *a grounding substitution ϕ for G', and*
2. *a covering D for $A_i\theta$ in G' wrt $|.|$,*

such that

3. *$I \models D\phi$, and*
4. *$|A_i\theta\phi| + 1 = max|[G']_I^i|$.*

From (2) above it follows that there exists a covering $C \subseteq \{A_1, \ldots, A_{i-1}, A_{i+1}, \ldots, A_m\}$ for A_i in G wrt $|.|$ such that

- *$A_s \in C$ and $D = C\theta \setminus \{A_s\theta\} \cup \{B_1, \ldots, B_n\}\theta$, or*
- *$A_s \notin C$ and $D = C\theta$.*

In the first case, by (3), $I \models (C\theta \setminus \{A_s\theta\} \cup \{B_1, \ldots, B_n\}\theta)\phi$ and consequently, since I is a model for c, $I \models (C\theta \setminus \{A_s\theta\} \cup H\theta)\phi$. But then $I \models C\theta\phi$ because $H\theta = A_s\theta$. In the second case, by (3), $I \models C\theta\phi$.

By (1), $\theta\phi$ is a grounding substitution for $\leftarrow A_1, \ldots, A_{s-1}, A_{s+1}, \ldots, A_m$. Let σ be a grounding substitution for $A_s\theta\phi$. Then $\theta\phi\sigma$ is a grounding substitution for G, such that $I \models C\theta\phi\sigma$ (since $C\theta\phi\sigma = C\theta\phi$). Hence $|A_i\theta\phi\sigma| + 1 \in |[G]_I^i|$ and as a result $max|[G]_I^i| \geq max|[G']_I^i|$.

Let $i \in [1,m]$, $i \neq s$, such that $|[G']_I^i| = \emptyset$. Then $max|[G']_I^i| = 0 \leq max|[G]_I^i|$. It follows that $max|[G]_I^i| \geq max|[G']_I^i|$ for all $i \in [1,m]$, $i \neq s$, completing the proof. □

7.3 The Transformation

The above result can be used to develop a program transformation which is able to derive correct and efficient programs from logical specifications. Basically, the idea is to transform a given program into one which is semi delay recurrent, but with equivalent declarative semantics. Then by adding safe delay declarations a program is obtained which terminates for all goals using a semi-local selection rule.

Section 7.3.1 formally introduces the transformation and proves the main results relating to termination of the transformed programs, and their semantic equivalence to the original programs. Some efficiency issues are considered in Section 7.3.2.

7.3.1 Termination

The following definition formalises the transformation exemplified in Section 7.1.4[4].

Definition 7.14 (semi delay recurrent transform *sdr*) For a predicate symbol p and a clause identifier c defined in a program P, let p^{sdr}, p^{depth} and c^{dec} denote predicate symbols not defined in P. Let p/k be a predicate. Then

$$sdr(p/k) = p(v_1, \ldots, v_k) \leftarrow p^{depth}(v_1, \ldots, v_k, d), p^{sdr}(v_1, \ldots, v_k, d)$$

where the d and v_i are variables for all $i \in [1, k]$. Let $c : p(t_1, \ldots, t_k) \leftarrow B_1, \ldots, B_n$ be a clause with $Rec = \{i_1, \ldots, i_m\} \subseteq \{1, \ldots, n\}$ representing the recursive body atoms of c in that $j \in Rec$ iff $rel(B_j) \simeq p$. Then

$$sdr(c) = \begin{cases} p^{sdr}(t_1, \ldots, t_k, \_) \leftarrow & \text{if } n = 0 \\ p^{sdr}(t_1, \ldots, t_k, d) \leftarrow c^{dec}(d, d_{i_1}, \ldots, d_{i_m}), B'_1, \ldots, B'_n & \text{otherwise} \end{cases}$$

where $vars(d, d_{i_1} \ldots d_{i_m}) \cap vars(c) = \emptyset$ and

- if $rel(B_i) \not\simeq p$ then $B'_i = B_i$,
- otherwise $B'_i = q^{sdr}(s_1, \ldots, s_l, d_i)$ given that $B_i = q(s_1, \ldots, s_l)$.

Then $sdr(P) = \{sdr(c) \mid c \text{ is a clause in } P\} \cup \{sdr(p) \mid p \text{ is a predicate in } P\}$. □

Example 7.5 Applying the *sdr* transform to the predicate Quicksort/2 of Section 7.1.2 results in the following clauses.

```
sdr(Quicksort/2) : Quicksort(x, y) ←
                       Quicksort^depth(x, y, d) ∧
                       Quicksort^sdr(x, y, d).

sdr(qs1) :         Quicksort^sdr([], [], _) ←
sdr(qs2) :         Quicksort^sdr([x|xs], ys, d) ←
                       Qs2^dec(d, d2, d3),
                       Partition(xs, x, l, b),
                       Quicksort^sdr(l, ls, d2),
                       Quicksort^sdr(b, bs, d3),
                       Append(ls, [x|bs], ys)
```

□

Each predicate c^{dec} introduced by the *sdr* transform is essentially a function mapping the depth bound in the head of a clause c to the depth bounds for the recursive calls in the body of c. To ensure termination and completeness, the definitions of these decrementation predicates need to be contrained. The following property is required for termination.

[4]The transformation and the results of this section assume that the declarative semantics of the transformed program will be given by a Herbrand interpretation. It is trivial to see how these can be adapted to apply in the case of the non-Herbrand interpretation considered in Section 7.1.4.

Definition 7.15 (termination property of c^{dec}) Let c^{dec} be a decrementation predicate introduced by the *sdr* transform. Let $|.|_\omega$ be a norm and let I be a model for c^{dec}. The predicate c^{dec} is *decreasing* wrt $|.|_\omega$ and I if, for all $c^{dec}(d, d_1, \ldots, d_n) \in I$, $|d|_\omega > |d_i|_\omega$ for all $i \in [1, n]$. □

Lemma 7.16 (termination) Consider the program $sdr(P) \cup Depth \cup Dec$ where $sdr(P)$ is obtained from a program P via the above transform, and *Depth* and *Dec* contain respectively definitions for each predicate p^{depth} and c^{dec} introduced by the transform such that

- no predicate in *Depth* depends on a predicate in $sdr(P) \cup Dec$
- no predicate in *Dec* depends on a predicate in $sdr(P) \cup Depth$

Let I be a model for $sdr(P) \cup Depth \cup Dec$ and suppose there exist level mappings $|.|_1$ and $|.|_2$ such that *Depth* is semi delay recurrent wrt $|.|_1$ and I, and *Dec* is semi delay recurrent wrt $|.|_2$ and I. Furthermore, suppose there exists a norm $|.|_\omega$ such that, for all $c^{dec} \in Dec$, c^{dec} is decreasing wrt $|.|_\omega$ and I.

Then there exists a level mapping $|.|$ such that $sdr(P) \cup Depth \cup Dec$ is semi delay recurrent wrt $|.|$ and I. □

Proof 10 *Let $|.|_\omega$ be the norm satisfying the above condition. Define the level mapping $|.|$ by*

$$|p^{sdr}(\bar{t}, d)| = |d|_\omega \qquad |p(\bar{t})| = 0 \qquad |p^{depth}(\bar{t})| = |p^{depth}(\bar{t})|_1 \qquad |c^{dec}(\bar{t})| = |c^{dec}(\bar{t})|_2$$

It is easy to show that for any predicate p in P and for any clause c in P, both $sdr(p)$ and $sdr(c)$ are semi delay recurrent wrt $|.|$ and I. □

The main consequence of Lemma 7.16 is that any program can be transformed into one which is semi delay recurrent and whose semantics are equivalent in a sense which will be examined shortly.

Observe that for a clause c the atom $c^{dec}(d, d_{i_1}, \ldots, d_{i_m})$ is the only atom in the body of $sdr(c)$ which is a cover for any other atom wrt the level mapping $|.|$ defined in Lemma 7.16. This means that after its resolution, an arbitrary amount of coroutining may take place between the body atoms of $sdr(c)$.

Example 7.6 Let P be the program consisting of the clauses qs_1 and qs_2. Then $sdr(P)$ comprises the three clauses of Example 7.5. Let *Depth* consist of the single clause

$$\mathsf{Quicksort}^{depth}(\_, \_, \_)$$

and *Dec* consist of the single clause

$$\mathsf{Qs}_2^{dec}(S(d), d, d)$$

The program $sdr(P) \cup Depth \cup Dec$ is semi delay recurrent and moreover, if I is the minimal model for this program and J is the minimal model for P then for any ground terms t_1 and t_2, $I \models \mathsf{Quicksort}(t_1, t_2)$ iff $J \models \mathsf{Quicksort}(t_1, t_2)$. □

By Theorem 7.11, the program $sdr(P) \cup Depth \cup Dec$ will be terminating for all goals under a semi local selection rule if, for each predicate, a delay declaration is added which is safe wrt the level mapping $|.|$ defined in the proof of Lemma 7.16. In the above example, this amounts to delaying goals of the form $\leftarrow$ Quicksort^{sdr}(x, y, d) until d is bounded wrt $|.|_\omega$. However, with the definition of Quicksort^{depth} in Example 7.6, all such goals will suspend, since d is never instantiated. Of course, the d parameter acts as a depth bound on the subsequent computation and it is the role of the introduced Quicksort^{depth} predicate to establish a bound which is large enough to allow that part of the search space containing correct answers to be searched. For example, the goal $\leftarrow$ Quicksort([1,2,3], y) will fail if the subgoal $\leftarrow$ Quicksort^{depth}([1,2,3], y, d) binds d to S(S(0)) since more than two recursive calls to the predicate Quicksort^{sdr} are required to obtain the correct answer substitution y/[1,2,3]. To ensure that completeness is not lost, the definition of Quicksort^{depth} needs to be constrained to ensure that sufficiently large depth bounds are determined.

Observe, however, that such *depth* predicates are only responsible for determining the initial depth bound. The depth bounds at subsequent steps in the computation are dependent on the decrementation predicates. Hence the definitions of these predicates must also be constrained. In this respect, it turns out that the following properties can be used to ensure completeness, and yet are sufficiently general enough to permit considerable flexibility in defining decrementation predicates.

Definition 7.17 (completeness properties of c^{dec}) Let c^{dec} be a decrementation predicate of arity n introduced by the *sdr* transform. Let $|.|_\omega$ be a norm and let I be a model for c^{dec}. The predicate c^{dec} is

- *well defined* wrt $|.|_\omega$ and I if for all $k \in \mathbf{N}$, there exists $c^{dec}(d, d_1, \ldots, d_n) \in I$ such that $|d|_\omega = k$.
- *monotonic* wrt $|.|_\omega$ and I if for any two ground atoms $c^{dec}(d, d_1, \ldots, d_n) \in I$ and $c^{dec}(d', d'_1, \ldots, d'_n) \in I$, such that $|d|_\omega > |d'|_\omega$ holds, $|d_i|_\omega \geq |d'_i|_\omega$ for all $i \in [1, n]$.
- *unbounded above* wrt $|.|_\omega$ and I if for every ground atom $c^{dec}(d, d_1, \ldots, d_n) \in I$, there exists $c^{dec}(d', d'_1, \ldots, d'_n) \in I$ such that $|d'_i|_\omega > |d_i|_\omega$ for all $i \in [1, n]$. □

The following lemma states that it is not necessary to determine precise depth bounds for the search space, but it is instead possible to estimate them by over approximation. They may not, of course, be under approximated without losing completeness. This is a useful result from a practical point of view, since it considerably simplifies the analysis required to determine the bounds.

Lemma 7.18 (over approximation) Let $sdr(P) \cup Depth \cup Dec$ be the program satisfying the conditions of Lemma 7.16. Let I be the minimal model for $sdr(P) \cup Depth \cup Dec$. Suppose there exists a norm $|.|_\omega$ such that for each decrementation predicate c^{dec} in Dec, c^{dec} is well defined wrt $|.|_\omega$ and I, decreasing wrt $|.|_\omega$ and I and monotonic wrt $|.|_\omega$ and I. Then the following hold

1. If $I \models p^{sdr}(t_1, \ldots, t_k, d)$ then there exists d' such that $I \models p^{sdr}(t_1, \ldots, t_k, d')$ and for all d'' such that $I \models p^{sdr}(t_1, \ldots, t_k, d'')$, $|d''|_\omega \geq |d'|_\omega$.
2. If $I \models p^{sdr}(t_1, \ldots, t_k, d)$ then for all $h > 0$, there exists d' such that $|d'|_\omega = |d|_\omega + h$ and $I \models p^{sdr}(t_1, \ldots, t_k, d')$. □

Proof 11 *For brevity, let* $R = sdr(P) \cup Depth$ *and let* $S = R \cup Dec$. *Let* I_{Dec} *denote the subset of the minimal model* I *for* S *comprising those atoms with predicate symbols belonging to predicates defined in Dec. Note that, since no predicate in Dec depends on any predicate in* R, I_{Dec} *is the minimal model for Dec. Then* $I = \cup_{k=0}^{\infty} I_S^k$ *where*

$$I_S^0 = \{p(\bar{t}) \mid p(\bar{t}) \leftarrow \text{ is a ground instance of a base clause in } R\} \cup I_{Dec}$$

$$I_S^{n+1} = \left\{ p(\bar{t}) \;\middle|\; \begin{array}{l} p(\bar{t}) \leftarrow B_1, \ldots, B_m \text{ is a ground instance of a clause in } R (m > 0) \\ \text{and } B_i \in \cup_{k=0}^{n} I_S^k \text{ for all } i \in [1, m] \end{array} \right\} \cup I_{Dec}$$

It follows immediately from Definition 7.14 that if $p^{sdr}(t_1, \ldots, t_k, d) \in I_S^0$ *then for every term* t *in the Herbrand universe of* S, $p^{sdr}(t_1, \ldots, t_k, t) \in I_S^0$. *Since* c^{dec} *is well defined wrt* $|.|_\omega$ *and* I, *and* c^{dec} *is decreasing wrt* $|.|_\omega$ *and* I, *there exists a Herbrand term* t *such that* $|t|_\omega = n$ *for all* $n \geq 0$. *It follows that for all* $n \geq 0$ *there exists* d' *such that* $p^{sdr}(t_1, \ldots, t_k, d') \in I_S^0$ *and* $|d'|_\omega = n$.

Fisrt show that if $p^{sdr}(t_1, \ldots, t_k, d) \in I_S^n$, *then* $|d|_\omega \geq n$. *The case for* $n = 0$ *follows from above. For the induction step, suppose that* $p^{sdr}(t_1, \ldots, t_k, d) \in I_S^{n+1}$ $(n \geq 0)$. *Then there exists a ground instance* $p^{sdr}(t_1, \ldots, t_k, d) \leftarrow c^{dec}(d, d_{i_1}, \ldots, d_{i_l}), B_1, \ldots, B_m$ *of a clause in* R, *(where for all* $j \in [1, m]$, $B_j = q_j^{sdr}(s_j^1, \ldots, s_j^{n_j}, d_j)$ *and* $j \in \{i_1, \ldots, i_l\}$ *iff* $rel(B_j) \simeq p^{sdr}$*), such that* $c^{dec}(d, d_{i_1}, \ldots, d_{i_l}) \in \cup_{k=0}^{n} I_S^k$ *and* $B_j \in \cup_{k=0}^{n} I_S^k$ *for all* $j \in [1, m]$. *By the induction hypothesis,* $|d_{i_j}|_\omega \geq n$ *for all* $j \in [1, l]$. *Hence* $|d|_\omega \geq n + 1$, *since* c^{dec} *is decreasing wrt* $|.|_\omega$ *and* I.

Next show that if $p^{sdr}(t_1, \ldots, t_k, d) \in I_S^n$, *then for all* $h > 0$ *there exists* d' *such that* $p^{sdr}(t_1, \ldots, t_k, d') \in I_S^n$ *and* $|d'|_\omega = |d|_\omega + h$. *Again the case for* $n = 0$ *follows from above. For the induction step, suppose that* $p^{sdr}(t_1, \ldots, t_k, d) \in I_S^{n+1}$ $(n \geq 0)$. *Then there exists a ground instance* $c^* : p^{sdr}(t_1, \ldots, t_k, d) \leftarrow c^{dec}(d, d_{i_1}, \ldots, d_{i_l}), B_1, \ldots, B_m$ *of a clause* c *in* R, *(where for all* $j \in [1, m]$, $B_j = q_j^{sdr}(s_j^1, \ldots, s_j^{n_j}, d_j)$ *and* $j \in \{i_1, \ldots, i_l\}$ *iff* $rel(B_j) \simeq p^{sdr}$*), such that* $c^{dec}(d, d_{i_1}, \ldots, d_{i_l}) \in \cup_{k=0}^{n} I_S^k$ *and* $B_j \in \cup_{k=0}^{n} I_S^k$ *for all* $j \in [1, m]$. *Since* c^{dec} *is well defined wrt* $|.|_\omega$ *and* I, *for all* $h > 0$ *there exists* $c^{dec}(d', d'_{i_1}, \ldots, d'_{i_l}) \in \cup_{k=0}^{n} I_S^k$ *such that* $|d'|_\omega = |d|_\omega + h$. *Then, by the monotonicity of* c^{dec} *wrt* $|.|_\omega$ *and* I, *it must be the case that* $|d'_{i_j}|_\omega \geq |d_{i_j}|_\omega$ *for all* $j \in [1, l]$. *Hence by the induction hypothesis, for all* $j \in [1, m]$ *such that* $rel(B_j) \simeq p^{sdr}$, $B'_j = q_j^{sdr}(s_j^1, \ldots, s_j^{n_j}, d'_j) \in \cup_{k=0}^{n} I_S^k$. *It follows that* $p^{sdr}(t_1, \ldots, t_k, d') \leftarrow c^{dec}(d', d'_{i_1}, \ldots, d'_{i_l}), B'_1, \ldots, B'_m$ *is a ground instance of* c, *where* $B'_i = B_i$ *if* $rel(B_i) \not\simeq p^{sdr}$, *such that* $c^{dec}(d', d'_{i_1}, \ldots, d'_{i_l}) \in \cup_{k=0}^{n} I_S^k$ *and* $B'_j \in \cup_{k=0}^{n} I_S^k$ *for all* $j \in [1, m]$. *Therefore,* $p^{sdr}(t_1, \ldots, t_k, d') \in I_S^{n+1}$ *proving the result.* □

The main equivalence result may now be stated. This includes both a completeness result which asserts that every logical consequence of the original program is also a logical consequence of the transformed program, and a soundness result which asserts the converse (restricted to the predicates defined in the original program).

Lemma 7.19 (equivalence) Let $sdr(P) \cup Depth \cup Dec$ be the program satisfying the conditions of Lemma 7.16. Let I be the minimal model for $sdr(P) \cup Depth \cup Dec$ and let J be the minimal model for P. For a predicate p defined in P, define

$$depth(p(\bar{t})) = min\{|d|_\omega \mid I \models p^{sdr}(\bar{t}, d)\}$$

Suppose the following conditions hold

1. there exists a norm $|.|_\omega$ such that for each decrementation predicate c^{dec} in *Dec*, c^{dec} is
   - well defined wrt $|.|_\omega$ and I,
   - decreasing wrt $|.|_\omega$ and I,
   - monotonic wrt $|.|_\omega$ and I, and
   - unbounded above wrt $|.|_\omega$ and I.
2. whenever $I \models p^{sdr}(\bar{t}, d)$ then there exists d', such that $|d'|_\omega \geq depth(p(\bar{t}))$ and $I \models p^{depth}(\bar{t}, d')$.

Then $I \models p(\bar{t})$ iff $J \models p(\bar{t})$. □

Proof 12 *The proof proceeds bottom up on the predicate dependency graph of P. First partition the predicates in P as follows*

1. $p \in \Sigma_{P_0}$ *iff for each clause* $p(t_1, \ldots, t_k) \leftarrow B_1, \ldots, B_m$ *defining p, either* $m = 0$ *or* $rel(B_j) \simeq p$ *for all* $j \in [1, m]$
2. $p \in \Sigma_{P_{n+1}}$ *iff* $p \notin \cup_{k=0}^{n} \Sigma_{P_k}$ *and for each clause* $p(t_1, \ldots, t_k) \leftarrow B_1, \ldots, B_m$ *defining p, either* $m = 0$, *or for all* $j \in [1, m]$ *either* $rel(B_j) \simeq p$ *or* $rel(B_j) \in \cup_{k=0}^{n} \Sigma_{P_k}$.

Example 7.7 *Consider the program*

S(x) $\leftarrow$ *P(x), Q.*
P(x) $\leftarrow$ *R(x), P(x).*
Q.
R(A).
R(B) $\leftarrow$ *R(A), T(B).*
T(B) $\leftarrow$ *T(B), R(A), U.*
U $\leftarrow$ *V.*
V $\leftarrow$ *U.*

Then $\Sigma_{P_0} = \{Q, U, V\}$, $\Sigma_{P_1} = \{R, T\}$, $\Sigma_{P_2} = \{P\}$ *and* $\Sigma_{P_3} = \{S\}$.

Suppose the predicates of P divide into max + 1 partitions $\Sigma_{P_0}, \ldots, \Sigma_{P_{max}}$. *Let* P_k *denote the subset of P consisting of the definitions of the predicates in* Σ_{P_k}. *Observe then that* $P = \cup_{k=0}^{max} P_k$. *The minimal model J for P may also be partitioned according to the predicate dependency graph such that* $J = \cup_{k=0}^{max} J_{P_k}$ *where*

$$J_{P_k} = \{p(\bar{t}) \mid p(\bar{t}) \in J \text{ and } p \in \Sigma_{P_k}\}$$

for all $k \in [0, max]$.

Given the way that sdr(P) is derived from P according to Definition 7.14, the partitions of P may be used to derive a partitioning of sdr(P). Let $\Sigma_{Q_k} = \{p^{sdr} \mid p \in \Sigma_{P_k}\}$ *and* $\Sigma_{R_k} = \Sigma_{P_k}$. *Let* Q_k *and* R_k *denote the subsets of sdr(P) consisting of the definitions of the predicates in* Σ_{Q_k} *and* Σ_{R_k} *respectively. Observe that* $Q_k = \{sdr(c) \mid c \text{ is a clause in } P_k\}$ *and* $R_k = \{sdr(p) \mid p \text{ is a predicate in } P_k\}$. *Hence* $sdr(P_k) = Q_k \cup R_k$ *and* $sdr(P) = \cup_{k=0}^{max} sdr(P_k)$.

Finally, the minimal model I for $sdr(P) \cup Depth \cup Dec$ *may be partitioned as follows. Let* I_{Dec} *and* I_{Depth} *denote the subsets of I comprising respectively those atoms with predicate*

symbols belonging to predicates defined in Dec and Depth respectively. Note that, since no predicate in Dec depends on any predicate in sdr(P) ∪ Depth, I_{Dec} is the minimal model for Dec. (A similar observation may be made regarding I_{Depth}, but this is not required for the proof).

The remainder of the minimal model $I_{sdr(P)} = I \setminus (I_{Dec} \cup I_{Depth})$, may also be partitioned according to the partitioning of the predicates in sdr(P) described above such that $I_{sdr(P)} = \cup_{k=0}^{max}(I_{Q_k} \cup I_{R_k})$ where

$$I_{Q_k} = \{p(\bar{t}) \mid p(\bar{t}) \in I \text{ and } p \in \Sigma_{Q_k}\} \text{ and } I_{R_k} = \{p(\bar{t}) \mid p(\bar{t}) \in I \text{ and } p \in \Sigma_{R_k}\}$$

for all $k \in [0, max]$.

To prove the result, it suffices to show that for all $k \in [0, max]$ if $p(\bar{t}) \in J_{P_k}$ then $p(\bar{t}) \in I_{R_k}$. This is proved by induction over k. Observe that for all $k \in [0, max]$

$$I_{R_k} = \left\{ p(\bar{t}) \;\middle|\; \begin{array}{l} p(\bar{t}) \leftarrow p^{depth}(\bar{t}, d), p^{sdr}(\bar{t}, d) \\ \text{is a ground instance of a clause in } R_k, \\ p^{depth}(\bar{t}, d) \in I_{Depth} \text{ and } p^{sdr}(\bar{t}, d) \in I_{Q_k} \end{array} \right\}$$

Each J_{P_k} and I_{Q_k} can in turn also be defined inductively, such that $J_{P_k} = \cup_{h=0}^{\infty} J_{P_k}^h$ and $I_{Q_k} = \cup_{h=0}^{\infty} I_{Q_k}^h$ where

$$J_{P_0}^0 = \{p(\bar{t}) \mid p(\bar{t}) \leftarrow \text{ is a ground instance of a base clause in } P_0\}$$

$$J_{P_0}^{n+1} = \left\{ p(\bar{t}) \;\middle|\; \begin{array}{l} p(\bar{t}) \leftarrow B_1, \ldots, B_m \text{ is a ground instance of a clause in } P_0 (m > 0), \text{ and} \\ B_j \in \cup_{h=0}^{n} J_{P_0}^h \text{ for all } j \in [1, m] \end{array} \right\}$$

$$J_{P_k}^0 = \{p(\bar{t}) \mid p(\bar{t}) \leftarrow \text{ is a ground instance of a base clause in } P_k\}$$

$$J_{P_k}^{n+1} = \left\{ p(\bar{t}) \;\middle|\; \begin{array}{l} p(\bar{t}) \leftarrow B_1, \ldots, B_m \text{ is a ground instance of a clause in } P_k (m > 0), \\ B_j \in \cup_{h=0}^{n} J_{P_k}^h \text{ for all } j \in [1, m] \text{ such that } rel(B_j) \simeq p, \text{ and} \\ B_j \in \cup_{h=0}^{k-1} J_{P_h} \text{ for all } j \in [1, m] \text{ such that } rel(B_j) \not\simeq p \end{array} \right\}$$

$$I_{Q_0}^0 = \{p^{sdr}(\bar{t}) \mid p^{sdr}(\bar{t}) \leftarrow \text{ is a ground instance of a base clause in } Q_0\}$$

$$I_{Q_0}^{n+1} = \left\{ p^{sdr}(\bar{t}) \;\middle|\; \begin{array}{l} p^{sdr}(\bar{t}) \leftarrow C, B_1, \ldots, B_m \text{ is a ground instance of a clause in } Q_0, \\ C \in I_{Dec}, \text{ and} \\ B_j \in \cup_{h=0}^{n} I_{Q_0}^h \text{ for all } j \in [1, m] \end{array} \right\}$$

$$I_{Q_k}^0 = \{p^{sdr}(\bar{t}) \mid p^{sdr}(\bar{t}) \leftarrow \text{ is a ground instance of a base clause in } Q_k\}$$

$$I_{Q_k}^{n+1} = \left\{ p^{sdr}(\bar{t}) \;\middle|\; \begin{array}{l} p^{sdr}(\bar{t}) \leftarrow C, B_1, \ldots, B_m \text{ is a ground instance of a clause in } Q_k, \\ C \in I_{Dec}, \\ B_j \in \cup_{h=0}^{n} I_{Q_k}^h \text{ for all } j \in [1, m] \text{ such that } rel(B_j) \simeq p, \text{ and} \\ B_j \in \cup_{h=0}^{k-1} I_{R_h} \text{ for all } j \in [1, m] \text{ such that } rel(B_j) \not\simeq p \end{array} \right\}$$

Inductive hypothesis A: If $p(\bar{t}) \in J_{P_k}$ then $p(\bar{t}) \in I_{R_k}$.

1. **Base case:** *($k = 0$) Inductive hypothesis B: If $p(\bar{t}) \in J_{P_0}^r$ then there exists d such that $p^{sdr}(\bar{t}, d) \in I_{Q_0}^r$.*

   (a) **Base case:** *($r = 0$) It follows immediately from Definition 7.14 that if $p(\bar{t}) \in J_{P_0}^0$ then for every d in the Herbrand universe of sdr(P)∪Depth∪Dec, $p^{sdr}(\bar{t}, d) \in I_{Q_0}^0$.*

(b) **Inductive case:** *($r > 0$) Now suppose $p(\bar{t}) \in J^{n+1}_{P_0}$, ($n \geq 0$). Then there exists a ground instance $p(\bar{t}) \leftarrow B_1, \ldots, B_m$ of a clause c in P_0 ($m > 0$) such that for all $j \in [1,m]$, $B_j = q_j(\overline{s_j}) \in \cup^n_{h=0} J^h_{P_0}$. By hypothesis B, it follows that for all $j \in [1,m]$, there exists d_j such that $q^{sdr}_j(\overline{s_j}, d_j) \in \cup^n_{h=0} I^h_{Q_0}$. Consider the predicate c^{dec} in the clause $sdr(c)$. Since c^{dec} is well defined wrt $|.|_\omega$ and I, and unbounded above wrt $|.|_\omega$ and I, there exists $c^{dec}(d, d'_1, \ldots, d'_m) \in I_{Dec}$ such that $|d'_j|_\omega \geq |d_j|_\omega$ for all $j \in [1,m]$. By Lemma 7.18, $B'_j = q^{sdr}_j(\overline{s_j}, d'_j) \in \cup^n_{h=0} I^h_{Q_0}$ for all $j \in [1,m]$. It follows that $p^{sdr}(\bar{t}, d) \leftarrow c^{dec}(d, d'_1, \ldots, d'_m), B'_1, \ldots, B'_m$ is a ground instance of $sdr(c) \in Q_0$ such that $c^{dec}(d, d'_1, \ldots, d'_m) \in I_{Dec}$ and $B'_j \in \cup^n_{h=0} I^h_{Q_0}$ for all $j \in [1,m]$. Hence $p^{sdr}(\bar{t}, d) \in I^{n+1}_{Q_0}$.*

Now supposing $p(\bar{t}) \in J_{P_0}$, it follows from the above that there exists d such that $p^{sdr}(\bar{t}, d) \in I_{Q_0}$. Then, by assumption, there exists d', such that $|d'|_\omega \geq depth(p(\bar{t}))$ and $I \models p^{depth}(\bar{t}, d')$ (i.e. $p^{depth}(\bar{t}, d') \in I_{Depth}$). By Lemma 7.18, $p^{sdr}(\bar{t}, d'') \in I_{Q_0}$ and as a logical consequence $p(\bar{t}) \in I_{R_0}$.

2. **Inductive case:** *($k > 0$) Inductive hypothesis C: If $p(\bar{t}) \in J^r_{P_k}$ then there exists d such that $p^{sdr}(\bar{t}, d) \in I^r_{Q_k}$.*

   (a) **Base case:** *($r = 0$) It follows immediately from Definition 7.14 that if $p(\bar{t}) \in J^0_{P_k}$ then for every d in the Herbrand universe of $sdr(P) \cup Depth \cup Dec$, $p^{sdr}(\bar{t}, d) \in I^0_{Q_k}$.*

   (b) **Inductive case:** *($r > 0$) Now suppose $p(\bar{t}) \in J^{n+1}_{P_k}$, ($n \geq 0$). Then there exists a ground instance $p(\bar{t}) \leftarrow B_1, \ldots, B_m$ of a clause c in P_k ($m > 0$), where*

      i. *for all $j \in [1,m]$ such that $rel(B_j) \simeq p$, $B_j = q_j(\overline{s_j}) \in \cup^n_{h=0} J^h_{P_k}$, and*

      ii. *for all $j \in [1,m]$ such that $rel(B_j) \not\simeq p$, $B_j \in \cup^{k-1}_{h=0} J_{P_h}$.*

      By hypotheses C and A respectively, it follows that

      i. *for all $j \in [1,m]$ such that $rel(B_j) \simeq p$, there exists d_j such that $q^{sdr}_j(\overline{s_j}, d_j) \in \cup^n_{h=0} I^h_{Q_k}$, and*

      ii. *for all $j \in [1,m]$ such that $rel(B_j) \not\simeq p$, $B'_j = B_j \in \cup^{k-1}_{h=0} I_{R_h}$.*

      Consider the predicate c^{dec} in the clause $sdr(c)$. Since c^{dec} is well defined wrt $|.|_\omega$ and I, and unbounded above wrt $|.|_\omega$ and I, there exists $c^{dec}(d, d'_{i_1}, \ldots, d'_{i_l}) \in I_{Dec}$ (where for all $j \in [1,m]$, $j \in \{i_1, \ldots, i_l\}$ iff $rel(B_j) \simeq p$) such that $|d'_j|_\omega \geq |d_j|_\omega$ for all $j \in [1,m]$. By Lemma 7.18, $B'_j = q^{sdr}_j(\overline{s_j}, d'_j) \in \cup^n_{h=0} I^h_{Q_k}$ for all $j \in [1,m]$ such that $rel(B_j) \simeq p$. It follows that $p^{sdr}(\bar{t}, d) \leftarrow c^{dec}(d, d'_{i_1}, \ldots, d'_{i_l}), B'_1, \ldots, B'_m$ is a ground instance of $sdr(c) \in Q_k$ such that $c^{dec}(d, d'_{i_1}, \ldots, d'_{i_l}) \in I_{Dec}$, $B'_j \in \cup^n_{h=0} I^h_{Q_k}$ for all $j \in [1,m]$ such that $rel(B_j) \simeq p$, and $B'_j \in \cup^{k-1}_{h=0} I_{R_h}$ for all $j \in [1,m]$ such that $rel(B_j) \not\simeq p$. Hence $p^{sdr}(\bar{t}, d) \in I^{n+1}_{Q_k}$.

   Now supposing $p(\bar{t}) \in J_{P_k}$, it follows from the above that there exists d such that $p^{sdr}(\bar{t}, d) \in I_{Q_k}$. Then, by assumption, there exists d', such that $|d'|_\omega \geq depth(p(\bar{t}))$ and $I \models p^{depth}(\bar{t}, d')$ (i.e. $p^{depth}(\bar{t}, d') \in I_{Depth}$). By Lemma 7.18, $p^{sdr}(\bar{t}, d') \in I_{Q_k}$ and as a logical consequence $p(\bar{t}) \in I_{R_k}$.

This proves the completeness result. The soundness result given by the converse can easily be proved in a similar manner to the above without relying on any properties of the decrementation predicates. □

This lemma states that the declarative semantics of the original and transformed programs (restricted to the predicates defined in the original program) coincide. Then, under the assumption that the transformed program is deadlock free (Marchiori & Teusink 1996), it can be guaranteed that all computed answers of this program are complete wrt the declarative semantics of the original program.

The problem now then is to define p^{depth} for each predicate p such that the condition of Lemma 7.19 holds. Observe the intuition behind the construction of the transformed program. Suppose it can be deduced that for a given goal G, all computed answers for G can be found in an SLD-tree of a certain depth, then the SLD-tree can be computed to that depth and no more, and all answers for G will surely have been found. In reality, the granularity is finer, relying not on the depth of the SLD-tree as a whole but rather on the lengths of individual branches. More precisely, for each predicate p/k the depth parameter d' in Lemma 7.19 is an upper bound on the number of calls to p/k along one particular branch of the tree. It will often be the case that this bound relates to the "input arguments" of the predicate, i.e. those arguments which are instantiated at the time the predicate is called. One natural approach therefore, is to use interargument relationships to capture this relation.

Example 7.8 Consider the following abstract version of the Append program where the list length norm has been applied to the arguments and the predicate has been augmented with a depth parameter.

```
Append^abs(0, x, x, 0).
Append^abs(x + 1, y, z + 1, d + 1) ←
    Append^abs(x, y, z, d).
```

The success set of this program is characterised by the set

$$\{\mathsf{Append}^{abs}(x, y, z, d) \mid x = z - y = d\}$$

from which it may be observed that d is bounded whenever x or z is. Append^{depth} may then be defined as follows.

```
Append^depth(x, _, z, d) ←
    Length(x, lx) ∧
    Length(z, lz) ∧
    OneOf(lx, lz, d).
```

The definition of OneOf/3 is non-recursive and hence terminating. It is defined to succeed once and instantiate d to either lx or lz in an obvious way.

The above characteristion of the success set may be derived automatically. The analysis of Benoy & King 1996 is one such example of a size relationship analysis based on abstract interpretation capable of deriving the above result. In addition, given suitable abstract programs it can also derive meaningful relationships for the Quicksort program which can then be used to form the definitions of the predicates SetDepth_Q/3, SetDepth_A/3 and SetDepth_P/4 in the program of Section 7.1.4. □

Example 7.9 Given the predicate Split from the program Mergesort

```
Split([], [], []).
Split([x|xs], [x|o], e) ← Split(xs, e, o).
```

the following abstract program may be obtained

Split^{abs}(0, 0, 0, 0).
Split^{abs}(xs + 1, o + 1, e, d + 1) $\leftarrow$ Split^{abs}(xs, e, o, d).

whose success set is characterised by the set

$$S = \{\text{Split}^{abs}(x, y, z, d) \mid d = x, 2y - 1 \leq d \leq 2y, 2z \leq d \leq 2z + 1\}$$

Observe that d is bounded whenever x, y or z is. This information (automatically inferred by the analysis of Benoy & King 1996) may be used to derive a program, according to the proposed transform, which terminates for any goal $\leftarrow$ Split(x, y, z) and, in addition, returns a set of complete answers wrt the declarative semantics whenever x, y or z is a list of determinate length and the remaining two arguments are (optionally) uninstantiated. Existing level mapping based approaches fail to prove termination for these three separate modes. These approaches only reason about the decrease in the level mapping of successive goals in a derivation. For the level mappings $|\text{Split}(t_1, t_2, t_3)|_2 = |t_2|$ and $|\text{Split}(t_1, t_2, t_3)|_3 = |t_3|$ the decrease only occurs on every second goal. A suitable level mapping which can be used to prove termination would be $|\text{Split}(t_1, t_2, t_3)| = min(|t_1|, 2|t_2|, 2|t_3| + 1)$ which is difficult to derive automatically with existing techniques. Although there is some similarity between the definition of $|.|$ and the description of the set S, it is important to remember that the information relating to the depths of derivatons captured by S only applies to successful derivations. Thus it cannot generally be relied upon as a proof of termination in itself (it is in this example, but only because there is a single body atom in the recursive clause). Termination is, instead, ensured through the described transformation procedure.

Another problematic predicate, similar in nature to the Split predicate, where arguments are exchanged in the recursive call, is examined in Mesnard 1995. Termination of that predicate in its various modes can also be ensured via the technique described here. □

7.3.2 Efficiency

The essential idea behind the described approach is to ensure termination by delaying possibly non-terminating goals until certain arguments of those goals become rigid. In theory, the rigidity checks necessary should not incur much more overhead than the delay declarations that are often used to assist termination. For example, checking rigidity of the first argument of the goal $\leftarrow$ Append([1,2,3], y, z) requires three Nonvar tests - exactly the same number that would be required if the goal were executed using the conventional delay declarations. There are additional costs due to unification and the calculation of the depth bound, but these costs could be minimised through careful implementation. Some sample programs have been naively implemented and tested, and some preliminary results are given below. The experiments have been carried out in SICStus Prolog (SICS 1995) on a Sparc 4.

Program P	Goal G	Length of list L	$P \cup \{G\}$		sdr(P) $\cup \{G\}$	
			one	all	one	all
8-queens	qn(_)	-	0.4s	6.8s	0.3s	5.3s
permsort	ps(L, _)	10	6.8s	∞	0.7s	0.7s
permsort	ps(_, L)	8	1.7s	10.5s	2.6s	10.8s
quicksort	qs(L, _)	4000	3.7s	4.5s	4.8s	6.0s
quicksort	qs(_, L)	8	12ms	∞	6ms	83.0s

The main overhead is due to the rigidity checks and the implementation in this respect is rather naive and could be improved. Even with the experimental implementation this overhead only reaches a maximum factor of about three for the simplest programs, e.g. Append. The power of the approach, however, lies in its scalability and it is here where potentially the most impressive performance gains are to be made. Preliminary tests indicate that the most benefit is obtained from larger programs where only one rigidity test is performed at the beginning of the program and the rest of the computation is bounded by the depth bounds. Then the transformed programs can outperform the original ones with the delay declarations, particularly as the amount of backtracking or coroutining increases.

7.4 Summary and Discussion

The aim of control generation is to automatically derive a computation rule for a program that is efficient but does not compromise program correctness. The problem has been effectively tackled here by transforming a program into a semantically equivalent one, introducing safe delay declarations and defining a flexible computation rule which ensures that all goals for the transformed program terminate. Furthermore, it has been shown that the answers computed by the transformed program are complete with respect to the declarative semantics. This is significant.

Beyond the theoretical aspects of the work, its practicality has been demonstrated. In particular, it has been shown how transformed programs can be easily implemented in a standard logic programming language and how such a program can be optimised to reduce the number of costly rigidity checks needed to ensure termination, dramatically improving its performance. Furthermore, with the proposed transformation, the termination problems caused by speculative output bindings are eliminated without the use of a local computation rule or other costly overhead. The coroutining behaviour which is then possible contributes significantly to the efficiency of the generated code.

In terms of correctness, only termination and completeness have been considered in this work, though other correctness issues also need investigating. The connection between acyclic modes, rigid terms and the occur check problem needs to be examined, since the check is never needed for acyclic moded goals (Naish 1993). Also, the example of Section 7.1.4.2 illustrates how the problem of deadlock freedom might be handled.

The efficiency issues also require further investigation. To some extent the issues of termination and performance have been separated but it is not now clear what role extra delay declarations might play in improving the performance of the transformed programs, or even whether other techniques such as multiple specialisation would be more appropriate.

8 Sonic Partial Deduction

8.1 Introduction

Control of partial deduction is divided into two levels. The local level guides the construction of individual SLDNF-trees while the global level manages the forest, determining which, and how many trees should be constructed. Each tree gives rise to a specialised predicate definition in the final program so the global control ensures a finite number of definitions are generated and also controls the amount of polyvariance, i.e. the number of specialised versions produced for each individual source predicate. The local control on the other hand determines what each specialised definition will look like.

Recent work on global control of partial deduction has reached a level of maturity where fully automatic algorithms can be described which offer a near optimal control of polyvariance and guarantee termination of the overall partial deduction process Leuschel *et al.* 1998. Such algorithms are parameterised by the local control component: an unfolding rule which describes how an incomplete SLDNF-tree should be constructed for a given goal and program. It is a requirement of any terminating partial deduction system that such trees are necessarily finite. Techniques developed to ensure finite unfolding of logic programs Bruynooghe *et al.* 1992, Martens *et al.* 1994, Martens & De Schreye 1996 have been inspired by the various methods used to prove termination of rewrite systems Dershowitz & Manna 1979, Dershowitz 1987. Whilst, by no means *ad hoc*, there is little direct relation between these techniques and those used for proving termination of logic programs (or even those of rewrite systems). This means that advances in the static termination analysis technology do not directly contribute to improving the control of partial deduction and the quality of specialised code produced by partial deduction systems. The work of this chapter aims to bridge this gap.

8.1.1 Offline versus Online Partial Deduction

Introduction of a static termination analysis phase into a partial deduction algorithm has the added benefit that unfolding decisions can be based on a *global analysis* of the program's behaviour, and can sometimes even be made before the actual specialisation phase itself. Such an *offline* approach has a number of advantages over its *online* counterpart where unfolding decisions are made at specialisation time.

The advantage which has been the focus of interest of many researchers is its usefulness in the automatic construction of compilers and compiler generators. Partial evaluation of a meta-interpreter with respect to an object program produces a "compiled" version of the object program where the interpretation overhead has been removed. Given a meta-interpreter, a *compiler* in this context is a specialised program dedicated to the "compilation" of object programs in the above sense. A *compiler generator*, or *cogen*, is a program which generates a compiler from a meta-interpreter.

Compiler generators can be automatically generated through self-application, i.e. through partial evaluation of a partial evaluator Futamura 1971. Self applicable partial evaluators for full languages are particularly difficult to contruct, however, and as a result this approach has recently been neglected in favour of a more promising one, known as the cogen approach, where the compiler generator is hand-written instead. This seemingly daunting task turns out to be not too difficult and offers a number of advantages over the indirect approach Birkedal & Welinder 1994.

In order to write a compiler generator by hand, one must first focus on the structure of the compilers that one would like to generate. Remember that a compiler is in effect a "partially evaluating meta-interpreter" where the "interpretation" overhead of the *partial evaluator* has been removed. In other words, the control which would be imposed on the meta-interpreter by the partial evaluator has been compiled into it effectively allowing direct (partial) execution of the meta-interpreter under this control regime. Any control decisions which can be made offline, i.e. independently of the object programs to be executed by the interpreter, can be hard-wired into the compiler and indeed should be in order to make the compiler as fast as possible. Thus, much can be contributed to the efficiency of compilers through the use of the offline approach with its separate static analysis phase.

Clearly these arguments still apply when the meta-interpreter, object program and object goal are replaced by an arbitrary program accepting static and dynamic inputs. For historical reasons, in the cogen approach a compiler is in fact called a *generating extension* and this terminology is adhered to here, not only for consistency with the literature but also for its wider applicability to arbitrary programs.

8.1.2 The Cogen Approach in Logic Programming

The construction of a cogen for a logic programming language (a subset of Prolog) was first described in Jørgensen & Leuschel 1996. A generating extension is obtained via a simple transformation of the source program which will briefly be described here without covering all of the details. Specifically, each atom in the body of a clause in the source program is marked as either *reducible* or *non-reducible*. Each clause in the source program appears in the generating extension, though slightly transformed. Atoms in the body of a clause marked as reducible in the source program also appear in the body of the corresponding clause in the generating extension. As a result these atoms will *always* be unfolded at partial evaluation time. Non-reducible atoms on the other hand are removed from the body during the transformation process; they will *never* be unfolded at partial evaluation time and together will form the leaves of the final SLD-tree (the final part of the transformation augments each head and body atom in the program with an additional argument to capture these leaf atoms at partial evaluation time). One problem with this approach, however, is that whilst it permits goals to be unfolded at normal execution speed, it can unduly restrict the amount of unfolding which takes place with a detrimental effect on the resulting specialised program.

Example 8.1 Consider the Append program below.

```
app1   Append([], x, x).
app2   Append([u|x], y, [u|z]) ←
           Append(x, y, z).
```

In the approach described in Jørgensen & Leuschel 1996 two generating extensions of this program are possible (see below). The first is obtained as a result of marking the body atom Append(x, y, z) in clause app_2 as reducible and the second is obtained by marking this atom as non-reducible.

```
Append([], x, x, []).
Append([u|x], y, [u|z], [leaves]) ←
    Append(x, y, z, leaves).
```

```
Append([], x, x, []).
Append([u|x], y, [u|z], [Append(x, y, z)]).
```

The fourth argument in each program is included to capture the leaves of the SLD-tree in the form of an unflattened nested list of atoms. Now consider the goal ← Append([1,2,3|x], y, z, leaves). Unfolding this goal wrt the first generating extension above leads to the construction of an SLD-tree of infinite depth. When the goal is unfolded wrt the second generating extension, the resulting SLD-tree is finite, but only a single unfolding step is performed and the opportunity for specialisation is missed.

The main problem with the above approach is that it is based on the concept of *binding times* which effectively classify *arguments* as static (known at specialisation time) or dynamic (unknown at specialisation time). This division is too coarse, however, to allow refined unfolding of goals containing partially instantiated data where some parts of the structure are known and others unknown. Instead, the key issue which needs to be considered is termination.

8.1.3 A Sonic Approach

This chapter proposes a flexible solution to the local termination problem for offline partial deduction of logic programs. Based on the cogen approach, the construction of a generating extension will be described which "compiles in" the local unfolding rule for a program and is capable of constructing maximally expanded SLD-trees of finite depth.

The technique builds directly on the work of Chapter 7. The link here is that the residual goals of a deadlocked computation are the leaves of an incomplete SLD-tree. The basic idea is to use static analysis to derive relationships between the sizes of goals and the depths of derivations. This depth information is incorporated in a generating extension and is used to accurately control the unfolding process. At specialisation time the sizes of certain goals are computed and the maximum depth of subsequent derivations is fixed according to the relationships derived by the analysis. In this way, termination is ensured whilst allowing a flexible and generous amount of unfolding. Section 8.2 shows how the transformation of Chapter 7 can be used directly to provide the basis of a generating extension which allows finite unfolding of bounded goals. A simple extension to the technique is described in Section 8.3 which also permits the safe unfolding of unbounded goals.

This is the first offline approach to partial deduction which is able to successfully unfold arbitrarily partially instantiated (including unbounded) goals such as the one encountered in Example 8.1 (Section 8.3). In fact, it is demonstrated that the method can, surprisingly, yield even better specialisation than (pure) online techniques. In particular, some problematic issues in unfolding, notably unfolding under a coroutining computation rule and the back propogation of instantiations Martens & De Schreye 1996, can be easily handled within the approach (Section 8.5). Furthermore,

it is the first offline approach which passes the KMP test (i.e., obtaining an efficient Knuth-Morris-Pratt pattern matcher by specialising a naive one), as demonstrated in Section 8.7.

An analysis which measures the depths of derivations may be termed a *sounding analysis*. Section 8.4 describes how such an analysis can be based on existing static termination analyses which compute level mappings and describes how the necessary depths may be obtained from these level mappings. Unfolding based on a sounding analysis then, is the basis of *sonic partial deduction*.

8.2 Unfolding Bounded Atoms

A fundamental problem in adapting techniques from the termination literature for use in controlling partial deduction is that the various analyses that have been proposed (see De Schreye & Decorte 1994 for a survey) are designed to prove *full* termination for a given goal and program, in other words guaranteeing finiteness of the complete SLD-tree constructed for the goal. For example, consider the goal ← Flatten([x, y, z], w) and the program Flatten consisting of the clauses app_1, app_2, $flat_1$ and $flat_2$.

```
flat1   Flatten([], []).
flat2   Flatten([e|x], r) ←
            Append(e, y, r) ∧
            Flatten(x, y).

app1    Append([], x, x).
app2    Append([u|x], y, [u|z]) ←
            Append(x, y, z).
```

A typical static termination analysis would (correctly) fail to deduce termination for this program and goal. Most analyses can infer that a goal of the form ← Flatten(x, y) will terminate if x is a rigid list of rigid lists, or if x is a rigid list and y is a rigid list. In the context of partial deduction however, such a condition for termination will usually be too strong. The problem is that the information relating to the goal, by the very nature of partial deduction, is often incomplete. For example, the goal ← Flatten([x, y, z], w), will not terminate but the program can be partially evaluated to produce the following specialised definition of Flatten/2.

```
Flatten([x, y, z], r) ←
    Append(x, r1, r ) ∧
    Append(y, r2, r1) ∧
    Append(z, [], r2).
```

The scheme described in Chapter 7 transforms the program Flatten into the following.

```
flat*   Flatten(x, y) ←
            SetDepth_F(x, d) ∧
            Flatten(x, y, d).

DELAY Flatten(_, _, d) UNTIL Ground(d).

flat1*  Flatten([], [], d) ← d ≥ 0.
```

$flat_2^*$ Flatten([e|x], r, d) $\leftarrow$ d $\geq$ 0 $\wedge$
Append(e, y, r) $\wedge$
Flatten(x, y, d $-$ 1).

app^* Append(x, y, z) $\leftarrow$
SetDepth_A(x, z, d) $\wedge$
Append(x, y, z, d).

DELAY Append(_, _, _, d) UNTIL Ground(d).

app_1^* Append([], x, x, d) $\leftarrow$ d $\geq$ 0.
app_2^* Append([u|x], y, [u|z], d) $\leftarrow$ d $\geq$ 0 $\wedge$
Append(x, y, z, d $-$ 1).

For now, assume that the (meta-level) predicate SetDepth_F(x, d) is defined such that it always succeeds instantiating the variable d to the length of the list x if this is found to be of determinate length and leaving d unbound otherwise. Note that a call to Flatten/3 will proceed only if its third argument has been instantiated as a result of the call to SetDepth_F(x, d). The purpose of this last argument is to ensure finiteness of the subsequent computation. More precisely, d is an upper bound on the number of calls to the recursive clause $flat_2^*$ *in any successful derivation.* Thus by failing any derivation where the number of such calls has exceeded this bound (using the test d $\geq$ 0), termination is guaranteed without losing completeness. The predicate SetDepth_A/3 is defined in a similar way, but instantiates d to the minimum of the lengths of the lists x and z, delaying if both x and z are unbounded.

The result of Chapter 7 guarantees that the above program will terminate for every goal (in some cases the program will deadlock). Moreover, given a goal of the form $\leftarrow$ Flatten(x, y) where x is a rigid list of rigid lists or where x is a rigid list and y is a rigid list, the program does not deadlock and produces all solutions to such a goal. In other words, both termination and completeness of the program are guaranteed.

Since the program is terminating for all goals, it can be viewed as a means of constructing a finite (possibly incomplete) SLD-tree for any goal. As mentioned above, it is indeed capable of complete evaluation but a partial evaluation for bounded goals may also be obtained. Quite simply, the deadlocking goals of the computation are seen to be the leaf nodes of an incomplete SLD-tree.

For example, the goal $\leftarrow$ Flatten([x, y, z], r) leads to deadlock with the residual goal $\leftarrow$ Append(x, r1, r, d1) $\wedge$ Append(y, r2, r1, d2) $\wedge$ Append(z, [], r2, d3). Removing the depth bounds, this residue can be used to construct a partial evaluation of the original goal resulting in the specialised definition of Flatten/2 above. Observe that the unfolding is achieved very efficiently through the direct execution of the transformed program. As discussed in Chapter 7 the main limiting factor in this respect is the calculation of the depth bounds. This will be examined in more detail in Section 8.6.

The approach, thus far, is limited in that it can only handle bounded goals. For unbounded goals the unfolding will deadlock immediately and it is not possible, for example, to specialise $\leftarrow$ Flatten([[], [a] | z], r) in a non-trivial way. This strong limitation will be overcome in the following sections.

8.2.1 Relation to Previous Approaches

The method proposed in Bruynooghe *et al.* 1991 (and further developed in Martens & De Schreye 1996) ensures the construction of a finite SLD-tree through the use of a measure function which associates with each node (goal) in the tree a weight from a well-founded set (see also Section 8.5.1). For example, the original measure function proposed by Bruynooghe *et al.* 1991 maps individual atoms to natural numbers and then defines the weight of a goal to be the weight of its selected atom.

Finiteness is ensured by imposing the condition that the weight of any goal is strictly less than the weight of its *direct covering ancestor*[1]. This last notion is introduced to prevent the comparison of unrelated goals which could precipitate the end of the unfolding process. Clearly, one should only compare the weights of goals whose selected atoms share the same predicate symbol. But this is not enough. Consider the atoms Append([1], y, r, 1) and Append([2], y1, y, 1) in the tree of Figure 8.1. Any sensible measure function would assign exactly the same weight to each atom. But, if these weights were compared, unfolding would be prematurely halted after four steps. Hence, this comparison must be avoided and this is justified by the fact that the atoms occur in separate "sub-derivations" of the main derivation. The direct covering ancestor of a goal G then is, loosely speaking, the "closest" ancestor G' occuring in the same sub-derivation where the selected atoms in G and G' share the same predicate symbol[2].

In the sonic approach, the above notions are dealt with implicitly. Figure 8.1 depicts the SLD-tree for the goal ← Flatten([[1], [2]], r, 2) and the transformed version of Flatten. The depth argument of each atom may be seen as a weight as described above. Note that the weight of any atom in a sub-derivation (except the first) is implicitly derived from the weight of its direct covering ancestor by the process of resolution. This conceptual simplicity eliminates the need to explicitly trace direct covering ancestors, improving performance of the specialisation process and removing a potential source of programming errors.

8.3 Unfolding Unbounded Atoms

The main problem with the above transformation is that it only allows the unfolding of bounded goals. Often, as mentioned in the introduction, to achieve good specialisation it is necessary to unfold unbounded atoms also. This is especially true in a logic programming setting, where partially instantiated goals occur very naturally even at runtime. This capability may be incorporated into the above scheme as follows. Although an atom may be unbounded, it may well have a *minimum* size. For example the length of the list [1,2,3|x] must be at least three regardless of how x may be instantiated. In fact, this minimum size is an accurate measure of the size of the part of the term which is partially instantiated and this may be used to determine an estimate of the number of unfolding steps necessary for this part of the term to be consumed in the specialisation process. For example, consider the Append/3 predicate and the goal ← Append([1,2,3|x], y, z). Given that the minimum size of the first argument is three it may be estimated that at least three unfolding steps must be performed. Now suppose

[1]Note that this concept has nothing to do with the *direct cover* relation (Definition 7.2).

[2]In fact, Bruynooghe *et al.* 1991 states a slightly more general condition which is useful for unfolding meta-interpreters, but the details are not important here.

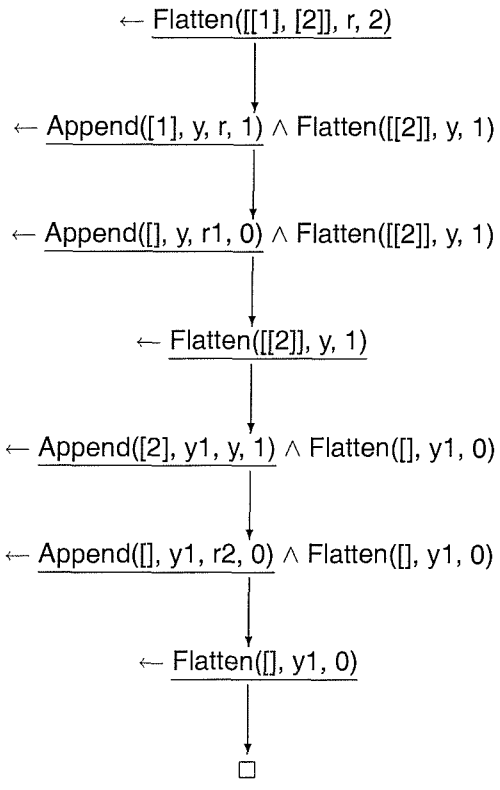


Figure 8.1: Unfolding of ← Flatten([[1], [2]], r, 2)

that the number of unfolding steps is fixed at one plus the minimum (this will usually give exactly the required amount of specialisation). The transformed Flatten program may now be used to control the unfolding by simply calling ← Append([1,2,3|x], y, z, 3). The problem here, of course, is that completeness is lost, since the goal fails if x does not become instantiated to []. To remedy this, an extra clause is introduced to capture the leaf nodes of the SLD-tree. The Append/3 predicate would therefore be transformed into the following.

app_1^* Append([], x, x, d) ← d ≥ 0.
app_2^* Append([u|x], y, [u|z], d) ← d ≥ 0 ∧
    Append(x, y, z, d − 1).
app_3^* Append(x, y, z, d) ← d < 0 ∧
    Append(x, y, z, _).

The call to Append/4 in the clause app_3^* immediately suspends since the depth argument is uninstantiated. The clause is only selected when the derivation length has exceeded the approximated length and the effect is that a leaf node (residual goal) is generated precisely at that point. For this reason, such a clause is termed a *leaf generator* in the sequel. Now for the goal ← Append([1,2,3|x], y, z, 3) the following resultants are obtained.

Append([1,2,3], y, [1,2,3|y], 3) ←
Append([1,2,3,u|x'], y, [1,2,3,u|z'], 3) ← Append(x', y, z', _)

Observe that the partial input data has been completely consumed in the unfolding process. In fact, in this example, one more unfolding step has been performed than is actually required to obtain an optimal specialisation, but this is due to the fact that the goal has been unfolded non-deterministically. In some cases, this non-deterministic unfolding may actually be desirable, but this is an orthogonal issue to termination (this issue will be re-examined in Section 8.7).

Furthermore, note that the SetDepth predicates must now be redefined to assign depths to unbounded atoms. In the case that the depths are derived from level mappings (see Section 8.4), which in turn are defined in terms of norms, this will most likely involve modifying the norm definitions such that variables map to zero instead of variables. Then, for example, $|$[1,2,3|x]$|_{\textit{list-length}} = 3$ and not $3 + x$.

Finally, a predicate such as SetDepth_A(x, z, d) must be defined such that d gets instantiated to the *maximum* of the minimum lengths of the lists x and z to ensure a maximal amount of unfolding. Recall the level mapping $|.|_4$ of Example 3.7 defined by $|\text{Append}(t_1, t_2, t_3)|_4 = min(|t_1|_{\textit{list-length}}, |t_3|_{\textit{list-length}})$ and consider the goal ← Append([1,2,3|x], y, z) from above. With a redefined list length norm mapping variables to zero, it becomes feasible to apply this level mapping to non-ground atoms. Then $|$Append([1,2,3|x], y, z)$|_4 = min(3, 0) = 0$. It is clearly inappropriate, however, to base the value of the depth bound on this level mapping, since it does not provide a measure of the structure present in the atom. This can easily be rectified by redefining $|.|_4$ such that $|\text{Append}(t_1, t_2, t_3)|_4 = max(|t_1|_{\textit{list-length}}, |t_3|_{\textit{list-length}})$. Note that the maximum value returned by this mapping will always be finite.

8.4 Deriving Accurate Depth Bounds from Level Mappings

The above transformations rely on a sounding analysis to determine the depths of derivations or unfoldings. Such an analysis may be based on exisiting termination

analyses which derive level mappings. To establish the link with the termination literature the *depth* argument in an atom during *unfolding* may simply be chosen to be the *level* of the atom with respect to some level mapping used in a termination proof. Whilst, in principle a depth bound for unfolding may be derived from any level mapping, in practice this can lead to excessive unfolding. The following example illustrates this.

Example 8.2 Consider again the Append program and the level mapping $|.|$ defined by $|\mathsf{Append}(x, y, z)| = 3 * |x|_{\text{list-length}}$. The program can be proven to be recurrent wrt $|.|$ and thus goals of the form Append(x, y, z) where x is a rigid list are guaranteed to terminate. If the upper bound on the number of derivation steps in a computation is defined to be equal to the level-mapping, a gross over-approximation is obtained. Given the goal $\leftarrow$ Append([1|x], y, z), the number of derivation steps will be estimated as three. Non-determinate unfolding wrt the clauses app_1^*, app_2^* and app_3^* then produces the following resultants (with the depth bounds removed)

Append([1], y, [1|y]) $\leftarrow$
Append([1,u], y, [1,u|y]) $\leftarrow$
Append([1,u,v], y, [1,u,v|y]) $\leftarrow$
Append([1,u,v,w|x], y, [1,u,v,w|z]) $\leftarrow$ Append(x, y, z)

This specialisation is clearly undesirable. The problem can be fixed here by using a determinate unfolding rule, but this may not always be the case. A more general solution is to consider the difference in the level mappings between the head and the (mutually recursive) body atoms. By subtracting the difference on each recursive call the number of unfolding steps may be accurately controlled.

Example 8.3 Consider clause app_2 of the Append program and the level mapping $|.|$ defined in Example 8.2. Since

$$|\mathsf{Append}([u|x], y, [u|z])| - |\mathsf{Append}(x, y, z)| = (3 \times |x|_{\text{list-length}} + 3) - (3 \times |x|_{\text{list-length}}) = 3$$

the clause app_2 may be transformed into the clause $app_2^\dagger$ below. Then an Append/3 atom whose size is measured wrt $|.|$ can be unfolded wrt app_1^*, $app_2^\dagger$ and app_3^* resulting in the desired specialisation.

$app_2^\dagger$ Append([u|x], y, [u|z], d) $\leftarrow$ d $\geq$ 0 $\wedge$ Append(x, y, z, d $-$ 3).

□

It is often the case that the head and recursive body atoms in a predicate contain distinct variables and thus the difference in their levels is an expression over these variables. Such expressions can often be reduced to a constant using interargument relationships.

Example 8.4 Consider the well known naive reverse predicate below.

rev_1 Reverse([], []).
rev_2 Reverse([x|xs], [y|ys]) $\leftarrow$
Delete(y, [x|xs], zs) $\wedge$
Reverse(zs, ys).

Given the interargument relationship $\mathsf{Delete}(x, y, z) : |y|_{\textit{list-length}} = |z|_{\textit{list-length}} + 1$ together with the level mapping $|.|$ defined by $|\mathsf{Reverse}(x, y)| = |x|_{\textit{list-length}} + 1$ and $|\mathsf{Delete}(x, y, z)| = |y|_{\textit{list-length}}$ the difference in the levels between the head and the recursive call of the clause rev_2 is $(1 + |\mathsf{xs}|_{\textit{list-length}} + 1) - (|\mathsf{zs}|_{\textit{list-length}} + 1) = (1 + |\mathsf{zs}|_{\textit{list-length}} + 1) - (|\mathsf{zs}|_{\textit{list-length}} + 1) = 1$. Note that this may be automatically derived using constraint technology. □

One problem remains in this example. For any goal ← Reverse(x, y), the level mapping $|.|$ over-approximates the number of unfolding steps by 1 each time which may lead to sub-optimal specialisation. Careful examination of the termination literature reveals that level mappings involving additive constants such as $|.|$ are needed in termination proofs where the recursive structure of the program is not fully exploited, such as in a proof of recurrency (De Schreye & Decorte 1994). For example, to prove that the Reverse predicate is recurrent wrt $|.|$ the inequality

$$|\mathsf{Reverse([x|xs], [y|ys])}| > |\mathsf{Delete(y, [x|xs], zs)}|$$

must hold and hence $|.|$ must be defined by $|\mathsf{Reverse}(x, y)| = |x|_{\textit{list-length}} + 1$ rather than $|\mathsf{Reverse}(x, y)| = |x|_{\textit{list-length}}$. The classes of bounded recurrent and semi delay recurrent programs, introduced in Chapters 6 and 7 respectively, allow termination proofs to be based on the recursive structure of a program. Additive constants are seldom needed in such proofs. Indeed, for directly recursive programs, they are completely unnecessary and for mutually recursive programs they can be minimised. The Reverse predicate, for example, is semi delay recurrent wrt the level mapping defined as in Example 8.4 but with $|\mathsf{Reverse}(x, y)| = |x|_{\textit{list-length}}$. It is straightforward to adapt existing termination analyses to derive these simpler level mappings which can then be used to give an accurate measure of the number of unfolding steps required for a given goal.

It may happen that the difference in levels between head and body atoms is not a constant, but is bounded by a constant n. In this case, it is safe to take the bound n as the difference as this will alow a large (though not necessarily maximal) amount of unfolding.

Finally the most problematic case arises when the difference is (bounded by) a variable expression which cannot be reduced to a constant. Here it may be possible to track the sizes of the relevant variables. This involves only a few extra arithmetic operations and *not* the calculation of a large number of term sizes and so incurs only a small performance penalty.

Example 8.5 Consider the Match program

m_1 Match([], _, _, _).
m_2 Match([a|ps], [a|ts], p, t) ←
    Match(ps, ts, p, t).
m_3 Match([a|v], [b|w], p, [x|t]) ←
    a ≠ b ∧
    Match(p, t, p, t).

and the level mapping $|.|$ defined by $|\mathsf{Match}(w, x, y, z)| = |x|_{\textit{list-length}} + (|z|_{\textit{list-length}})^2$. The difference between the head of the clause m_3 and its recursive body atom is not a constant:

$$
\begin{aligned}
|\mathsf{Match([a|v], [b|w], p, [x|t])}| - |\mathsf{Match(p, t, p, t)}| &= (1 + w + 1 + 2t + t^2) - (t + t^2) \\
&= (1 + w) + (1 + t)
\end{aligned}
$$

where $w = |\mathsf{w}|_{\textit{list-length}}$ and $t = |\mathsf{t}|_{\textit{list-length}}$. Tight control of the unfolding process can still be achieved however by transforming Match into the following, where extra arguments are added to track the sizes of the second and fourth arguments which in turn can be used to calculate a more accurate depth bound for each recursive call.

m_1^* Match([], _, _, _, (size_2, size_4, d)) ←
    $\mathsf{d} \geq 0 \wedge \mathsf{size}_2 \geq 0 \wedge \mathsf{size}_4 \geq 0$.
m_2^* Match([a|ps], [a|ts], p, t, (size_2, size_4, d)) ←
    $\mathsf{d} \geq 0 \wedge \mathsf{size}_2 \geq 0 \wedge \mathsf{size}_4 \geq 0 \wedge$
    Match(ps, ts, p, t, ($\mathsf{size}_2 - 1$, size_4, $\mathsf{d} - 1$)).
m_3^* Match([a|_], [b|_], p, [_|t], (size_2,size_4,d)) ←
    $\mathsf{d} \geq 0 \wedge \mathsf{size}_2 \geq 0 \wedge \mathsf{size}_4 \geq 0 \wedge$
    $\mathsf{a} \neq \mathsf{b} \wedge$
    Match(p, t, p, t, ($\mathsf{size}_4 - 1$, $\mathsf{size}_4 - 1$, $\mathsf{d} - \mathsf{size}_2 - \mathsf{size}_4$)).

In this program, the argument d keeps track of the level of each Match atom. The level of the recursive call of clause m_3^* is calculated from the level of the head by subtracting the sizes of the second and fourth head arguments. The necessary expression (i.e. $\mathsf{d} - \mathsf{size}_2 - \mathsf{size}_4$) can be obtained automatically, using, for example, constraint technology, from the difference in the levels of the head and the recursive call as calculated above (note that the size of the second argument in the head is $1 + w$ and the size of the fourth argument is $1 + t$). □

It is not clear when such a transformation would be generally applicable. It is important to note, however, that finiteness can always be guaranteed; the problems raised above relate only to the quality of the specialisation and, as mentioned earlier, this is also dependent to some extent on the control of determinacy. Although this has been touched upon in Gallagher 1993 this is still a relatively unexplored area in the context of partial deduction. Many of the problems above may disappear altogether with the right balance of bounded and determinate unfolding. Finally, note that the problem of deriving a tight upper bound on the number of derivation steps in a computation is also useful in the context of cost analysis (Debray & Lin 1991).

8.5 Offline versus Online Unfolding

This section compares the power of sonic partial deduction with existing online techniques. The most interesting conclusion of this study is that the choice of an offline approach does not necessarily entail the sacrifice of unfolding potential. On the contrary, in some cases the unfolding behaviour is better with the proposed method than with the most recent online ones. This is demonstrated through some simple examples which illustrate known problematic unfolding issues (Martens & De Schreye 1996).

8.5.1 Measure Functions and Level Mappings

Online unfolding methods, e.g. Martens & De Schreye 1996, Bruynooghe *et al.* 1991, Martens *et al.* 1994 use *measure functions* to assign *weights* to atoms and goals. Unfolding is controlled by ensuring that weights are strictly decreasing at each unfolding step. In the seminal online work Bruynooghe *et al.* 1991, weights were assigned to individual atoms using *set based* measure functions of the form

$$|p(t_1, \ldots, t_n)|_{p,S} = |t_{a_1}| + \ldots + |t_{a_m}|$$

where $S = \{a_1, \ldots, a_m\} \subseteq \{1, \ldots, n\}$ and $|t|$ counts the number of (non 0-ary) functors in the term t. The subset S of argument positions for each predicate is determined dynamically during the unfolding process. Clearly, such a function corresponds to a restricted form of level mapping and in principle, the level mapping could be derived *a priori* using static analysis. Much depends of course on the power of the analysis and also to what extent the decreasing weights of goals is dependent on the program input rather than the structure of the program itself. In many cases, however, current termination analysis techniques are able to derive exactly the same level mappings that are obtained through online unfolding.

8.5.2 Lexicographical Priorities

Set based measure functions can lead to overly restrictive unfolding as the following example from Martens & De Schreye 1996 illustrates.

Example 8.6 Consider the ProduceConsume program

pc_1 ProdCons([x|xs], []) ←
    ProdCons(xs, [x]).
pc_2 ProdCons(x, [y|ys]) ←
    ProdCons(x, ys).

and the goal ← ProdCons([1,2|xs], []). Figure 8.2 depicts a finite incomplete SLD-tree illustrating the desired unfolding for this goal (the additional third argument in each atom should be ignored at this point). As Martens & De Schreye 1996 points out, there is no subset S for which the set based measure function $|.|_{\mathsf{ProdCons},S}$ is decreasing for each successive atom in this tree (excluding the last node).

In order to obtain the desired unfolding, Martens and De Schreye refine their measure functions by introducing the notion of a *partition based* measure function. Such a function maps an atom to an ordered n-tuple where each element in the tuple is obtained by applying a set based measure function to the atom. By using the lexicographical ordering to compare n-tuples, this refinement effectively allows priorities to be assigned amongst the arguments of an atom. Figure 8.2 shows the 2-tuples assigned to the atoms in the SLD-tree by the function $|.|_{\mathsf{ProdCons},(\{1\},\{2\})}$ defined by

$$|\mathsf{ProdCons}(x, y)|_{\mathsf{ProdCons},(\{1\},\{2\})} = (|x|_{\mathsf{ProdCons},\{1\}}, |y|_{\mathsf{ProdCons},\{2\}})$$

The problem is easily handled within the framework proposed here by the choice of a level mapping which ensures termination. The above program is recurrent wrt the level mapping $|.|$ defined by $|\mathsf{ProdCons}(x, y)| = 2 * |x|_{\textit{list-length}} + |y|_{\textit{list-length}}$. Then

$$|\mathsf{ProdCons}([x|xs], [])| - |\mathsf{ProdCons}(xs, [x])| = 1$$

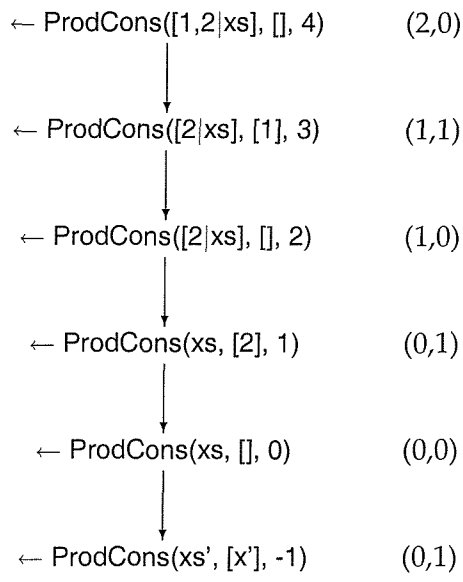


Figure 8.2: *Unfolding of* ← ProdCons([1,2|xs], [], 4)

$$|\text{ProdCons(x, [y|ys])}| - |\text{ProdCons(x, ys)}| = 1$$

and the clauses pc_1 and pc_2 can be transformed into

pc_1^* ProdCons([x|xs], [], d) ←
ProdCons(xs, [x], d − 1).
pc_2^* ProdCons(x, [y|ys], d) ←
ProdCons(x, ys, d − 1).

Now |ProdCons([1,2|xs], [])| = 4 and the goal ← ProdCons([1,2|xs], [], 4) can be unfolded wrt the clauses pc_1^*, pc_2^* and a leaf generator to produce the SLD-tree depicted in Figure 8.2. Notice how the priority assigned to the first argument of ProdCons/2 by the lexicographical ordering is captured by the co-efficient 2 in the level mapping |.|. In fact, exactly the same result may be obtained using any other level mapping |.|′ defined by $|\text{ProdCons}(x,y)|' = a * |x|_{\textit{list-length}} + b * |y|_{\textit{list-length}}$ where $a > b > 0$ are arbitrary integers. Of course, the generating extension and goal are different in each case. Also note that such a level mapping can be automatically derived using current termination analysis technology, e.g. Decorte & De Schreye 1997.

8.5.3 Well-quasi Orders and Homeomorphic Embedding

It turns out that, for the online approach, well-founded orders as used in Sections 8.5.1 and 8.5.2 are sometimes too rigid or (conceptually) too complex. Recently, *well-quasi orders* have therefore gained popularity to ensure online termination of program manipulation techniques (Bol 1991, Sahlin 1993, Sørensen & Glück 1995, Glück *et al.* 1996, Jørgensen *et al.* 1996, Leuschel & Martens 1996, Vanhoof & Martens 1997, Leuschel *et al.* 1998).

The additional power of well-quasi orders stems from the fact that incomparable elements are allowed within sequences of goals during unfolding (while approaches based upon well-founded orders have to impose strict decreases). For example, consider a sequence of goals $G_0, \ldots, G_n$ obtained during unfolding. Unfolding based on

a well founded order would require $G_0 > \ldots > G_n$. Thus, unfolding will be halted at this point if the next goal in the sequence G_{n+1} is such that $G_n \leq G_{n+1}$. On the other hand, when using a well-quasi order, it is permissible for elements in the sequence, such as G_n and G_{n+1}, to be incomparable. The goal G_{n+1} will be the final goal in the sequence only if for some $i < n+1$, $G_i \leq G_{n+1}$. The following definition formalises this idea.

Definition 8.1 (well-quasi order) An ordered set $S(\leq)$ is called *well-quasi ordered* iff for any infinite sequence $e_1, e_2, \ldots$ of elements of S, there exist elements e_i and e_j with $i < j$ such that $e_i \leq e_j$. □

A simple example of a well-quasi order is the *homeomorphic embedding* relation $\trianglelefteq$ defined below. The intuition behind this relation is that $s \trianglelefteq t$ if s can be obtained from t by "striking out" parts of t. For example, P(A) can be obtained from P(F(A)) by striking out the function symbol F and thus P(A) $\trianglelefteq$ P(F(A)). Note that this is a generalisation of the subterm relation.

Definition 8.2 (homeomorphic embedding) 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 for all $i \in [1, n]$, $s_i \trianglelefteq t_i$.

The power of well-quasi orders can be seen in the homeomorphic embedding relation which will, for example, allow an unfolding step from the goal ← P([], [A]) to the goal ← P([A], []) and *vice versa*, since the goal atoms are incomparable. No well founded order will allow both of these unfoldings. On the formal side, Leuschel 1998 shows that the homeomorphic embedding relation is strictly more powerful than any online approach using *monotonic well-founded orders* or *simplification orders*. These terms are defined below.

Definition 8.3 (monotonic well-founded order) A well-founded order $<$ on expressions is *monotonic* iff the following hold:

1. $x \not> y$ for all variables x, y;
2. $s \not> f(t_1, \ldots, t_n)$ whenever f is a function symbol and $s \not> t_i$ for some i;
3. $f(s_1, \ldots, s_n) \not> f(t_1, \ldots, t_n)$ whenever $s_i \not> t_i$ for all $i \in [1, n]$. □

Definition 8.4 (simplification ordering) A *simplification ordering* is a well-founded order $<$ on expressions which satisfies the following:

1. $f(t_1, \ldots, s, \ldots, t_n) < f(t_1, \ldots, t, \ldots, t_n)$ if $s < t$ (replacement property);
2. $t < f(t_1, \ldots, t, \ldots, t_n)$ (subterm property);
3. $s\theta < t\phi$ if $s < t$ for all variable renaming substitutions θ and ϕ (invariance under variable replacement). □

The results of Leuschel 1998 covers the approaches of Bruynooghe *et al.* 1992, Martens *et al.* 1994 and Martens & De Schreye 1996 as described in Sections 8.5.1 and 8.5.2. Furthermore, there is no well-founded order, monotonic or not, which is strictly more powerful than $\trianglelefteq$. In practice this means that there will be cases where $\trianglelefteq$ is more powerful than the sonic approach based upon well-founded orders.

Nonetheless, well-quasi orders are more costly to implement (at every unfolding step, a comparison is required with *every* ancestor while well-founded orders only require a comparison with the covering ancestor due to transitivity). Moreover, the well-founded orders used by the sonic approach are not restricted to be monotonic and do not have to be simplification orders. They are thus incomparable in power to $\trianglelefteq$. For example, the list length norm $|.|_{\textit{list-length}}$ is neither monotonic nor a simplification order, and indeed, given t_1 = [1,2,3] and t_2 = [[1,2,3],4] then $|t_1|_{\textit{list-length}} = 3 > |t_2|_{\textit{list-length}} = 2$ although $t_1 \trianglelefteq t_2$ (because t_1 can be obtained from t_2 by striking out parts of the term). In other words $|.|_{\textit{list-length}}$ will admit the sequence t_1, t_2 while $\trianglelefteq$ does not. As will be shown below, there are other cases where the sonic approach is more powerful than the simple approach of using homeomorphic embedding on covering ancestors.

8.5.4 Coroutining

The increased power offered by partition based measure functions can still be insufficient when unfolding under a coroutining computation rule. The following example, again from Martens & De Schreye 1996, illustrates the problem.

Example 8.7 Consider the program Co-ProduceConsume below

```
cpc1  ProduceConsume(x, y) ←
          Produce(x, y) ∧
          Consume(y).

cpc2  Produce([], []).
cpc3  Produce([x|xs], [x|ys]) ←
          Produce(xs, ys).

cpc4  Consume([]).
cpc5  Consume([x|xs]) ←
          Consume(xs).
```

and the goal ← ProduceConsume([1,2|x], y). Figure 8.3 illustrates the desired unfolding for this goal and program (again the additional third argument in each atom should be momentarily ignored). Observe that any (sensible) measure function which only considers the selected atom when assigning a weight to a goal will map the two goals containing the selected atoms Consume([1|y']) and Consume([2|y"]) to the same weight and consequently unfolding would stop on reaching the second of these goals. Observe that an unfolding rule based upon $\trianglelefteq$ *would* allow the Consume([2|y"]) to be unfolded (Consume([1|y"]) $\not\trianglelefteq$ Consume([2|y"]) as 1 and 2 are uncomparable). However, if the initial goal is slightly changed to ProduceConsume([1,1|x], y) then the same problem also arises for $\trianglelefteq$ (now Consume([1|y"]) $\trianglelefteq$ Consume([1|y"]) and further unfolding is prevented). □

The solution proposed in Martens & De Schreye 1996 is to further refine partition based measure functions to take into account other atoms in a goal besides the selected

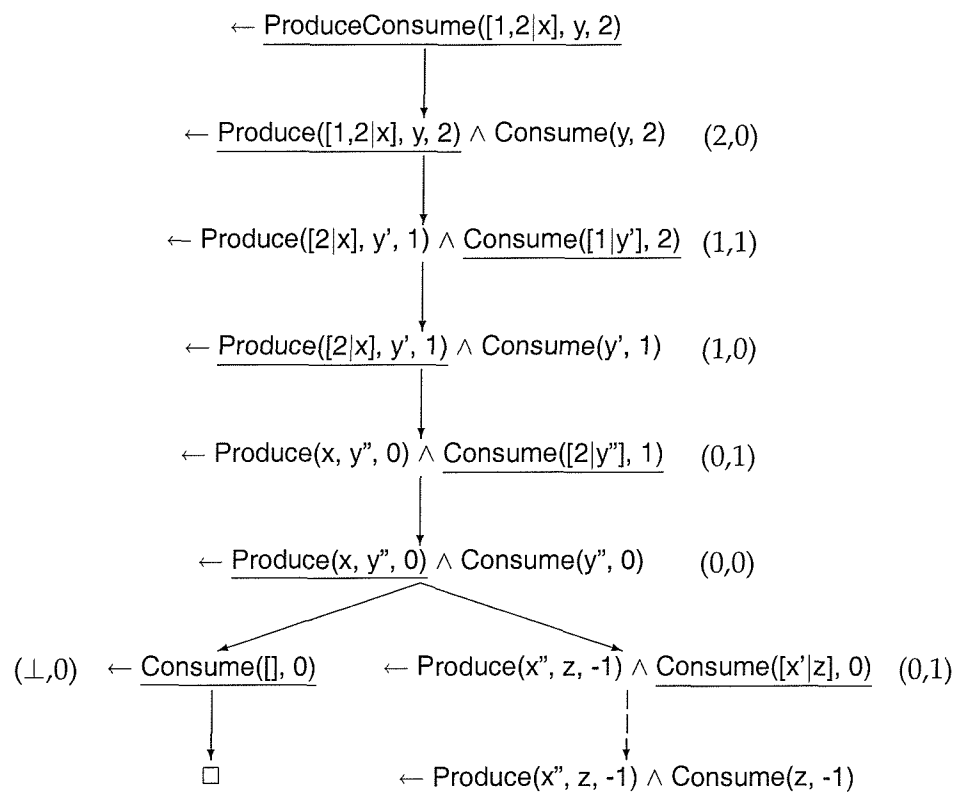


Figure 8.3: Unfolding of ← ProduceConsume([1,2|x], y, 2)

one. The details are somewhat complicated and consequently a full description is not given here. Figure 8.3 shows one possible assignment of weights to the goals in the SLD-tree under the scheme of Martens & De Schreye 1996. The weight associated to each goal is a 2-tuple where the first argument of the tuple is the size of the first argument of the Produce atom in the goal and the second argument is the size of the (first) argument of the Consume atom in the goal. The symbol $\perp$ is used to register the disappearance of the Produce atom, and in addition the ordering on the natural numbers is extended by defining $\perp < 0$.

An offline approach may exploit information from a static analysis to accurately control the unfolding in this example. In particular, interargument relationships allow depth information to be shared between the coroutining atoms in the computation. The interargument relationships

$$\text{ProduceConsume(x, y)} : x' = y' \text{ and } \text{Produce(x, y)} : x' = y'$$

may be derived for the program where $x' = |\text{x}|_{\textit{list-length}}$ and $y' = |\text{y}|_{\textit{list-length}}$. Let $|.|$ be the level mapping defined by $|\text{Produce}(x, y)| = |x|_{\textit{list-length}}$ and $|\text{Consume}(y)| = |y|_{\textit{list-length}}$. Then for any *successful* refutation of the goal ← Produce(x, y) ∧ Consume(y) the equation $|\text{Produce(x, y)}| = |\text{x}|_{\textit{list-length}} = |\text{y}|_{\textit{list-length}} = |\text{Consume(y)}|$ must hold. Hence the program can be transformed into the following (leaf generators for Produce/3 and Consume/2 omitted).

```
cpc*   ProduceConsume(x, y) ←
           SetDepth_PC(x, y, d) ∧ ProduceConsume(x, y, d).
```

cpc_1^* ProduceConsume(x, y, d) $\leftarrow$
Produce(x, y, d) $\wedge$ Consume(y, d).

$prod_1^*$ Produce([], [], d) $\leftarrow$ d $\geq$ 0.
$prod_2^*$ Produce([x|xs], [x|ys], d) $\leftarrow$ d $\geq$ 0 $\wedge$ Produce(xs, ys, d $-$ 1).

$cons_1^*$ Consume([], d) $\leftarrow$ d $\geq$ 0.
$cons_2^*$ Consume([x|xs], d $-$ 1) $\leftarrow$ d $\geq$ 0 $\wedge$ Consume(xs, d).

In this program, the predicate SetDepth_PC(x, y, d) is effectively defined by the equation $d = max(|\text{Produce(x, y)}|, |\text{Consume(y)}|)$. By choosing the maximum of the levels of the two atoms (which is always finite - see Section 8.3) the greatest potential for unfolding is obtained. Thus the initial goal $\leftarrow$ ProduceConsume([1,2|x], y) gives $max(|\text{Produce([1,2|x], y)}|, |\text{Consume(y)}|) = max(2, 0) = 2$ and consequently the goal $\leftarrow$ Produce([1,2|x], y, 2) $\wedge$ Consume(y, 2) is obtained. Unfolding this goal wrt the above program leads to the construction of the whole SLD-tree depicted in Figure 8.3. Using the context considering partition based measure functions of Martens & De Schreye 1996 the final unfolding step on the right hand branch of the tree (indicated by the dashed arrow) is not permitted since the weight of this goal is the same as the weight of its direct covering ancestor $\leftarrow$ Produce(x, y", 0) $\wedge$ Consume([2|y"], 1).

The key issue here is not that a single extra unfolding step is obtained in this example but the fact that this demonstrates that the unfolding capability of an offline technique may surpass that of an online one and the reason for this. The "sharing" of depth information between atoms is possible through the use of interargument relationships which describe the success set of the program. Information relating to the success set is not available to a (pure) online technique. Thus in sonic partial deduction a strictly broader context is considered than in the online case when making unfolding decisions. Finally it is worth remarking that the derivation of interargument relationships forms a core part of many of the termination analyses found in the literature.

8.5.5 Back Propagation

A generating extension for the naive reverse program using Append is shown below.

rev^* Rev(x, y) $\leftarrow$
SetDepth_R(x, y, d) $\wedge$
Rev(x, y, d).

rev_1^* Rev([], [], d) $\leftarrow$
d $\geq$ 0.
rev_2^* Rev([x|xs], y, d) $\leftarrow$
d $\geq$ 0 $\wedge$
Rev(xs, z, d $-$ 1) $\wedge$
App(z, [x], y, d $-$ 1).
rev_3^* Rev(x, y, d) $\leftarrow$
d $<$ 0 $\wedge$
Rev(x, y, _).

Unfolding the goal Rev([1,2|x], y) wrt this program results in an SLD-tree with associated resultants $r_1, \ldots, r_4$ below. To give some idea of how these are obtained, the SLD-derivation associated to r_2 is roughly depicted in Figure 8.4.

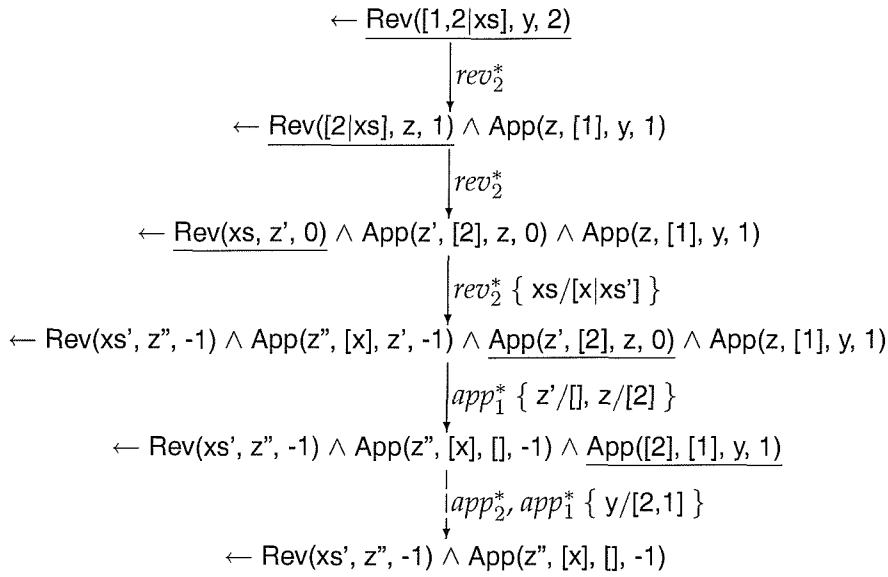


Figure 8.4: Unfolding of ← Rev([1,2|xs], y, 2)

r_1 Rev([1,2|x], y) ← x = [] ∧ y = [2,1].
r_2 Rev([1,2|x], y) ← x = [a|b] ∧ y = [2,1] ∧ Rev(b,c) ∧ App(c,[a],[]).
r_3 Rev([1,2|x], y) ← x = [a|b] ∧ y = [c,1] ∧ App(d,[2],[]) ∧ Rev(b,e) ∧
App(e,[a],[c|d]).
r_4 Rev([1,2|x], y) ← x = [a|b] ∧ y = [c,d|e] ∧ App(f,[1],e) ∧ App(g,[2],[d|f]) ∧
Rev(b,h) ∧ App(h,[a],[c|g]).

Observe that r_2 and r_3 both contain atoms of the form App(x, [y], []) in their right hand sides. These atoms clearly lead to failure but this is not identified during the unfolding process. The decision to leave them as residual atoms takes place before they become instantiated enough to be unfolded. More precisely, when one of these atoms is first encountered it is of the form App(x, [y], z) and should not be unfolded further since there is danger of non-termination. Later in the computation z becomes bound to [] but the atom App(x, [y], []) is no longer a candidate for selection. This problem also arises in the online approach when using the measure functions described in the previous sections to control unfolding [3]. It is termed the *back propagation* problem in Martens & De Schreye 1996 since it is caused by a reverse flow of data.

To solve the problem Martens & De Schreye 1996 suggests yet another, even more complicated measure function refinement. The details of this refinement are not presented and it is not clear to what extent it satisfactorily deals with the problem. The solution in the present context is much simpler and consists of always unfolding any atoms that are bounded. Such atoms will lead to a (possibly empty) resultant whose atoms have predicate symbols lower down in the predicate dependency graph than the predicate symbol of the initial atom. It might also be possible to unfold these atoms further or as a result other atoms in the original resultant may have become

[3]The unfolding and the resultants obtained are slightly different, but the resultants still contain atoms of the form App(x, [y], []) in their right-hand sides.

bounded and can also be selected for unfolding. Note that this process is guaranteed to terminate.

Here again a greater unfolding potential is realised through the availability of global information, i.e. the knowledge that a bounded atom can be safely unfolded a finite number of steps. A pure online technique does not have this "look ahead" capability; it can only compare the present goal with the ones it has encountered in the past with no clue as to what may occur in the future computation.

8.5.6 Other related work

8.5.6.1 Loop checking

Early work on termination in logic programming focused on detecting loops at run-time (Brough & Walker 1984, Covington 1985a, Covington 1985b, Nute 1985, Poole & Goebel 1985, van Gelder 1987). Simple adaptations of these techniques were proposed (Apt *et al.* 1989, Benkerimi & Lloyd 1990) for controlling unfolding during partial deduction. Benkerimi & Lloyd 1990, for example, give four criteria for controling the unfolding. At each unfolding step the selected literal is compared with the one previously selected on the same branch of the SLDNF-tree and unfolding is halted if the descendent literal is a variant of, an instance of, more general than or unifies with the ancestor literal. As illustrated in Bruynooghe *et al.* 1991 these criteria are not comprehensive enough to prevent infinite unfolding. Bol 1993 describes some so called *complete* loop checks which ensure termination of unfolding. It is shown in Bol 1991, however, that it is not enough to look only at the selected literal; the context of the goal is needed, which makes the check more expensive.

As it is, the majority of loop checks rely on comparing the current goal with every preceding goal in the derivation, resulting in a number of checks which is quadratic in the length of the derivation. Some notable exceptions include the Tortoise and Hare technique of van Gelder 1987, which is not actually complete, and those proposed by Bol 1991. Of these, it is suggested that the "triangular" loop check is perhaps, in general, the most efficient though this requires something on the order of $5n$ checks for a derivation of length n. Since each check can involve the comparison of goals, atom for atom, term for term, the cost of these loop checking techniques is still quite high.

Another major disadvantage of the linear time loop checks proposed in Bol 1991 is their random nature. Since they do not compare all goals, and are not tailored specifically to suit a given program and goal, when a loop occurs, it is largely a matter of luck as to when it is detected. In terms of partial deduction, this can lead to an unneccessary explosion of the search space during unfolding.

Finally, it may be remarked that these techniques are all designed to detect loops at run-time which in the partial deduction context translates as an online approach to unfolding. They cannot be used, for example, to ensure in advance, that a given goal, or class of goals can be completely unfolded.

8.5.6.2 Finiteness Analysis

Offline partial evaluation has been studied extensively in functional programming though for some time consideration of termination was largely neglected. The partial evaluation stage is preceded by a binding time analysis which annotates each argu-

ment in the program as either static or dynamic. Functions with static arguments can be evaluated whilst residual code is produced for those with dynamic ones. The termination issue is addressed by *generalising* variables, that is by changing their binding time annotation from static to dynamic (Holst 1991, Anderson & Holst 1996). This occurs whenever execution of a piece of static code may lead to non-termination. Whilst this works well for functional programs, the static/dynamic divisions do not translate well for logic programs (see Section 8.1) and thus this approach to ensuring termination is not generally suitable for logic programming.

8.5.7 Offline vs. Online Conclusion

This section has compared sonic partial deduction with the state of the art online unfolding techniques as described in Martens & De Schreye 1996. It has been shown that the approach is able to handle a variety of examples which are known to present difficulties in unfolding. The method is able to handle these examples in a uniform manner, whereas the work of Martens & De Schreye 1996 requires the use of increasingly complex measure functions to handle them. This increasing complexity introduces with it increasing overhead as well as the growing risk of programmer error in the actual coding. This is an important issue when considering the construction of a tool which one would like to prove terminating.

Not only is the approach much simpler, it also offers potential for unfolding unfulfilled by online methods. The reason why offline techniques can permit more unfolding than online ones is the fact that they consider the global context. A global analysis can infer information which may not be available locally when deciding on a particular atom to unfold. A number of fairly complex examples have been examined in the previous sections. The following is a simpler example:

```
UpToN(n, n, [n]).
UpToN(x, n, [x|xs]) ←
    x < n ∧
    UpToN(x + 1, n, xs).
```

Unfolding the goal ← UpToN(1, 3, x) leads to two other goals: ← UpToN(2, 3, x) and ← UpToN(3, 3, x). The only difference between the atoms in these goals is in the first argument which is increasing in value. To determine that the sequence of goals is finite (under a left-to-right computation rule) requires the global information that the first argument is bounded by the second argument.

Of course, an online technique may still be able to make refined unfolding decisions based on the availability of concrete data, not available to an offline one. Clearly, then, a more powerful technique may be obtained by a combination of these approaches.

8.6 Implementation

In the sonic approach the main limiting factor in the efficiency of the specialisation process is the calculation of the required depth bounds. The "greater than zero" tests on the depth bounds and the decrementation of them in the subsequent computation incur minimal overhead. Thus, by paying attention to the calculation of these bounds the generating extensions can be tuned for maximum efficiency.

This hand-crafting of generating extensions, is an important step in the development of fast specialisers. Experience with the self-application approach shows that fast compilers (or generating extensions) do not come about by accident. Some care needs to be taken to ensure that both the partial evaluator and the interpreter are amenable to specialisation. This amenability has been particularly difficult to achieve in the case of the partial evaluator which must be self-applied.

With regard to the Futamura projections, great emphasis has been placed on how to obtain an efficient compiled version of a program, but not how to obtain efficient compilers or compiler generators. The hope was that by focusing on the result of compilation, i.e. the quality of specialised code, such compilers and cogens would be obtained "for free" through self-application. Given that this approach has not delivered the expected goods, however, it is now necessary to consider how to write efficient generating extensions and cogens. Arguably, this is the right approach even if one were aiming towards self-application anyway; if it were not known how to write an efficient generating extension by hand, it would be extremely fortuitous to generate one automatically.

In this section, concrete issues relating to implementation are examined based on the findings of a simple empirical study. The aim of the study was to investigate ways in which to code a generating extension, based on the sonic approach, in Prolog. The main issues discussed are: atom selection; efficient calculation of the depth bounds; avoiding speculative output bindings and achieving argument indexing; and how to incorporate the global control. A prototype has been built based on the results of this section.

8.6.1 Atom selection

To understand the problem of atom selection consider the goal G = ← Append(x, y, z) ∧ Append([1,2,3], w, x) for the Append program. Suppose the first atom in this goal is selected for unfolding. An estimate of zero for the size of this atom would be obtained by taking the maximum of the minimum estimates for the list lengths of x and z. Nondeterminate unfolding with a depth of zero then leads to the following two goals:

← Append(x', y, z') ∧ Append([1,2,3], w, [u|x'])
← Append([1,2,3], w, [])

The second goal leads to failure. In the first goal, the second atom can be selected, and unfolding with a depth of three results in the following goal:

← Append([2,3|w], y, z')

It appears that two more unfolding steps whould be possible at this point (consuming the terms 2 and 3), but observe that the atom in this goal is simply a more instantiated version of an atom which has already been unfolded. As such it should not be reselected for unfolding, since doing so repeatedly can endanger termination. If, however, the second atom in the original goal G had been selected first followed by the first atom, then these unfolding steps could have been performed.

The problem here is similar to the coroutining and back propagation problems previously discussed. Clearly, there is a dependency between the size of the arguments x and [1,2,3] in the goal G, which could be exploited to obtain the desired

unfolding even when selecting the atom Append(x, y, z) first. If dependencies such as this are not identified during analysis, however, then the order of selection of atoms in a goal may significantly affect the result of the unfolding process.

Controlling the selection of atoms is non-trivial, however, and can greatly increase the time spent unfolding. In the prototype, the simplest option was adopted: unfolding atoms under the normal, left-to-right Prolog computation rule, calculating an atom's depth when it is first encoutered without regard to whether or not a better estimate might be obtained by delaying the atom. The results of Section 8.7 suggest that this approach works well in practice and it is unlikely that a more sophisticated approach is required.

8.6.2 Depth bound calculations

Two means of calculating depth bounds are required: one for (possibly) unbounded atoms and one for bounded atoms. The first of these is the simplest. All that is required is a predicate for each norm used and one for each level mapping. In the calculation of the norms, variables should always map to zero to obtain a minimum estimate of the size of a term. The level mappings may be defined as a "disjunction" representing various alternative modes for which a given predicate is terminating. All that is required is the maximum of the possible levels for an atom as discussed in Section 8.3.

Let $|.|$ be a level mapping defined for all atoms A with predicate symbol p/n as follows:

$$|A| = min(|A|_1, \ldots, |A|_m)$$

where each $|.|_i$ is a level mapping defined as follows:

$$|p(t_1, \ldots, t_n)|_i = f_i(||t_1||_1, \ldots, ||t_n||_n)$$

where each $||.||_j$ is a type-linear norm and $f_i : \mathbf{N}^n \mapsto \mathbf{N}$ is a (monotonic) function.

The above scheme can be used to define a large number of level mappings, including all those commonly found in the termination literature. The *min* function in the level mapping definition allows termination to be proven for a predicate when used in different modes (see Example 3.7). For the level mapping to be used to estimate the size of unbounded atoms, this *min* function must be changed to a *max* function and, in addition, variables must be mapped to zero by the norms (see Section 8.3). Translation into Prolog then results in the following program schema:

```
set_depth_p_n_u(X1,...,Xn,L):-
    '||.||1'(X1,SX1),
    ⋮
    '||.||n'(Xn,SXn),
    'f1'(SX1,...,SXn,L1),
    ⋮
    'fm'(SX1,...,SXn,Lm),
    max([L1,...,Lm],L).
```

Obviously, this general schema can be specialised for each level mapping. For example, the following is a specialised instance of the above schema for the level mapping $|.|$ defined by $|\mathsf{Append}(x,y,z)| = min(|\mathsf{Append}(x,y,z)|_1, |\mathsf{Append}(x,y,z)|_2)$ where $|\mathsf{Append}(x,y,z)|_1 = |x|_{\textit{list-length}}$ and $|\mathsf{Append}(x,y,z)|_2 = |z|_{\textit{list-length}}$:

```
set_depth_append_3_u(X,Z,L):-
    list_length(X,LX),
    list_length(Z,LZ),
    max(LX,LZ,L).
```

Turning to the calculation of the norms, since only a minimum estimate of the size of a term is needed, this is a case where variables should be mapped to zero. The definition of a type-linear norm can be translated directly into Prolog. For example, the list length norm can be defined by the following predicate:

```
list_length(V,D,D).
    var(V),!.
list_length([],D,D).
list_length([_|Y],D_in,D_out):-
    D_in1 is D_in + 1,
    list_length(Y,D_in1,D_out).
```

The norm which sums the lengths of the sublists of a list can then be defined as follows:

```
sum_sublist_length(V,D,D).
    var(V),!.
sum_sublist_length([],D,D).
sum_sublist_length([X|Y],D_in,D_out):-
    list_length(X,D_in,D_mid),
    sum_sublist_length(Y,D_mid,D_out).
```

It is possible that, it may be useful to know if an atom is actually bounded, and it is straightforward to modify these predicate definitions to determine this, with trivial overhead. Future experimentation will determine whether or not this information can be used to improve the unfolding. For example, with the level mapping definition above, if `X` and `Z` were found to be rigid it might be more useful to bind `L` to their minimum. In fact, in the case of the Append predicate this will not make any difference, but it gives an idea of how such information might be useful.

Recall that, in the proposed scheme, atoms which have been unfolded are delayed and will be awoken only if they become bounded. Where the level mapping $|.|$ is defined as the minimum of level mappings $|.|_1$ to $|.|_m$ as above, an atom will be bounded wrt $|.|$ if it is bounded wrt $|.|_i$ for some $i \in [1, m]$. This in turn will depend on the rigidity of the arguments which the level mapping is defined in terms of. For example, an Append/3 atom will be bounded wrt the level mapping $|.|$ defined earlier if either its first argument or its third argument is rigid.

It is straightforward to define a predicate which measures the size of a term wrt a norm, but delays until the term is rigid. Because the term may become rigid incrementally, i.e. various parts of the term may become instantiated over time, constraints are a convenient mechanism to handle the calculation. For example, the following is an implementation of a norm which measures the size of a binary tree (the expressions in braces are constraints):

```
:- block tree_size(-, ?).

/* delay tree_size(A, B) until A is instantiated */
```

```
tree_size(leaf, D):-
    {D = 0}.
tree_size(node(X, _, Z), D):-
    tree_size(X, Dx),
    tree_size(Z, Dz),
    {D = Dx + Dz + 1}.
```

Constraints are a particularly expensive mechanism to use, however, for such time-critical calculations. The above norm can alternatively be implemented as follows:

```
:- block tree_size(-, ?, ?), tree_size(?, -, ?).

/* delay tree_size(A, B, C) until A and B are instantiated */

tree_size(leaf, D, D).
tree_size(node(X, _, Z), D_in, D_out):-
    D_in1 is D_in + 1,
    tree_size(X, D_in1, D_mid),
    tree_size(Z, D_mid, D_out).
```

The difference between these two implementations is significant as the following table shows. Timings are also included for implementations of the list length norm. All timings performed on a Sparc 4 using SICStus Prolog 3 # 3, and rigid terms as input.

Norm	Term Size	'{}' Time(ms)	'is' Time(ms)
list_length	10000	1550	10
list_length	100000	16920	40
tree_size	32767	6670	50
tree_size	65535	13750	100

8.6.3 Speculative output bindings and argument indexing

The following program is a simple (incomplete) implementation of a generating extension for the naive reverse program of Section 8.5.5. The predicate `set_depth_rev_u/3` is used to estimate the level of a possibly unbounded `rev/2` atom. This predicate always succeeds and never suspends. Its counterpart `set_depth_rev_b/3` found in the leaf generator clause for `rev` is defined similarly but will suspend if the atom is not bounded, i.e. it will only compute a level for bounded atoms. The predicates `set_depth_append_u/3` and `set_depth_append_b/3` are defined similarly.

```
rev(X, Y):-
    set_depth_rev_u(X, Y, Level),
    rev(X, Y, Level, unbounded).

rev([], [], D, _):-
    D >= 0.
rev([X|Xs], Y, D, Mode):-
    D >= 0,
```

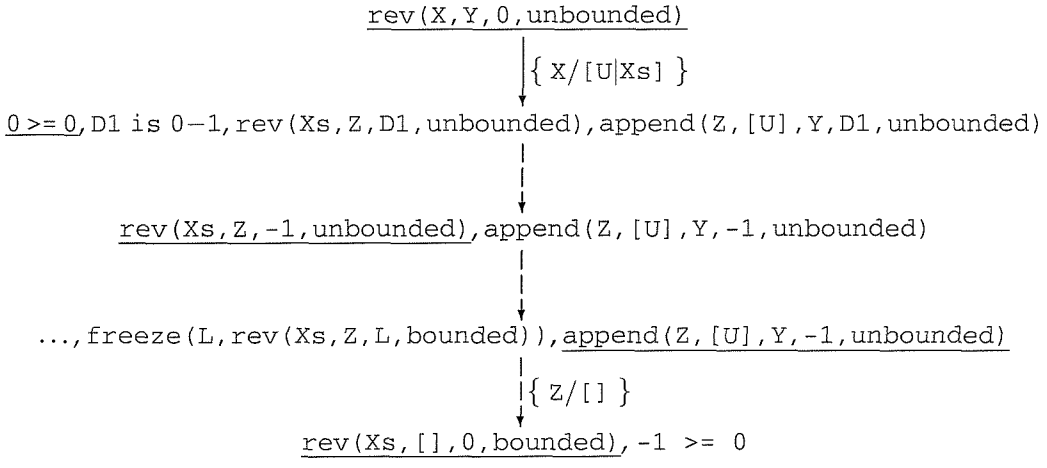
```
    D1 is D - 1,
    rev(Xs, Z, D1, Mode),
    append(Z, [X], Y, D1, Mode).
rev(X, Y, D, unbounded):-
    D < 0,
    set_depth_rev_b(X, Y, Level),
    freeze(Level, rev(X, Y, Level, bounded)).

append(X, Y, Z):-
    set_depth_append_u(X, Z, Level),
    append(X, Y, Z, Level, unbounded).

append([], X, X, D, _):-
    D >= 0.
append([X|Xs], Y, [X|Zs], D, Mode):-
    D >= 0,
    D1 is D - 1,
    append(Xs, Y, Zs, D1, Mode).
append(X, Y, Z, D, unbounded):-
    D < 0,
    set_depth_append_b(X, Z, Level),
    freeze(Level, append(X, Y, Z, Level, bounded)).
```

Observe that this simple program implements the solution to the back propagation problem described in Section 8.5.5. In fact, for any goal of the form `rev(x, y)` where `x` or `y` is a non-rigid list of arbitrary length, the program will prune all failing branches.

Unfortunately, the problem of speculative output bindings rears its ugly head once again, with the effect this time, not being non-termination, but poor performance. Consider, for example, the computation which results from the call `rev(X, Y)`. After finding the first solution the following segment of the computation is reached:



This last goal arises because the call `append(Z,[U],Y,-1,unbounded)` matches the first clause of `append` and speculatively binds `Z` to `[]`. This binding causes the `rev` atom in the goal to become bounded and as a result it is selected. The call `-1`

`>= 0`, which will cause the whole goal to fail, is not reached until this redundant sub-computation is completed. Furthermore, the call to `-1 >= 0` is repeatedly selected due to backtracking over the call `rev(Xs,[],0,bounded)` (observe that it matches two clauses of `reverse`).

As will be seen shortly, these speculative output bindings are a source of great inefficieny. In order to avoid them, the obvious solution is to ensure that the depth bounds are range checked before any output bindings are made. Delaying unification *à la* Naish 1993, until after the range check has been made is one possibility. In this case, the second clause of `append`, for example, would be transformed into the following:

```
append(A1, Y, A3, D, Mode):-
    D >= 0,
    A1 = [X|Xs],
    A3 = [X|Zs],
    D1 is D - 1,
    append(Xs, Y, Zs, D1, Mode).
```

The adequacy of this approach depends very much on the sophistication of the available indexing mechanism. Many implementations of Prolog will not be able to perform the deep indexing that would be required here. Hence, in order to take advantage of the performance benefits of indexing, an alternative approach is called for. The one suggested here is to introduce an auxilliary clause which simply handles the range checks for depth bounds. The `append` predicate, for example, would become:

```
append(X, Y, Z):-
    set_depth_append_u(X, Z, Level),
    append(X, Y, Z, Level, unbounded).

append_aux(X, Y, Z, D, Mode):-
    (D >= 0 ->
        append(X, Y, Z, D, Mode)
    ;
        Mode = unbounded,
        set_depth_append_b(X, Z, Level),
        freeze(Level, append(X, Y, Z, Level, bounded))
    ).

append([], X, X, _, _).
append([X|Xs], Y, [X|Zs], D, Mode):-
    D1 is D - 1,
    append_aux(Xs, Y, Zs, D1, Mode).
```

A comparison of these three versions reveals the potential impact that speculative output bindings and argument indexing can have on the efficiency of the generating extension. The programs were all executed with a goal of the form `rev(`x`, Y)` where x was an open list of elements (i.e. the tail of the list was a variable). Running on interpreted code under SICStus Prolog 2.1 #9 on a Sparc 4, the results, for the 1, 2, 5, 10, 20, 200 and 500 element list respectively, were as follows: Original version (with

speculative output bindings) – 4ms, 7ms, 17ms, 45ms, 143ms, 10792ms, 65527ms; delayed unification version – 4ms, 5ms, 11ms, 27ms, 76ms, 4813ms, 29755ms; auxilliary clause version – 3ms, 4ms, 9ms, 22ms, 56ms, 2918ms, 17516ms.

8.6.4 Global control

When an atom is unfolded it gives rise to an SLD-tree which describes the unfolding. The majority of approaches to global control rely on an abstract representation of the SLD-trees for atoms to control polyvariance. In order, for the local control to plug in to the global component it must return some such representation. Trace-terms (Gallagher & Lafave 1996) and characteristic trees (e.g. Leuschel *et al.* 1998) have emerged as the main contenders for abstracting SLD-trees. Characteristic trees are hard to generate directly in the cogen approach, since they require meta-level information regarding which atom in a goal has been selected. Hence, trace-terms, which do not require such information, appear to be the perfect choice in this situation. As an example, trace-terms can be incorporated into the `reverse` predicate as follows:

```
rev([], [], _, _, rev1).
rev([X|Xs], Y, D, Mode, rev2(Rev, App)):-
    D1 is D - 1,
    rev_aux(Xs, Z, D1, Mode, Rev),
    append_aux(Z, [X], Y, D1, Mode, App).
```

Trace-terms only abstract single derivations, however, and must be combined in some way to form trace-term trees. In order to construct the trace-term tree, each individual trace-term must be saved to avoid its loss on backtracking for alternative solutions. This introduces, what is probably, the greatest expense of the whole approach. In a Prolog based implementation this overhead seems difficult to avoid and no solution is suggested here. It is an area which requires further investigation, however, in order to improve the efficiency of the overall specialisation process.

8.7 Experiments and Benchmarks

To gauge the efficiency and power of the sonic approach, a prototype implementation has been devised and integrated into the ECCE partial deduction system (Leuschel 1996, Leuschel 1997, Leuschel *et al.* 1998). The latter is responsible for the global control and code generation and calls the sonic prototype for the local control. A comparison has been made with ECCE under the default settings, i.e. with ECCE also providing the local control using its default unfolding rule. For the global control, both specialisers used conjunctive partial deduction (Leuschel *et al.* 1996, Glück *et al.* 1996) and characteristic trees (Leuschel *et al.* 1998).

In the cogen approach, it is very covenient to build trace terms (Gallagher & Lafave 1996) for use in the global control and this was incorporated into the sonic prototype. As ECCE employs characteristic trees in a certain format, however, a conversion from trace terms into characteristic trees had to be added. Such a conversion will be unnecessary in an improved version of ECCE which is also able to handle trace terms.

All the benchmarks are taken from the DPPD library (Leuschel 1996) and were run on a Power Macintosh G3 266 Mhz with Mac OS 8.1 using SICStus Prolog 3 #6 (Macintosh version 1.3). Tables 8.1 shows the total specialisation time for each benchmark without post-processing. This total specialisation time includes not only the time spent in unfolding during specialisation but also the additional time needed by the global control (provided by ECCE) to guide the overall specialisation process. Table 8.2 shows only the time spent in unfolding during specialisation. In Table 8.1 the times to produce the generating extensions for the sonic approach are not included, as this is still done by hand. It is possible to automate this process and one purpose of hand-coding the generating extensions was to gain some insight into how this could be best achieved. In any case, in situations where the same program is repeatedly respecialised, this time will become insignificant anyway. The precision of the timings, which were performed using the `statistics/2` predicate, seems to be approximately 1/60th of a second, i.e., about 16.7 ms. Hence "0 ms" in Table 8.2 should most likely be interpreted as "less than 16 ms". The runtimes for the residual programs appear in Table 8.3, which, for a more comprehensive comparison, also includes the results obtained by MIXTUS.

Benchmark	sonic + ECCE	ECCE
advisor	**17** ms	150 ms
applast	83 ms	**33** ms
doubleapp	50 ms	**34** ms
map.reduce	**33** ms	50 ms
map.rev	**50** ms	67 ms
match.kmp	300 ms	**166** ms
matchapp	**66** ms	83 ms
maxlength	**184** ms	200 ms
regexp.r1	**34** ms	400 ms
relative	**50** ms	166 ms
remove	**367** ms	400 ms
remove2	1049 ms	**216** ms
reverse	50 ms	50 ms
rev_acc_type	316 ms	**83** ms
rotateprune	**67** ms	183 ms
ssupply	**34** ms	100 ms
transpose	**50** ms	467 ms
upto.sum1	**33** ms	284 ms
upto.sum2	**50** ms	83 ms

Table 8.1: Specialisation times (total w/o post-processing)

The sonic prototype implements a more agressive unfolding rule than the default determinate unfolding rule of ECCE. This is at the expense of total transformation time (see Table 8.1), as it often leads to increased polyvariance, but consequently the speed of the residual code is often improved, as can be seen in Table 8.3.[4] Default ECCE settings more or less guarantee no slowdown, and this is reflected in Table 8.3,

[4] A more agressive unfolding rule, in conjunctive partial deduction, did not lead to improved speed under compiled code of Prolog by BIM; see Leuschel 1997. So, this also depends on the quality of the indexing generated by the compiler.

Benchmark	sonic + ECCE	ECCE
advisor	0 ms	33 ms
applast	0 ms	16 ms
doubleapp	0 ms	0 ms
map.reduce	0 ms	17 ms
map.rev	0 ms	34 ms
match.kmp	0 ms	99 ms
matchapp	0 ms	33 ms
maxlength	0 ms	67 ms
regexp.r1	0 ms	383 ms
relative	0 ms	166 ms
remove	34 ms	201 ms
remove2	33 ms	50 ms
reverse	16 ms	33 ms
rev_acc_type	0 ms	32 ms
rotateprune	0 ms	99 ms
ssupply	0 ms	67 ms
transpose	16 ms	400 ms
upto.sum1	0 ms	168 ms
upto.sum2	0 ms	66 ms

Table 8.2: Specialisation times (unfolding)

whereas the general lack of determincay control in the prototype sonic unfolding rule leads to two small slowdowns.

There is plenty of room for improvement, however, on these preliminary results. The sonic approach is flexible enough to allow determinacy control to be incorporated within it, and this extra layer of control could help to guarantee no slowdown. Also, the sonic prototype has been built on the philosophy of "unfold finitely as much as possible". This bull-in-a-china-shop approach actually pays off much better than expected, but the results also indicate that some refinements might also lead to better specialisation times and more efficient residual code. There is plenty of scope for variation within the prototype, which would allow these refinements to be made. The only potential problem is in identifying when it would be appropriate to use them.

All in all, the sonic approach provides extremely fast unfolding combined with very good specialisation capabilities. It is surprising that the sonic approach outperformed the (albeit conservative) default unfolding of ECCE. Also observe that the sonic approach even improves upon the `match.kmp` benchmark and passes the KMP test (even better than the online system does). The sonic approach is thus the first offline approach to our knowledge which passes the KMP test.[5] If it were possible to extend the sonic approach to the global control as well, one would hopefully obtain an extremely efficient specialiser producing highly optimised residual code.

[5] One might argue that the global control is still online. Note, however, that for KMP no generalisation and thus no global control is actually needed.

Benchmark	Original	sonic + ECCE	ECCE	MIXTUS
advisor	1541 ms 1	483 ms 3.19	426 ms 3.62	471 ms
applast	1563 ms 1	491 ms 3.18	471 ms 3.32	1250 ms
doubleapp	1138 ms 1	700 ms 1.63	600 ms 1.90	854 ms
map.reduce	541 ms 1	100 ms 5.41	117 ms 4.62	383 ms
map.rev	221 ms 1	71 ms 3.11	83 ms 2.66	138 ms
match.kmp	4162 ms 1	1812 ms 2.30	3166 ms 1.31	2521 ms
matchapp	1804 ms 1	771 ms 2.34	1525 ms 1.18	1375 ms
maxlength	217 ms 1	283 ms 0.77	208 ms 1.04	213 ms
regexp.r1	3067 ms 1	396 ms 7.74	604 ms 5.08	
relative	9067 ms 1	17 ms 533.35	1487 ms 6.10	17 ms
remove	3650 ms 1	4466 ms 0.82	2783 ms 1.31	2916 ms
remove2	5792 ms 1	4225 ms 1.37	3771 ms 1.54	3017 ms
reverse	8534 ms 1	6317 ms 1.35	6900 ms 1.24	
rev_acc_type	37391 ms 1	26302 ms 1.42	26815 ms 1.39	25671 ms
rotateprune	7350 ms 1	5167 ms 1.42	5967 ms 1.23	5967 ms
ssupply	1150 ms 1	79 ms 14.56	92 ms 12.50	92 ms
transpose	1567 ms 1	67 ms –	67 ms –	67 ms
upto.sum1	6517 ms 1	4284 ms 1.52	4350 ms 1.50	4716 ms
upto.sum2	1479 ms 1	1008 ms 1.47	1008 ms 1.47	1008 ms

Table 8.3: Speed of the residual programs (in ms, for a large number of queries, interpreted code) and Speedups

9 Conclusion

The staging of a program's input, or alternatively its division into static and dynamic parts, is the fundamental basis of program specialisation. In specialisation of functional programs, this division is explicitly captured by classifying arguments as static or dynamic. Classification of arguments in the logic programming setting is less satisfactory, however, due to the ubiquity of partially instantiated data structures, particularly during partial deduction. An alternative approach is required to the problem of ensuring finite unfolding during partial deduction, specifically tailored to deal with the peculiarities of logic programming.

The viewpoint adopted here has been that a theory for termination of unfolding should arise naturally out of a theory for full termination of programs. In fact, the former should really be a generalisation of the latter given that full evaluation is simply a special case of partial evaluation. While this thesis has not sought to develop a theory of termination for partial deduction as such, it has developed the existing theory for full termination in order to provide a basis for a practical technique for ensuring finite unfolding.

To begin with, this has required focusing on the recursive structure of termination proofs. A first notion of "partial termination" can be obtained by considering the strongly connected components (SCCs) of the predicate dependency graph of a program. Loops can occur in any of the SCCs. If execution of a program leads to loops in some SCCs but not in others, the program may be said to partially terminate, and in terms of partial deduction, the non-looping SCCs can be unfolded.

The notions of bounded recurrency and bounded acceptability introduced in this thesis provide a foundation for the construction of termination proofs based on the recursive structure of programs. While facilitating proofs of full termination in general, the focus on recursion leads the way to considering termination of the individual SCCs when only partial input is supplied. Proofs based on recurrency or acceptability provide no support for this.

Since coroutining logic programs accurately model the unfolding process, developing a theory of termination for them is key to providing a theoretical underpinning for finite unfolding. The class of semi delay recurrent programs captures a useful subset of coroutining programs, where, as before, the emphasis is on the recursive structure to facilitate termination proofs and to allow individual SCCs to be considered. Moreover, programs which have been proven to be bounded acceptable can easily be transformed into semi delay recurrent versions. The advantage of doing so is that the strict left-to-right computation rule can be relaxed, and a more flexible one adopted without danger of non-termination. This gives the opportunity to use the transformed program as an "unfolding machine", which simply handles the unfolding of bounded goals, as part of a generating extension. This idea was taken a step further in the last chapter by extending the machine to also handle unbounded goals.

The most significant contribution of this thesis then, is in establishing a link be-

tween the fields of partial deduction and static termination analysis. A direct consequence of drawing on the static termination literature, rather than loop checking, say, is that the result lends itself naturally to offline partial deduction. The proof of concept is provided in the results of the previous chapter. The sonic approach represents a significant step forward in the offline partial deduction technology for logic programs, being the first offline approach to successfully unfold arbitrarily instantiated goals and, as a result, the first to pass the KMP test.

A full implementation of the proposed cogen together with comprehensive experimentation and benchmarking are now needed to drive the work forward. Even at this stage, however, there are a number of issues which remain unresolved some of which have arisen through the limited experimentation which has already been carried out.

- Without any determinacy control there is a possibility that the specialised program may be slower than the original. Clearly, then this is an important issue which must be addressed in the development of a practical specialiser. It is orthogonal to the termination issue, however, and as such there should not be any problem in incorporating determinacy control within the proposed framework.

- Having seen that the same specialised programs can be produced using different unfolding strategies (e.g. sonic vs. ECCE) raises the question of how the global and local control really interrelate. Obtaining the right balance could significantly affect the efficiency of the specialisation process.

- The techniques presented have been designed only for definite logic programs. There are a number of non-trivial issues relating to both termination and specialisation which would need to be addressed when extending the techniques to deal with normal logic programs.

- There is much potential for combining offline and online unfolding strategies to obtain more efficient and more powerful specialisers. How to combine the two and obtaining the right balance are non-trivial problems.

- This thesis has really only considered how to make the local control offline. No work has been done on effective offline global control for logic programs, and it remains to be seen whether or not there is much to be gained from this.

- The philosophy adopted in the design of the unfolding algorithms here has been "finitely unfold as much as possible". Determinacy issues aside, this may not always be desirable. Prolific unfolding may well lead to huge residual programs with no significant improvement in performance. Only extensive experimentation will reveal whether this philosophy is well founded or if there is a need to manage the code explosion/performance improvement tradeoff.

Bibliography

ALPUENTE, M., M. FALASCHI, P. JULIÁN, & G. VIDAL 1997. "Spezialisation of Lazy Functional Logic Programs", *in Proceedings of PEPM'97, the ACM Sigplan Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pp. 151–162, Amsterdam, The Netherlands. ACM Press.

ANDERSON, P.H. & C.K. HOLST 1996. "Termination analysis for offline partial evaluation of a higher order functional language", *in* Cousot, R. & Schmidt, D. A., (eds.), *Proceeding of the Third International Symposium on Static Analysis, SAS'96*, Lecture Notes in Computer Science 1145, pp. 67–82, Aachen, Germany. Springer-Verlag.

APT, K.R. & M. BEZEM 1990. "Acyclic programs", In Warren & Szeredi 1990, pp. 617–633.

APT, K.R., R.N. BOL, & J.W. KLOP 1989. "On the safe termination of Prolog programs", In Levi & Martelli 1989, pp. 353–368.

APT, K.R. & D. PEDRESCHI 1990. "Studies in pure Prolog: Termination", *in Proceedings Esprit Symposium on Computational Logic*, pp. 150–176, Brussels. Springer-Verlag.

APT, K.R. & D. PEDRESCHI 1994. "Modular termination proofs for logic and pure Prolog programs", *in* Levi, G., (ed.), *Proceedings of the Fourth International School for Computer Science Researchers*. Oxford University Press.

BENKERIMI, K. & J.W. LLOYD 1990. "A partial evaluation procedure for logic programs", In Debray & Hermenegildo 1990, pp. 343–358.

BENOY, F. & A. KING 1996. "Inferring argument size relations with CLP($\mathcal{R}$)", In Gallagher 1996, pp. 204–223.

BEZEM, M. 1989. "Characterizing termination of logic programs with level mappings", *in* Lusk, E. L. & Overbeek, R. A., (eds.), *Proceedings of the North American Conference on Logic Programming*, pp. 69–80, Cleveland, Ohio, USA. MIT Press.

BEZEM, M. 1993. "Strong termination of logic programs", *Journal of Logic Programming*, 15 (1 & 2): 79–97.

BIRKEDAL, L. & M. WELINDER 1994. "Hand-writing program generator generators", *in* Hermenegildo, M. & Penjam, J., (eds.), *Proceedings of the Sixth International Symposium on Programming Language Implementation and Logic Programming*, Lecture Notes in Computer Science 844, pp. 198–214, Madrid, Spain. Springer-Verlag.

BOL, R. N. 1991. *Loop checking in logic programming*. PhD thesis, CWI, Amsterdam. CWI Tract 112.

BOL, R. N. 1993. "Loop checking in partial deduction", *Journal of Logic Programming*, 16 (1 & 2): 25–46.

BOSSI, A., N. COCCO, & M. FABRIS 1992. "Typed norms", *in* Krieg-Brükner, (ed.), *Fourth European Symposium on Programming*, LNCS 582, pp. 73–92, Rennes, France. Springer-Verlag.

BOSSI, A., N. COCCO, & M. FABRIS 1994. "Norms on terms and their use in proving universal termination of a logic program", *Theoretical Computer Science*, 124: 297–328.

BROUGH, D. & A. WALKER 1984. "Some practical properties of a Prolog interpreter", *in Proceedings of the International Conference on Fifth Generation Computer Systems*, Tokyo, Japan. Association for Computing Machinery.

BRUYNOOGHE, M., D. DE SCHREYE, & B. MARTENS 1991. "A general criterion for avoiding infinite unfolding during partial deduction of logic programs", In Saraswat & Ueda 1991, pp. 117–131.

BRUYNOOGHE, M., D. DE SCHREYE, & B. MARTENS 1992. "A General Criterion for Avoiding Infinite Unfolding During Partial Deduction", *New Generation Computing*, 11 (1): 47–79.

BRUYNOOGHE, M., M. LEUSCHEL, & K. SAGONAS 1998. "A polyvariant binding-time analysis for off-line partial deduction", *in* Hankin, C., (ed.), *Proceedings of the European Symposium on Programming (ESOP'98)*, Lecture Notes in Computer Science 1381, pp. 27–41. Springer-Verlag.

BURSTALL, R.M. & J. DARLINGTON 1977. "A transformation system for developing recursive programs", *Journal of the ACM*, 24 (1): 44–67.

CAVEDON, L. 1989. "Continuity, consistency, and completeness properties for logic programs", In Levi & Martelli 1989, pp. 571–584.

CODISH, M. & C. TALBOCH 1997. "A semantic basis for termination analysis of logic programs and its realisation using symbolic norm constraints", *in* Hanus, M., Heering, J., & Meinke, K., (eds.), *Proceedings of the Sixth International Joint Conference on Algebraic and Logic Programming, ALP'97 – HOA'97*, Lecture Notes in Computer Science 1298, pp. 31–45, Southampton, UK. Springer-Verlag.

COLMERAUER, A., H. KANOUI, P. ROUSSEL, & R. PASERO 1973. "Un systeme de communication homme-machine en francais", Technical report, Groupe de recherche en Intelligence Artificielle, Université d'Aix-Marseille.

COVINGTON, M. A. 1985a. "Eliminating unwanted loops in Prolog", *SIGPLAN Notices*, 20 (1).

COVINGTON, M. A. 1985b. "A further note on looping in Prolog", *SIGPLAN Notices*, 20 (8).

DANVY, O., R. GLÜCK, & P. THIEMANN, (eds.) 1996. *Proceedings of the 1996 Dagstuhl seminar on partial evaluation*, Lecture Notes in Computer Science 1110, Schloß Dagstuhl. Springer-Verlag.

DE SCHREYE, D. 1998. "Personal communication".

DE SCHREYE, D. & S. DECORTE 1994. "Termination of logic programs: The never-ending story", *Journal of Logic Programming*, 19 & 20: 199–260.

DE SCHREYE, D., K. VERSCHAETSE, & M. BRUYNOOGHE 1992. "A framework for analysing the termination of definite logic programs with respect to call patterns", *in FGCS'92*, pp. 481–488. MIT Press.

DEBRAY, S. & M. HERMENEGILDO, (eds.) 1990. *Proceedings of the 1990 North American Conference on Logic Programming*, Austin. ALP, MIT Press.

DEBRAY, S.K. & N.-W. LIN 1991. "Automatic complexity analysis of logic programs", In Furukawa 1991, pp. 599–613.

DEBRAY, S.K., N.-W. LIN, & M. HERMENEGILDO 1990. "Task granularity analysis in logic programs", *in Proceedings ACM SIGPLAN'90 conference on programming language design and implementation*, pp. 174–188.

DECORTE, S. & D. DE SCHREYE 1997. "Demand-driven and constraint-based automatic left-termination analysis for logic programs", In Naish 1997, pp. 78–92.

DECORTE, S. & D. DE SCHREYE 1998. "Termination analysis: Some practical properties of the norm and level mapping space", *in* Jaffar, J., (ed.), *Proceedings of the 1998 Joint International Conference and Symposium on Logic Programming*, pp. 235–249, Manchester, England. The MIT Press.

DECORTE, S., D. DE SCHREYE, & M. FABRIS 1993. "Automatic inference of norms: A missing link in automatic termination analysis", *in* Miller, D., (ed.), *Proceedings of the 1993 International Logic Programming Symposium*, pp. 420–436, Vancouver, Canada. The MIT Press.

DECORTE, S., D. DE SCHREYE, & M. FABRIS 1994. "Exploiting the power of typed norms in automatic inference of interargument relations", Technical report, Dept. computer science, K.U.Leuven.

DERSHOWITZ, N. 1987. "Termination of Rewriting", *Journal of Symbolic Computation*, 3: 69–116.

DERSHOWITZ, N. & Z. MANNA 1979. "Proving termination with multiset orderings", *Communications of the ACM*, 22 (8): 465–476.

ERSHOV, A.P. 1982. "Mixed Computation: Potential applications and problems for study", *Theoretical Computer Science*, 18: 41–67.

FUCHS, NORBERT, (ed.) 1997. *Proceedings of the 7th International Workshop on Logic Program Synthesis and Transformation (LOPSTR'97)*, Lecture Notes in Computer Science 1463, Leuven, Belgium. Springer-Verlag.

FURUKAWA, KOICHI, (ed.) 1991. *Proceedings of the Eighth International Conference on Logic Programming*, Paris, France. The MIT Press.

FUTAMURA, Y. 1971. "Partial evaluation of a computation process - n approach to a compiler-compiler", *Systems, Computers, Controls*, 2 (5): 45–50.

GALLAGHER, J.P. 1993. "Tutorial on specialisation of logic programs", *in Proceedings of PEPM'93, the ACM Sigplan Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pp. 88–98. ACM Press.

GALLAGHER, J., (ed.) 1996. *Proceedings of the 6th International Workshop on Logic Program Synthesis and Transformation (LOPSTR'96)*, Lecture Notes in Computer Science 1207, Stockholm, Sweden. Springer-Verlag.

GALLAGHER, J. & A. DE WAAL 1994. "Fast and precise regular approximations of logic programs", *in* Van Hentenryck, P., (ed.), *Proceedings of the Eleventh International Conference on Logic Programming*, pp. 599–613, Massachusetts Institute of Technology. The MIT Press.

GALLAGHER, J.P. & L. LAFAVE 1996. "Regular approximation of computation paths in logic and functional languages", In Danvy *et al.* 1996, pp. 115–136.

GLÜCK, R., J. JØRGENSEN, B. MARTENS, & M.H. SØRENSEN 1996. "Controlling conjunctive partial deduction of definite logic programs", *in* Kuchen, H. & Swierstra, S., (eds.), *Proceedings of the International Symposium on Programming Languages, Implementations, Logics and Programs (PLILP'96)*, LNCS 1140, pp. 152–166, Aachen, Germany. Springer-Verlag. Extended version as Technical Report CW 226, K.U. Leuven. Accessible via `http://www.cs.kuleuven.ac.be/~dtai`.

GRÖGER, G. & L. PLÜMER 1992. "Handling of mutual recursion in automatic termination proofs for logic programs", *in* Apt, K., (ed.), *Proceedings of the 1992 Joint International Conference and Symposium on Logic Programming*, pp. 336–350, Washington, USA. The MIT Press.

GURR, C.A. 1994. *A self-applicable partial evaluator for the logic programming language Gödel*. PhD thesis, University of Bristol.

GURR, C.A. 1995. "Personal communication on the literature on termination analyses".

HAREL, D. 1989. *The Science of Computing*. Addison-Wesley.

HILL, P.M. & J.W. LLOYD 1994. *The Gödel programming language*. MIT Press.

HOLST, C.K. 1991. "Finiteness Analysis", *in* Hughes, J., (ed.), *Functional Programming Languages and Computer Architectures*, Lecture Notes in Computer Science 523, pp. 473–495, Cambridge, Massachusetts, USA. Association for Computing Machinery, Springer-Verlag.

JANSSENS, G. & M. BRUYNOOGHE 1992. "Deriving Descriptions of Possible Values of Program Variables by Means of Abstract Interpretation", *Journal Logic Programming*, 13: 205–258.

JONES, N.D., C.K. GOMARD, & P. SESTOFT 1993. *Partial evaluation and automatic program generation*. Prentice Hall.

JØRGENSEN, J. & M. LEUSCHEL 1996. "Efficiently generating efficient generating extensions in Prolog", In Danvy *et al.* 1996, pp. 238–262. Extended version also available as Technical Report CW 221, K.U. Leuven. Accessible via `http://www.cs.kuleuven.ac.be/~dtai`.

JØRGENSEN, J., M. LEUSCHEL, & B. MARTENS 1996. "Conjunctive partial deduction in practice", In Gallagher 1996, pp. 59–82. Also in the Proceedings of BENELOG'96. Extended version as Technical Report CW 242, K.U. Leuven.

KAWAMURA, T. & T. KANAMORI 1988. "Preservation of stronger equivalence in unfold/fold logic program transformation", *in* Institute for New Generation Computer Technology (ICOT), (ed.), *Proceedings of the International Conference on Fifth Generation Computer Systems*, 2, pp. 413–421, Tokyo, Japan. Springer-Verlag.

KNUTH, D.E., J.H. MORRIS, & V.R. PRATT 1977. "Fast pattern matching in strings", *SIAM Journal of Computation*, 6 (2): 323–350.

KOMOROWKSI, J. 1981. *A specification of an abstract Prolog machine and its application to partial evaluation.* PhD thesis, Linköping University, Sweden. Linköping Studies in Science and Technology Dissertations 69.

KOMOROWSKI, J. 1992. "An introduction to partial evaluation", *in* Pettorossi, A., (ed.), *Proceedings Meta'92*, Lecture Notes in Computer Science 649, pp. 49–69. Springer-Verlag.

KOWALSKI, R.A. 1974. "Predicate logic as a programming language", *in Information Processing 74*, pp. 569–574, Stockholm. North-Holland Pub. Co.

KOWALSKI, R.A. 1979. "Algorithm = Logic + Control", *Communications of the ACM*, 22 (7): 424–436.

LAFAVE, L. & J. GALLAGHER 1997. "Constraint-based partial evaluation of rewriting-based functional logic programs", In Fuchs 1997, pp. 168–188.

LEUSCHEL, M. 1996. "The ECCE partial deduction system and the DPPD library of benchmarks". Obtainable via `http://www.ecs.soton.ac.uk/~mal`.

LEUSCHEL, M. 1997. *Advanced Techniques for Logic Program Specialisation.* PhD thesis, K.U. Leuven. Accessible via `http://www.ecs.soton.ac.uk/~mal`.

LEUSCHEL, M. 1998. "On the power of homeomorphic embedding for online termination", *in* Levi, G., (ed.), Static Analysis. *Proceedings of SAS'98*, LNCS 1503, pp. 230–245, Pisa, Italy. Springer-Verlag.

LEUSCHEL, M., D. DE SCHREYE, & A. DE WAAL 1996. "A conceptual embedding of folding into partial deduction: Towards a maximal integration", *in* Maher, M., (ed.), *Proceedings of the 1996 Joint International Conference and Symposium on Logic Programming*, pp. 319–332, Bonn, Germany. The MIT Press. Extended version available as Technical Report CW 225, K.U. Leuven. Accessible via `http://www.cs.kuleuven.ac.be/~dtai`.

LEUSCHEL, M. & B. MARTENS 1996. "Global control for partial deduction through characteristic atoms and global trees", In Danvy *et al.* 1996, pp. 263–283. Extended version available as Technical Report CW 220, K.U. Leuven. Accessible via `http://www.cs.kuleuven.ac.be/~dtai`.

LEUSCHEL, M., B. MARTENS, & D. DE SCHREYE 1998. "Controlling generalisation and polyvariance in partial deduction of normal logic programs", *ACM Transactions on Programming Languages and Systems*, 20 (1): 208–258.

LEVI, G. & M. MARTELLI, (eds.) 1989. *Proceedings of the Sixth International Conference on Logic Programming*, Lisbon. The MIT Press.

LINDENSTRAUSS, N. & Y. SAGIV 1997. "Automatic termination analysis of logic programs", In Naish 1997, pp. 63–77.

LLOYD, J.W. 1987. *Foundations of Logic Programming*. Springer-Verlag, second edition.

LLOYD, J.W., (ed.) 1995. *Proceedings of the 1995 International Logic Programming Symposium*, Portland, USA. The MIT Press.

LLOYD, J. W. & J. C. SHEPHERDSON 1991. "Partial evaluation in logic programming", *Journal of Logic Programming*, 11: 217–242.

LÜTTRINGHAUS-KAPPEL, S. 1993. "Control generation for logic programs", *in* Warren, D. S., (ed.), *Proceedings of the Tenth International Conference on Logic Programming*, pp. 478–495, Budapest, Hungary. The MIT Press.

MARCHIORI, E. 1996. "Personal communication".

MARCHIORI, E. & F. TEUSINK 1995. "Proving termination of logic programs with delay declarations", In Lloyd 1995, pp. 447–461.

MARCHIORI, E. & F. TEUSINK 1996. "Proving deadlock freedom of logic programs with dynamic scheduling", *in* Boer, F. & M.Gabbrielli, (eds.), *JICSLP'96 Post-Conference Workshop W2 on Verification and Analysis of Logic Programs*, Bonn. TR-96-31, University of Pisa, Italy.

MARTENS, B. & D. DE SCHREYE 1996. "Automatic finite unfolding using well-founded measures", *Journal of Logic Programming*, 28 (2): 89–146.

MARTENS, B., D. DE SCHREYE, & T. HORVÁTH 1994. "Sound and complete partial deduction with unfolding based on well-founded measures", *Theoretical Computer Science*, 122 (1–2): 97–117.

MARTIN, J.C. & A. KING 1997. "Generating efficient, terminating logic programs", *in* Bidoit, M. & Dauchet, M., (eds.), *Proceedings of the Seventh International Joint Conference on Theory and Practice of Software Development (TAPSOFT'97)*, Lecture Notes in Computer Science 1214, Lille, France. Springer-Verlag.

MARTIN, J.C., A. KING, & P. SOPER 1996. "Typed norms for typed logic programs", In Gallagher 1996, pp. 224–238.

MARTIN, J.C. & M. LEUSCHEL 1999. "Sonic partial deduction", *in Proceedings of the Third International Ershov Conference on Perspectives of System Informatics*, pp. 101–112, Lecture Notes in Computer Science 1755, Novosibirsk, Russia. Springer-Verlag.

MESNARD, F. 1995. "Towards Automatic Control for CLP($\mathcal{X}$) Programs", *in LOPSTR'95*. Springer-Verlag.

NAISH, L. 1993. "Coroutining and the construction of terminating logic programs", *in Australian Computer Science Conference*, Brisbane.

NAISH, L., (ed.) 1997. *Proceedings of the Fourteenth International Conference on Logic Programming*, Leuven, Belgium. The MIT Press.

NUTE, D. 1985. "A programming solution to certain problems with loops in Prolog", *SIGPLAN Notices*, 20 (8).

PETTOROSSI, A., M. PROIETTI, & S. RENAULT 1996. "Enhancing partial deduction via unfold/fold rules", In Gallagher 1996, pp. 146–168.

PLÜMER, L. 1990a. *Termination proofs for logic programs.* Lecture Notes in Artificial Intelligence 446. Springer-Verlag.

PLÜMER, L. 1990b. "Termination proofs for logic programs based on predicate inequalities", In Warren & Szeredi 1990, pp. 634–648.

PLÜMER, L. 1991. "Automatic termination proofs for Prolog programs operating on nonground terms", In Saraswat & Ueda 1991, pp. 503–517.

POOLE, D. & R. GOEBEL 1985. "On eliminating loops in Prolog", *SIGPLAN Notices*, 20 (8).

SAHLIN, D. 1993. "Mixtus: An automatic partial evaluator for full Prolog", *New Generation Computing*, 12 (1): 7–51.

SARASWAT, V. & K. UEDA, (eds.) 1991. *Proceedings of the 1991 International Logic Programming Symposium*, San Diego, USA. MIT Press.

SEDGEWICK, R. 1990. *Algorithms in C.* Addison-Wesley.

SEKI, H. 1989. "Unfold/fold transformation of stratified programs", In Levi & Martelli 1989, pp. 554–568.

SICS 1995. *SICStus Prolog User's Manual.* Intelligent Systems Laboratory, SICS, PO Box 1263, S-164 28 Kista, Sweden. Accessible via SICStus Prolog home page at `http://www.sics.se/ps/sicstus.html`.

SØRENSEN, M.H. & R. GLÜCK 1995. "An algorithm of generalization in positive supercompilation", In Lloyd 1995, pp. 465–479.

TAMAKI, H. & T. SATO 1984. "Unfold/Fold Transformations of Logic Programs", *in* Tärnlund, S.-Å., (ed.), *Proceedings of the Second International Conference on Logic Programming*, pp. 127–138, Uppsala, Sweden.

ULLMAN, J. D. & A. VAN GELDER 1988. "Efficient tests for top-down termination of logical rules", *Journal ACM*, 35 (2): 345–373.

VAN GELDER, A. 1987. "Efficient loop detection in Prolog using the Tortoise-and-Hare technique", *Journal of Logic Programming*, 4 (1): 23–32.

VAN HENTENRYCK, P., A. CORTESI, & B. LE CHARLIER 1994. "Type Analysis of Prolog Using Type Graphs", *in PLDI'94*, pp. 337–348. ACM Press.

VAN LEEUWEN, J., (ed.) 1990. *Handbook of Theoretical Computer Science: Volume B.* Elsevier.

VAN ROY, P. 1984. "A Prolog Compiler for the PLM", Master's thesis, Computer Science Division, University of California, Berkeley.

VANHOOF, W. & B. MARTENS 1997. "To Parse or Not To Parse", In Fuchs 1997, pp. 322–342. Also as Technical Report CW 251, K.U.Leuven.

VASAK, T. & J. POTTER 1986. "Characterisation of terminating logic programs", *in IEEE Symposium on Logic Programming*, pp. 140–147.

VERSCHAETSE, K. & D. DE SCHREYE 1991. "Deriving termination proofs for logic programs, using abstract procedures", In Furukawa 1991, pp. 301–315.

VERSCHAETSE, K., S. DECORTE, & D. DE SCHREYE 1992. "Derivation of linear size relations by abstract interpretation", *in* Bruynooghe, M. & Wirsing, M., (eds.), *Proceedings of the Fourth International Symposium on Programming Language Implementation and Logic Programming*, Lecture Notes in Computer Science 631, Leuven, Belgium. Springer-Verlag.

VIEILLE, L. 1989. "Recursive query processing: The power of logic", *Theoretical Computer Science*, 69 (1): 1–53.

WARREN, D.H.D. & P. SZEREDI, (eds.) 1990. *Proceedings of the Seventh International Conference on Logic Programming*, Jerusalem. The MIT Press.