Compositional Logic Programming

Richard McPhee

Worcester College

Submitted for the degree of Doctor of Philosophy

Trinity Term 2000

Oxford University Computing Laboratory
Programming Research Group

Compositional Logic Programming

Richard McPhee
Worcester College

Submitted for the degree of Doctor of Philosophy
Trinity Term 2000

Abstract

Relational program derivation has gathered momentum over the last decade with the development of many specification logics. However, before such relational specifications can be executed in existing programming languages, they must be carefully phrased to respect the evaluation order of the language. In turn, this requirement inhibits the rapid prototyping of specifications in a relational notation. The aim of this thesis is to bridge the gap between the methodology and practice of relational program derivation by realising a compositional style of logic programming that permits specifications to be phrased naturally and executed declaratively.

The first contribution of this thesis is the identification of a collection of desiderata that sets out the particular language requirements necessary to support our notion of compositionality. Thus, from the outset, we differentiate between the execution of specifications and programs, the latter an enterprise best left to the likes of Prolog and Mercury.

Compositionality is obtained in this thesis by translating higher-order elements of functional programming style into the logic paradigm. By doing so, these elements become enriched with the extra expressiveness fostered by nondeterminism, logic variables, and program converses. Another contribution of this thesis is the demonstration that a curried representation of programming terms in a compositional logic language is sufficient to provide the desired higher-order facilities without the need for either extra-logical predicates or higher-order unification.

A further contribution of this thesis is the rediscovery of fair SLD-resolution as a fundamental way to guarantee termination of compositional programs within the confines of resolution. Unfortunately, though, fair SLD-resolution using the 'breadth-first' computation rule exhibits efficiency behaviour that is difficult to predict. Consequently, this thesis proposes and implements two novel versions of fair SLD-resolution that overcome the deficiencies of the breadth-first computation rule.

The first strategy, based on a new formulation of tabled evaluation, restores efficiency by eliminating redundant computation and also inherits the extra termination benefits intrinsic to tabled evaluation. The second strategy, called prioritised fair SLD-resolution, selects literals in a goal from those whose resolution is known to terminate at the expense of the others. Prioritised resolution centres around an original adaptation of an existing termination analysis for logic programs. Although termination analysis has recently been used to verify the correctness of logic programs, its application in this thesis to improve the efficiency of compositional programs is new.

To my parents, Elizabeth and Roderick

Care, mad to see a man sae happy,
E'en drown'd himsel amang the nappy!
As bees flee hame wi' lades o' treasure,
The minutes wing'd their way wi' pleasure:
Kings may be blest, but Tam was glorious.
O'er a' the ills o' life victorious!

Robert Burns, *Tam O'Shanter*.

Contents

Acknowledgements

Many people deserve thanks for their help during my years at Oxford, none more so than Oege de Moor for his continual support and guidance. I would also like to thank Mike Spivey for supervising me whilst Oege was on sabbatical at Microsoft. The encouragement offered by the members of the Algebra of Programming Research Group helped greatly: Richard Bird, Sharon Curtis, and Jesús Ravelo, in particular. Special thanks are due to Naomi Lindenstrauss and Alexander Serebrenik for invaluable conversations regarding termination analysis and detailed comments on Chapter 5. Other people who have helped along the way by discussing ideas and reading portions of this thesis are: Jonathan Hill, Christopher Hogger, Graham Hutton, Geraint Jones, Jim Lipton, John Lloyd, Wayne Luk, Corin Pitcher, Olivier Ridoux, Andrew Simpson, Charles Stewart, and David S. Warren. I am indebted to Richard Bird and Graham Hutton for their fastidious examination of the thesis and for their numerous comments that helped improve the quality of the final draft.

More people than I can mention have made my time at Oxford a happy one, although the following handful deserve particular thanks: Adrian and Emma, Charles, Hendrik, Dave K., Andy, Jon, Ali, Charlie, Brian, Tori, Chuck and Pia, Karl and Laura, Dave G., Jim and Jim. I would like to thank Kate beyond words for her unending support, patience, and love during the writing of this thesis. Finally, I owe everything to my family: Rob, Sheila, Tony, and my parents, Rod and Liz, especially.

Chapter 1

Introduction

One of the primary strengths of functional programming languages is their ability to split up programs into small, mind-sized pieces. Through the use of higher-order operators, or *combinators*, one can capture common idioms of program composition. Apart from the aesthetic benefits and the increased readability of programs, employing combinators encourages the formulation of general mathematical results about programs (Backus 1985). In turn, such results facilitate concise verification of the equivalence of programs and permit the derivation of functional programs from mathematical specifications (Bird 1987). Also, because combinators are defined by the user, they can be tailored to specific applications: for example, to parsing (Hutton 1992*b*), pretty-printing (Hughes 1995), graphical user interfaces (Hallgren & Carlsson 1995), or capturing patterns of parallel computation (Darlington, Field, Harrison, Kelly, Sharp, Wu & While 1993).

Recently, members of the program derivation community (Bird & de Moor 1996, Berghammer & Von Karger 1995, Aarts, Backhouse, Hoogendijk, Voermans & Van der Woude 1992, Jones & Sheeran 1990) have shown that it is often useful to use a relational calculus to derive programs from specification, rather than a purely functional one, since a relational calculus provides a natural treatment of nondeterminism and function converses; these two elements combine to form a calculus whose primitive constructs are more powerful than those available in a functional one. Furthermore, the flexibility of a relational calculus can help to clarify and simplify the structure of proofs made in the calculus (Bird & de Moor 1994).

The main drawback is that, of all the relational calculi being developed (Bird & de Moor 1996, Möller 1993, Backhouse & Hoogendijk 1993, Jones & Sheeran 1990, Schmidt & Ströhlein 1988), no language exists in which the relational expressions being derived can be directly executed. Two notable exceptions are (Cattrall 1993, Hutton 1992*a*), although both of these are more concerned with executing certain types of *programs* than with the prototyping of *specifications*. The absence of a widely accepted notion of relational computation prohibits the prototyping of specifications in a relational notation and hinders the understanding of why some relational transformations lead to an efficient program and others do not. Of the existing declarative programming paradigms, the logic paradigm allows direct execution of very high-level specifications—through the application of nondeterminism and program converses—whilst the functional paradigm requires a specification to be phrased more algorithmically due to the distinction between input and output. However, functional languages provide more support for composing specifications.

The logic programming community has also recognised the benefits of compositionality and the issue has been explored in the design of functional logic (Hanus, Kuchen & Moreno-Navarro 1995, Lloyd 1995, Hanus 1994, Moreno-Navarro & Roderiguez-Artalejo 1992, Reddy 1987, Bellia & Levi 1986) and higher-order logic languages (Nadathur & Miller 1988). Moreover, this work focuses on executable languages and even addresses questions of efficient execution. Nevertheless, such languages still fail to support the direct execution of the relational expressions derived in the calculi above.

The main deficiency of these new logic languages and, indeed, the traditional languages like Prolog (Colmerauer, Kanoui, Pasero & Roussel 1973) and Mercury (Somogyi, Henderson & Conway 1995), for executing relational programs stems from their predominant use of the left-to-right computation rule which effectively bans the use of relational composition and converse. Programs executed in such languages often loop infinitely, unless extra-logical annotations are added to them. By doing so, however, the declarative underpinnings of relational programming are lost, as is the ability to reason abstractly about such programs.

Despite the widespread use of the left-to-right computation rule in the logic programming community, its limitations have been acknowledged by the introduction of coroutining computation rules (Lüttringhaus-Kappel 1993, Naish 1992, Naish 1985) in languages such as NU-Prolog (Thom & Zobel 1987). Coroutining rules attempt to overcome the inflexibility of the left-to-right one by selecting literals in a goal depending upon the pattern of instantiation of their arguments. The result is increased flexibility that allows more programs to terminate than does the left-to-right computation rule. However, automatically determining precisely which instantiation patterns for a program will lead to its efficient execution is a difficult undertaking (Lüttringhaus-Kappel 1992). Consequently, a coroutining logic language can often rely too heavily on the programmer's particular choice of instantiation patterns to facilitate efficient computation.

An alternative approach to address the shortcomings of the left-to-right computation rule when executing compositional programs was presented by (Hamfelt & Nilsson 1996, Nilsson & Hamfelt 1995) who abandoned explicit recursion in favour of providing structurally recursive combinators over the list data type. To encapsulate recursion of this nature in Prolog, (Hamfelt & Nilsson 1996) provided multiple versions of each combinator and used groundness analysis (Apt & Etalle 1993) to determine at run-time the most appropriate implementation to use during the resolution of the selected literal. The disadvantage of such an approach to realising compositional logic programming is that every recursive data type must be analysed to determine all of its possible combinator forms and, furthermore, their use is restricted to ground arguments. Moreover, despite the virtues of compositionality, there are occasions when programming only with combinators is inconvenient and explicit recursion is preferable: an observation that is also true in functional programming.

Another deficiency of most existing logic languages, that becomes evident when writing compositional programs in them, is their lack of adequate support for higher-order programming constructs. One recent attempt to rekindle interest in higher-order logic programming in Prolog (Naish 1996) suggests once more the use of the extra-logical construct first introduced by (Warren 1982) which has largely been overlooked by the community over the years. This construct provides a more faithful analogy to the higher-order programming style found in the functional paradigm than the one adopted by languages like Mercury. Nevertheless, the application of cumbersome extra-logical predicates by the programmer hinders the declarative formulation of programs. Furthermore, any style of higher-order programming in Prolog inherits the left-to-right computation rule and, in turn, the poor termination behaviour of compositional programs.

The major exception to the lack of support for compositionality in logic programming is provided by the higher-order language λProlog (Nadathur & Miller 1988) which allows higher-order programs to be constructed naturally, without the need for extra-logical predicates. This advantage over Prolog led (Gegg-Harrison 1995) to propose a general characterisation of structural recursion in λProlog, suitable for certain common data types, through the provision of a handful of combinators. Although these combinators form a solid basis for writing compositional programs, the inheritance of the left-to-right computation rule in λProlog once again hinders their execution.

All the above approaches to support compositional programming are based upon the computational process of resolution. However, alternative computation mechanisms have recently been suggested for logic programming. For example, (Lipton & Broome 1994) propose a computational model based upon rewriting relational expressions directly using a set of relational axioms. This line of work, though, is still in development and it remains to be seen whether it can provide an effective method of evaluating relational expressions.

Contributions of this thesis

The aim of this thesis is to design and implement a compositional logic programming language that overcomes the problems of executing relational specifications in existing logic languages and, ultimately, to bridge the gap between the methodology and practice of relational programming. In the following definition, we pin down the notion of compositionality adopted in this thesis.

Definition 1.1 (Compositional logic programming). A *combinator* is a predicate that takes one or more partially applied predicates as arguments. A *compositional logic programming language* allows programs to be constructed using both the primitive connectives of the language and user-defined combinators such that the termination behaviour of the program is independent of the way in which it is composed. $\diamondsuit$

The first contribution of this thesis is the identification and realisation of a compositional style of logic programming. Compositional logic languages benefit from being able to execute high-level specifications that are bereft of the algorithmic detail commonly required in existing programming languages. Consequently, a compositional logic language frees the programmer from the burden of writing programs with regard to the language's execution order. The downside of this declarative style, however, is that a compositional language can never be expected to compete in terms of efficiency with programming languages that have a rigid evaluation order. Even though such languages are unable to execute naturally the style of programs proposed in this thesis, from the outset we emphasise that the purpose of a compositional language is to prototype specifications, rather than to execute algorithms; once a specification is refined to an actual program, it should be implemented in an efficient programming language designed for that purpose.

Despite the attempts discussed earlier to introduce higher-order programming into the logic community, its limited acceptance can be attributed to an incomplete analysis of the language features necessary to support faithfully such a style of logic programming. This thesis identifies, for the first time, a particular set of requirements that complements compositional logic programming. One such requirement is the curried representation of terms in the language to allow the partial application of predicates. True, Prolog permits this style of partial application—albeit as something of an afterthought—though it requires the cumbersome use of extra-logical predicates. The use of curried terms permits the majority of higher-order programming constructs required for compositionality, yet without the need for unnatural syntax. Moreover, the impact of curried terms on the operational mechanisms of a logic language, in particular the unification algorithm, is insignificant. Although the higher-order logic language λProlog also supports a curried syntax, the price paid for this is the adoption of higher-order unification which is an undecidable procedure in general. The curried representation of terms proposed in this thesis permits a compositional style of programming without the need for either extra-logical predicates or higher-order unification.

Another contribution of this thesis is the identification that a *fair* computation rule is crucial for the support of compositionality. We rediscover fair SLD-resolution (Lassez & Maher 1984) from logic programming and prove that it guarantees the termination of a logic program whenever any other resolution strategy would terminate for it. Therefore, fair SLD-resolution rectifies the infinite looping of compositional programs often experienced in Prolog whenever possible within the confines of resolution. Despite its theoretical elegance, however, fair SLD-resolution using the 'breadth-first' computation rule, as opposed to Prolog's depth-first (left-to-right) one, results in efficiency behaviour that is difficult to predict. Moreover, a prior attempt to rectify the efficiency problems of fair SLD-resolution (Janot 1991) fails to generalise to compositional programs. This thesis contributes two novel implementations of resolution that provide better support for compositional programming than existing techniques.

The first method, based on recording (or tabling) the results of individual resolutions, extends existing work on the tabled evaluation of logic programs (Chen, Swift & Warren 1995, Warren 1992, Tamaki & Sato 1986) to improve the termination behaviour of fair SLD-resolution in practice. Such behaviour is achieved both by eliminating redundant subcomputations in a resolution and by expanding the theoretical class of terminating programs beyond that of SLD-resolution. Secondly, we suggest a novel fair computation rule that selects literals in a goal depending on whether their resolution is known to terminate. We achieve this by adapting an existing termination test for logic programs (Lindenstrauss & Sagiv 1997) to use the new computation rule and we craft a static analysis of logic programs to determine the set of terminating literals for a program. Although termination analysis has recently been used to verify the correctness of logic programs (Speirs, Somogyi & Søndergaard 1997), the application of it to improve the execution efficiency of logic programs is novel.

In this thesis, only definite programs—those free of occurrences of negation—are considered in order to keep the discussion focussed on the important issues regarding compositionality. However, the compositional languages presented in this thesis can be extended to encompass normal programs—those that may contain negative literals—by implementing negation-as-failure through the use of a "safe" computation rule (Lloyd 1987). A safe computation rule selects negative literals for resolution only when they are ground so as to avoid introducing unsound answer substitutions. A fair computation rule can be extended with the notion of safeness by simply delaying the resolution of all negative literals until they are ground and, once selected for resolution, must be resolved away completely so that the success or failure of its resolution can be noted.

Overview of the thesis

The thesis is organised into seven chapters. In the remainder of this chapter, we present two substantial programming examples—implemented in the actual compositional language developed in this thesis—to demonstrate the desirability of the novel style of compositional logic programming. Previous attempts to promote a combinatory style of logic programming (Hamfelt & Nilsson 1996, Gegg-Harrison 1995) have lacked adequate examples which illustrate the practical benefits of constructing programs from combinators. The first example provides a set of combinators that serves to build both parsers and pretty-printers simultaneously, and the second is an implementation of the relational hardware description language Ruby (Jones & Sheeran 1990).

Chapter 2 establishes a collection of desiderata for a compositional programming language and uses it to examine the suitability of various programming paradigms for supporting declarative relational programs. The paradigms considered are functional, logic, and higher-order logic programming. After the investigation, we explain why each of these paradigms fails to satisfy fully our requirements for a relational programming language.

In Chapter 3, we identify the language features necessary to support relational programming in a logic language: a curried representation of terms which allows a natural syntax for compositional programs; and a fair computation rule which guarantees the termination of programs whenever possible by any other resolution strategy. We also explain why the choice of search strategy in a compositional language is orthogonal to the choice of computation rule. We then show that despite the elegant theoretical benefits of fair SLD-resolution, its efficiency behaviour in practice is difficult to predict. We end the chapter by discussing why coroutining computation rules fail to provide an adequate basis for compositional programming.

In Chapter 4, we review the notion of tabled evaluation of logic programs (Tamaki & Sato 1986) and introduce 'fair' tabling techniques, called fair OLDT-resolution, to overcome the practical limitations of fair SLD-resolution experienced when executing compositional programs. Tabling is able to achieve this by eliminating redundant computation and avoiding the cycles that are intrinsic to left-recursive programs. Fair tabling techniques also address the termination problems experienced by existing tabling systems that inherit the left-to-right computation rule from Prolog. However, the drawback is that fair tabled evaluation loses the desirable termination property of fair SLD-resolution which guarantees the termination of a query should any other resolution strategy terminate for it.

In Chapter 5, we reinstate the termination property of fair SLD-resolution yet improve upon its efficiency behaviour by developing a novel fair computation rule called prioritised fair SLD-resolution. We present a new computation rule that selects literals for resolution if its resolution is known to terminate; this determination is made via an original adaptation of an existing termination test for logic programs (Lindenstrauss & Sagiv 1997).

The penultimate chapter presents a comparison between the performance of four resolution strategies—SLD-resolution, fair SLD-resolution, fair OLDT-resolution, and prioritised fair SLD-resolution—when executing various compositional programs in terms of the number of resolution steps taken to complete their computation successfully. Finally, Chapter 7 presents the conclusions of the thesis and suggests possible areas of future work.

Three appendices to the thesis contain respective implementations of the resolution strategies presented in Chapters 3, 4, and 5 in an effort to formalise the important algorithms and data structures presented in the thesis, allow comparisons to be made between the three strategies, and provide a platform from which more efficient implementations can be launched.

1.1 Combinator parsing and pretty-printing

In functional programming, the preferred method of writing parsers is through the use of *parsing combinators* (Hutton 1992*b*). The reasons for this preferential position are the speed of creating such parsers, the simplicity of their structure, and the ease with which the parsers can be modified. Combinator parsers in functional languages make essential use of compositionality and higher-order programming. Ambiguous grammars are dealt with by appealing to the classic "list of successes" approach (Wadler 1985), which encapsulates a form of backtracking. Traditional logic programming languages are, for the most part, unsuitable for a combinatory approach to programming since features like currying and higher-order programming are not generally supported.

In this example, it is shown how our compositional logic programming language proves adequate for implementing combinator parsers. As we shall see, our parsers are also more symmetrical than their functional counterparts; later, we run our parsers "backwards"—taking full advantage of the expressive power of the logic language—thus producing a pretty-printer for our grammar without expending any additional programming effort.

Primitive parsers

The structural simplicity of combinator parsers stems from the hierarchical nature in which they are derived; from a discrete set of primitive parsing constructs, we can build elaborate parsers for sophisticated grammars. Each parser is represented by a predicate in the logic program. The "input" to our parsers is a list of characters and the "output" consists of both the converted part of the input string, called the *value*, and the unconsumed remainder of the input. A pair is represented by (a, b) in the language and lists are constructed by the symbols *nil*, which denotes the empty list, and ':', which denotes infix cons.

The first primitive parser is

$$\mathit{succeed}\ V\ X\ V\ X\ \ :-\ \ .$$

that consumes no input and always succeeds with some predetermined value V. In a logic programming language, we do not require a parser that always fails; we simply inherit the failure of a parser from the lack of a proof for a parse, i.e., if a parser fails then the interpreter simply indicates *no*. The parser

$$\mathit{satisfy}\ P\ (A : X)\ V\ Y\ \ :-\ \ P\ A,\ \mathit{succeed}\ A\ X\ V\ Y\ .$$

provides a good illustration of this phenomenon since any A that does not satisfy the predicate P will result in failure of *satisfy*.

Another primitive parser

$$\mathit{literal}\ A\ X\ V\ Y\ \ :-\ \ \mathit{satisfy}\ (\mathit{eq}\ A)\ X\ V\ Y\ .$$

succeeds when the first character in the input string X matches A. The predicate *eq* determines the equality of two terms by unifiability. The curried use of *eq* above demonstrates a particularly convenient syntactic notation corresponding to that found in functional programming.

By now the explicit replication of logic variables in our predicate definitions may appear tedious: after all, in functional programming we simply omit redundant parameters. Indeed, there is no fundamental reason for preventing this form of η-conversion of terms in our language. Such syntactic changes to our language compliment compositionality and their inclusion in a complete language implementation would be desirable.

Parsing combinators

One of the aims of combinator parsing is to facilitate an easy translation from a BNF description of a grammar to its parser. To achieve this, we provide a number of higher-order predicates, taking parsers to parsers, that correspond to the various constructs of BNF. For instance, we can define the choice operator of BNF as the higher-order predicate *or*:

$$\begin{array}{l}(P_1\ \$\mathit{or}\ P_2)\ X\ V\ Y\ \ :-\ \ P_1\ X\ V\ Y.\\(P_1\ \$\mathit{or}\ P_2)\ X\ V\ Y\ \ :-\ \ P_2\ X\ V\ Y.\end{array}$$

The unary operator '$' permits the infix use of a predicate. Additionally, we define the sequential composition of two parsers using the predicate *then*:

$$(P_1\ \$\mathit{then}\ P_2)\ X\ (\mathit{pair}\ V\ W)\ Y\ \ :-\ \ P_1\ X\ V\ Z,\ P_2\ Z\ W\ Y.$$

To build useful parsers, we provide for the direct manipulation of values returned by a parser. For example, as we proceed with a parse we could envisage the construction of an abstract syntax tree or some immediate semantic evaluation of the parsed expressions. The parser P $\$\mathit{using}$ F performs this task by applying the arbitrary predicate F to the value returned by the parser P. The new value is then passed on to the next stage in the parse.

$$(P\ \$\mathit{using}\ F)\ X\ V\ Y\ \ :-\ \ P\ X\ W\ Y,\ F\ W\ V.$$

The following parser *many* embodies the repetitious application of some parser P to the input, analogous to the BNF construct P^*.

$$\begin{array}{ll}\mathit{many}\ P\ X\ V\ Y\ \ :-\ \ (((P\ \$\mathit{then}\ (\mathit{many}\ P))\ \$\mathit{using}\ \mathit{cons})\ \mathit{or}\\ \qquad\qquad\qquad\qquad\qquad(\mathit{succeed}\ \mathit{nil}))\ X\ V\ Y.\end{array}$$

The predicate *cons* is defined as $\mathit{cons}\ (A, X)\ (A : X)\ \ :-\ .$ As well as the construct P^*, BNF also provides the notation P^+ which selects strictly greater than zero items from term P. The corresponding parser *some* is a simple variant of the parser *many* and is defined below:

$$some\ P\ X\ V\ Y\ \ :-\ \ ((P\ \$then\ (many\ P))\ \$using\ cons)\ X\ \ V\ \ Y.$$

A grammar invariably contains certain reserved words and symbols which, although vital for guiding the logical structure of expressions in the parse, play no further role after performing this duty. Therefore, it is useful to discard the values of these items during the course of a parse. The two combinators *xthen* and *thenx* perform exactly this task and are defined below:

$$\begin{aligned}(P_1\ \$xthen\ P_2)\ X\ \ V\ \ Y\ \ &:-\ \ ((P_1\ \$then\ P_2)\ \$using\ snd)\ X\ \ V\ \ Y.\\ (P_1\ \$thenx\ P_2)\ X\ \ V\ \ Y\ \ &:-\ \ ((P_1\ \$then\ P_2)\ \$using\ fst)\ X\ \ V\ \ Y.\end{aligned}$$

The parser *xthen* discards the first component of the value pair, using $snd\ (A, B)\ B\ \ :-$, and *thenx* discards the second component, using $fst\ (A, B)\ A\ \ :-$, before proceeding with the remainder of the parse.

Parsing simple, numerical expressions

We now give an example of parsing the expressions of a small calculator using the combinators presented earlier. The BNF grammar for the calculator is as follows:

$$\begin{aligned}&e ::= t + t \mid t - t \mid t\\ &t ::= f * f \mid f/f \mid f\\ &f ::= digit^{+} \mid (e)\end{aligned}$$

Of course, '/' and '−' are not associative and, to keep matters simple, we require that extra parentheses are inserted in expressions to explicitly dictate associativity. Therefore, an expression like $1 - 2 - 3$ does not parse but $1 - (2 - 3)$ or $(1 - 2) - 3$ will.

The translation from the above grammar to its parser is straightforward since each BNF construct in the grammar has an analogous combinator in the program. The parser is presented in Figure 1.1. The predicate *parse* can be used to parse numerical expressions into their abstract syntax trees which are built from the constructors *add*, *sub*, *mul*, *div*, and *num*. As an example, we can write the parser

$$\#\ \ :-\ \ parse\ \text{“}2 + (4 - 1) * 3\text{”}\ \ V\ \ Y.$$

This produces the following solutions:

$$\begin{aligned}V\ &=\ add\ (num\ \text{“2”})\ (mul\ (sub\ (num\ \text{“4”})\ (num\ \text{“1”}))\ (num\ \text{“3”}))\\ Y\ &=\ \text{“ ”}\ ;\\ V\ &=\ add\ (num\ \text{“2”})\ (sub\ (num\ \text{“4”})\ (num\ \text{“1”}))\\ Y\ &=\ \text{“}*3\text{”}\ ;\\ V\ &=\ num\ \text{“2”}\\ Y\ &=\ \text{“}+(4-1)*3\text{”}\ ;\\ &no\end{aligned}$$

An expression that cannot be parsed is rejected by the parser, e.g., the query

$$\#\ \ :-\ \ parse\ \text{“}+(4 - 1) * 3\text{”}\ \ V\ \ Y.$$

fails.

```
e X V Y :− ((((t $then ((literal '+') $xthen t)) $using plus) $or
    ((t $then ((literal '−') $then t)) $using minus)) $or t) X V Y.

t X V Y :− ((((f $then ((literal '∗') $xthen f)) $using times) $or
    ((f $then ((literal '/') $xthen f)) $using divide)) $or f) X V Y.

f X V Y :− ((number $using value) $or
    ((literal '(') $xthen (e $thenx (literal ')')))) X V Y.

digit A :− elem A "0123456789".          minus (pair X Y) (sub X Y) :− .
value X (num X) :− .                     times (pair X Y) (mul X Y) :− .
plus (pair X Y) (add X Y) :− .           divide (pair X Y) (div X Y) :− .

number X V Y :− some (satisfy digit) X V Y.
parse X V Y :− e X V Y.
```

Figure 1.1: The combinator parser for a simple calculator.

Pretty-printing

One of the most interesting and appealing aspects of programming a parser in a logic programming language is that, to some extent, pretty-printing comes free with the parsing program. Pretty-printing is the converse operation of parsing and, by exploiting the declarative nature of the logic language, we immediately obtain a pretty-printer by running our parser backwards. The following query illustrates this point.

```
# :− parse X (add (num "2")
                  (mul (sub (num "4") (num "1")) (num "3"))) " ".        (1.1)
```

This query produces the infinite sequence of solutions

$$\begin{aligned} X &= \text{“}2 + (4 - 1) * 3\text{”} \; ; \\ X &= \text{“}2 + (4 - (1)) * 3\text{”} \; ; \\ X &= \text{“}2 + (4 - ((1))) * 3\text{”} \; . \\ &\textit{yes} \end{aligned}$$

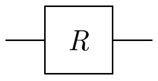
There are, of course, an infinite number of possible pretty-prints for the expression since we can indefinitely supply redundant parentheses in the manner shown above. As we shall see later in this thesis, it is important for our logic programming language to employ a particular computation rule in order to terminate whenever possible and, for the programs that are unable to terminate, provide a fair enumeration of the possible solutions. So, for example, the strategy would avoid the immediate report of solutions formed by consecutively inserting parentheses in (1.1). (This point is discussed further in Section 3.3.)

Our pretty-printer inherits all its knowledge regarding sophisticated layout, e.g., off-side rules and indentation conventions, from the parser. However, since our parser is relatively simple, our pretty-printer is consequently simple also. Achieving the complexity of a pretty-printer written exclusively for the task, like (Hughes 1995), would involve extending the parser of this section with additional combinators to capture the required features. Although we do not describe the details here, combinator parsers allow such extensions to be carried out easily.

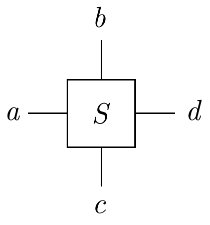
1.2 Ruby

In this example, we show how to implement the Ruby (Jones & Sheeran 1993, Jones & Sheeran 1990, Sheeran 1989) relational language in the actual compositional language developed in the subsequent chapters of the thesis. Until now, only the purely functional subset of Ruby has been implemented (Hutton 1993); in this section, we implement the full relational remainder. Ruby is a relational language for describing hardware circuits. The behaviour of a circuit is specified by a relational program which can then be transformed into an equivalent program amenable to implementation in hardware. A circuit in Ruby is represented as a *binary relation* between simple data values.

The data values on which Ruby circuits operate consist solely of integers, and both tuples and lists of integers. A relation in Ruby can be given pictorially, corresponding directly to its interpretation as a circuit. For example, the circuit associated with the relation R can be represented as the single node:



The convention regarding data flow in such a diagram is that domain values appear at the left and top sides of the node and range values appear at the bottom and right sides. In addition, multiple wires to and from a node are read from left to right, and from bottom to top. Therefore, a relation $(a, b)\ S\ (c, d)$—taking tuples to tuples—may be represented by the circuit



Since data flows from left to right in a circuit, we read our relational expressions from left to right, too.

Ruby expressions are composed hierarchically from primitive relations and higher-order predicates that take relations to relations. The primitive relations, defined in Figure 1.2, are used to construct and deconstruct the data values of a circuit. (Of these relations, though, only *id*, *outl*, and *outr* are essentially primitive in the presence of relational intersection (Bird & de Moor 1996).) These primitive relations provide us with the bare bones of the Ruby language whilst the real expressive power is obtained from higher-order predicates. The most fundamental of these are sequential composition, par-

$A \ \$id \ A$:– .
$(pair \ A \ B) \ \$swap \ (pair \ B \ A)$:– .
$(pair \ A \ B) \ \$outl \ A$:– .
$(pair \ A \ B) \ \$outr \ B$:– .

$A \ \$fork \ (pair \ A \ A)$:– .
$A \ \$null \ nil$:– .
$A \ \$wrap \ (A : nil)$:– .

$app \ (pair \ nil \ Y) \ Y$:– .
$app \ (pair \ (A : X) \ Y) \ (A : Z)$:– $app \ (pair \ X \ Y) \ Z$.
$(pair \ A \ X) \ \$apl \ Y$:– $app \ (pair \ (A : nil) \ X) \ Y$.
$(pair \ X \ A) \ \$apr \ Y$:– $app \ (pair \ X \ (A : nil)) \ Y$.

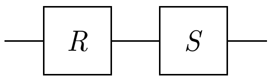
$(pair \ (pair \ A \ B) \ C) \ \$lsh \ (pair \ A \ (pair \ B \ C))$:– .
$(pair \ A \ (pair \ B \ C)) \ \$rsh \ (pair \ (pair \ A \ B) \ C)$:– .

Figure 1.2: The implementation of the primitive Ruby relations.

allel composition, and converse. The sequential, forward composition of two relations R and S is given by

$(R \ \$comp \ S) \ A \ C$:– $R \ A \ B, \ S \ B \ C$.

The sequential composition of two circuits looks like this:



The parallel composition of two relations R and S is expressed by

$(R \ \$par \ S) \ (pair \ A \ C) \ (pair \ B \ D)$:– $R \ A \ B, \ S \ C \ D$.

and simply places R above S. Every relation R has a well-defined converse *conv* R, given straightforwardly by

$conv \ R \ A \ B$:– $R \ B \ A$.

Two commonly used higher-order predicates are simple shorthands, given by

$fst \ R \ A \ B$:– $(R \ \$par \ id) \ A \ B$.
$snd \ R \ A \ B$:– $(id \ \$par \ R) \ A \ B$.

Since Ruby is a language for designing circuits, a plethora of predicates exist for combining circuit elements. For two circuits, denoted respectively by the relations R and S, the predicate *beside* joins R and S together in the following fashion:

$(R \ \$beside) \ S \ A \ B$:–
    $(rsh \ \$comp \ ((fst \ R) \ \$comp \ (lsh \ \$comp \ ((snd \ S) \ \$comp \ rsh)))) \ A \ B$.

The beauty of Ruby is exemplified when we define the *dual* of existing predicates in terms of themselves, providing a convenient way to produce new circuits. We can obtain the dual of any higher-order predicate by first taking the converse of all its arguments and then taking the converse of the entire expression. For example, we can manufacture the predicate *below* in terms of its dual predicate *beside* by defining

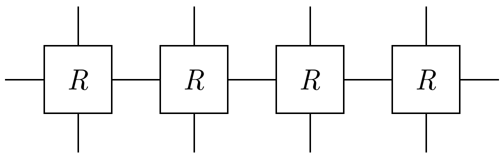
$$(R\ \$below\ S)\ A\ B\ \ :-\ \ conv\ ((conv\ R)\ \$beside\ (conv\ S))\ A\ B.$$

Both *beside* and *below*, interpreted as circuits, are shown in Figure 1.3.

Using *beside*, we can create a new predicate *row* which forms a linear array, cascading the circuit R a number of times. The predicate *row* takes strictly non-empty lists in its pair arguments and is defined as follows:

$$\begin{aligned} &row\ R\ A\ B\ :-\ ((snd\ (conv\ wrap))\ \$comp\ (R\ \$comp\ (fst\ wrap)))\ A\ B.\\ &row\ R\ A\ B\ :-\ ((snd\ (conv\ apl))\ \$comp\ ((R\ \$beside\ (row\ R))\ \$comp\ (fst\ apl)))\\ &\qquad\qquad A\ B. \end{aligned}$$

Notice that *row* is defined as a disjunction. The first clause succeeds only when the second component of the domain tuple is the singleton list. The second clause fails in this case since its recursive call to *row* receives a domain pair argument whose second component is the empty list and such calls to *row* always fail. An instance of *row* is shown below:



The predicate *col*, forming a vertical column of circuits, is defined in terms of its dual predicate *row* as

$$col\ R\ A\ B\ :-\ conv\ (row\ (conv\ R))\ A\ B.$$

Finally, the two predicates *rdl* and *rdr* relate tuples of values to single values and correspond to the circuit equivalent of “reducing” (or “folding”) in functional programming. They are defined as

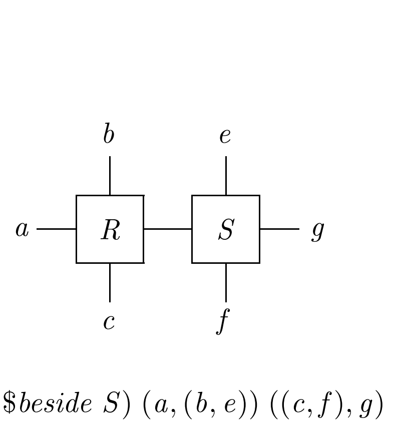
$$\begin{aligned} &rdl\ R\ A\ B\ :-\ ((row\ (R\ \$comp\ (conv\ outr)))\ \$comp\ outr)\ A\ B.\\ &rdr\ R\ A\ B\ :-\ ((col\ (R\ \$comp\ (conv\ outl)))\ \$comp\ outl)\ A\ B. \end{aligned}$$

respectively.

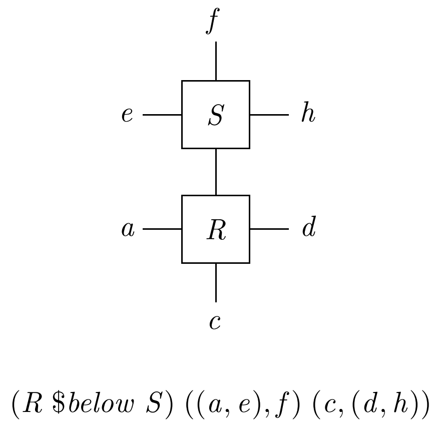
An attractive aspect of using a compositional logic language to implement Ruby is the ease with which Ruby operators translate from abstract mathematics into a concrete program. This exemplifies the declarative nature of our language.

A simple sorting circuit in Ruby

We now present a circuit for sorting an arbitrary, non-empty list of natural numbers using the selection sort algorithm. Two alternative methods for describing the sorting circuit have been given in the past, one by (Sheeran & Jones 1987) and the other by (Hutton



$(R\ \$beside\ S)\ (a,(b,e))\ ((c,f),g)$

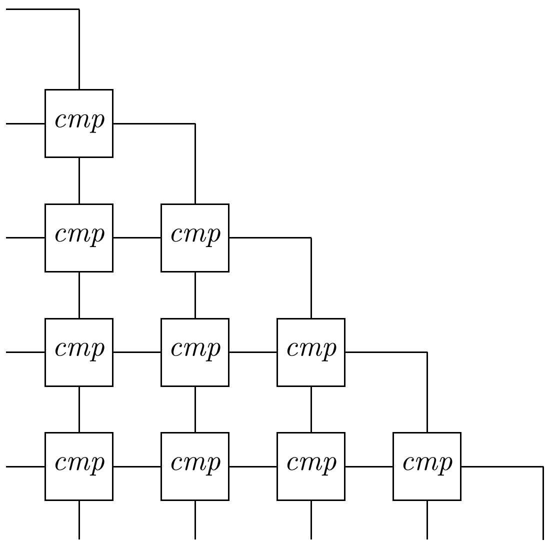


$(R\ \$below\ S)\ ((a,e),f)\ (c,(d,h))$

Figure 1.3: The circuits corresponding to *beside* and *below*.

1992*a*) which has also been implemented. We implement the former presentation here since it is more relational than the latter.

The input to the circuit is a list of naturals and the output is the same list but sorted according to the relation *cmp*, defined shortly. The sorting circuit for the relation *sort* is illustrated below.



The implementation of Ruby in our compositional programming language adopts a symbolic representation of natural numbers since the language lacks a primitive implementation of them. The symbolic representation adopted is that of Peano natural numbers: zero is represented by 0, one by s 0, two by s (s 0), and so on, although we will often abbreviate Peano naturals to actual numerals in the following.

The implementations of the natural number comparison operators are given below where *leq* represents the numerical relation $\leq$, *geq* is $\geq$, *lt* is $<$, and *gt* is $>$.

$$\begin{array}{ll}
\textit{star}\ R\ A\ B\ \ :\!-\ \ \textit{id}\ A\ B. & \textit{leq}\ A\ B\ \ :\!-\ \ \textit{star}\ (\textit{conv succ})\ A\ B. \\
\textit{star}\ R\ A\ B\ \ :\!-\ \ (R\ \$\textit{comp}\ (\textit{star}\ R))\ A\ B. & \textit{geq}\ A\ B\ \ :\!-\ \ \textit{conv leq}\ A\ B. \\
 & \textit{lt}\ A\ B\ \ :\!-\ \ ((\textit{conv succ})\ \$\textit{comp leq})\ A\ B. \\
\textit{succ}\ A\ (s\ A)\ \ :\!-\ \ . & \textit{gt}\ A\ B\ \ :\!-\ \ (\textit{succ}\ \$\textit{comp geq})\ A\ B.
\end{array}$$

The *leq* relation is implemented in Ruby as the reflexive, transitive closure of the the converse of the successor relation *succ*. The comparison operator *cmp* is defined in terms of the following relations *min* and *max* that return the minimum and maximum elements in a pair, respectively:

$$\begin{array}{ll}
\textit{min}\ (\textit{pair}\ A\ B)\ B\ \ :\!-\ \ \textit{leq}\ A\ B. & \textit{max}\ (\textit{pair}\ A\ B)\ B\ \ :\!-\ \ \textit{geq}\ A\ B. \\
\textit{min}\ (\textit{pair}\ A\ B)\ A\ \ :\!-\ \ \textit{gt}\ A\ B. & \textit{max}\ (\textit{pair}\ A\ B)\ A\ \ :\!-\ \ \textit{lt}\ A\ B.
\end{array}$$

$$\textit{cmp}\ A\ B\ \ :\!-\ \ (\textit{fork}\ \$\textit{comp}\ (\textit{min}\ \$\textit{par max}))\ A\ B.$$

A non-empty list of naturals is sorted using the following *sort* relation:

$$\begin{array}{l}
\textit{sort}\ A\ B\ \ :\!-\ \ ((\textit{conv wrap})\ \$\textit{comp wrap})\ A\ B. \\
\textit{sort}\ A\ B\ \ :\!-\ \ ((\textit{conv}\ (\textit{rdr}\ (\textit{conv}\ ((\textit{conv apr})\ \$\textit{comp}\ (\textit{col cmp})))))\ \$\textit{comp} \\
\qquad\qquad\qquad\qquad\qquad ((\textit{snd}\ (\textit{conv wrap}))\ \$\textit{comp apr}))\ A\ B.
\end{array}$$

To the uninitiated, relational expressions can appear rather terse and the ability to execute such expressions is clearly a useful instruction tool. For example, given the above program, we can query

$$\#\ \ :\!-\ \ \textit{sort}\ (2:4:0:1:3:5:\textit{nil})\ B. \tag{1.2}$$

which produces the single answer $A = 0 : 1 : 2 : 3 : 4 : 5 : nil$. The truly relational aspect of the programming language is illustrated by the following query in which the converse of *sort* is computed, producing every permutation of the sorted list $0 : 1 : 2 : 3 : 4 : 5 : nil$:

$$\#\ \ :\!-\ \ \textit{conv sort}\ (0:1:2:3:4:5:\textit{nil})\ B. \tag{1.3}$$

Naturally there are more efficient ways of permuting a list of numbers but (1.3) serves as an exemplar of running a program backwards.

1.3 Summary

Programming in a compositional style results in concise programs, encourages reuse of code by identifying common patterns of recursion, and facilitates the development of formal results about programs. However, such a style of programming is, for the most part, alien to the logic programming community. A compositional logic language inherits all the above benefits familiar from functional programming and is also enriched with the extra expressiveness obtained from nondeterminism, logic variables, and program converses. As a result, a truly compositional logic language is able support the execution of declarative specifications. This chapter proposed such a language and presented two substantial programming examples to illustrate the expressive benefits of a compositional style of logic programming.

In the remainder of this thesis, we present several novel discoveries that realise faithfully this form of logic programming. The first discovery is that a curried representation of terms in a language permits the higher-order features necessary to support compositionality, without the need for either extra-logical predicates or higher-order unification. The second discovery is that fair SLD-resolution guarantees the termination of logic programs whenever any other resolution strategy would terminate. The proof of this result is presented in this thesis. Finally, two efficient versions of fair resolution are described and implemented in this thesis to overcome the poor efficiency of fair SLD-resolution.

Chapter 2

Survey of Existing Languages

Two example programs were presented in the previous chapter to motivate the desired style of compositional programming and to illustrate the level of expressiveness required in an actual language. In this chapter, three existing programming paradigms are evaluated for their suitability of supporting compositionality. Each paradigm is examined according to a collection of desiderata, identified in Section 2.1, which includes: the ease with which high-level specifications can be represented as programs; the extent to which compositional programs can be written and executed without regard to the evaluation order of the language; and, finally, the efficiency with which compositional programs are executed in the language. The programming paradigms considered in this chapter are functional (Gofer), first-order logic (Prolog), and higher-order logic (λProlog).

Functional programming, considered first, is used to implement an elementary relational calculus where relations are represented as set-valued functions in order to capture nondeterminism. However, it becomes apparent that such a representation of relations is unable to support a general implementation that respects the aforementioned desiderata of a compositional language.

We then examine first-order logic programming and review the important processes of unification and resolution since they provide the foundations upon which later chapters are built. Furthermore, we detail the problems associated with implementing relations in Prolog, one of which is its lack of support for higher-order programming. This limitation is addressed by the higher-order logic language λProlog which we consider subsequently. There, we describe the method of higher-order unification which is a crucial component of λProlog's operational mechanism. However, both these logic paradigms share a fundamental shortcoming for implementing a truly compositional language, namely the left-to-right computation rule they both adopt. Despite this, we that λProlog can support a partial implementation of the relational language Ruby, albeit through the critical use of extra-logical annotations to the program. The implementation is far from satisfactory since the heuristics employed in it are unable to generalise to other compositional programs.

A possible fourth contender for review is functional logic programming (Hanus et al. 1995, Hanus 1994, Moreno-Navarro & Roderiguez-Artalejo 1992, Bellia & Levi 1986) which invariably employs the computational mechanism of narrowing (Bellia, Bugliesi & Occhiuto 1990, Reddy 1987), a blend of rewriting for evaluating functional expressions and resolution for evaluating predicates. However, the functional logic paradigm is subsumed by the logic paradigm with regards to compositionality in that functional logic languages also employ the left-to-right computation rule. Thus, the findings of this chapter for logic languages are equally applicable to functional logic ones.

We conclude by noting that each of the three paradigms considered falls short of the language requirements set out in Section 2.1 for the support of compositionality. However, the investigation of this chapter identifies that first-order logic programming provides a promising starting point in the quest for a compositional language. The material presented in this chapter paves the way towards subsequent chapters in which alternative computation rules are developed that better support compositionality than the predominant left-to-right one.

2.1 The desiderata of a compositional programming language

The primary purpose of the compositional style of programming advocated in this thesis is to support the prototyping of high-level specifications. Consequently, a fundamental requirement of any compositional language is that it should allow high-level relational specifications to be easily translated from abstract mathematical notation into programs. Thus, the syntax of the language should unobtrusively permit the phrasing of specifications as programs. This requires the language to support higher-order programming constructs since they aid compositionality by facilitating abstraction during program construction, as exemplified in the programs of Chapter 1.

The second feature of a compositional language is more subtle: since specifications are intrinsically devoid of algorithmic detail, a compositional language should allow programs to be formulated independently of the actual evaluation order employed by the language. This requirement underpins the desired declarative nature of the language by avoiding the need for compositional programs to be constructed with regard to the particular context in which they are used. However, by removing the need for the programmer to consider the evaluation order of the language, we must ensure that the execution of a compositional program terminates whenever possible. In the absence of such a property, it can be difficult to differentiate between programs that are unable to terminate under any circumstances and those that can under a specific order of evaluation.

The final requirement, which is related to the previous one, is that the language must ensure that programs are executed with predictable efficiency behaviour. That is, the programmer should be able to expect the evaluation of a compositional program to proceed with comparable efficiency–for example, to within some constant factor of an appropriate measure—to one designed for execution in a language with a fixed order of evaluation.

The desiderata discussed above are independent of the actual operational mechanisms employed by any particular compositional language. However, in this thesis we restrict our attention to evaluating mechanisms drawn exclusively from functional, first-order logic, and higher-order logic programming, since the most common languages based on these paradigms appear to fall short of satisfying our desiderata for a compositional language. Despite this decision, it may nevertheless be the case that a novel model of execution exists which provides a better foundation for evaluating compositional programs than any of the aforementioned ones. For example, (Lipton & Broome 1994) propose a computational model based upon rewriting relational expressions directly using a set of relational axioms; although this work has yet to evolve into an actual implementation, it may yet prove to be a suitable operational foundation for compositional programming.

2.2 Functional programming languages

Functional languages provide an unobtrusive syntax for compositional programs and possess a well-defined and theoretically simple computational model. For these reasons, functional languages appear to be good candidates for supporting compositional programming. Moreover, functional languages have been used in the past to implement relational calculi (Collins & Hogg 1997, Cattrall 1993, Hutton 1992*a*, MacLennan 1983). However, each of these calculi fail to capture the full generality of executing specifications whilst simultaneously satisfying the criteria of a compositional language set out in the previous section.

For example, the interpreter presented by (Cattrall 1993), which is implemented in the lazy functional language Miranda (a trademark of Research Software Limited), uses the type information of relational programs to infer their most appropriate underlying representation: either association lists, set-valued functions, or characteristic functions. However, the programmer must often use language primitives to restrict the number of elements in the range or domain of a relation before applying certain operators like relational converse. The reason for this is due to the limitations of implementing relations as set-valued functions, which are discussed in Section 2.2.3.

Alternatively, the Ruby interpreter (Hutton 1992*a*), which is implemented in the functional language LML, represents relations using techniques other than set-valued functions. The interpreter provides primitive definitions of relational composition, converse, and product, which are sufficient to allow all other Ruby combining forms to be defined directly as functions in LML. Nevertheless, this implementation executes only a subset of arbitrary Ruby specifications, as discussed in Section 2.2.4.

One recent exception to the problem of providing a general implementation of relations in a functional language has been addressed by (Seres & Spivey 1999) who have developed an embedding of Prolog in the functional language Haskell. This novel approach to implementing Prolog interprets its logical connectives as Haskell functions which, in turn, permits well-known algebraic properties of the functions to be inherited into the embedded Prolog program. Moreover, (Seres & Spivey 1999) have even considered how to integrate aspects of fairness into their implementation, advocated earlier by (McPhee & de Moor 1996).

In the remainder of this section, we illustrate the difficulties of implementing a compositional language using the natural representation of relations as set-valued functions. We use Gofer as the implementation language and perform the study by introducing a relational calculus and then translate it into a Gofer program using a simple data refinement of sets to lists.

2.2.1 Relations

A *binary relation* $R : A \to B$, of type A to B, is a subset of the Cartesian product $A \times B$ of sets A and B. The notation aRb denotes that the pair $(a, b) \in R$. The composition of two relations $R : B \to C$ and $S : A \to B$, written as $R \circ S : A \to C$, is defined as follows:

$$a(R \circ S)c = \exists b. aSb \wedge bRc$$

Every relation $R : A \to B$ has a well defined converse $R^\circ : B \to A$ defined straightforwardly as

$$b R^\circ a = aRb$$

Converse is contravariant, i.e., reverses composition, in that $(R \circ S)^\circ = S^\circ \circ R^\circ$ and is also an involution since $(R^\circ)^\circ = R$.

The final two operations used are the intersection of two relations $R, S : A \to B$, written as $R \cap S : A \to B$, and their union, written as $R \cup S : A \to B$. These operations are defined predictively as follows

$$a(R \cap S)b = aRb \wedge aSb$$
$$a(R \cup S)b = aRb \vee aSb$$

The small collection of relational operators above is already sufficient for writing interesting compositional programs. In the next section, we implement relations as set-valued functions in the functional programming language Gofer.

2.2.2 An implementation of relations as set-valued functions in Gofer

One essential difference between a relation and a function is that a relation may relate a single domain value to several range values whereas a function, by definition, maps each domain value to precisely one range value. To accommodate the nondeterminism of relations in a functional language, we appeal to an alternative mathematical model of relations as *set-valued functions*. The set-valued function representation of relations is particularly suitable since it provides an ostensibly natural way to emulate relations in a functional language: by considering sets as lists, we can model a relation in a functional language as a list-valued function. Hence, a relation is implemented as a function that takes a single domain value as its argument and returns a list of all related range values. We present the implementation in the functional programming language, Gofer.

Sets in Gofer

We implement sets as lists in Gofer using the following type synonym with the data type invariant that a set does not contain duplicate elements.

```
> type Set a = [a]
```

The invariant is enforced by an appropriate implementation of *setify*, i.e.,

```
> setify :: [a] → Set a
> setify = nub
```

where the Gofer library function *nub* removes duplicate elements from a list.

We define the function *set*, which applies a function to all values in a set, as follows.

```
> set :: (a → b) → Set a → Set b
> set = map
```

From an implementation point of view, we do not employ *setify* here to avoid testing equality of lists.

The function *inject* makes a set out of any value and is defined as

```
> inject :: a → Set a
> inject a = [a]
```

The function *union* returns the union of a set of sets and is implemented as follows:

```
> union :: Set (Set a) → Set a
> union = setify . foldr (++) []
```

By implementing *union* using the library function *foldr*, we arrive at a simple definition and restore the data type invariant for the resulting set using *setify*.

Two utility functions that operate on pairs of sets are defined next. The first calculates the union of two sets in terms of list concatenation as follows:

```
> cup :: (Set a, Set a) → Set a
> cup = setify . uncurry (++)
```

The second computes the intersection of two sets by retaining only those range elements common to both:

```
> cap :: (Set a, Set a) → Set a
> cap (a,b) = (setify . filter (flip elem b)) a
```

In this section we have chosen to implement the set data type using lists, however, the functions presented above abstract over the actual implementation selected.

Relations in Gofer

Using the implementation of sets given in the previous section, we define a relation to be a set-valued function using the following type synonym:

```
> type Rel a b = a → Set b
```

The function *unit* is the identity relation which converts any value to a singleton set, where the data type invariant for sets is trivially satisfied:

```
> unit :: Rel a a
> unit = inject
```

We implement the composition of two relations using the function

```
> comp :: Rel b c → Rel a b → Rel a c
> comp r s = union . set r . s
```

The implementations of the union and intersection of set-valued functions depend on the function *split* which forms a pair of sets.

```
> split :: Rel a b → Rel a c → a → (Set b, Set c)
> split r s a = (r a, s a)
```

Using *split*, the union of two relations is programmed as

$> join :: Rel\ a\ b \rightarrow Rel\ a\ b \rightarrow Rel\ a\ b$
$> join\ r\ s = cup\ .\ split\ r\ s$

The intersection of two relations is computed by the function *meet* whose implementation is given below.

$> meet :: Rel\ a\ b \rightarrow Rel\ a\ b \rightarrow Rel\ a\ b$
$> meet\ r\ s = cap\ .\ split\ r\ s$

The relational combinators above allow for the definition of many other useful relational programs in a natural, declarative manner.

An additional benefit of implementing relations in Gofer is that we obtain the built-in Gofer data types—like integers, pairs, and lists—for free. For example, the relation *succ*, which relates an integer to its successor, uses the Gofer built-in type *Int* and is programmed as

$> succ :: Rel\ Int\ Int$
$> succ = lift\ (1+)$

The abstraction function *lift* converts any Gofer function into a relation and is defined as follows:

$> lift :: (a \rightarrow b) \rightarrow Rel\ a\ b$
$> lift\ f = unit\ .\ f$

We already used Gofer's built-in pair data type in the definition of *split* and we use it again to program the right projection of a pair as relation

$> outr :: Rel\ (a,b)\ b$
$> outr = lift\ snd$

The Gofer list data type is used frequently in the following relational programs and we define the list construction relation *cons* that adds an element to a list as follows:

$> cons :: Rel\ (a,[a])\ [a]$
$> cons = lift\ (uncurry\ (:))$

The relational combinators implemented thus far are enough to allow the implementation of interesting relations. For example, the reflexive, transitive closure of a relation is programmed declaratively as

$> close :: Rel\ a\ a \rightarrow Rel\ a\ a$
$> close\ r = unit\ `join`\ ((close\ r)\ `comp`\ r)$

Using this definition, we can specify the relation *geq* which computes the set of integers greater than or equal to a given integer.

$> geq :: Rel\ Int\ Int$
$> geq = close\ succ$

Similarly, the relation

```
> leq :: Rel Int Int
> leq = close (lift (flip (−) 1))
```

relates an integer to all the integers less than or equal to it.

We can implement a relational fold over lists as follows:

```
> fold :: b → Rel (a,b) b → Rel [a] b
> fold c r [] = unit c
> fold c r (x:xs) = (union . set (curry r x) . fold c r) xs
```

Many relational programs can be expressed in terms of *fold*. For example, the length of a list can be calculated by the relation

```
> len :: Rel [a] Int
> len = fold 0 (succ `comp` outr)
```

Therefore, in Gofer, we can query

```
? len [1..5]
[5]
```

Notice that *len* always returns a singleton set since *len* is a function. However, *fold* is able to return a set that contains more than one element, thus capturing the nondeterministic behaviour that lends the extra expressiveness to relations. For example, all the subsequences of a list can be computed using the function

```
> subseqs :: Rel [a] [a]
> subseqs = fold [] (cons `join` outr)
```

The nondeterminism of *subseqs* arises from the fact that, at each step in the relational fold, we decide whether to include the current element of the list, using *cons*, or exclude it, using *outr*. So, the query

```
? len (subseqs [1..5])
[32]
```

determines that there are 32 subsequences, i.e., 2^5, of [1,2,3,4,5] as expected.

So, implementing relations as set-valued functions in Gofer permits the natural expression of relational composition, union, and intersection. However, the conspicuous absence of an implementation of relational converse is not an accident: representing relations as set-valued functions does raise significant problems since not all the essential features of sets can be implemented in Gofer. The following section details these problems.

2.2.3 Problems with implementing relations as set-valued functions

In the previous section, we demonstrated that pleasingly declarative relational programs can be written in a functional language by representing relations as list-valued functions. However, these relational programs ignored an implementation of converse. Indeed, despite the declarative implementation obtained, representing relations as set-valued functions presents two substantial problems.

First of all, calculating the converse of a relation is a nontrivial operation. To illustrate this point, consider the relation *outr* which was defined in the previous section. The converse of *outr*—a relation that takes an element and returns *any* tuple whose second component is that element—is nondeterministic since the second component of the resultant tuple may be any element of the appropriate type. Functional languages like Gofer require each applicable value to be explicitly enumerated and if there is an infinite number of such values, then the set-valued function will also be infinite.

Secondly, introducing relations that return infinite lists is problematic itself since the obvious definitions of the set-valued relational operations, presented in the previous section, could result in non-terminating programs even when the solution set is finite. For example, the set of all numbers can be specified as

$> top :: Set\ Int$
$> top = join\ leq\ geq\ 0$

Then, simple expressions like *elem* 5 *top* fail to terminate since *leq* enumerates the infinite number of integers less than 0 before considering those greater than 0.

However, the exploitation of lazy evaluation when joining infinite lists in Gofer can help make an implementation of relations more declarative. To explain, consider the following function that interleaves two lists by alternating ("twiddling") their elements:

$> twiddle :: [a] \rightarrow [a] \rightarrow [a]$
$> twiddle\ [\,]\ ys = ys$
$> twiddle\ (x{:}xs)\ ys = x : twiddle\ ys\ xs$

The function *twiddle* is 'fair' when joining two lists in the sense that it always considers elements in both of its arguments, whereas ++ may not if its first argument is infinite. With this knowledge, the occurrence of concatenation in the function *join* can be replaced by *twiddle* to permit the fair union of infinite relations. Similarly, *union* can be redefined to use *twiddle*, therefore, allowing *comp* to compose infinite relations. In this case, the expression *elem* 5 *top* would terminate with the answer *True*.

Despite improving the termination behaviour for the relational operators above, the same trick of using *twiddle* cannot be applied to the function *meet* since it is obliged to traverse the entire length of a list, finite or otherwise, in the search for duplicates. For example, to determine the intersection of *geq* and *leq*, which is a finite relation, the following query could be used

$?\ meet\ geq\ leq\ 4$
$[4$

However, Gofer does not terminate here since all the elements of an infinite list must be enumerated. So, a direct implementation of relations as set-valued functions—complete with relational converse—in a functional language is unsatisfactory since relations that operate on infinite domains often require the enumeration of infinite lists, even when the relation itself is finite. Consequently, many simple relations can fail to terminate.

2.2.4 An existing implementation of Ruby in LML

In this section, we give a brief overview of an existing implementation of a Ruby interpreter (Hutton 1993, Hutton 1992a). We refer to this implementation as the 'interpreter' to avoid confusion with the Ruby implementation presented in Chapter 1.

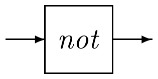
The analogy between circuits and Ruby expressions in the interpreter is more rigorous than in Section 1.2. In particular, each Ruby primitive in the interpreter has identified inputs and outputs, just as in any real circuit. On the other hand, the programs of Section 1.2 capture the specification of Ruby expressions—rather than their interpretation as real circuits—and, consequently, can specify circuits that are unable to be described directly in hardware. Thus, the Ruby implementation of Section 1.2 permits the execution of Ruby specifications whilst the interpreter executes Ruby programs, i.e., actual circuits.

The interpreter operates on the subset of Ruby specifications that adhere to a definition of being 'executable', i.e., those specifications that are 'causal relations' (Hutton 1992a). Essentially, a Ruby circuit is executable if each component of the circuit is a function (rather than a relation) and the value of each 'wire' connecting components in the circuit can be determined by these functions. Moreover, the value of each wire is determined by only one circuit component and the value of an input wire must not depend upon the value of any output wire from the same component, unless delay components are used. (Delay components are used in sequential circuits to allow their input value during one clock cycle of the circuit to be returned as their output value during a subsequent clock cycle. However, as is the case with (Hutton 1992a), we make no use of them in the following discussion.) An executable circuit need not be a functional program and the interpreter is able to compute Ruby circuits that contain uses of relational converse, for example.

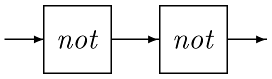
Since each Ruby circuit component in the interpreter has a notion of directionality, i.e., each component has defined inputs and outputs, two components can only be composed if the output value of the first follows the input direction of the second. For example, consider the *not* primitive for logical negation, defined in the syntax of Section 1.2 as:

not true false :− .
not false true :− .

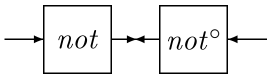
The circuit corresponding to this definition is shown below where the data flows from left to right, as indicated by the direction of the arrows.



The composition of two circuits in the interpreter is only possible if the arrows of each circuit to be composed flow in the same direction. Thus, the composition *not* \$*comp not* of two *not* components is executable as the following circuit shows:



However, the Ruby specification *not* \$*comp* (*conv not*) is not executable since, in its representation as a circuit below, the wire connecting both components is driven twice:



(The label *not*° in the circuit above represents *conv not*.) Consequently, the interpreter rejects this program since it is a Ruby specification rather than an executable program.

On the other hand, this specification can be executed in the compositional logic language of Section 1.2 using a query of the following form:

$$\# \ :- \ (not \ \$comp \ (conv \ not)) \ true \ A.$$

The result of this query is $A = true$.

The primary benefit of the Ruby interpreter is to execute circuits that are realisable in hardware whilst that of the compositional language of Section 1.2 is to execute specifications. The Ruby language is designed to provide a method of describing the abstract behaviour of a circuit in a high-level notation and then permit its transformation to an 'executable' program through the application of various algebraic laws. Therefore, the interpreter and the implementation of Section 1.2 complement each other and illustrate the distinct roles of a programming language and a specification language.

2.3 First-order logic programming languages

The example compositional programs presented in Chapter 1 made fundamental use of a higher-order syntax to allow partially applied relations to be passed as arguments to other relations. Functional programming, as illustrated in the previous section, provides support for higher-order programming that is both natural for the programmer and semantically well-founded. However, first-order logic programming, as its name suggests, lacks the same solid theoretical foundation for higher-order constructs. Therefore, an examination of first-order logic programming, using Prolog, is necessary to evaluate the extent to which it satisfies the desiderata for compositional programming given at the beginning of the chapter.

In this section, we review the aspects of first-order logic programming that will be used pervasively in subsequent chapters: the structure of terms, substitutions, unification, and SLD-resolution. We restrict our attention to definite programs only, i.e., those free of negated literals, since the extension to include negative literals in definite programs is straightforward (Lloyd 1987). The material contained in this section is standard from the literature and further details can be found in (Hogger 1990, Lloyd 1987), for example.

2.3.1 Terms, programs, goals, and substitutions

Let us begin by defining the structure of terms and programs in first-order logic languages.

Definition 2.1 (Terms and literals). Let X be a set of variables and Ω a set of constant symbols. The *terms* (or *literals*) of the programming language are defined inductively by the grammar

$$\begin{aligned} \mathbb{T}_\Omega(X) ::= {} & x \\ | \ & f(t_1, \ldots, t_n) \end{aligned}$$

where $x \in X$, $f \in \Omega$, and $t_1, \ldots, t_n \in \mathbb{T}_\Omega(X)$ for $n \geq 0$. In the case where $n = 0$, we write f rather than $f(\,)$. $\diamondsuit$

In untyped logic programming languages, like Prolog, the arity of a constant symbol f can vary between its occurrences in a program, although such behaviour is prohibited in strongly typed languages, like Mercury.

Definition 2.2 (Definite clauses and programs). A *definite clause*, or just *clause*, is a formula $A :- B_1, \ldots, B_n$, for $n \geq 0$. The literal A is called the *head* of the clause and the outermost constant in A is called a *predicate* symbol. The literals $B_1, \ldots, B_n$ form the *body* of the clause. When $n = 0$, we write the clause as $A :-$ and refer to it as a *fact*. Finally, a *program* is a set of clauses. $\diamondsuit$

Definition 2.3 (Goal clauses). A *goal clause*, also known as a *query* or just a *goal*, is a formula $\# :- B_1, \ldots, B_n$ where $B_1, \ldots, B_n$ are literals, for $n \geq 0$. The symbol $\#$ is a unique predicate symbol that cannot appear in a program. When $n = 0$, we write the goal as $\# :-$ and refer to it as the *empty clause*. $\diamondsuit$

Most presentations of first-order logic programming in the literature make a distinction between terms and literals: terms being that subset of literals devoid of predicate symbols. In the current review, no such distinction is made and, in Section 3.1, we argue that such a distinction prohibits the natural adoption of higher-order programming constructs in logic languages.

The operational framework of a first-order logic language requires a procedure to determine whether or not a query succeeds with respect to a given program. The resulting *proof procedure* must match literals in a goal with clauses in a program and apply the technique of choosing only those values for the variables of a literal when an appropriate value becomes apparent. In other words, logic variables are 'place holders' until the proof procedure can determine suitable values for them. *Substitutions* bind variables to terms and can be considered as functions, as defined next.

Definition 2.4 (Substitutions). Let X and Y be sets of variables. A substitution $\phi : X \to \mathbb{T}_\Omega(Y)$ is a total function mapping variables to terms. For $X = \{x_1, \ldots, x_n\}$ and $n \geq 0$, we represent ϕ using the notation $\{x_1 \mapsto s_1, \ldots, x_n \mapsto s_n\}$ where variables x_i map to terms s_i, for $1 \leq i \leq n$. (For brevity, we will ignore pairs of the form $x_i \mapsto x_i$, simply assuming their existence.) The set Y is given by $\bigcup_{i=1}^{n} \mathrm{vars}\ s_i$, where the function $\mathrm{vars} : \mathbb{T}_\Omega(Y) \to \mathsf{P}\,Y$ takes a term and returns the set of variables contained in that term. $\diamondsuit$

The result of a computation in a logic language is a substitution generated by the matching process mentioned above. The proof procedure uses these substitutions to systematically replace variables in a term by other terms. The *instance* of a term under a substitution is obtained according to the definition below.

Definition 2.5 (Application of substitutions). Given $\phi = \{x_1 \mapsto s_1, \ldots, x_n \mapsto s_n\} : X \to \mathbb{T}_\Omega(Y)$, we define a function $[\phi] : \mathbb{T}_\Omega(X) \to \mathbb{T}_\Omega(Y)$ (pronounced "apply ϕ") that determines the *instance* of a term t under ϕ as follows:

$$t[\phi] = \begin{cases} \phi\ x, & \text{if } t = x \in X \\ f(t_1[\phi], \ldots, t_n[\phi]), & \text{if } t = f(t_1, \ldots, t_n) \end{cases}$$

That is, when ϕ is applied to a variable, that variable gets replaced by whatever ϕ maps it to and when ϕ is applied to a composite term, ϕ is applied to each subterm. $\diamondsuit$

Substitutions obey several important algebraic properties that aid the construction of the desired proof procedure. In particular, we can define the composition of two substitutions; given two substitutions $\sigma : X \to \mathbb{T}_\Omega(Y)$ and $\phi : Y \to \mathbb{T}_\Omega(Z)$, there exists another substitution $\phi \circ \sigma : X \to \mathbb{T}_\Omega(Z)$, called the *composition* of σ and ϕ, such that the identity

$$(\phi \circ \sigma)\ x = (\sigma x)[\phi] \tag{2.1}$$

holds. Furthermore, it is easy to prove that

$$t[\phi \circ \sigma] = t[\sigma][\phi]$$

for all terms t and that the composition of substitutions is associative.

The identity substitution $\iota : X \to \mathbb{T}_\Omega(X)$ maps each variable to itself and is defined simply as $\iota\ x = x$. The substitution ι is the unit of composition since $\iota \circ \phi = \phi = \phi \circ \iota$. The proofs of the above well-known properties of substitutions are straightforward and we omit the details.

2.3.2 Unification

As mentioned in the previous section, the matching of literals in a goal with clauses in a program is the key component of the proof procedure used to determine whether a goal is a logical consequence of a program. The matching process is known as *unification* (Paterson & Wegman 1978) and unifying two terms produces a substitution such that the terms become identical after applying this substitution over them. Moreover, the substitution created by the unification algorithm is the 'most general' one possible in the sense that all other unifiers are an instance of it. Most general unifiers prevent the generation of useless instances of literals in the proof procedure and help keep the search space that a computer must examine as small as possible.

Definition 2.6 (Most general unifiers). Let $D = \{(s_1, t_1), \ldots, (s_n, t_n)\}$ be a set of pairs of terms from $\mathbb{T}_\Omega(X)$, for $n \geq 1$, and $\phi : X \to \mathbb{T}_\Omega(Y)$ be a substitution. We say that ϕ *unifies*, or *is a unifier of*, D if $s_i[\phi] = t_i[\phi]$, for all $1 \leq i \leq n$. Moreover, we call ϕ a *most general unifier* if, for every other unifier $\psi : X \to \mathbb{T}_\Omega(Y)$ of D, there exists a substitution $\sigma : Y \to \mathbb{T}_\Omega(Y)$, such that $\psi = \sigma \circ \phi$. The most general unifier ϕ of two terms is unique up to the renaming of the variables in Y. $\diamondsuit$

An algorithm that determines the most general unifier of two terms is given in Figure 2.1. By a slight abuse of notation, we understand the application of a substitution ϕ over a set D of pairs of terms by defining

$$D[\phi] = \{(s[\phi], t[\phi]) \mid (s, t) \in D\}. \tag{2.2}$$

The gist of the algorithm is that ok = false if the terms u and v are not unifiable, otherwise ok = true. In the latter case, we let ϕ be a unifier of u and v; moreover, we write $\psi = \sigma \circ \phi$, for ψ some other unifier of the disagreement set D and σ some substitution. Therefore, ϕ accumulates the most general unifier of u and v as the computation progresses whilst D represents the disagreement set containing those parts of u and v pending unification. The invariant maintained by the algorithm, which is used in a proof of the correctness of *unify*, is as follows.

```
$unify\ u\ v =$
  let $D := \{(u, v)\}, \phi := \iota, ok :=$ true
  while $ok$ and $D \neq \{\,\}$ do
    let $D' \cup \{(s, t)\} = D$
      if $s = x = t \longrightarrow D := D'$
      $[]$ $s = x \notin t \longrightarrow D := D'[x \mapsto t], \phi := \{x \mapsto t\} \circ \phi$
      $[]$ $t = x \notin s \longrightarrow D := D'[x \mapsto s], \phi := \{x \mapsto s\} \circ \phi$
      $[]$ $s = f(t_1, \ldots, t_n)$ and $t = f(s_1, \ldots, s_n) \longrightarrow D := D' \cup \{(t_i, s_i) \mid 1 \leq i \leq n\}$
      $[]$ otherwise $\longrightarrow ok :=$ false
      fi
  od
  if $ok \longrightarrow$ Just $\phi$
  $[]$ otherwise $\longrightarrow$ Nothing
  fi
```

Figure 2.1: The unification algorithm for two terms.

Proposition 2.1. *The invariant of the* while *loop in the unification algorithm, illustrated in Figure 2.1, is as follows:*

1. *If u and v have a unifier then $ok =$ true.*
2. *If $ok =$ true then some substitution ψ unifies u and v if and only if there exists a substitution σ such that $\psi = \sigma \circ \phi$ and ϕ unifies D.*

The function *unify*, therefore, returns either 'Nothing', signalling that the disagreement pair is not unifiable, or 'Just ϕ' where ϕ is the most general unifier of the terms u and v as established by the following lemma, whose proof is well known and is omitted.

Lemma 2.1. *If two terms u and v have a unifier, then they have a most general unifier. Moreover, the function unify u v determines this fact.*

Let us now step through an example unification of two terms. Consider the computation of *unify* $f(a, g(X), Y)\ f(Y, g(h(Z)), V)$. After the first iteration of the while loop in the algorithm of Figure 2.1, we obtain the disagreement set

$$D_1 = \{(a, Y), (g(X), g(h(Z))), (Y, V)\}$$

On the next iteration of the loop, we can select any disagreement pair from D_1; suppose we select (Y, V). The unification of this pair produces the single substitution $\phi_1 = \{Y \mapsto V\}$ and we apply this over the remaining pairs in D_1 to form the new disagreement set

$$D_2 = \{(a, V), (g(X), g(h(Z)))\}$$

Suppose we select (a, V) at the subsequent unification step. The resulting substitution is $\phi_2 = \{ V \mapsto a\}$ which we apply over D_2 to obtain the new set

$$D_3 = \{(g(X), g(h(Z)))\}$$

There is only one pair to choose at the next unification step which results in the formation of the set

$$D_4 = \{(X, h(Z))\}$$

The final unification step produces the substitution $\phi_3 = \{X \mapsto h(Z)\}$ and the empty disagreement set. Therefore, the most general unifier of the two terms $f(a, g(X), Y)$ and $f(Y, g(h(Z)), V)$ is $\phi = \phi_3 \circ \phi_2 \circ \phi_1$. The application of ϕ over both terms produces the unified term $f(a, g(h(Z)), a)$.

In this section, we have reviewed an algorithm to calculate the most general unifier for two terms if it exists. What we now require is a proof procedure to check whether a goal is a logical consequence of a program. The systematic search for the proof of a goal, with respect to a given program, is centred around the process of *resolution* and we discuss this in the following section.

2.3.3 SLD-resolution

In this section, the computational strategy of resolution is reviewed. Logic programming languages, like Prolog, use a particularly simple form of resolution, called *SLD-resolution* (Selected literal using Linear resolution for Definite clauses), that allows a programmer to write efficient programs. The drawback of using SLD-resolution, as we shall see shortly, is that the search for a proof of a goal can often fail to terminate even for logically simple queries. In other words, a Prolog programmer must be cognisant with the manner in which programs are executed by Prolog and construct programs accordingly. Consequently, the declarative nature of Prolog programs is often severely restricted.

Resolution is called an ‘inference rule’ because it derives information about a goal and a program. A choice exists of how to perform resolution; for example, individual clauses in the program could be selected to match with literals in a goal or, alternatively, program clauses could be matched together to form new clauses to resolve a goal with. The former of these methods, called *linear resolution*, involves resolving a literal in the goal with a clause taken from the program, producing a new goal which we continue to resolve in the same manner, and forms the basis of resolution for languages like Prolog. The soundness and completeness of resolution are proved, for example, by (Apt & van Emden 1982) where ‘soundness’ means that each derived resolvent is a logical consequence of the program and ‘completeness’ means that all such consequences of the program may be derived by resolution.

Linear resolution itself allows various choices at each resolution step during the proof of a goal: if a goal has more than one literal, any literal from the goal may be selected for resolution and, moreover, any program clause may be chosen to resolve with the selected literal. The method of selecting program clauses is called the *search strategy* and the goal literal called the *computation rule* (or *selection strategy*). It is, however, well known that any applicable program clause or goal literal may be selected and still retain the aforementioned soundness and completeness properties of resolution (Spivey

1996, Hogger 1990, Lloyd 1987, Apt & van Emden 1982). With this in mind, both the search strategy and the computation rule can be fixed in advance to allow the systematic search for proofs of a goal.

The proof of a goal can be visualised in the form of a tree structure, with each node labelled with a goal and each branch representing the result of a resolution step. Given the myriad of choices possible in a resolution, it is not difficult to imagine that this tree can quickly become enormous. From the point of view of systematically searching for a proof of a goal, a priority is to reduce the size of search space that a machine must examine before successfully finding an answer. One resolution-based method of searching for proofs that restricts the size of a search tree is called *SLD-resolution*, adopting linear resolution and a predetermined computation rule and search strategy. We now give a few definitions to formalise the use of SLD-resolution for searching for a proof of a query with respect to some program.

Definition 2.7 (SLD-resolution). Let $C = A :- B_1, \ldots, B_m$, for $m \geq 0$, be a definite clause, and $G = \# :- A_1, \ldots, A_n$, for $n \geq 1$, a goal such that A_1 and A are unifiable with most general unifier ϕ. Then the *SLD-resolvent* of C and G is the goal

$$G' = (\# :- B_1, \ldots, B_m, A_2, \ldots, A_n)[\phi]$$

The substitution ϕ is called the *substitution of the SLD-resolution.* $\diamondsuit$

A sequence of SLD-resolutions forms an *SLD-derivation* where the selected literal, the substitution of the SLD-resolution, and the program clause used to resolve with, are recorded for each resolution step.

Definition 2.8 (SLD-derivation). For a program P and a goal G, an *SLD-derivation* of $P \cup \{G\}$ is a (possibly infinite) sequence of triples $(G_1, C_1, \phi_1), \ldots, (G_n, C_n, \phi_n)$, for $n \geq 1$, where C_n denotes a 'variant' of a definite clause in P, G_n is a goal (we distinguish $G_1 = G$), and ϕ_n is a substitution. Moreover, for $1 \leq i < n$, G_{i+1} is *derived from* G_i and C_i via substitution ϕ_i by an SLD-resolution. $\diamondsuit$

A variant clause C is identical to its corresponding program clause except that all variables in C are renamed to avoid clashes with variables in the current derivation.

The basic method of searching for a proof of a goal $G = \# :- A_1, \ldots, A_n$ with respect to a program P involves interpreting the symbol '#' in a special way. In particular, '#' is imagined as a unique predicate symbol of no arguments; then the literals in G are repeatedly resolved with relevant clauses from P until the process either fails or derives the empty clause $\# :-$, from Definition 2.3. This process, called *refutation*, utilises SLD-resolution and the unification algorithm presented earlier to determine the most general values of any free variables in G. The definition of this process, called *SLD-refutation*, is as follows.

Definition 2.9 (SLD-refutation). For a program P and a goal G, an *SLD-refutation* of G is an SLD-derivation for $P \cup \{G\}$ starting at G and ending with $G_n = \# :-$, for $n \geq 1$. If the substitutions of the SLD-resolutions in the derivation are $\phi_1, \ldots, \phi_n$, then the *answer substitution of the refutation* is $\phi = \phi_n \circ \ldots \circ \phi_1$. $\diamondsuit$

The search tree formed from an SLD-resolution is commonly known as an *SLD-tree*, and we define them as follows.

Definition 2.10 (SLD-trees). Let P be a program and G a goal. An SLD-tree for $P \cup \{G\}$ is a (possibly infinite) tree with each node labelled with a goal and each branch labelled with a substitution so that the following conditions are satisfied:

1. The root node of the tree is G.
2. Let $G' = \# :- A_1, \ldots, A_n$, for $n \geq 1$, be a node in the tree. For each program clause $A :- B_1, \ldots, B_m$, for $m \geq 0$, such that A_1 and A are unifiable with most general unifier ϕ, the node has a child $\# :- (B_1, \ldots, B_m, A_2, \ldots, A_n)[\phi]$. We label the arc connecting G' to a child with the substitution of the SLD-resolution and call A_1 the *selected literal*.
3. Nodes labelled with the empty goal $\# :-$ have no children.

Each sequence of nodes in an SLD-tree is clearly either an SLD-refutation or an SLD-derivation. ♢

The shape of the SLD-tree generated for a query depends on the particular computation rule employed in the SLD-resolution; indeed, the choice of rule can have a tremendous influence on the size of the corresponding SLD-tree. Nevertheless, every SLD-tree is essentially the same with respect to the refutations it contains. In other words, every SLD-tree contains the same number of refutations, irrespective of the computation rule used in the resolution steps. This result is a reformulation of the completeness of SLD-resolution.

In this section, we have only considered the resolution of definite programs, i.e., those free of negation, since extending a logic language to include negative literals is straighforward (Lloyd 1987). The essential alteration is the use of a *safe* computation rule (Lloyd 1987) which selects a negative literal in a goal only when it is ground (variable free) to ensure its resolution cannot result in an unsound answer substitution. Now that we have reviewed the important aspects of SLD-resolution, we can evaluate Prolog against the desiderata for a compositional language.

2.3.4 Higher-order logic programming in Prolog

The use of higher-order functions in functional programming results in concise programs, encourages program reuse, and, by abstracting common patterns of recursion, promotes the development of formal results about programs. On the other hand, Prolog programs are typically constructed from first-order predicates, with the flow of data between literals in a goal facilitated through the use of logic variables rather than from combinators. Indeed, a first-order syntax actively hinders the adoption of a compositional style of programming in Prolog. Superficially, then, Prolog may appear to lack the desiderata of compositional programming. Therefore, in this section, Prolog is assessed on its ability to support compositionality both in terms of syntax and operational behaviour.

To illustrate the traditional, stylistic differences between logic and functional programs, let us consider how we might write the well known insertion sort algorithm in both paradigms. An experienced functional programmer often writes programs using higher-order functions to express recursive algorithms over particular data structures.

So, one might expect the following definition of insertion sort in the functional language, Gofer.

$$\begin{array}{ll}
> & \mathit{insert}\ a\ [\,] \ = \ [a] \\
> & \mathit{insert}\ a\ (b : x) \\
> & \quad \mid a \leq b \ = \ a : b : x \\
> & \quad \mid \mathit{otherwise} \ = \ b : \mathit{insert}\ a\ x \\
 & \\
> & \mathit{isort} \ = \ \mathit{foldr}\ \mathit{insert}\ [\,]
\end{array}$$

Note that *isort* is defined using a standard pattern of recursion over lists, embodied by the function *foldr*.

On the other hand, a Prolog programmer must often express recursion explicitly since higher-order constructs are alien to pure Prolog. A typical Prolog implementation of insertion sort is as follows:

$$\begin{array}{l}
\mathit{insert}(A, [\,], [A]) \ . \\
\mathit{insert}(A, [B|Y], [A|B|Y]) \ :- \ A \leq B. \\
\mathit{insert}(A, [B|Y], [B|Z]) \ :- \ A > B, \ \mathit{insert}(A, Y, Z). \\
 \\
\mathit{isort}([\,], [\,]) \ . \\
\mathit{isort}([A|X], Y) \ :- \ \mathit{isort}(X, Z), \ \mathit{insert}(A, Z, Y).
\end{array}$$

The absence of higher-order predicates makes abstraction impossible in pure Prolog, resulting in programs that regularly contain many identical instances of the same recursive patterns that could otherwise be abstracted out into combinators. Nevertheless, the benefits of higher-order programming have not entirely escaped the logic programming community and, over the years, Prolog implementations have included ad-hoc support for higher-order programming features at the syntactic level. However, to say that such attempts have failed to catch on would be quite an understatement.

The reason for the apparent apathy towards higher-order programming in Prolog can be attributed to its lack of appropriate theoretical foundations. Consequently, this has fostered an obscure syntax for programming with higher-order predicates in Prolog and such programs are notoriously hard to decipher in comparison to their functional counterparts. Two predominant methods of supporting higher-order predicates in Prolog exist, both involving the use of meta-level predicates. We discuss both in turn below.

The 'call' primitive in Prolog

The meta-level predicate *call* is often provided as a primitive in many implementations of Prolog, including Mercury (Somogyi et al. 1995), to support higher-order programming. In such cases, $\mathit{call}(Q)$ treats Q as a goal and evaluates it. For example, if Q is $\mathit{isort}([3, 2, 1], Y)$ then $\mathit{call}(Q)$ is equivalent to $\mathit{isort}([3, 2, 1], Y)$. Supplementary versions of *call* are also provided to cater for arguments greater than one, such that $\mathit{call}(Q, A_1, \ldots, A_n)$ is evaluated by passing the literals $A_1, \ldots, A_n$ as extra arguments to Q before evaluating it. Using the *call* notation, then, the higher-order predicate *foldr* can be defined in Prolog as

$$\begin{array}{l} \mathit{foldr}(F, C, [\,], C). \\ \mathit{foldr}(F, C, [A|X], B) \ :- \ \mathit{foldr}(F, C, X, D),\ \mathit{call}(F, A, D, B). \end{array}$$

From this, the *isort* predicate defined at the beginning of the section can be reformulated as

$$\mathit{isort}(X, Y) \ :- \ \mathit{foldr}(\mathit{insert}, [\,], X, Y).$$

resulting in a program much more similar to its functional counterpart.

However, the main drawbacks of using *call* are two-fold: firstly, programs must explicitly contain occurrences of *call*, unnecessarily cluttering the code and reducing the aesthetic benefits of higher-order programming; secondly, the higher-order features that *call* provides are less expressive than those found in functional programming languages. In particular, predicates are not first class objects in such logic programs since they cannot be returned as the result of a predicate. The following example illustrates this point.

Example 2.1. Consider the Prolog program below that defines relational composition and the addition predicate for Peano natural numbers.

$$\mathit{compose}(R, S, A, C) \ :- \ \mathit{call}(S, A, B),\ \mathit{call}(R, B, C).$$

$$\begin{array}{l} \mathit{add}(0, N, N)\ . \\ \mathit{add}(s(M), N, s(K)) \ :- \ \mathit{add}(M, N, K). \end{array}$$

The query

$$\# \ :- \ \mathit{foldr}(\mathit{compose}(\mathit{add}, \mathit{add}(1)), 0, [1], A). \tag{2.3}$$

is intended to add 1 to each value in the given list and then sum this new list. However, when executed in Prolog, the query fails, returning the non-informative answer *no*. To see why this is so, consider the SLD-tree for (2.3) shown in Figure 2.2. Each leaf of the tree ends in failure since *compose* is defined for only four arguments, rather than five. The crucial point is that $\mathit{compose}(\mathit{add}, \mathit{add}(1))$ should return a predicate as a result that can then be applied to the correct number of arguments. $\diamondsuit$

The main competitor to the *call* primitive for supporting higher-order programming is the *apply* primitive, detailed next. The *apply* literal addresses the problem with *call* discussed in Example 2.1.

The 'apply' primitive in Prolog

A method of dealing with higher-order programming in Prolog, similar to the *call* primitive above, was suggested by (Warren 1982) that used the meta-level predicate *apply*. The use of *apply* has recently been championed by (Naish 1996) as the superior of the two approaches since it provides a more faithful analogy of higher-order functional programming than does *call* by allowing predicates to return partially applied predicates as

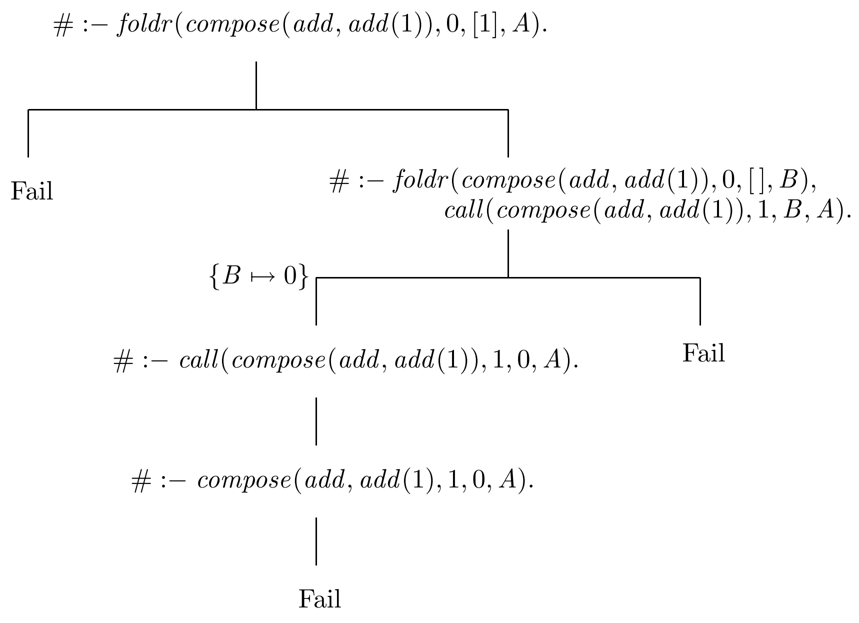

$\# :- \mathit{foldr}(\mathit{compose}(\mathit{add}, \mathit{add}(1)), 0, [1], A).$
Fail
$\# :- \mathit{foldr}(\mathit{compose}(\mathit{add}, \mathit{add}(1)), 0, [\,], B), \mathit{call}(\mathit{compose}(\mathit{add}, \mathit{add}(1)), 1, B, A).$
$\{B \mapsto 0\}$
$\# :- \mathit{call}(\mathit{compose}(\mathit{add}, \mathit{add}(1)), 1, 0, A).$
Fail
$\# :- \mathit{compose}(\mathit{add}, \mathit{add}(1), 1, 0, A).$
Fail

Figure 2.2: The SLD-tree for $\# :- \mathit{foldr}(\mathit{compose}(\mathit{add}, \mathit{add}(1)), 0, [1], A)$ implemented using *call*.

results. The literal $\mathit{apply}(F, A, G)$ takes a predicate F as its first argument and attempts to evaluate $F(A, G)$. When $F(A, G)$ is a first-order term, the behaviour of *apply* and *call* are identical. However, if $F(A, G)$ remains partially applied, *apply* binds G instead to an appropriate representation of $F(A)$.

Occurrences of *call* in a program can be replaced by *apply*, though applications of *apply* must be cascaded to replace uses of *call* that have more than three arguments. The central technique is to use currying to pass each argument in turn to the higher-order predicate, until it is fully applied. In particular, $n - 2$ occurrences of *apply* are necessary to replace a single use of *call* with n arguments. Although generally more occurrences of *apply* must appear in the body of a clause than with *call*, the trade-off is that the resulting programs behave in a manner similar to analogous functional programs. Let us illustrate with an example.

Example 2.2. Using *apply*, we can recast the definitions of *foldr* and *compose* as follows:

$$\mathit{foldr}(F, C, [\,], C).$$
$$\mathit{foldr}(F, C, [A|X], B) \;:-\; \mathit{foldr}(F, C, X, D),\; \mathit{apply}(F, A, G),\; \mathit{apply}(G, D, B).$$

$$\mathit{compose}(R, S, A, C) \;:-\; \mathit{apply}(S, A, B),\; \mathit{apply}(R, B, C).$$

Using these new definitions, query (2.3) can be successfully executed, producing the answer $A = 2$. To illustrate the reason for this, consider the corresponding SLD-tree shown in Figure 2.3. The node labelled (*) in the figure can, this time, be success-

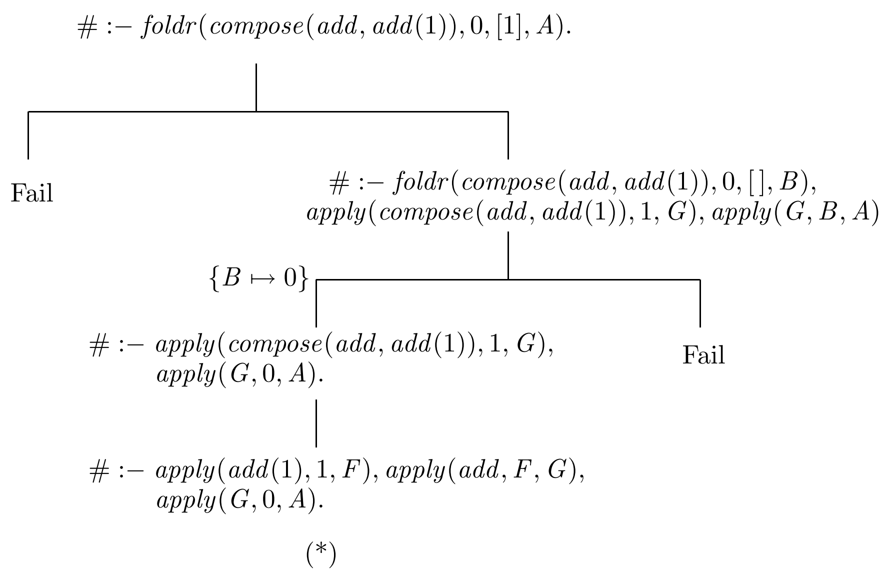

$\# :- foldr(compose(add, add(1)), 0, [1], A).$
Fail
$\# :- foldr(compose(add, add(1)), 0, [\,], B),$
$apply(compose(add, add(1)), 1, G), apply(G, B, A).$
$\{B \mapsto 0\}$
$\# :- apply(compose(add, add(1)), 1, G),$
$apply(G, 0, A).$
Fail
$\# :- apply(add(1), 1, F), apply(add, F, G),$
$apply(G, 0, A).$
(*)

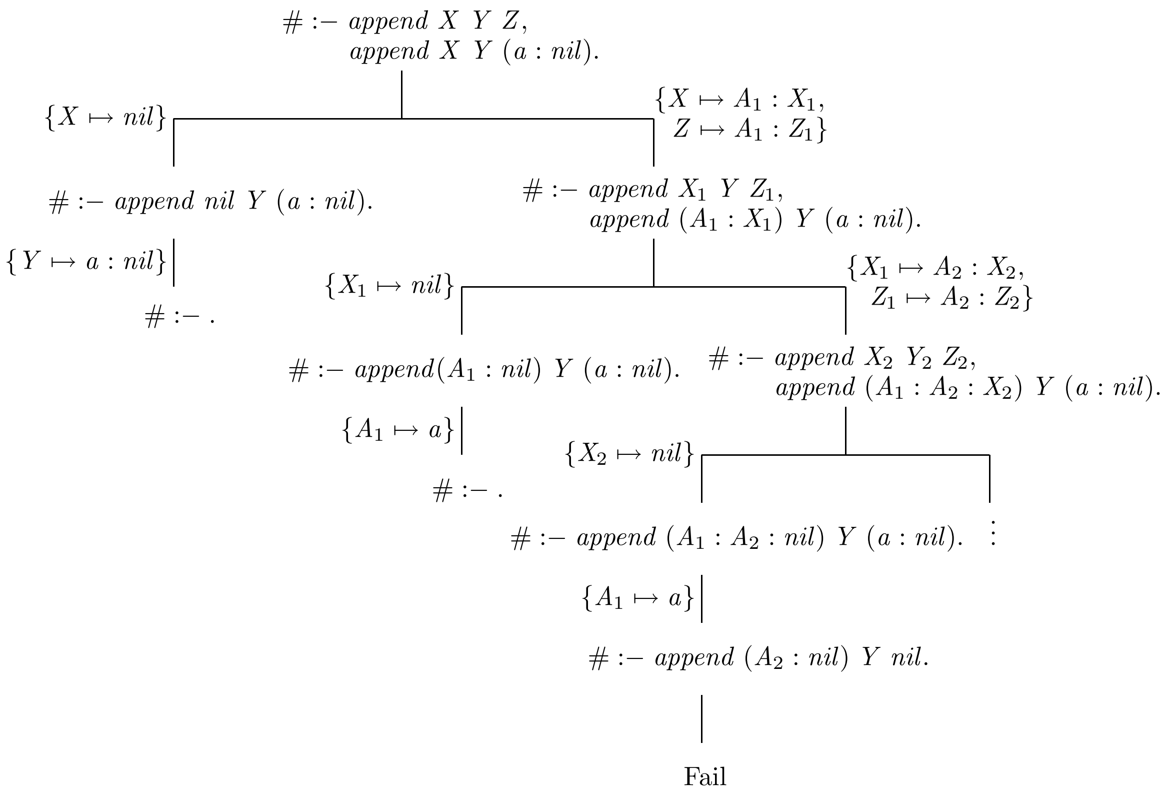
Figure 2.3: The SLD-tree for $\# :- foldr(compose(add, add(1)), 0, [1], A)$ implemented using *apply*.

fully executed: the subgoal $apply(add(1), 1, F)$ can be executed directly, binding F to the value 2. The next subgoal, $apply(add, F, G)$, is instantiated to $apply(add, 2, G)$, resulting in G being bound to the partially applied literal $add(2)$. The final subgoal becomes $apply(add(2), 0, A)$ which can be evaluated directly to produce the final answer $A = 2$. ♢

2.3.5 The limitations of SLD-resolution

The previous section demonstrated that Prolog can support a compositional style of programming at the syntactic level, albeit a little obtrusively. However, syntax is only one component of the desiderata that characterises the ability of a language to support compositionality. The proof of the pudding lies in the successful execution and termination of relational programs. Unfortunately, the resolution strategy adopted by Prolog exhibits intrinsic deficiencies when executing relational programs and we discuss them in the remainder of this section.

Most implementations of logic programming languages, whose computational model is based on SLD-resolution, employ a left-to-right (or depth-first) computation rule. So, for the goal $\# :- A_1, \ldots, A_n$, the leftmost literal A_1 is always selected at each resolution step and any introduced literals take its place. However, the search for an SLD-refutation can sometimes be substantially more difficult, often resulting in non-termination, when using a left-to-right computation rule than some other one. The following example demonstrates this phenomenon.



# :− append X Y Z, append X Y (a : nil).
{X ↦ nil}
{X ↦ A1 : X1, Z ↦ A1 : Z1}
# :− append nil Y (a : nil).
# :− append X1 Y Z1, append (A1 : X1) Y (a : nil).
{Y ↦ a : nil}
# :− .
{X1 ↦ nil}
{X1 ↦ A2 : X2, Z1 ↦ A2 : Z2}
# :− append(A1 : nil) Y (a : nil).
# :− append X2 Y2 Z2, append (A1 : A2 : X2) Y (a : nil).
{A1 ↦ a}
# :− .
{X2 ↦ nil}
# :− append (A1 : A2 : nil) Y (a : nil).
{A1 ↦ a}
# :− append (A2 : nil) Y nil.
Fail

Figure 2.4: The SLD-tree for $\# :- \ \mathit{append}\ X\ \ Y\ \ Z, \mathit{append}\ X\ \ Y\ \ (a : \mathit{nil})$.

Example 2.3. Consider the following standard definition in a logic language of the predicate *append* that concatenates two lists:

$$\begin{array}{l} \mathit{append}\ \mathit{nil}\ \ Y\ \ Y\ \ :-\ \ . \\ \mathit{append}\ (A : X)\ \ Y\ \ (A : Z)\ \ :-\ \ \mathit{append}\ X\ \ Y\ \ Z. \end{array}$$

Furthermore, consider the following query:

$$\#\ \ :-\ \ \mathit{append}\ X\ \ Y\ \ Z,\ \ \mathit{append}\ X\ \ Y\ \ (a : \mathit{nil}). \tag{2.4}$$

The left-to-right computation rule of Prolog effectively enumerates all lists X and Y which, when concatenated, form Z, and then checks each of them to see if X and Y concatenate to form the list $a : \mathit{nil}$. The SLD-tree for this query, depicted in Figure 2.4, illustrates that the resolution is infinite since there are an infinite number of such lists X, Y, and Z. ♢

By initially selecting the rightmost literal in (2.4), only two possibilities for X and Y would be enumerated, therefore dramatically reducing the search required before discovering all SLD-refutations and terminating. In fact, the use of a left-to-right computation rule means that SLD-resolution can be incomplete, in a practical sense, when searching for a proof of a goal. Despite the limitations of the left-to-right computation rule, it remains predominant in logic programming owing almost entirely to the ease of its implementation and the fact that programmers can write more efficient programs once educated about the order of evaluation.

Although ostensibly pathological in nature, queries like (2.4)—where infinite subgoals occur before finite ones in a goal—arise remarkably often in compositional programs. Therefore, the combination of a left-to-right computation rule and depth-first search is unsuitable for supporting compositional logic programs. Nevertheless, resolution and unification naturally capture the desirable feature of nondeterminism prevalent in compositional programs.

In the following section, we turn our attention to the higher-order logic programming language λProlog (Nadathur & Miller 1988) whose resolution strategy is similar to that of Prolog's but adopts a more sophisticated representation of terms and, consequently, a more elaborate unification algorithm. Again, λProlog utilises the left-to-right computation rule and, therefore, inherits the same problems with termination of compositional programs. However, a partial solution to this problem is illustrated that relies on an extra-logical primitive and could be equally be implemented for Prolog.

2.4 Higher-order logic programming languages

Higher-order logic languages (Nadathur & Miller 1998, Qian 1994, Prehofer 1994) appear to satisfy immediately at least some of the desiderata for compositional languages. In particular, languages like λProlog (Nadathur & Miller 1988) employ a higher-order logic as their semantic basis which provides natural support for the usual syntactic higher-order constructs—currying and passing predicates as arguments to other predicates—pervasive in compositional programs.

In this chapter, we give an introduction to λProlog, describing the higher-order terms employed and the programming benefits associated with them. We then discuss the process of higher-order unification—fundamental to the computation of programs in λProlog—and present several examples that illustrate the computation of higher-order unifiers. Beyond that, we complete the analysis of λProlog against the desiderata for a compositional language by implementing the relational language Ruby, from Section 1.2. However, λProlog retains the left-to-right computation rule and so suffers the same termination problems as Prolog when executing compositional programs. Despite this clear shortcoming in terms of compositionality, this section demonstrates that λProlog supports a partial solution to the problem of termination for compositional programs, at least for the Ruby implementation.

2.4.1 Terms of the language

The notion of higher-order logic programming is captured using a generalisation of definite clauses (Nadathur & Miller 1998). The terms of these clauses are described in a higher-order logic, Church's simple theory of types (Church 1940) in the case of λProlog, and a degree of familiarity with this is assumed in the current section, especially with the typing of terms. Basically, the terms are λ-terms which, informally, have the concrete syntax $\lambda x_1, \ldots, x_n.A_1, \ldots, A_m$, for $n, m \geq 0$. Here, the variables $x_1 \ldots x_n$ form the *binder* of the term—often abbreviated to $\vec{x}$—and $A_1, \ldots, A_m$ are the terms of its *body*. The usual meanings of abstraction and application of λ-terms are given next.

Definition 2.11 (λ-conversion). Let S and T be λ-terms. The process of λ-*conversion*, which shows how to convert one λ-term to another, is given as the reflexive, transitive closure of the following rules:

$$\begin{array}{rlll} \lambda x.\,T & \lambda\text{-converts to} & \lambda y.\,T[x \mapsto y], \text{provided } y \notin \mathrm{fv}(T) & \alpha\text{-conversion} \\ (\lambda x.\,T)\ S & \lambda\text{-converts to} & T[x \mapsto S] & \beta\text{-conversion} \\ \lambda x.\,T\ x & \lambda\text{-converts to} & T, \text{provided } x \notin \mathrm{fv}(T) & \eta\text{-conversion} \end{array}$$

The function ‘fv’ returns the set of free variables in a λ-term. The expression $T[x \mapsto S]$ substitutes S for the free occurrences of x in T, ensuring that any bound variables in T are renamed to avoid the capture of any free variables in S. ♢

With respect to the rules of λ-conversion, the equality of two λ-terms S and T is defined as follows:

$$S = T \ \equiv\ S\ \lambda\text{-converts to } T \text{ or } T\ \lambda\text{-converts to } S$$

A λ-term is said to be in *normal form* if it can be written as $\lambda\vec{x}.P\ A_1\ \ldots\ A_n$, where $n \geq 0$, P is a constant or variable of type $\alpha_1 \to \ldots \alpha_n \to \beta$ with β being of non-functional type, and each A_i is a term that can be written in a similar way. A term can be rewritten to this form by performing all applicable β-conversions, and also by applying the η-conversion rule in the reverse direction, known as η-*expansion*, to supply explicitly all arguments to terms of functional type.

Functions in λProlog are irreducible constants used to construct data types and have no equational definition in a program. Consequently, a λ-term in λProlog is evaluated using only the rules of λ-conversion rather than the usual rewriting found in functional programming where free variables are replaced by the right hand side of an appropriate equation defined in the program. The benefit of restricting β-conversion to encode only the most basic form of substitution is that unification can be used to determine the actual *value* of a function variable.

To illustrate this important deviation from functional programming, let us define a procedure *mapf* that takes a function as its first argument, and applies it to each element of its second argument.

$$\begin{array}{l} \mathit{mapf}\ \ F\ \ \mathit{nil}\ \ \mathit{nil}\ \ {:}{-}\ \ . \\ \mathit{mapf}\ \ F\ \ (A : X)\ \ ((F\ \ A) : Y)\ \ {:}{-}\ \ \mathit{mapf}\ \ F\ \ X\ \ Y\ \ . \end{array}$$

From this definition and assuming ‘+’ is some predefined constructor function, we can query

$$\#\ \ {:}{-}\ \ \mathit{mapf}\ \ F\ \ (1 : 2 : 3 : \mathit{nil})\ \ (1 + 1 : 2 + 1 : 3 + 1 : \mathit{nil})\ . \tag{2.5}$$

This query succeeds in λProlog with F bound to $\lambda x.x + 1$. Here, the binding for F is determined from the rule for β-conversion which constructs an appropriate λ-term. Finding values of this kind for higher-order variables depends critically on the aforementioned weakened form of function evaluation.

On the other hand, predicates in λProlog embody the 'real' evaluation phases in a program since it is they that are defined by clauses in the program. For example, consider the analogy of *mapf* that instead takes a predicate as an argument:

$$\begin{array}{l} \mathit{map}\ P\ \mathit{nil}\ \mathit{nil}\ \ {:}-\ \ . \\ \mathit{map}\ P\ (A : X)\ (B : Y)\ \ {:}-\ \ P\ A\ B,\ \mathit{map}\ P\ X\ Y\ . \end{array}$$

and the predicate

$$\mathit{succ}\ M\ (M + 1)\ \ {:}-\ \ .$$

Then, the query

$$\#\ \ {:}-\ \ \mathit{map}\ P\ (1 : 2 : 3 : \mathit{nil})\ (1 + 1 : 2 + 1 : 3 + 1 : \mathit{nil})\ . \tag{2.6}$$

attempts to enumerate possible *predicative* bindings for P, rather than functional ones as in (2.5). Possible solutions could be $P = \lambda xy.\mathit{succ}\ x\ y$, $P = \lambda xy.\mathit{succ}\ x\ Z, \mathit{succ}\ x\ y$, and so on. In fact, since an infinite number of predicates could be generated to solve such a query, it seems reasonable to limit the positions that a predicate variable can occupy.

With reference to a clause of the form $A\ {:}-\ G$, there are four alternative locations that a predicate variable can occupy in a definite clause: (1) the head of the atomic formula A; (2) an argument position of the atomic formula A; (3) the head of an atomic goal formula in G; (4) an argument position of an atomic goal formula in G. These four locations, labelled correspondingly, are illustrated in the following example clause:

$$\mathit{map}\ \overset{\overset{1}{\downarrow}}{}\ \overset{\overset{2}{\downarrow}}{P}\ (A : X)\ (B : Y)\ {:}-\ \overset{\overset{3}{\downarrow}}{P}\ A\ B, \mathit{map}\ \overset{\overset{4}{\downarrow}}{P}\ X\ Y.$$

Only (1) seems an unrealistic location for a predicate variable since the clause would be anonymous. Therefore, predicate variables are prohibited from assuming such positions. Despite this, predicate variables can still occupy such positions in goal queries, i.e., position 3. Therefore, queries such as (2.6) remain legitimate. The reason for permitting such queries—even though, by themselves, they cannot be expected to enumerate meaningful solutions—is that they can be coerced into providing meaningful bindings for predicate variables by constraints imposed by other conjuncts in the goal; such techniques depend upon the left-to-right computation rule employed by λProlog and prove valuable in meta-programming (Nadathur & Miller 1998) but we will make no use of such techniques in what follows.

2.4.2 Higher-order unification

As in first-order logic, the notion of computation in higher-order logic corresponds to solving a query with respect to a given program by a process of resolution. The main difference between first-order and higher-order logic languages is that unification in higher-order languages is performed over λ-terms where the equality between these terms is based on the rules of λ-conversion, rather than on Clark's axioms of equality (Lloyd 1987), i.e., syntactic equality of terms. In particular, the problem of unifying two λ-terms is equivalent to unifying their normal forms. In the remainder of this section we describe the process of *higher-order unification*, i.e., the unification of λ-terms. These λ-terms are assumed to be simply typed (Church 1940).

Unification aims to determine whether an arbitrary disagreement set has any unifiers and, if so, to provide one such unifier. In a higher-order setting, finding a unifier for an arbitrary disagreement set of simply typed λ-terms is undecidable (Huet 1973). Moreover, a unifiable disagreement set may not have a most general unifier. Despite these problems, (Huet 1975) developed a sound and complete method of finding unifiers for arbitrary disagreement sets. By sound, we mean that any substitution computed for a disagreement set by the procedure is a unifier for it, and, by complete, we mean that a unifier is returned for a disagreement set if one exists. In the case where a unifier does not exist for a disagreement set, however, the procedure may not terminate. For a strictly first-order disagreement set, the procedure always returns the most general unifier.

The algorithm hinges upon the observation that, for some disagreement sets, it is relatively straightforward to discover at least one unifier for the set or to establish that it is not unifiable. The unification procedure uses an iterative application of two functions in an attempt to transform a disagreement set to one where the above determination can be made. The following lemma, due to (Huet 1975), provides the basis for the first of these functions. The notation $U(D)$ denotes the set of unifiers for a disagreement set D and, for a literal A, if the head of A is other than a variable we say that A is *rigid* otherwise we say A is *flexible*.

Lemma 2.2. *Let $s = \lambda\vec{x}.f\ A_1\ \dots\ A_n$ and $t = \lambda\vec{x}.g\ B_1\ \dots\ B_m$ be two rigid, simply typed λ-terms in normal form with the same type. Then $\phi \in U(\{(s,t)\})$ if and only if $f = g$, which implies that $n = m$, and $\phi \in U(\{(\lambda\vec{x}.A_i, \lambda\vec{x}.B_i) \mid 1 \leq i \leq n\})$.*

The lemma above suggests that we can either determine whether no unifiers exist for a disagreement set D or that we can reduce the problem to finding unifiers for the arguments of each pair in D. For two arbitrary λ-terms, we use α-conversion to ensure they have identical binders. From the lemma, we construct the function *simplify*, whose definition follows.

Definition 2.12 (Simplify). Let D be a disagreement set. The function *simplify*, depicted in Figure 2.5, transforms D into either the symbol 'fail' if D does not have a unifier, or a new disagreement set consisting solely of flexible-flexible or flexible-rigid pairs. $\diamondsuit$

The first stage in computing the unifiers for a disagreement set D involves evaluating *simplify* D. If the result is 'fail' then no unifiers exist for D. Alternatively, if the result is an empty set, or a set consisting only of flexible-flexible pairs, then at least one trivial unifier can be found for D. However, more work needs to be performed if the new set contains any flexible-rigid pairs. For such a pair, we fabricate substitutions that attempt to make the heads of the two terms identical. The function *match* undertakes this task.

Definition 2.13 (Match). Let s be a flexible term and t a rigid term, both in normal form with the same type. The function *match* takes (s,t) to a set of substitutions and is defined in Figure 2.6. $\diamondsuit$

The role of *match* is to determine a set of substitutions, each substitution forming a partial unifier, for the pair of terms and, hence, bring the search for a complete unifier

```
$simplify\ D =$
  let $D' = \{\,\}$
  while $D \neq \{\,\}$ do
    let $D'' \cup \{(s, t)\} = D$ and $D := D''$
    if $flexible\ s \longrightarrow D' := D' \cup \{(s, t)\}$
    $[]$ $flexible\ t \longrightarrow D' := D' \cup \{(t, s)\}$
    $[]$ otherwise $\longrightarrow$
        let $s = \lambda\vec{x}.f\ A_1\ \ldots\ A_n$ and $t = \lambda\vec{x}.g\ B_1\ \ldots\ B_m$ be in normal form
        if $f \neq g \longrightarrow$ return fail
        $[]$ otherwise $\longrightarrow D := D \cup \{(\lambda\vec{x}.A_i, \lambda\vec{x}.B_i) \mid 1 \leq i \leq n\}$
        fi
    fi
  od
  return $D'$
```

Figure 2.5: The function *simplify*.

```
$match\ (s, t) =$
  let $s = \lambda\vec{x}.F\ A_1\ \ldots\ A_n$ and $t = \lambda\vec{x}.G\ B_1\ \ldots\ B_m$ and $S = \{\,\}$
  let $F : \alpha_1 \to \cdots \to \alpha_n \to \beta$ where $\beta$ is primitive
  if $G \notin \vec{x} \longrightarrow$
      let $w_1, \ldots, w_n$ and $H_1, \ldots, H_m$ be fresh variables
      $S := S \cup \{(F, \lambda w_1, \ldots, w_n.G\ (H_1\ w_1\ \ldots\ w_n)\ \ldots\ (H_m\ w_1\ \ldots\ w_n))\}$
  fi
  for $i = 1$ to $n$ do
    if $\alpha_i = \gamma_1 \to \cdots \to \gamma_k \to \beta \longrightarrow$
        let $w_1, \ldots, w_n$ and $H_1, \ldots, H_k$ be fresh variables
        $S := S \cup \{(F, \lambda w_1, \ldots, w_n.w_i\ (H_1\ w_1\ \ldots\ w_n)\ \ldots\ (H_k\ w_1\ \ldots\ w_n))\}$
    fi
  od
  return $S$
```

Figure 2.6: The function *match*.

```
$unify\ u\ v =$
  let $D := simplify\ \{(u, v)\}$, $\phi := \iota$ $ok$ = false
  while $ok$ and $D \neq \{\,\}$ do
     let $D' \cup \{(s, t)\} = D$
        if $match\ (s, t) = \{\,\} \longrightarrow ok :=$ false
        [] otherwise $\longrightarrow$
              let $\psi \in match\ (s, t)$
              $D := simplify\ (D'[\psi]), \phi := \psi \circ \phi$
        fi
  od
  if $ok \longrightarrow$ Just $\phi$
  [] otherwise $\longrightarrow$ Nothing
  fi
```

Figure 2.7: The unification algorithm for two λ-terms.

closer to a solution. The function *match* makes essential use of type information to determine the number of arguments each higher-order term requires. The following lemma, due to (Huet 1975), states that *match* achieves this goal.

Lemma 2.3. *Let s be a flexible term and t be a rigid term where both terms have the same type and are in normal form. If there exists a substitution $\phi \in U(\{(s,t)\})$ then there exists a substitution $\psi \in match\ (s, t)$ such that $\phi = \sigma \circ \psi$, for some substitution σ.*

The functions *simplify* and *match* provide the necessary machinery to describe a unification algorithm for two λ-terms based on an iterative use of them. The resulting function, *unify*, is shown in Figure 2.7. Notice that the algorithm for *unify* is nondeterministic in two places. The first is the selection of a pair from D and the second is the choice of substitution ψ. All such choices should be explored in an actual implementation of *unify*.

Example higher-order unifications

In this section, we step through two examples to illustrate the computation of higher-order unifiers. For the first example, consider the computation of $unify\ (g\ (F\ 1)\ (F\ 2))\ (g\ 1\ 2)$ where the variable F has type $nat \to nat$ (assuming that nat is already declared as the type of natural numbers). We begin by generating the disagreement set

$$D = \{(F\ 1, 1), (F\ 2, 2)\}$$

according to the algorithm of Figure 2.7. Next, we select either of the flexible-rigid terms in D, using *match* to generate a set of substitutions. Suppose we choose the first pair and compute

$$match\ (F\ 1, 1) = \{F \mapsto \lambda w.1, F \mapsto \lambda w.w\} \tag{2.7}$$

of possible substitutions. Here, we note that the argument 1 to F is of type *nat*. Consequently, variable w in the generated λ-abstraction $\lambda w.w$ is a first-order variable and is, therefore, bereft of any arguments in the body.

We may arbitrarily select either of these substitutions to apply in the remainder of the unification step. By choosing $\phi = \{F \mapsto \lambda w.1\}$, we generate the new disagreement set

$$D_1 = D[\phi] = \{((\lambda w.1)\ 1, 1), ((\lambda w.1)\ 2, 2)\}.$$

We can simplify D_1 by β-converting the first component of each pair, resulting in the set

$$D_1 = \{(1, 1), (1, 2)\}.$$

To complete the unification step we call *simplify* D_1 which fails in this instance since the rigid-rigid pair $(1, 2)$ cannot be unified. At this stage, we backtrack and choose the alternative substitution $\phi = \{F \mapsto \lambda w.w\}$ from (2.7). This time, we obtain the disagreement set

$$D_2 = D[\phi] = \{((\lambda w.w)\ 1, 1), ((\lambda w.w)\ 2, 2)\}$$

which, by β-converting each pair, reduces to

$$D_2 = \{(1, 1), (2, 2)\}.$$

The call to *simplify* D_2 succeeds and returns D_2 unchanged. The unifier for D is, therefore, the substitution $\{F \mapsto \lambda w.w\}$.

A more involved example is to determine the unifier for the terms $g\ (F\ 1)$ and $g\ (2 + 1)$, for variable $F : nat \to nat$ and '+' a constructor. The initial disagreement set is constructed using

$$simplify\ \{(g\ (F\ 1), g\ (2 + 1))\} = \{(F\ 1, 2 + 1)\}$$

and we next compute

$$match\ (F\ 1, 2 + 1) = \{F \mapsto \lambda w.(H_1\ w) + (H_2\ w), F \mapsto \lambda w.w\} \tag{2.8}$$

from the algorithm of Figure 2.6. The second of these substitutions, $\psi = F \mapsto \lambda w.w$, produces

$$D_1 = D[\psi] = \{((\lambda w.w)\ 1, 2 + 1)\}$$

and, by a single β-conversion, we obtain

$$D_1 = \{(1, 2 + 1)\}.$$

Here, *simplify* D_1 fails since 1 and $2 + 1$ are not unifiable. On backtracking to the substitution $\phi_1 = \{F \mapsto \lambda w.(H_1\ w) + (H_2\ w)\}$ in (2.8), we produce the set

$$D_2 = D_1[\phi_1] = \{((\lambda w.(H_1\ w) + (H_2\ w))\ 1, 2 + 1)\}$$

which β-converts to

$$D_2 = \{((H_1\ 1) + (H_2\ 1), 2 + 1)\}.$$

Since $((H_1\ 1) + (H_2\ 1), 2 + 1)$ is a rigid-rigid pair, we derive

$$D_3 = \mathit{simplify}\ D_2 = \{(H_1\ 1, 2), (H_2\ 1, 1)\}.$$

Therefore, *simplify* reduces the problem of finding unifiers for the two terms in D_2 to that of finding unifiers for their arguments. We now repetitiously perform a unification step on either of the pairs in D_3. The first pair results in the substitutions

$$\mathit{match}\ (H_1\ 1, 2) = \{H_1 \mapsto \lambda v.2, H_1 \mapsto \lambda v.v\}. \tag{2.9}$$

However, the second substitution $\{H_1 \mapsto \lambda v.v\}$ is inappropriate since

$$\begin{array}{ll} & \mathit{simplify}\ (\{(H_1\ 1, 2), (H_2\ 1, 1)\}[H_1 \mapsto \lambda v.v]) \\ = & \quad \{\text{application of substitution}\} \\ & \mathit{simplify}\ \{((\lambda v.v)\ 1, 2), (H_2\ 1, 1)\} \\ = & \quad \{\beta\text{-conversion}\} \\ & \mathit{simplify}\ \{(1, 2), (H_2\ 1, 1)\} \\ = & \quad \{\text{application of } \mathit{simplify}\} \\ & \text{fail.} \end{array}$$

Therefore, we choose the other substitution $\phi_2 = \{H_1 \mapsto \lambda v.2\}$ from (2.9), and, in a similar fashion as above, derive

$$D_3 = \{(2, 2), (H_2\ 1, 1)\}.$$

Selecting the second pair, we obtain the set of substitutions

$$\mathit{match}\ (H_2\ 1, 1) = \{H_2 \mapsto \lambda u.1, H_2 \mapsto \lambda u.u\}.$$

This time, either choice of substitution is suitable. Suppose we choose $\phi_3 = \{H_2 \mapsto \lambda u.u\}$. We derive

$$\begin{array}{ll} & \mathit{simplify}\ (\{(2, 2), (H_2\ 1, 1)\}[H_2 \mapsto \lambda u.u]) \\ = & \quad \{\text{application of substitution}\} \\ & \mathit{simplify}\ \{(2, 2), ((\lambda u.u)\ 1, 1)\} \\ = & \quad \{\beta\text{-conversion}\} \\ & \mathit{simplify}\ \{(2, 2), (1, 1)\} \\ = & \quad \{\text{application of } \mathit{simplify}\} \\ & \{\,\}. \end{array}$$

We are done and the unifier of D is obtained from the composition of each substitution in the computation, i.e.,

$$\begin{array}{cl}
 & \phi_3 \circ \phi_2 \circ \phi_1 \\
= & \quad \{\text{definitions}\} \\
 & \{H_2 \mapsto \lambda u.u\} \circ \{H_1 \mapsto \lambda v.2\} \circ \{F \mapsto \lambda w.(H_1\ w) + (H_2\ w)\} \\
= & \quad \{\text{composition of substitutions}\} \\
 & \{F \mapsto \lambda w.((\lambda v.2)\ w) + ((\lambda u.u)\ w)\} \\
= & \quad \{\text{two } \beta\text{-conversions}\} \\
 & \{F \mapsto \lambda w.2 + w\}.
\end{array}$$

2.4.3 An implementation of Ruby in λProlog

The higher-order logic that underpins λProlog allows compositional programs to be written naturally without the extra-logical annotations necessary in its first-order counterparts. However, λProlog deviates markedly from existing practice by adopting higher-order unification, a process which is undecidable in general. Moreover, λProlog also employs the left-to-right computation rule which was identified in Section 2.3.5 as prohibitive for the support of compositionality.

In this section, we present an implementation of Ruby (Section 1.2) in λProlog which demonstrates that Ruby specifications translate naturally into λProlog programs and attempts to overcome the problem of the left-to-right computation rule. The former objective is unsurprising since the compositional language of Chapter 1 adopted a similar syntax to that of λProlog. Therefore, in this section we concentrate on the novel aspects of the Ruby implementation. In Section 2.2.4, we reviewed an existing implementation of the functional subset of Ruby; here, we show how a λProlog implementation of Ruby encompasses the full specification component of Ruby. No familiarity with λProlog is assumed in the discussion.

The language λProlog is polymorphically typed and is curried in the same sense as in many functional programming languages like Haskell. Type signatures are mandatory for all function and predicate symbols. Type variables in λProlog are denoted by identifiers beginning with an upper case letter. The type of propositions, i.e., truth values or booleans, is the special type 'o' and, consequently, the type signature of a predicate in λProlog always terminates with this type. Thus, functions (or constructors) can be distinguished from predicates by their types.

The pair data type can be defined in λProlog using the following program:

kind *pair* type $\rightarrow$ type $\rightarrow$ type.
type *pair* $A \rightarrow B \rightarrow (pair\ A\ B)$.

The 'kind' operator introduces a new type constructor of arity one less than the number of occurrences of the keyword 'type' in the declaration. In the case above, *pair* constructs a pair from two argument types. The type of function and predicate symbols are declared via a 'type' declaration. In the program above, the constructor function *pair* takes two arguments of type A and B, and returns a value of type *pair* A B.

The following λProlog program declares the data type *nat* of Peano natural numbers:

kind nat type.
type 0 nat.
type s $nat \rightarrow nat$.

Natural numbers are thus represented by 0, $s\ 0$, $s\ (s\ 0)$, and so on, although we will abbreviate such expressions to actual integers in what follows. With the declaration of naturals above, the addition of natural numbers can be defined in Ruby as follows.

type $add\ (pair\ nat\ nat) \rightarrow nat \rightarrow o$.

$add\ (pair\ 0\ N)\ N$.
$add\ (pair\ (s\ M)\ N)\ (s\ K)\ :-\ add\ (pair\ M\ N)\ K$.

In addition to pairs and natural numbers, an implementation of Ruby in λProlog requires the following declaration of lists:

kind $list$ type $\rightarrow$ type.
type nil $list\ A$.
type $:$ $A \rightarrow (list\ A) \rightarrow (list\ A)$.

where *nil* denotes the empty list and ':' denotes infix cons. The function symbol ':' takes two arguments, one of type A and the other of type *list* A, and returns a result of type *list* A.

An attractive aspect of using λProlog to implement Ruby is the ease with which Ruby operators translate into λProlog programs; the higher-order functions translate almost identically from their abstract Ruby definitions to their corresponding implementations in λProlog. Consequently, the λProlog implementation of Ruby is almost identical to that of Section 1.2. The only, yet significant, difference is that the implementation of Ruby in λProlog depends critically on the use of an extra-logical primitive $flex : A \rightarrow o$ that succeeds if its argument is a variable. The use of *flex* attempts to overcome the inherent problem of the left-to-right computation rule adopted in λProlog. Recall from Section 2.3.5 that a left-to-right computation rule adversely affects the ability to run predicates "in reverse," i.e., solving for variables in either the range or domain positions of a predicate. Such behaviour is vital to satisfy the requirement that each relation in Ruby has a well defined converse.

The fundamental Ruby primitive of sequential composition is directly affected by the problematic behaviour of the computation rule. To see why, consider the following naive implementation of sequential composition in λProlog:

type $comp\ (A \rightarrow B \rightarrow o) \rightarrow (B \rightarrow C \rightarrow o) \rightarrow A \rightarrow C \rightarrow o$.
$comp\ R\ S\ A\ C\ :-\ R\ A\ B,\ S\ B\ C$.

Consider the following query:

$\#\ :-\ comp\ add\ (conv\ add)\ A\ (pair\ 1\ 1)$.

After one resolution step, it produces the resolvent

$$\# \;:-\; \mathit{add}\ A\ B,\ \mathit{conv}\ \mathit{add}\ B\ (\mathit{pair}\ 1\ 1)\ . \tag{2.10}$$

In the subsequent resolution step, the left-to-right computation rule of λProlog selects the literal *add A B* which can be satisfied in an infinite number of ways. Consequently, the resolution of (2.10) enters an infinite loop. However, the literal *conv add B* (*pair* 1 1) can be satisfied in only one way, namely by binding B to 2, which when applied to *add A B* results in the literal *add A* 2 whose resolution is finite. Thus, the evaluation of (2.10) can be made to terminate by selecting the second literal for resolution instead of the first one. As discussed in Section 2.1, one criterion for a compositional language is that its termination behaviour should not depend upon the order in which relations are composed; so, for a language like λProlog to support a compositional style of programming, we must avoid selecting infinite literals over finite ones.

A solution to this problem in λProlog is relatively straightforward although admittedly less declarative than one might prefer: in the definition of *comp*, its third argument A is checked to see whether it is uninstantiated. If so, the literals in the body of *comp* are resolved in the reverse order to avoid slipping into an infinite computation. The assumption made is that if A is uninstantiated then the final argument C will be instantiated. The revised implementation of composition is shown in Figure 2.8. The appearance of the cut '!' in the first clause of *comp* avoids the second clause being attempted should the first one fail. Another point to note is that some type obscuring takes place to overcome the fact that a Ruby value comprises either a number, a tuple, or a list. An alternative way of tackling this problem is to declare a new type of all possible Ruby values, hence, distinguishing each value using constructors.

The implementation of Ruby in λProlog exhibits pleasing termination behaviour when executing the following sorting circuit, originally presented in Section 1.2:

```
sort A B  :-  (wrap comp (conv wrap)) A B.
sort A B  :-  (apr comp (snd (conv wrap)) comp
        (conv (rdr (conv ((col cmp) comp (conv apr)))))) A B.
```

For example, both queries (1.2) and (1.3) terminate as desired when using the definition of *comp* in Figure 2.8 but fail to terminate when using the naive definition.

2.5 Summary

In this chapter, three existing programming paradigms were examined against a collection of criteria to determine how well they supported the style of compositional programming proposed in Chapter 1. The criteria encompassed matters of: the ease with which high-level specifications can be translated into programs of the language; the declarative nature of the resulting programs, in particular, that their termination behaviour should not depend upon how the program was composed; and, finally, the efficiency of the language.

The three paradigms considered were functional, first-order logic, and higher-order logic. Although functional programming allowed high-level specifications to be phrased naturally as programs, it was able to capture only a subset of the expressiveness required for compositional relational programming. First-order logic programming proved

```
kind anonymous type.

type comp (A → B → o) → (B → C → o) → A → C → o.
type untype A → anonymous.
type uninstantiated A → o.
type flexible anonymous → o.

comp R S A C :− uninstantiated A, !, S B C, R A B.
comp R S A C :− R A B, S B C.

uninstantiated A :− flexible (untype A) .

flexible (untype A) :− flex A, !.
flexible (untype (pair A B)) :− flexible (untype A), flexible (untype B) .
flexible (untype (A : nil)) :− flexible (untype A), !.
flexible (untype (A : X)) :− flexible (untype A), flexible (untype X) .
```

Figure 2.8: The implementation of sequential composition in λProlog.

to be more expressive than functional programming by naturally supporting nondeterminism and program converses. However, first-order logic programming lacked the elegant higher-order programming constructs that accompany functional and higher-order logic languages. Consequently, specifications often required some syntactic massaging during their transformation to programs which resulted in their obfuscation.

Whereas Prolog suffers from a cumbersome syntax, λProlog allows the natural representation of high-level specifications as programs. The price paid, though, is that λProlog adopts higher-order unification which is an undecidable procedure in general. However, the main limitation for supporting compositionality in both these logic languages is their use of the inflexible left-to-right computation rule which causes many simple compositional programs to fail to terminate. Despite this behaviour, we demonstrated that the problem of the left-to-right computation rule in λProlog can be partially solved in an implementation of the relational language, Ruby. The implementation relied on a heuristic method of reordering literals in the body of a clause. Unfortunately, such heuristics fail to generalise to arbitrary compositional programs and hinder the construction of truly declarative programs.

The exploration of this chapter suggests that logic programming forms an appropriate starting point for creating a compositional programming language. The deficiencies of logic programming that must be addressed, however, are two-fold: the lack of elegant higher-order programming constructs that eliminate the need for extra-logical predicates; and, secondly, the need for a computation rule that guarantees termination of compositional programs whilst retaining their efficient execution. These issues are addressed in the remainder of the thesis.

Chapter 3

Compositionality in Logic Languages

In this chapter, we present the language features necessary for a logic language to support the compositional style of programming proposed in the previous chapters. The first language feature we introduce is a curried syntax for the terms of the language which allows specifications to be phrased naturally without the need for extra-logical predicates, as is the case with Prolog and Mercury. We show that the adoption of curried terms in a first-order language does not affect the definition of SLD-resolution in any way and necessitates only superficial changes to the standard unification algorithm.

The most significant language feature which facilitates compositionality in a logic language is the use of *fair* SLD-resolution. Fair SLD-resolution was introduced to the logic programming community by (Lassez & Maher 1984) to provide a theoretical understanding of negation-as-failure. We augment their results by proving that the fair SLD-resolution of a query is guaranteed to terminate whenever any other resolution strategy would also terminate for it. Thus, fair SLD-resolution rectifies the poor termination behaviour caused by the left-to-right computation rule and, consequently, provides a method by which compositional programs can be constructed independently of the evaluation order of the language. A summary of this work appears in (McPhee & de Moor 1996).

Despite the theoretical elegance of fair SLD-resolution, however, we show that the use of the 'breadth-first' fair computation rule often exhibits poor execution efficiency when evaluating compositional programs. In a previous, though entirely independent, treatment of fair SLD-resolution, (Janot 1991) introduced indexed fair SLD-resolution to overcome the efficiency problems associated with the breadth-first computation rule. Although indexed fair SLD-resolution does improve performance over the breadth-first computation rule, we show that it fails to support the desiderata of a compositional language outlined in Chapter 2.

One often cited alternative to the left-to-right computation rule in the logic programming community is the use of *coroutining* computation rules (Lüttringhaus-Kappel 1993, Naish 1992). Coroutining ostensibly offers improved termination behaviour over the left-to-right one whilst simultaneously maintains efficient execution. We examine this class of computation rules and show that, like indexed fair SLD-resolution, they fall short of allowing a general implementation of compositional programming.

3.1 A higher-order syntax

Higher-order programming constructs aid compositionality by encouraging abstraction during program construction, as exemplified in the programs presented in Chapter 1. In this section, we employ a curried syntax for logic programs which naturally allows variables to appear as the head of a literal, previously reserved only for predicate symbols (Definition 2.1). Consequently, compositional programs can be written without extra-logical predicates—currently required in languages like Prolog—since higher-order variables can be instantiated directly during the application of a unifier over a goal. The new definition of terms is as follows.

Definition 3.1 (Curried terms). Let X be a set of variables and Ω a set of constant symbols. The *terms*, $\mathbb{T}'_\Omega(X)$, of the programming language are defined inductively by the grammar

$$\begin{aligned}\mathbb{T}'_\Omega(X) ::= {} & x \\ \mid {} & c \\ \mid {} & s\ t\end{aligned}$$

where $x \in X$, $c \in \Omega$, and $s, t \in \mathbb{T}'_\Omega(X)$. The application of two terms is written by juxtaposition and associates to the left. $\diamondsuit$

The definitions of definite clauses, programs, goal clauses, and substitutions from Section 2.3.1 each abstract over the actual form of literals and, as such, are unaffected by the adoption of curried terms. Furthermore, the change to curried terms necessitates only superficial alterations to the application of a substitution over a term (Definition 2.5). A unification algorithm for curried terms is depicted in Figure 3.1 which contains only minor differences to the standard one from Figure 2.1. Moreover, the correctness proof of the new unification algorithm is identical to the one for the standard algorithm.

Owing to the fact that a change to curried terms is entirely syntactic, it is straightforward to show that the class of programs that can be expressed using curried terms is precisely the class of pure Prolog programs that may also contain occurrences of the *call* extra-logical predicate. We define the set of *call* terms that may appear in a pure Prolog program as follows.

Definition 3.2 (Call terms). Let X be a set of variables and Ω a set of constant symbols. The *call* terms, $\mathbb{C}_\Omega(X)$, are defined by the grammar

$$\mathbb{C}_\Omega(X) ::= \mathit{call}(t_1, \ldots, t_n)$$

where $n \geq 1$ and each $t_i \in \mathbb{T}_\Omega(X)$, for $1 \leq i \leq n$. $\diamondsuit$

As discussed in Section 2.3.4, the *call* literal is provided by most Prolog implementations to permit higher-order logic variables to appear in programs; without *call*, such programs would be syntactically incorrect. The semantic interpretation of a *call* literal during a resolution step is given by the following equations:

$$\begin{aligned} & \mathit{call}(f(s_1, \ldots, s_m), t_1, \ldots, t_n) \\ = {} & \mathit{call}(f, s_1, \ldots, s_m, t_1, \ldots, t_n) \\ = {} & f(s_1, \ldots, s_m, t_1, \ldots, t_n)\end{aligned}$$

```
$unify\ u\ v =$
  let $D := \{(u, v)\}, \phi := \iota, ok :=$ true
  while $ok$ and $D \neq \{\,\}$ do
    let $D' \cup \{(s, t)\} = D$
      if $s = x = t \longrightarrow D := D'$
      $[\!]$ $s = x \notin t \longrightarrow D := D'[x \mapsto t], \phi := \{x \mapsto t\} \circ \phi$
      $[\!]$ $t = x \notin s \longrightarrow D := D'[x \mapsto s], \phi := \{x \mapsto s\} \circ \phi$
      $[\!]$ $s = s_1\ s_2$ and $t = t_1\ t_2 \longrightarrow D := D' \cup \{(s_1, t_1), (s_2, t_2)\}$
      $[\!]$ otherwise $\longrightarrow ok :=$ false
      fi
  od
  if $ok \longrightarrow$ Just $\phi$
  $[\!]$ otherwise $\longrightarrow$ Nothing
  fi
```

Figure 3.1: The new unification algorithm for two terms.

where $m, n \geq 0$, f is a constructor of arity $m+n$, and each s_i and t_i are terms. Whenever a *call* literal is selected for resolution, it must be in either of the forms above otherwise the resolution step fails.

A simple equivalence exists between curried terms and regular terms augmented with the *call* predicate, characterised by the translation function $\alpha : \mathbb{T}_\Omega(X) \cup \mathbb{C}_\Omega(X) \to \mathbb{T}'_\Omega(X)$ whose definition is given below.

$$\alpha(t) = \begin{cases} x, & \text{if } t = x \in X \\ f\ (\alpha(t_1))\ \ldots\ (\alpha(t_n)), & \text{if } t = f(t_1, \ldots, t_n) \\ f\ (\alpha(s_1))\ \ldots\ (\alpha(s_m))\ (\alpha(t_1))\ \ldots\ (\alpha(t_n)), & \\ \qquad \text{if } t = \mathit{call}(f(s_1, \ldots, s_m), t_1, \ldots, t_n) & \end{cases} \tag{3.1}$$

It is straightforward to show that the function α has an inverse modulo the equality of *call* literals stated above. In order to formalise the aforementioned equivalence, we introduce a few additional concepts.

Given a substitution $\phi = \{x_1 \mapsto s_1, \ldots, x_n \mapsto s_n\} : X \to \mathbb{T}_\Omega(Y)$, by a slight abuse of notation we define the substitution $\alpha(\phi) : X \to \mathbb{T}'_\Omega(Y)$ as follows:

$$\alpha(\phi) = \{x_1 \mapsto \alpha(s_1), \ldots, x_n \mapsto \alpha(s_n)\}$$

For a term $t \in \mathbb{T}_\Omega(X)$, it is easy to prove that

$$\alpha(t[\phi]) = (\alpha(t))[\alpha(\phi)] \tag{3.2}$$

The following lemma proves the equivalence of programs written using curried terms and those written using standard terms augmented with the *call* predicate.

Lemma 3.1. *Every pure Prolog program augmented with the call extra-logical primitive can be equivalently expressed using curried terms.*

Proof. It suffices to show that for any SLD-derivation $D = G_1, G_2, \ldots,$ in Prolog, an equivalent one exists using curried terms. Consider two successive triples G_i and G_{i+1} in D. Let A_1 be the selected literal in G_i and $A :- B_1, \ldots, B_m$ be the selected clause. Then the resolvent of the resolution step is the goal

$$\# :- B_1[\phi], \ldots, B_m[\phi]$$

where ϕ is the substitution of the resolution. By applying α to this resolvent, we obtain the resolvent that should be computed in the corresponding derivation using curried terms:

$$\# :- \alpha(B_1[\phi]), \ldots, \alpha(B_m[\phi]) \tag{3.3}$$

Now, in the equivalent derivation using curried terms, the selected literal is $\alpha(A_1)$ and the chosen clause is $\alpha(A) :- \alpha(B_1), \ldots, \alpha(B_m)$. The resolvent of this resolution step is

$$\# :- (\alpha(B_1))[\phi'], \ldots, (\alpha(B_m))[\phi']$$

where $\phi' = \alpha(\phi)$, a fact that is clear from the similarity of the two unification algorithms. By m applications of (3.2), the previous resolvent is equivalent to

$$\# :- \alpha(B_1[\phi]), \ldots, \alpha(B_m[\phi])$$

which is identical to the resolvent (3.3). Since this equivalence holds for all such triples in D, we have the desired result. ♢

3.2 Fair SLD-resolution

The main problem with logic programming languages with respect to supporting compositionality is the use of the left-to-right computation rule. As discussed in Section 2.3.5, the use of this rule often causes non-termination of even simple logic programs. Consequently, the programmer must be aware of the restrictions imposed by the rule, therefore reducing the declarative nature of programs. Moreover, the same program will generally fail to terminate for a variety of argument instantiations. The main aim of compositional logic programming, however, is to relieve the programmer from such burdens. Thus, an alternative computation rule must be discovered that promotes, rather than restricts, declarative programming.

Fortunately, a class of computation rules exists that guarantees the termination of a logic program whenever *any* other computation rule would do so. The class of rules, called *fair* computation rules, ensures that every literal in a goal is selected for resolution after a finite number of resolution steps. A fair SLD-derivation captures this behaviour and is formalised in the following definition, originally due to (Lassez & Maher 1984).

Definition 3.3 (Fair SLD-derivation). An SLD-derivation of a goal $\# :- A_1, \ldots, A_m$, is *fair* if it is either failed or, for every literal A_i, for $1 \leq i \leq m$, (some further instantiation of) A_i is selected within a finite number of resolution steps. ♢

A simple way to institute a fair computation rule is to queue the literals pending resolution in a goal, i.e., add the literals introduced by a resolution step to the end of the current goal. This is called the *breadth-first* fair computation rule. A simple adjustment to Definition 2.7 suffices to define fair SLD-resolution.

Definition 3.4 (Fair SLD-resolution). Let $C = A \mathrel{:-} B_1, \ldots, B_m$, for $m \geq 0$, be a definite clause, and $G = \# \mathrel{:-} A_1, \ldots, A_n$, for $n \geq 1$, a goal such that A_1 and A are unifiable with most general unifier ϕ. Then the *SLD-resolvent* of C and G is the goal

$$G' = (\# \mathrel{:-} A_2, \ldots, A_n, B_1, \ldots, B_m)[\phi]$$

The substitution ϕ is called the *substitution of the SLD-resolution.* $\diamondsuit$

Varying the computation rule in an SLD-resolution does not affect the answer substitutions contained in the corresponding SLD-tree—the number of success nodes in an SLD-tree is invariant with respect to the computation rule—but rather alters the shape of the SLD-tree. Therefore, the choice of computation rule has a significant effect on the overall efficiency of a resolution. The most important observation of fair SLD-resolution is that if any computation rule can be used to construct a finite SLD-tree for a query, then every fair computation rule constructs a finite tree. In other words, if an infinite loop in a program can be avoided by *some* computation rule then *any* fair computation rule will avoid it. In order to prove this claim, we will require the following two lemmas.

The first lemma, stated and proved by (Lloyd 1987), demonstrates that the order in which subgoals are selected for resolution is irrelevant in a derivation. Given a program clause C, the notation $C^{:-}$ represents the literals in the body of C.

Lemma 3.2 (Switching Lemma). *For a program P and a goal G, suppose that $P \cup \{G\}$ has an SLD-derivation $(G, C_1, \phi_1), \ldots, (G_n, C_n, \phi_n)$, for $n \geq 1$. Suppose also that, for $1 < k < n$, we have the goals*

$$\begin{aligned} G_{k-1} &= (\# \mathrel{:-} A_1, \ldots, A_i, \ldots, A_j, \ldots, A_m) \\ G_k &= (\# \mathrel{:-} A_1, \ldots, C_k^{:-}, \ldots, A_j, \ldots, A_m)[\phi_k] \\ G_{k+1} &= (\# \mathrel{:-} A_1, \ldots, C_k^{:-}, \ldots, C_{k+1}^{:-}, \ldots, A_m)[\phi_{k+1} \circ \phi_k] \end{aligned}$$

Then there exists an SLD-derivation of $P \cup \{G\}$ in which A_j is selected in G_{k-1} instead of A_i, and A_i is selected in G_k instead of A_j.

Fair SLD-resolution was introduced by (Lassez & Maher 1984) to provide a description of negation-as-failure. They showed that fair SLD-resolution is complete with respect to finite failure: whenever some SLD-resolution of a goal finitely fails, then every fair SLD-resolution of that goal also finitely fails. Despite this result, they neglected to formulate an equivalent result regarding finite success: whenever some SLD-resolution of a goal finitely succeeds, then every fair SLD-resolution of that goal also finitely succeeds. We provide precisely such a result in the following novel lemma and corollary. A similar result has been independently stated and proved by (Janot 1991).

Lemma 3.3. *For a program P and a goal G, if there exists an infinite fair SLD-tree for $P \cup \{G\}$ then every SLD-tree for $P \cup \{G\}$ is infinite.*

Proof. Let $D = G_1, G_2, \ldots,$ be an infinite, fair SLD-derivation in the fair SLD-tree for $P \cup \{G\}$, and R be some arbitrary computation rule. It suffices to show that D can be transformed into some other infinite SLD-derivation that uses R as its computation rule. By fairness, we know that all the literals of G are selected in D after a finite number of steps. Moreover, it must be the case that R selects at least one literal of G. We proceed by induction over the length n of the new infinite derivation.

For $n = 1$, let A_1 be the selected literal in G_1 of D and B_1 be the selected literal in G_1 using R. Since D is fair, a further instantiated version of B_1 must be selected in G_k, for $k > 1$. By k applications of the Switching Lemma, we can construct an infinite, fair SLD-derivation, D_1, where B_1 is selected in G_1 instead of A_1.

For $n > 1$, let $D_n = G_1, G_2, \ldots, G_n, \ldots,$ be an infinite, fair SLD-derivation with the same initial $n-1$ selected literals as in the derivation via R, by the induction hypothesis. Let A_i be the selected literal in G_n of D_n, and B_i the selected literal in G_n for the derivation via R. The first $n - 1$ selected literals are the same in both derivations and so B_i must be selected in goal G_{k+p}, for some $p \geq 0$, since D_n is fair. By p applications of the Switching Lemma, we can construct a new infinite, fair SLD-derivation D_{n+1} where B_i is selected in G_n instead of A_i, and so selects the same initial n literals as in the derivation via R.

Therefore, for an arbitrary computation rule R, we can construct an infinite SLD-derivation $D' = G_1, G_n, \ldots, G_i, \ldots,$ via R, such that for all $i \geq 1$, there is an infinite fair SLD-derivation D_j, for $j > i$, that selects the same i initial literals as D'. Since R is arbitrary, then all SLD-derivations must be infinite. $\diamondsuit$

The following corollary to Lemma 3.3 establishes the termination behaviour of fair SLD-resolution.

Corollary 3.1. *For a program P and a goal G, if some SLD-tree for $P \cup \{G\}$ is finite, then every fair SLD-tree for $P \cup \{G\}$ is finite.*

Proof. Immediate from Lemma 3.3 by the contrapositive law. $\diamondsuit$

The corollary above guarantees that the fair SLD-resolution of a goal will terminate if it it is possible to do so using *any* other computation rule. The following example provides an illustration of this valuable behaviour.

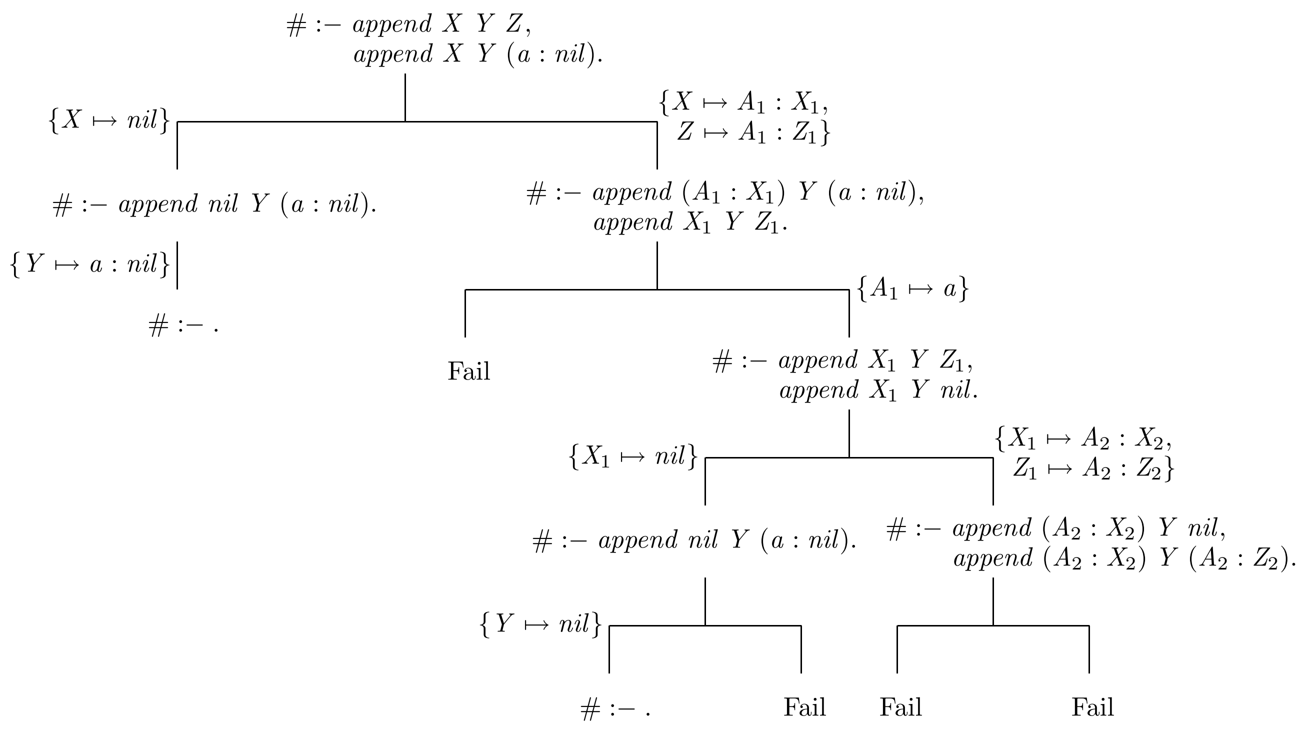
Example 3.1. Consider again the predicate *append* that concatenates two lists:

$$\begin{array}{l} append\ nil\ Y\ Y\ \ :-\ \ . \\ append\ (A : X)\ Y\ (A : Z)\ \ :-\ \ append\ X\ Y\ Z. \end{array}$$

Furthermore, consider the following query which was initially presented in Example 2.3.

$$\#\ :-\ \ append\ X\ Y\ Z,\ append\ X\ Y\ (a : nil).$$

In that example, we saw that evaluating the query using the left-to-right computation rule resulted in an infinite SLD-tree (which was depicted in Figure 2.4). However, the corresponding fair SLD-resolution of this query using the breadth-first computation rule is is finite, producing the SLD-tree shown in Figure 3.2. $\diamondsuit$



$\# :- \mathit{append}\ X\ Y\ Z, \mathit{append}\ X\ Y\ (a : \mathit{nil}).$
$\{X \mapsto \mathit{nil}\}$
$\{X \mapsto A_1 : X_1, Z \mapsto A_1 : Z_1\}$
$\# :- \mathit{append}\ \mathit{nil}\ Y\ (a : \mathit{nil}).$
$\# :- \mathit{append}\ (A_1 : X_1)\ Y\ (a : \mathit{nil}), \mathit{append}\ X_1\ Y\ Z_1.$
$\{Y \mapsto a : \mathit{nil}\}$
$\{A_1 \mapsto a\}$
$\# :- .$
Fail
$\# :- \mathit{append}\ X_1\ Y\ Z_1, \mathit{append}\ X_1\ Y\ \mathit{nil}.$
$\{X_1 \mapsto \mathit{nil}\}$
$\{X_1 \mapsto A_2 : X_2, Z_1 \mapsto A_2 : Z_2\}$
$\# :- \mathit{append}\ \mathit{nil}\ Y\ (a : \mathit{nil}).$
$\# :- \mathit{append}\ (A_2 : X_2)\ Y\ \mathit{nil}, \mathit{append}\ (A_2 : X_2)\ Y\ (A_2 : Z_2).$
$\{Y \mapsto \mathit{nil}\}$
$\# :- .$
Fail
Fail
Fail

Figure 3.2: The fair SLD-tree for $\# :- \mathit{append}\ X\ Y\ Z, \mathit{append}\ X\ Y\ (a : \mathit{nil})$.

The above properties of fair SLD-resolution are essential in a compositional language that requires the termination behaviour of programs to be independent of the way in which they are composed from smaller ones. To see why fair SLD-resolution satisfies this requirement, consider the following predicate that relates a binary tree to its frontier:

$$
\begin{aligned}
&\mathit{frontier}\ (\mathit{tip}\ A)\ (A : \mathit{nil})\ :-\ .\\
&\mathit{frontier}\ (\mathit{bin}\ S\ T)\ Z\ :-\\
&\quad \mathit{frontier}\ S\ (A : X),\ \mathit{frontier}\ T\ (B : Y),\ \mathit{append}\ (A : X)\ (B : Y)\ Z.
\end{aligned}
$$

Now, suppose we attempt to solve the following goal using the left-to-right computation rule:

$$\#\ :-\ \mathit{frontier}\ T\ (a : b : c : \mathit{nil})\ . \qquad (3.4)$$

The proof search in this instance reports one solution for T and then enters an infinite loop since it must resolve the literal *frontier* $S\ (A : X)$ which has an infinite number of solutions. By contrast, a fair SLD-resolution discovers both solutions for T and then terminates.

The standard solution to this problem of entering an infinite loop in a language like Prolog is to write another predicate

$$
\begin{aligned}
&\textit{cfrontier}\ (A : \textit{nil})\ (\textit{tip}\ A)\ \ {:-}\ \ .\\
&\textit{cfrontier}\ Z\ (\textit{bin}\ S\ T)\ \ {:-}\\
&\quad \textit{append}\ (A : X)\ (B : Y)\ Z,\ \textit{cfrontier}\ (A : X)\ S,\ \textit{cfrontier}\ (B : Y)\ T.
\end{aligned}
$$

However, rewriting a program according to the context in which it is used defies the notion of compositionality identified in Section 2.1. Without the ability to write programs without regard for the evaluation order of the language, we would be unable to implement the elementary composition operator, converse:

$$\textit{conv}\ R\ A\ B\ \ {:-}\ \ R\ B\ A\ .$$

A minimum healthiness requirement for the two programs above is the equivalence

$$\textit{conv frontier} = \textit{cfrontier}.$$

However, the two sides of this equation exhibit different behaviours under a left-to-right computation rule, as can be seen from the execution of query (3.4). On the other hand, fair SLD-resolution guarantees that, for programs that exhibit only finite nondeterminism, such surprises do not occur. Therefore, a fair computation rule is necessary to implement relational converse, and this argument can be made for other elementary composition operators such as relational composition and intersection.

3.3 Search strategies

In Section 2.3.5, we demonstrated that depth-first search, coupled with the left-to-right computation rule, hampered compositionality by causing the non-termination of many programs. Depth-first search is incomplete because it can plummet down an infinite search path before discovering all proofs of a query. One way to tackle this problem—without adding extra-logical control annotations to programs—is to adopt a complete search strategy. Breadth-first search is one such complete strategy but it suffers from the well-known disadvantage of being space-inefficient.

An alternative search strategy that attempts to address the incompleteness of depth-first search is *bounded depth-first search* (Korf 1985) which limits the depth to which an SLD-tree may be explored. The depth bound indicates the maximum number of resolution steps that can be performed in any SLD-derivation with the result that only a finite portion of a possibly infinite search space is actually considered. The choice of value for the depth bound, however, is an inherent problem in bounded search: if the chosen bound is too small then some solutions may be missed. On the other hand, if the bound is too large, much unnecessary computation may be performed.

One way to address this problem is to search with an increasing depth bound. That is, we begin by searching for all solutions up to some initial depth bound and then successively increment the depth bound, searching for solutions up to the new depth. The resulting strategy, called *depth-first iterative deepening* (Stickel 1988, Korf 1985), emulates breadth-first search except that previously visited nodes are not retained; rather, we commence the search again from the original goal each time the depth bound is increased. The consequential redundant computation incurred by iterative deepening might, at first sight, seem terribly inefficient. However, considering that the search space can increase exponentially between increments of the depth bound, it becomes apparent that the repeated work constitutes a small proportion of the overall size of the search space.

According to the suggestions of (Korf 1985), search strategies should be compared by considering trade-offs in terms of space, time, and the length of solution paths. Moreover, Korf argues that, of all the exhaustive search strategies, depth-first iterative deepening fairs the best on these three criteria. Nevertheless, the primary aim of compositional programming is to produce terminating programs whenever possible, i.e, to construct finite SLD-trees. Unfortunately, every exhaustive search strategy will fail to terminate when traversing an infinite SLD-tree: different search strategies only affect the number of solutions found before committing themselves to an infinite branch in the tree. Since fair computation rules are guaranteed to construct finite SLD-trees whenever possible, the choice of search strategy becomes less significant and we simply adopt depth-first search for its easy and efficient implementation.

Nevertheless, compositional programs that exhibit unbounded nondeterminism can never have a finite SLD-tree, irrespective of the computation rule used. In such instances, the choice of search strategy becomes essential for the fair enumeration of the infinite number of solutions. To illustrate this point, let us return to the pretty-printing example of Section 1.1 and consider the following query in which the parser is run backwards to produce a pretty-print of an abstract syntax tree:

$$\begin{aligned} \# \;:-\;\; & \mathit{parse}\ X\ (\mathit{add}\ (\mathit{num}\ \text{“2”}) \\ & \qquad (\mathit{mul}\ (\mathit{sub}\ (\mathit{num}\ \text{“4”})\ (\mathit{num}\ \text{“1”}))\ (\mathit{num}\ \text{“3”})))\ \text{“ ”}. \end{aligned}$$

The query produces an infinite sequence of solutions:

$$\begin{aligned} X &= \text{“}2 + (4 - 1) * 3\text{”}\ ; \\ X &= \text{“}2 + (4 - (1)) * 3\text{”}\ ; \\ X &= \text{“}2 + (4 - ((1))) * 3\text{”}\ . \\ & \mathit{yes} \end{aligned}$$

In this example, the use of depth-first search results in successive pretty-prints that contain extra parentheses around the same subexpression. By using a complete search strategy, like breadth-first search or depth-first iterative deepening, a wider variety of pretty-prints would be obtained. Therefore, in an ideal compositional logic language, both a fair computation rule and a complete search strategy would be desirable to enhance the ability of a programmer to construct compositional programs without regard for the evaluation order of the language.

3.4 The limitations of fair SLD-resolution

Fair SLD-resolution satisfies the language requirement which states that the termination behaviour of a compositional program should not rely on the way in which it is composed. However, the fair SLD-resolution of a query is only guaranteed to terminate whenever any other SLD-resolution of it does also. As we explain in this section, one limitation of

SLD-resolution is that, irrespective of the choice of computation rule, it is often unable to terminate for the class of left-recursive programs, of which many compositional programs are an instance. Another limitation of fair SLD-resolution that we discuss in this section is that its efficiency behaviour when executing compositional programs can be rather difficult to predict.

3.4.1 Left-recursive programs

Although fair SLD-resolution is guaranteed to construct a finite SLD-tree whenever any other computation rule would, fairness alone is not enough to remove all possible infinite branches in an SLD-tree. In particular, it is a trivial undertaking to construct programs that loop infinitely for any computation rule. For example, the program

$$loop \;:-\; loop.$$

and the query # :− *loop* is one such contrived case. However, the inability of resolution to solve queries of this form is not confined solely to such pathological cases. In compositional programming, logically correct specifications can fail to terminate when executed under any form of resolution, as illustrated in the following example.

Example 3.2. Consider the task of computing the reflexive, transitive closure of a relation using a calculus similar to that presented in Section 2.2.1. The basic combinators are illustrated below:

$$\begin{array}{lll} id\ A\ A\ :-\ . \quad & succ\ (s\ N)\ N\ :-\ . \quad & (R\ \$cup\ S)\ A\ B\ :-\ R\ A\ B. \\ & & (R\ \$cup\ S)\ A\ B\ :-\ S\ A\ B. \\ \multicolumn{3}{l}{(R\ \$comp\ S)\ A\ C\ :-\ R\ A\ B,\ S\ B\ C.} \end{array}$$

The transitive closure of a relation is computed by the predicate *close*:

$$close\ R\ A\ B\ :-\ (id\ \$cup\ ((close\ R)\ \$comp\ R))\ A\ B.$$

Now, consider using fair SLD-resolution to solve the following query which has only one solution, namely $\{A \mapsto 0\}$:

$$\#\ :-\ close\ succ\ 0\ A. \qquad (3.5)$$

The fair SLD-tree of this query is shown in Figure 3.3. The branch labelled (1) of the tree ends in success, binding A to 0. Branch (2) ends in failure since the goal # :− *id* 0 (*s* A) cannot be satisfied. Finally, branch (3) does not terminate because a similar search tree to the one above is repeated infinitely. Indeed, resolution always results in non-termination for (3.5), irrespective of the computation rule. ◇

Query (3.5) illustrates an example of a *left-recursive* program. Termination is impossible since the proof of *close succ* 0 A depends on a variant of itself, i.e., the literal *close succ* 0 B. The notoriety of left-recursion stems partially from the fact that little can be done to overcome this problem short of rewriting the program or adopting a novel computation mechanism that can detect and avoid infinite branches in a proof tree. (Such a computation mechanism is developed in the following chapter that does

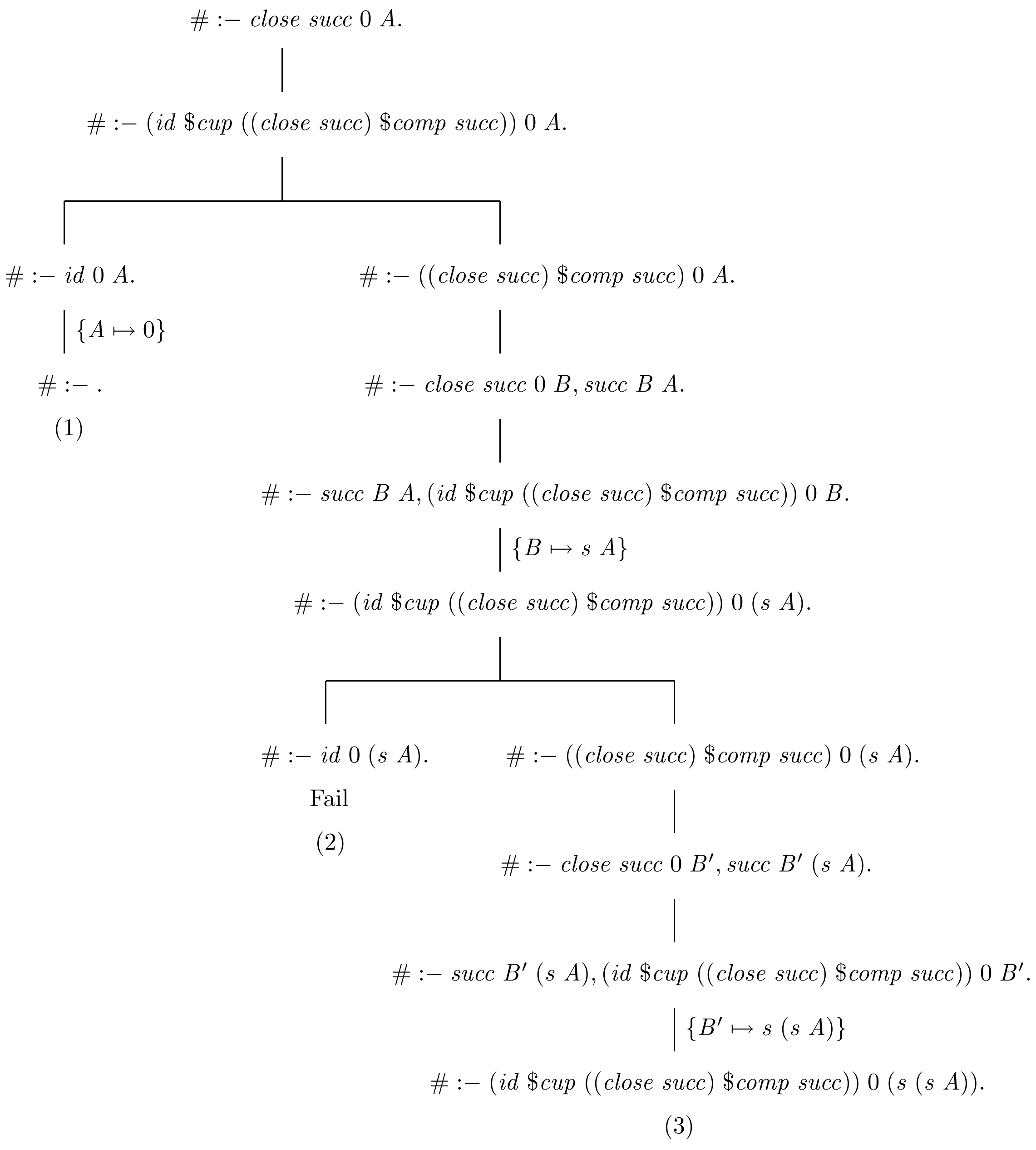

$\# :- \textit{close succ } 0\ A.$
$\# :- (\textit{id } \$\textit{cup } ((\textit{close succ}) \$\textit{comp succ}))\ 0\ A.$
$\# :- \textit{id } 0\ A.$
$\{A \mapsto 0\}$
$\# :- .$
(1)
$\# :- ((\textit{close succ}) \$\textit{comp succ})\ 0\ A.$
$\# :- \textit{close succ } 0\ B, \textit{succ } B\ A.$
$\# :- \textit{succ } B\ A, (\textit{id } \$\textit{cup } ((\textit{close succ}) \$\textit{comp succ}))\ 0\ B.$
$\{B \mapsto s\ A\}$
$\# :- (\textit{id } \$\textit{cup } ((\textit{close succ}) \$\textit{comp succ}))\ 0\ (s\ A).$
$\# :- \textit{id } 0\ (s\ A).$
Fail
(2)
$\# :- ((\textit{close succ}) \$\textit{comp succ})\ 0\ (s\ A).$
$\# :- \textit{close succ } 0\ B', \textit{succ } B'\ (s\ A).$
$\# :- \textit{succ } B'\ (s\ A), (\textit{id } \$\textit{cup } ((\textit{close succ}) \$\textit{comp succ}))\ 0\ B'.$
$\{B' \mapsto s\ (s\ A)\}$
$\# :- (\textit{id } \$\textit{cup } ((\textit{close succ}) \$\textit{comp succ}))\ 0\ (s\ (s\ A)).$
(3)

Figure 3.3: The fair SLD-tree for the query # :− *close succ* 0 *A*.

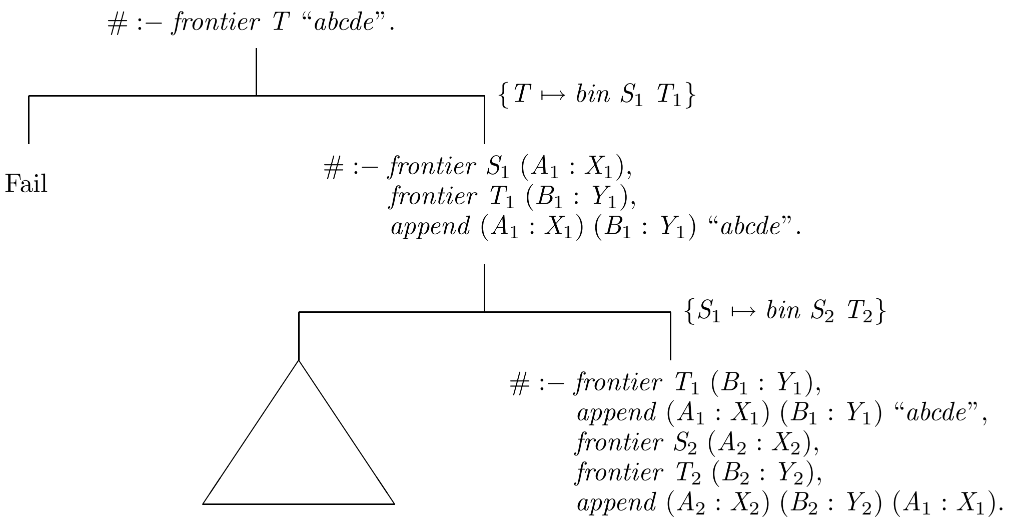

$\# :- \textit{frontier}\ T$ "abcde".
$\{T \mapsto \textit{bin}\ S_1\ T_1\}$
Fail
$\# :- \textit{frontier}\ S_1\ (A_1 : X_1)$, $\textit{frontier}\ T_1\ (B_1 : Y_1)$, $\textit{append}\ (A_1 : X_1)\ (B_1 : Y_1)$ "abcde".
$\{S_1 \mapsto \textit{bin}\ S_2\ T_2\}$
$\# :- \textit{frontier}\ T_1\ (B_1 : Y_1)$, $\textit{append}\ (A_1 : X_1)\ (B_1 : Y_1)$ "abcde", $\textit{frontier}\ S_2\ (A_2 : X_2)$, $\textit{frontier}\ T_2\ (B_2 : Y_2)$, $\textit{append}\ (A_2 : X_2)\ (B_2 : Y_2)\ (A_1 : X_1)$.

Figure 3.4: A fragment of the fair SLD-tree for # :− *frontier T* "*abcde*".

allow both # :− *loop* and (3.5) to terminate.) The predicate *close* from Example 3.2 can be rewritten as follows, replacing left- with right-recursion:

$$\textit{close}\ R\ A\ B\ :-\ (\textit{id}\ \$\textit{cup}\ (R\ \$\textit{comp}\ (\textit{close}\ R)))\ A\ B.$$

This alteration is enough to permit (3.5) to terminate, as desired.

3.4.2 Efficiency and the choice of fair computation rule

The idyllic theoretical foundations of fair SLD-resolution unfortunately incur a severe efficiency penalty in practice: the breadth-first computation rule often generates many branches in the fair SLD-tree that are destined to finitely-fail and, hence, contribute nothing to the search for refutations. The exploration of these dead-end derivations can have an adverse affect on the efficiency of the computation. The following example illustrates the problem.

Example 3.3. Consider again the *frontier* predicate from Section 3.2 and the following query that enumerates all the binary trees that have "*abcde*" as their leaves.

$$\#\ :-\ \textit{frontier}\ T\ \text{"}\textit{abcde}\text{"}. \tag{3.6}$$

A fragment of the fair SLD-tree for (3.6) is depicted in Figure 3.4. However, searching this tree, in practical terms, takes such a long time that the computation is rendered virtually infeasible. The main reason for the explosive growth of the search tree arises from the fact that variants of the literal *frontier* S_1 $(A_1 : X_1)$ are solved redundantly many times in the computation. Moreover, the proofs of such literals are cyclic since they introduce two further variants of themselves at each resolution step. ♢

The only requirement for fairness in a resolution is that each literal in a goal is selected for resolution after a finite number of resolutions steps; the breadth-first computation rule of Definition 3.4 is perhaps the most naive method of achieving fairness. However, one can imagine alternative fair computation rules that exhibit better efficiency behaviour in practice than the breadth-first one. Indeed, (Janot 1991), who independently identified the significance of fair SLD-resolution, suggested several alternative fair computation rules to improve the efficiency of fair SLD-resolution. The most successful of these is the *indexed* fair computation rule which performs a bounded number of resolution steps using the left-to-right computation rule before performing a resolution step using the breadth-first one.

Indexed fair SLD-resolution attaches a natural number—the so-called *index*—to each literal in the body of a clause. In terms of notation, a literal A that is indexed with the number n is denoted $A[n]$. The definition of indexed fair SLD-resolution is given by (Janot 1991) as follows.

Definition 3.5 (Indexed fair SLD-resolution). Let $C = A :- B_1[i_1], \ldots, B_m[i_m]$, for $m \geq 0$, be an indexed definite clause and $G = \# :- A_1[j_1], \ldots, A_n[j_n]$, for $n \geq 1$, an indexed goal such that A_1 and A are unifiable with most general unifier ϕ. Then the *SLD-resolvent* of C and G is the goal

$$G' = \begin{cases} (\# :- A_2[j_2], \ldots, A_n[j_n], B_1[i_1], \ldots, B_m[i_m])[\phi], & \text{if } j_1 = 1 \\ (\# :- B_1[k_1], \ldots, B_m[k_m], A_2[j_2], \ldots, A_n[j_n])[\phi], & \text{if } j_1 > 1 \end{cases}$$

where $k_p = \min(j_1 - 1, i_p)$, for $1 \leq p \leq m$. ◇

So, if the index of the selected literal is equal to 1, new literals are introduced at the end of the goal. Otherwise, the new literals are introduced at the beginning of the goal, but their respective indices must be strictly less than the selected literal's index. By doing so, only a finite number of literals can be added to the beginning of a goal before using a breadth-first resolution step to maintain fairness.

Although (Janot 1991) demonstrated substantial improvements to the efficiency of fair SLD-resolution using indexing, the process of determining appropriate indices for literals is an arbitrary endeavour based solely upon trial-and-error observations for various values of index for a particular query. Indexed fair SLD-resolution falls short of the criteria for a compositional logic language identified at the beginning of Chapter 2: the heuristic nature of determining indices places too much of a burden on the programmer and provides little or no consistency in execution efficiency of different programs. Therefore, the declarative nature of compositional programs is hampered when using indexed fair SLD-resolution. In Chapters 4 and 5, we present novel resolution strategies that attempt to satisfy the requirements of compositionality more accurately than the computation rules suggested by (Janot 1991).

3.5 Using coroutining to improve the efficiency of fair SLD-resolution

A key criterion for the support of compositionality in a logic language is the adoption of a flexible computation rule to overcome the termination problems of the left-to-right one. Coroutining computation rules have been introduced by (Lüttringhaus-Kappel 1993, Naish 1992, Naish 1985) to address precisely this problem of the left-to-right computation rule whilst simultaneously maintaining efficiency. A natural response to the efficiency problems inherent with fair SLD-resolution is to suggest the use of a coroutining computation rule over a fair one. In this section, therefore, we review coroutining computation rules and examine the extent to which they can be used to overcome the practical limitations of fair SLD-resolution.

3.5.1 The benefits of a coroutining computation rule

The successful resolution of a literal depends crucially on the terms that its arguments are bound to. Certain bindings will contribute more information to the computation by instantiating variables in other subgoals and may also allow the computation to terminate. It is well known in the logic programming community that selecting literals whose arguments are bound to terms rather than variables can be significantly more flexible than Prolog's left-to-right computation rule, facilitating the termination of more programs than Prolog. In particular, the use of a coroutining computation rule (Lüttringhaus-Kappel 1993, Naish 1992, Ullman & Gelder 1988) attempts to overcome the limitations of the left-to-right computation rule by selecting literals driven purely by the data flow, i.e., the interactions of subgoals by the instantiation of common variables.

Let us illustrate coroutining computation rules by examining once again the *append* predicate:

$$\begin{array}{l} \textit{append nil } Y\ Y\ \ {:}{-}\ \ . \\ \textit{append } (A : X)\ Y\ (A : Z)\ \ {:}{-}\ \ \textit{append } X\ Y\ Z. \end{array}$$

Consider the query

$$\#\ {:}{-}\ \ \textit{append } X\ Y\ Z,\ \textit{append } X\ Y\ (a : \textit{nil}).$$

A coroutining computation rule would avoid selecting the leftmost subgoal *append* $X\ Y\ Z$ since all its arguments are variables. Instead, *append* $X\ Y\ (a : \textit{nil})$ would be selected: its third argument is bound to a term other than a variable and, moreover, the variables X and Y would receive 'meaningful' bindings after a resolution step.

The idea of a meaningful binding can be defined using various criteria. For example, the clauses of a predicate could be examined to identify those arguments used for structural induction. Then, literals with these arguments bound to a term may reduce the size of the remaining proof and, therefore, would be selected for resolution over others. The resolution process that performs coroutining computation is reviewed in the following section.

3.5.2 SLDF-resolution

A coroutining computation rule makes use of information extracted from a program to determine those instances of a predicate that are likely to maximise dataflow in a resolution step. A *call set* for a program provides a representation of those *callable literals* that may be selected during a resolution step. For the time being, we assume that the call set already exists for a program, although we return to how it can be automatically constructed in Section 3.5.4. We now present several definitions that permit the description of a computation rule that selects literals according to the call set. In particular, the call set in coroutining is simply a set of literals and a literal in a goal is callable if an instance of it appears in the call set.

Definition 3.6 (Call set). A *call set* $\mathbb{C}$ is a set of literals. ♢

Definition 3.7 (Callable literals). Let A be a literal and $\mathbb{C}$ be a call set. Then A is *callable* with respect to $\mathbb{C}$ if and only if $A[\phi] \in \mathbb{C}$ for some substitution ϕ. ♢

Since call sets are likely to be infinite, we will write only the most general form of each literal in the set, simply assuming every instance of the general form exists in it. The following example illustrates this point.

Example 3.4. A possible call set for *append* that contains only the most general form of each literal is:

$$\begin{aligned}\{&append\ nil\ Y\ Z,\ append\ (A : X)\ Y\ Z,\\ &\quad append\ X\ Y\ nil,\ append\ X\ Y\ (B : Z)\}.\end{aligned} \tag{3.7}$$

Each literal in the set has its first or third argument bound to a constructor. The selection of such a literal in a goal ensures that the size of the remaining problem is decreased at each subsequent resolution step. ♢

Adopting a computation rule that selects only those literals in a goal that are instances of those in the call set could result in no literal being selected at all. For example, suppose the call set (3.7) is used for *append* and consider the following query:

$$\#\ :-\ append\ X\ (a : b : c : nil)\ Y,\ append\ X\ Y\ Z.$$

Here, no literal can be selected because neither subgoal is an instance of a literal in (3.7). In this case, the goal is *floundered* and its evaluation is considered unsuccessful. Thus, a coroutining computation rule is a partial function that takes a goal and a call set, and returns a selected literal from the goal.

Definition 3.8 (Coroutining computation rule). Let $G = \#\ :-\ A_1, \ldots, A_n$, for $n \geq 1$, be a goal and $\mathbb{C}$ a call set. A computation rule $R_{\mathbb{C}}$ is a partial function mapping G to the leftmost callable literal A_i, for $1 \leq i \leq n$, with respect to $\mathbb{C}$. We say that G is *floundered* if no literal in G is callable. ♢

The notion of a goal being floundered is adapted from SLDNF-resolution (SLD-resolution with Negation as Failure) (Hogger 1990, Lloyd 1987), occurring there when a goal comprises only non-ground negative literals. The notion of a floundered goal is generalised by *SLDF-resolution* (SLD-resolution with Floundering), defined next, where a goal of positive literals is floundered if it contains no callable literals.

Definition 3.9 (SLDF-resolution). Let $C = A\ :-\ B_1, \ldots, B_m$, for $m \geq 0$, be a definite clause, $G = \#\ :-\ A_1, \ldots, A_n$, for $n \geq 1$, a goal, and $R_{\mathbb{C}}$ a computation rule. There are two cases to consider:

1. Suppose the selected literal is $R_{\mathbb{C}}(G) = A_i$, for $1 \leq i \leq n$, and that A_1 and A are unifiable with most general unifier ϕ. The *SLDF-resolvent* of C and G is the goal

$$G' = (\#\ :-\ B_1, \ldots, B_m, A_1, \ldots, A_{i-1}, A_{i+1}, \ldots, A_n)[\phi]$$

2. Suppose that G is *floundered*. Then the resolution is said to be floundered, also.

Therefore, an SLDF-resolution is the same as an SLD-resolution, except that the last goal in any derivation may be floundered. An SLDF-resolution may be infinite, successful, or failed in the same way as an SLD-resolution, or floundered. $\diamondsuit$

In practice, SLDF-resolution is efficient and terminates more often than corresponding Prolog programs. However, the main drawback of SLDF-resolution is that the computation of a goal can flounder if the call set fails to contain enough general forms of literals. Therefore, the analysis for determining bindings of arguments is of utmost importance. In the following section, the programming language NU-Prolog, that uses a coroutining computation rule, is reviewed and a method of automatically constructing the call set for a program is presented.

3.5.3 NU-Prolog

NU-Prolog (Thom & Zobel 1987) is a variant of Prolog that utilises the coroutining computation rule of Definition 3.8. In this section, we give a brief overview of NU-Prolog, demonstrating its strengths and weaknesses.

In NU-Prolog, the notion of a call set of literals is captured by new syntactic constructs called *when-declarations* that annotate logic programs, dictating the circumstances under which a literal is able to be selected for resolution. The following definition formalises when-declarations.

Definition 3.10 (When-declarations). A *when-clause* is a statement of the form

A when W

where A is an unrestricted literal, i.e., no variable occurs in A more than once, and W is a *when-formula*, defined as follows:

$W ::= \text{true} \mid X \mid \text{ground}(X) \mid W \text{ and } W \mid W \text{ or } W$

where X is a variable that occurs in A. Finally, a *when-declaration* is a set of when-clauses. $\diamondsuit$

For a when-clause A when W, the meaning of W is given by the associated truth value of an instance of W. This idea is formulated in the following definition.

Definition 3.11 (Interpretation of when-declarations). Let A when W be a when-clause and ϕ a substitution. The truth value of $W[\phi]$ is defined as follows:

$$W[\phi] = \begin{cases} \mathit{true}, & \text{if } W = \text{true} \\ \mathit{true}, & \text{if } W = X \text{ and } X[\phi] \text{ is not a variable} \\ \mathit{true}, & \text{if } W = \text{ground}(X) \text{ and } X[\phi] \text{ is a ground term} \\ W_1[\phi] \wedge W_2[\phi], & \text{if } W = W_1 \text{ and } W_2 \\ W_1[\phi] \vee W_2[\phi], & \text{if } W = W_1 \text{ or } W_2 \\ \mathit{false}, & \text{otherwise} \end{cases}$$

A literal B is *callable* if $B = A[\phi]$ and $W[\phi]$ is *true*, or B does not unify with A. $\diamondsuit$

Example 3.5. Suppose, for the predicate *append* we add the following when-declaration to the program:

$$append(X, Y, Z) \text{ when } X \text{ or } Z.$$

Then the literals $append([\,], X, Y)$ and $append(X, Y, [A, B, C])$ are callable since '$[\,]$' and '$[A, B, C]$' are not variables. Alternatively, the literal $append(X, [a, b, c], Y)$ is not callable since both the first and third arguments are variables. ♢

The simplicity of NU-Prolog's incarnation of SLDF-resolution is particularly pleasing since the control information gained by when-declarations is separate from the logic of the programs. That is, the insertion of when-declarations in a program does not affect its declarative nature. Recall from the list of criteria for a compositional logic language (Chapter 2) that the programmer should be liberated from all control information. Let us now illustrate the benefits of coroutining, with respect to compositionality, by looking at an example NU-Prolog program.

Example 3.6. Consider the following NU-Prolog program that implements quicksort:

$$\begin{array}{l}
partition(A, [\,], [\,], [\,]).\\
partition(A, [B|X], Y, [B|Z]) \;:-\; A < B,\ partition(A, X, Y, Z).\\
partition(A, [B|X], [B|Y], Z) \;:-\; A \geq B,\ partition(A, X, Y, Z).\\
\\
qsort([\,], [\,]).\\
qsort([A|X], Y) \;:-\\
\quad partition(A, X, X_1, X_2),\ append(Y_1, [A|Y_2], Y),\ qsort(X_1, Y_1),\ qsort(X_2, Y_2).
\end{array}$$

The predicates $<$ and $\geq$ respectively embody the less-than and greater-than-or-equal-to relations over integers. Now suppose we add the following when-declarations to the program:

$$\begin{array}{l}
append(X, \_, Y) \text{ when } X \text{ or } Y\\
partition(\_, X, Y, Z) \text{ when } X \text{ or } (Y \text{ and } Z)\\
qsort(X, Y) \text{ when } X \text{ or } Y
\end{array}$$

The first when-declaration states that a literal of *append* should only be selected for resolution when its first or third argument is not a variable. The second clause states that a literal of *partition* should be selected either when its second argument is not a variable or when both its third and fourth arguments are not variables. The final when-declaration states that a literal of *qsort* may be selected in the cases where either its first or second arguments are not variables.

With these when-declarations, *qsort* can be used to sort lists of integers and also to permute sorted lists since the SLDF-tree of such queries is finite and non-floundering. Moreover, 'mixed-mode' queries—where both arguments are bound to partial lists—can be successfully computed. For example, the query

$$\# \;:-\; qsort([4, A, 6, B, C, 5], [1, 2, 3|Y]).$$

terminates with the answer substitution $\{A \mapsto 3, B \mapsto 2, Z \mapsto 1, Y \mapsto [4, 5, 6]\}$ and the five other answer substitutions that bind X, Y, and Z to the remaining permutations of 1, 2, and 3. ♢

```
$generate\ P =$
  let $\mathbb{C}_0 := patterns\ P$ and $finished := false$
  while not $finished$ do
    let $\mathbb{C} := \mathbb{C}_0$ and $accept := true$
    while $\mathbb{C} \neq \{\,\}$ and $accept$ do
      let $A \in \mathbb{C}$ and $\mathbb{C} := \mathbb{C} - \{A\}$
      $accept := finite\ (P \cup \{\# :- A\}, \mathbb{C}_0)$
    done
    if not $accept \longrightarrow$
      let $B \in \mathbb{C}_0$ and $\mathbb{C}' \subseteq\ successors\ B$
      $\mathbb{C}_0 := (\mathbb{C}_0 - \{B\}) \cup \mathbb{C}'$
    fi
    $finished := accept$
  done
```

Figure 3.5: The algorithm for determining a call set for coroutining.

The call set is crucial for the efficient and terminating execution of queries: the remaining problem being to determine it automatically. In what follows, we review a method presented by (Lüttringhaus-Kappel 1993) for computing a call set for a coroutining computation rule.

3.5.4 Generating a call set automatically

An algorithm was presented by (Lüttringhaus-Kappel 1992) to determine automatically a call set $\mathbb{C}$ for a coroutining computation rule with respect to a given program. The main criterion for a literal's inclusion in $\mathbb{C}$ is that it has a finite SLDF-tree. The algorithm proceeds by maintaining a set of candidate literals and simulating their execution to determine if their SLDF-tree is finite. Those literals that appear to be infinite are refined by incrementally binding more arguments until the literals either become finite or are deemed to be infinite. The algorithm terminates when every literal in the candidate set has a finite SLDF-tree at which time the candidate set is returned as the call set. The algorithm is shown in Figure 3.5.

The candidate call set $\mathbb{C}_0$ for the program P initially comprises the most general form of each predicate in P, i.e., where the arguments to each literal are bound to a variable. The function *patterns* creates precisely this set for P. There are several nondeterministic choices in the abstract algorithm that affect the efficiency of an implementation. Some of these issues are resolved in (Lüttringhaus-Kappel 1993) by employing heuristics to reduce the size of the search space. Naturally, there are occasions when other heuristics would result in a more effective call set. The most important choices are discussed below.

In the while loop, each literal in $\mathbb{C}$ is checked for a finite SLDF-tree with respect to the current candidate call set $\mathbb{C}_0$ using the function *finite*. However, since determining whether an arbitrary SLDF-tree is finite is undecidable, the check is approximated using loop detection techniques (Bol 1991) to test for infinite recursion. The resolution of the current literal is simulated using $\mathbb{C}_0$ as the call set in the coroutining computation rule. Essentially, a loop is assumed to exist in the SLDF-tree if any variant literal appears twice in a derivation. Although this heuristic works well in general, loops in an SLDF-tree can be caused by other means and therefore the heuristic may fail to detect them. To envelope this type of loop, (Lüttringhaus-Kappel 1993) employs a bound on the size of the SLDF-tree constructed in the simulated execution. The role of the depth bound is to reject infinite literals, although it often excludes valuable finite ones.

If the current SLDF-tree is assumed to be infinite, there is a choice of which literal to refine. The best choice is often the literal that caused the loop in the SLDF-tree. A poor choice of literal might likely cause the same loop in a subsequent SLDF-tree or result in a set under which some goals will flounder that could otherwise be successfully executed with a different set.

Interestingly, the algorithm of Figure 3.5 makes crucial use of type information to allow the refinement of literals by adding appropriate constructors; such is the behaviour of the *successors* function. Historically, strong type information has been ignored by the logic programming community, considered too restrictive on programs. However, a recent trend in the logic community has acknowledged the advantages of type-checking logic programs (Naish 1996, Lüttringhaus-Kappel 1992), notwithstanding languages like Mercury (Somogyi et al. 1995) and λProlog (Nadathur & Miller 1988) that are already strongly typed.

Finally, the successive refinements of literals and the size of the candidate set must be considered. Literals are refined until they reach a certain term depth: if the literal can be refined beyond this term depth, it is assumed to be infinite and it is removed from the candidate set. Also, new literals are admitted to the set only if they are not instances of literals already there. Let us illustrate these considerations by examining several examples of computing call sets.

Example 3.7. Consider again the *append* predicate:

$$\begin{aligned}&\mathit{append}\ \mathit{nil}\ Y\ Y\ :\!-\ .\\&\mathit{append}\ (A : X)\ Y\ (A : Z)\ :\!-\ \mathit{append}\ X\ Y\ Z.\end{aligned}$$

The initial candidate set, $\mathbb{C}_0$, comprises the most general form of *append*, i.e., the single literal $\mathit{append}\ X\ Y\ Z$. We commence by checking whether the SLDF-tree for $\#\ :\!-\ \mathit{append}\ X\ Y\ Z$ is finite. It is easy to see that, in fact, the tree is infinite with a repetitious call to a variant of $\mathit{append}\ X\ Y\ Z$. According to the discussion above, this literal is refined by selecting a single variable in the literal and binding it to either *nil* or $A_1 : X_1$. This process is repeated for each variable X, Y, and Z, and then each refined literal is included in $\mathbb{C}_0$:

$$\begin{aligned}\mathbb{C}_0 = \{&\mathit{append}\ \mathit{nil}\ Y\ Z,\ \mathit{append}\ (A_1 : X_1)\ Y\ Z,\ \mathit{append}\ X\ \mathit{nil}\ Z,\\&\mathit{append}\ X\ (A_1 : X_1)\ Z,\ \mathit{append}\ X\ Y\ \mathit{nil},\ \mathit{append}\ X\ Y\ (A_1 : X_1)\}.\end{aligned} \tag{3.8}$$

From this new set, any literal may be selected; suppose it is *append X nil Z*. Repeating the process above, the SLDF-tree for *append X nil Z* is again found to be infinite with a variant of this literal causing the loop. Refining this literal produces the new literals

$$\mathbb{C}' = \{\mathit{append\ nil\ nil}\ Z,\ \mathit{append}\ (A_2 : X_2)\ \mathit{nil}\ Z,\ \mathit{append}\ X\ \mathit{nil\ nil},\\ \mathit{append}\ X\ \mathit{nil}\ (A_2 : X_2)\}.$$

However, all of these literals are covered by more general literals in $\mathbb{C}_0$ and so are excluded from $\mathbb{C}_0$.

Now suppose *append* X $(A_1 : X_1)$ Z is selected from (3.8). Again, the corresponding SLDF-tree contains a loop. Furthermore, all successive refinements to this literal—*append* X $(A_1 : \mathit{nil})$ Z, *append* X $(A_1 : A_2 : X_2)$ Z, and so on—exhibit similar looping behaviour. This situation is dealt with by cutting-off further refinement when the size of these terms reaches some finite bound: on reaching the bound the the original literal is deemed infinite and it is removed, along with its successors, from $\mathbb{C}_0$. Of course, this bound is a heuristic and can mean that some literals with finite SLDF-trees are excluded from the set. Finally, $\mathbb{C}_0$ contains the literals in (3.7) that all have finite, though possibly floundering, SLDF-trees. ♢

Sometimes there can be an infinite number of literals in a call set that have finite SLDF-trees. The following example illustrates this point.

Example 3.8. Consider the *reverse* program below.

reverse nil nil :− .
reverse $(A : X)$ Y :− *reverse* X Z, *append* Z $(A : \mathit{nil})$ Y.

By tracing through the algorithm of Figure 3.5, the following call set is computed for *reverse*:

$$\{\mathit{reverse\ nil}\ Y,\ \mathit{reverse}\ (A_1 : X_1)\ Y,\ \mathit{reverse}\ X\ \mathit{nil},\ \mathit{reverse}\ X\ (A_1 : \mathit{nil}),\\ \mathit{reverse}\ X\ (A_1 : A_2 : \mathit{nil}),\ \mathit{reverse}\ X\ (A_1 : A_2 : A_3 : \mathit{nil}),\ \ldots\}$$

In theory, the set is infinite: when the first argument to *reverse* is a variable,the second argument must be a *complete* list, i.e., one terminated with *nil*. Since only finite lists may be represented by when-declarations, the refinement is again cut-off at some bound but this time the literals are included in the set. ♢

The previous examples illustrate the limitations of the algorithm. Firstly, literals are deemed infinite using heuristics and, secondly, there may be an infinite number of literals in a call set. In the latter case, there may be a general recursive scheme that describes such infinite sequences, like the idea of a *complete* list in Example 3.8. However, the algorithm itself does not provide a general method of spotting when arguments must be complete. The following example considers again the *qsort* predicate from Example 3.6 and illustrates the practical problems with the algorithm.

Example 3.9. The call set computed by the algorithm for predicate *qsort* is:

$$\{qsort([\,], Y),\ qsort([A_1|X_1], Y),\ qsort(X, [\,]),\ qsort(X, [A_1]),\\ qsort(X, [A_1, A_2]),\ qsort(X, [A_1, A_2, A_3]), \ldots\}$$

The value of the depth-bound on terms dictates the number of admitted literals in the sequence $\{qsort(X, [\,]),\ qsort(X, [A_1]),\ \ldots\}$. If the bound is too small, e.g., the value 5, then queries like

$$\#\ :-\ \ qsort\ X\ [4, 6, 2, 3, 1, 5].$$

may flounder if the second argument exceeds the bound on the term size. $\diamondsuit$

The generic shape of an argument is of prime importance in coroutining computation rules, as Example 3.9 illustrates. However, the algorithm of Figure 3.5 used to generate a call set for a program is not suitable for abstracting over terms in this way. In fact, it is the very nature of coroutining that is to blame for this problem; coroutining is designed to compute incrementally, interleaving the resolution of subgoals depending on the instantiations of their variables. Therefore, restricting attention to one resolution step at a time limits the ability to abstract naturally over the size of terms. (In Chapter 5, we introduce a computation rule that replaces the local nature of a coroutining computation rule with a global view of a literal's termination behaviour. The call set, then, comprises literals whose SLD-resolution is guaranteed to terminate.)

3.6 Summary

In Chapter 2, we identified a collection of requirements that are necessary for a language to support the style of compositional programming proposed in Chapter 1. One requirement was that the language should allow specifications to be phrased naturally in the syntax of the language, which requires the ability to define higher-order predicates. In this chapter, we presented a curried syntax for the terms of a logic language that replaces the need for higher-order programming constructs to be introduced via extra-logical predicates, as is currently the case in languages such as Prolog and Mercury. Moreover, we showed that the addition of curried terms to a language provides the same higher-order programming capability as that afforded by the extra-logical *call* predicate from Prolog.

A second language requirement identified in Chapter 2 is that a compositional program should be able to be written without regard for the evaluation order employed by the language. In this chapter we reviewed the notion of fair SLD-resolution and showed that it satisfied this language criterion by proving that every fair SLD-resolution of a goal terminates whenever any other SLD-resolution of the same goal would do so also.

We also explained that, although the choice of computation rule alone affects the termination characteristics of a logic language, the choice of search strategy can help to augment the declarative nature of the language. In particular, when a compositional program exhibits unbounded nondeterminism, a fair search strategy, like breadth-first search or depth-first iterative deepening, can ensure that solutions from all parts of the search tree are discovered, whereas an unfair search strategy, like depth-first search, may commit to enumerating an infinite number of related solutions contained in a restricted part of the search tree.

Despite the theoretical elegance of fair SLD-resolution, we demonstrated that it suffers from poor efficiency behaviour when executing simple compositional programs. Therefore, we reviewed two possible methods of overcoming the efficiency problems of fair SLD-resolution: a previous, though independent, treatment of fair SLD-resolution, called indexed fair SLD-resolution; and coroutining. However, we showed that both fell short of satisfying the desiderata for a compositional language. Firstly, indexed fair SLD-resolution relied upon the use of heuristics to prune the search tree and required extra annotations to each program clause. In the case of coroutining, it often computed a call set that was inadequate to allow efficient evaluation of compositional programs. Moreover, the computed set was often 'unsafe' in the sense that the resolution of a goal could terminate using an unsafe call set but may loop infinitely for any instantiation of it. Therefore, when using coroutining it is difficult to predict whether a query will execute efficiently and whether it will terminate.

Chapter 4

Tabling Logic Programs

Fair SLD-resolution, which we reviewed in the previous chapter, constructs a finite SLD-tree for a query should any such tree exist. However, as discussed in Section 3.4, the practicalities of fair SLD-resolution can be less convincing than the theory suggests: the search tree for many fair computations can grow so large—yet still theoretically finite—that traversing the tree in reasonable time becomes practically impossible. The perpetrator of this adverse efficiency behaviour is often the redundant evaluation of previously computed subgoals. Moreover, we saw that another limitation of fair SLD-resolution, inherited from SLD-resolution, is its inability to terminate for left-recursive programs, of which many compositional programs are an instance.

One possible solution to these problems is to record, or *table*, the result of resolving each literal during an evaluation (Chen et al. 1995, Warren 1992, Tamaki & Sato 1986). Then, on repetitious calls to variants of these 'tabled' literals, their solutions are obtained from the table rather than by recomputing them. Tabling has already been integrated into some logic programming systems, such as XSB (Sagonas, Swift & Warren 1994), and the reasons for doing so are two-fold: firstly, the class of terminating programs is extended beyond that possible under SLD-resolution by permitting left-recursive programs to terminate; secondly, redundant computation is eliminated since the solutions to previously computed literals are recorded in the table. Owing to these factors, tabling languages like XSB are extremely efficient, often executing programs in fewer resolution steps than in regular Prolog implementations. However, systems like XSB adopt the same left-to-right computation rule as Prolog and, consequently, inherit the same poor termination behaviour when executing compositional programs.

In this chapter, we present a straightforward extension to an existing tabling system that uses the left-to-right computation rule (Tamaki & Sato 1986) to perform *fair* tabled resolution instead. We then employ fair tabled evaluation as the computation strategy of our compositional language and show that it overcomes the non-termination of left-recursive programs experienced when using fair SLD-resolution and also improves upon its execution efficiency. However, we also discuss that the trade-off for this behaviour is that compositional programs are no longer guaranteed to terminate in the way that they are when executed using fair SLD-resolution.

4.1 Adding tabling to logic languages

Existing logic languages are incomplete for at least two reasons: the first is the left-to-right consideration of subgoals in a goal; and the second is due to entering a cycle in a single computation path where the same subgoal is satisfied infinitely many times. One main reason for adopting tabulation techniques in logic languages like XSB (Sagonas et al. 1994) is to avoid recomputing previously evaluated literals and, consequently, overcome the problem of entering cyclic evaluations.

In analogy with techniques from functional programming (Bird 1980), tabling in logic programming involves recording the evaluation of literals during the course of a computation. The situation is more complicated than in functional programming, however, since logic programs may have more than one solution. Therefore, a table in logic language is regarded as a finite mapping from literals to sets of their answers. The main motivation for adopting tabling in logic languages like XSB is to increase their class of terminating programs by detecting, and removing, cyclic evaluations (Warren 1992). In such systems, it is a pleasant side-effect that redundant evaluation of subgoals is avoided. However, we find the converse to be true: our motivation for adopting tabling rests with the elimination of redundant subcomputations since such behaviour is intrinsic to fair resolution. Let us begin this section with a motivating example to develop an intuition for the tabled evaluation of logic programs.

4.1.1 An example tabled evaluation

Tabling broadens the class of programs for which a logic language can terminate beyond that of SLD-resolution. Moreover, tabling can drastically improve the efficiency of execution of many programs by eliminating redundant computation. In this section, we detail an example OLDT-resolution that illustrates both of these points.

Consider the following program for the predicate *path* that determines the transitive relationships between the vertices a, b, and c in a directed graph:

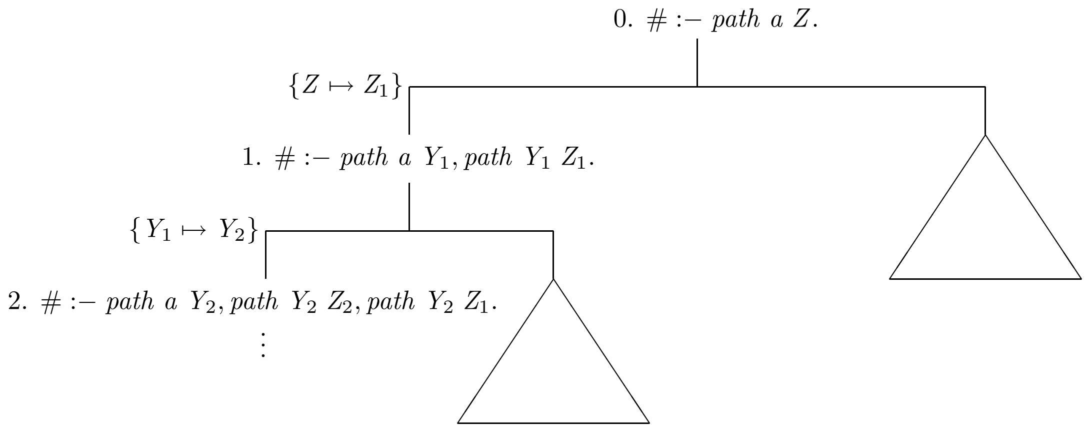
$$arc\ a\ b\ :-\ .\qquad arc\ b\ c\ :-\ .$$

$$\begin{aligned} &path\ X\ Z\ :-\ path\ X\ Y,\ path\ Y\ Z.\\ &path\ X\ Z\ :-\ arc\ X\ Z.\end{aligned}$$

Even though the graph above is free of cycles, the SLD-resolution of any literal of *path* loops infinitely, irrespective of the choice of computation rule or the instantiation of the literal's arguments; such SLD-trees always contain a cyclic derivation since every literal of *path* introduces another *path* literal whose resolution depends on itself. For example, consider the SLD-resolution of the query

$$\#\ :-\ path\ a\ Z. \tag{4.1}$$

The SLD-tree of this query is depicted in Figure 4.1. We can see from the figure that the SLD-resolution of (4.1) enters an infinite loop without finding any solutions. Clearly, the culprit is the cyclic call to variants of *path a Z* along the left branch of the tree. However, as illustrated below, OLDT-resolution is able to rectify this problem of infinite recursion and produce a terminating computation for (4.1).



0. # :− path a Z.
$\{Z \mapsto Z_1\}$
1. # :− path a Y_1, path Y_1 Z_1.
$\{Y_1 \mapsto Y_2\}$
2. # :− path a Y_2, path Y_2 Z_2, path Y_2 Z_1.

Figure 4.1: The SLD-tree for # :− *path a Z*.

In a tabled evaluation, it often makes sense to restrict those predicates in a program whose computations are tabled. For example, in the case of the program above, we choose not to table the *arc* predicate since it is defined solely in terms of facts, and so its computation will always terminate. Indeed, existing implementations of tabled logic languages, like XSB, use dependency analysis to decide upon the set of predicates whose evaluation should be tabled. These predicates, like *path* above, are referred to as 'tabled predicates'.

An OLDT-tree for a query is a forest of SLD-trees; the OLDT-tree of (4.1) is illustrated in Figure 4.2. The numbering of nodes in the tree helps identify paths in the derivation. A node also labelled with '*' identifies the first occurrence of a tabled predicate; we call such a node an 'answer node'.

Since *path* is a tabled predicate and no variant of *path a Z* already occurs as the root of a tree, the tabled evaluation of (4.1) commences by creating an answer node, i.e., node 0*, with root # :− *path a Z*. Using depth-first search, *path a Z* is resolved with the first of its program clauses to generate node 1. From here, the leftmost literal *path a Y* is selected only to discover that it is a variant of the answer node 0*. Node 1 is called an 'active node' and its further evaluation is delayed until answers have been discovered for *path a Y* via the answer node 0*.

Since no progress from node 1 can be made for the time being, we backtrack to node 0* and perform a resolution with the second program clause of *path*. This step generates node 2 and *arc a Z* is selected for resolution. As mentioned earlier, *arc* is defined solely in terms of facts and there is no benefit to be gained by tabling its execution. Node 2 is called an 'interior node' and a resolution step is performed on it. We arrive at node 3 to discover a refutation for *path a Z*. At this point, the answer *path a b* for literal *path a Z* is inserted into the table to obtain

$$path\ a\ Z \mapsto \{path\ a\ b\} \tag{4.2}$$

Next, we backtrack to the delayed, active node 1. On re-encountering an active node, the action taken depends upon the state of the table: any answers that have been entered

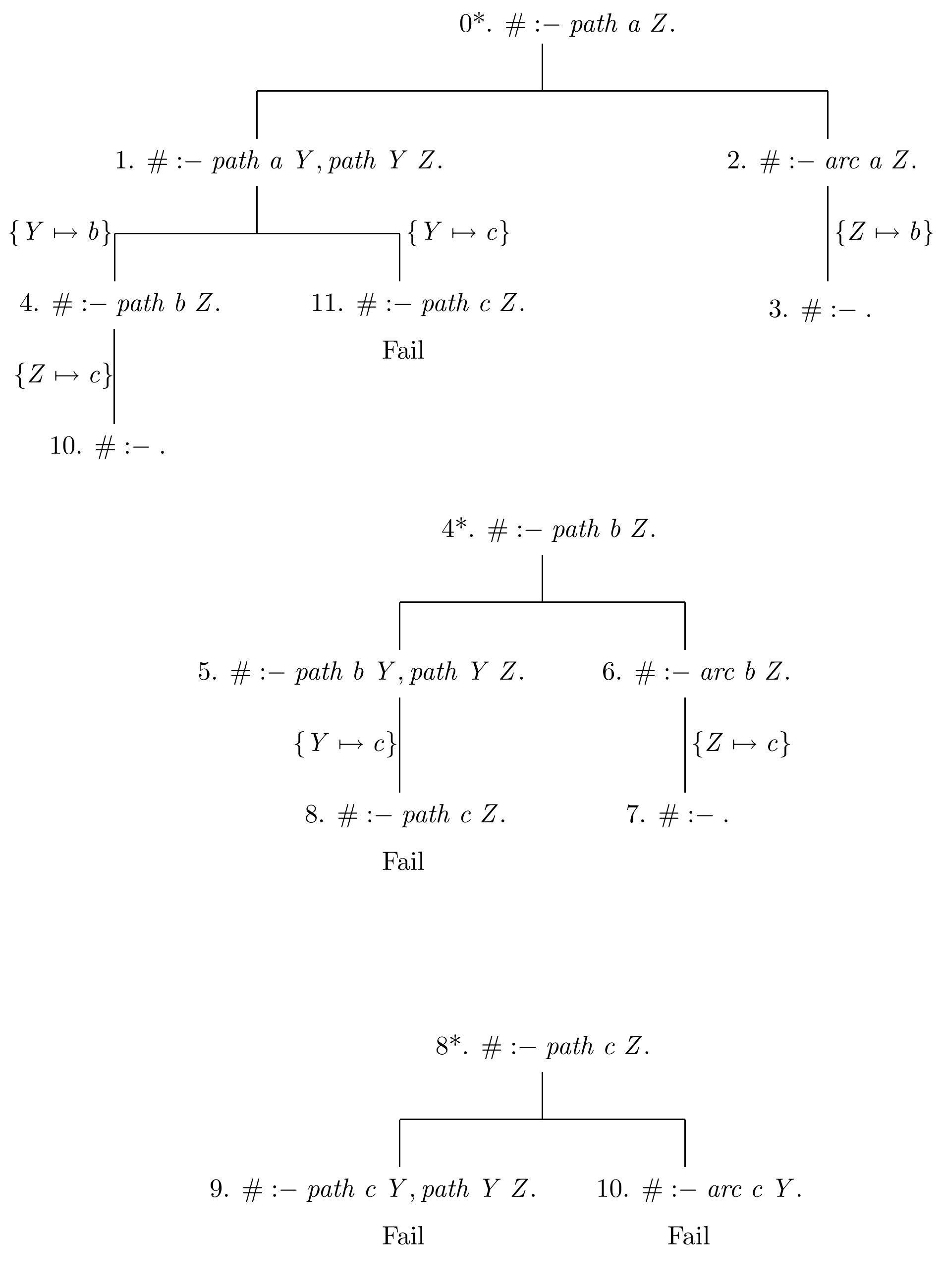


Figure 4.2: The OLDT-tree for # :− *path a Z*.

into the table since the previous visit to the active node are applied to its goal; conversely, if no new answers have appeared in the table then the active node simply fails, provided the associated answer node is 'completely evaluated'. We formalise the concept of an answer node being completely evaluated later in the chapter. Here, there is an answer for $path\ a\ Z$ in table (4.2), and it is used to obtain node 4. Since a variant of $path\ b\ Z$ does not already form the root of a tree, node 4* is identified as an answer node.

The computation from node 4* continues and node 5 is generated. Again, node 5 is an active node and it is delayed until answers for $path\ b\ Z$ have been found elsewhere. We backtrack to node 4*, and derive nodes 6 and 7 which provide a refutation for $path\ b\ Z$. The table (4.2) is extended with this solution, resulting in

$$\begin{array}{l} path\ a\ Z \mapsto \{path\ a\ b\} \\ path\ b\ Z \mapsto \{path\ b\ c\} \end{array} \tag{4.3}$$

We now return to node 5 with one new answer for $path\ b\ Z$ entered in the table. Using this answer, node 8 is formed with node 8* as its corresponding answer node. The development of this tree is similar to that of node 4*, first delaying node 9 but this time failing at node 10. We backtrack to node 9 but there are no answers for literal $path\ c\ Y$ in the table. Since the tree for $path\ c\ Z$ is completely evaluated—it has been resolved with all program clauses and its only remaining child is cyclic—the delayed node 9 fails, terminating the tree of node 8*. At this point, node 8* is identified as a 'complete node'.

Now all the remaining nodes in the collection of trees are active, namely nodes 8 and 4. Backtracking from node 8* to node 8, the table is examined only to find that no answers exist for $path\ c\ Z$ and so node 8 fails. Consequently, node 4* is deemed a complete node and we backtrack to node 4. This time, a previously untried answer for $path\ b\ Z$ exists in the table. Using this, a refutation is discovered for $path\ a\ Z$, the new answer being $path\ a\ c$. The success of this refutation depends on whether a variant of the new answer has been discovered previously for $path\ a\ Z$. After checking table (4.3), we see that the answer $path\ a\ c$ does not appear so it is indeed a new answer. Adding this to the table results in

$$\begin{array}{l} path\ a\ Z \mapsto \{path\ a\ b,\ path\ a\ c\} \\ path\ b\ Z \mapsto \{path\ b\ c\} \end{array} \tag{4.4}$$

Tracing back up the tree, the second answer for $path\ a\ Y$ is applied to node 1, creating node 11. Now, the literal $path\ c\ Z$ is the root of a complete node but has no answers in the table. Therefore, $path\ c\ Z$ is a finitely-failed literal and so node 11 fails. At this point, all possible answers for node 0* have been generated and the resolution of the original query (4.1) terminates with the following substitutions for Z:

$$\begin{array}{l} Z\ =\ b; \\ Z\ =\ c; \\ no. \end{array}$$

Therefore, OLDT-resolution has constructed a finite OLDT-tree when the corresponding SLD-tree is infinite.

4.1.2 The left-to-right computation rule in tabled evaluation

A logic language that uses tabled evaluation is guaranteed to terminate if an SLD-resolution using the same computation rule would make only finitely many different calls and generate only finitely many different answers (Warren 1992). Therefore, tabling eliminates non-termination due to making the same call infinitely many times—as was the case with query (4.1)—resulting in the termination of a larger class of programs than SLD-resolution alone. Nevertheless, a tabling system that employs the left-to-right computation rule, as is the case with XSB (Sagonas et al. 1994), is still susceptible to the inherent problem of building infinite search trees. Let us now discuss an example which illustrates this problem in the context of tabled evaluation.

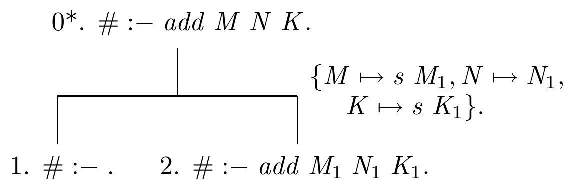
Example 4.1. Consider the program to add Peano natural numbers:

$$\begin{aligned}&\mathit{add}\ 0\ N\ N\ \ {:}{-}\ \ .\\&\mathit{add}\ (s\ M)\ N\ (s\ K)\ \ {:}{-}\ \ \mathit{add}\ M\ N\ K.\end{aligned}$$

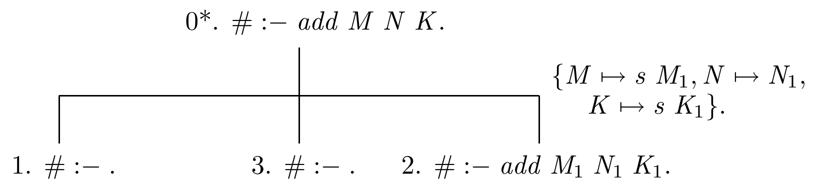
and the following query, where the value 50 abbreviates the corresponding Peano natural number:

$$\#\ \ {:}{-}\ \ \mathit{add}\ M\ N\ K,\ \mathit{add}\ M\ N\ 50. \tag{4.5}$$

The tabled execution of this query using a left-to-right computation rule will not terminate. To explain why, consider the evaluation of the leftmost literal *add* M N K, only. The first resolution step produces a refutation for *add* M N K with the answer literal *add* 0 N N. After the next resolution step, the search tree rooted at *add* M N K is



Node 2 is active since the literal *add* M_1 N_1 K_1 is a variant of *add* M N K. However, an answer exists in the table for this variant, namely *add* 0 N N. On returning this answer to node 2, the above tree becomes



with a new refutation at node 3 and associated answer *add* 1 N_2 $(s\ N_2)$. This answer does not already appear in the table so it is inserted there. The resolution proceeds again from node 2 with the new answer returned to it. Each subsequent resolution step produces a new, distinct answer that is used to produce yet another new answer. Since there are an infinite number of triples (M, N, K) such that $M + N = K$, the tabled resolution attempts to enumerate them all which forces non-termination of the query. ♢

The main problem in the example above is that the leftmost, infinite subgoal is always selected for resolution over the rightmost, finite one. In terms of producing terminating computations—necessary for compositional programming—we require the ability to select literals other than the leftmost one. We identify such a class of flexible rules for tabled evaluation in the following section.

4.1.3 Flexible computation rules in tabled evaluation

The tabled evaluation of logic programs described by (Tamaki & Sato 1986) enforces the use of the left-to-right computation rule. However, as we saw in the previous section, this computation rule is undesirable for the execution of compositional programs since it can cause them to loop infinitely. We saw in Chapter 3 that a fair computation strategy is preferable since it permits such programs to terminate whenever possible. However, a concept of fairness for tabled evaluation, analogous to that for fair SLD-resolution, cannot be obtained by simply adopting the breadth-first computation rule.

The reason for this is that tabled evaluation imposes a restriction on the computation rules that can be used naturally with it. Specifically, tabling permits the use of any *arbitrary* but *fixed* computation rule. By arbitrary, we mean that any literal can be selected initially from a goal. The fixed requirement means that, at subsequent resolution steps, the further arbitrary selection of literals is restricted to only those literals introduced at the previous resolution step. The reason for the above restriction is that a tabled evaluation must be able to detect precisely when a refutation for an individual literal occurs and, hence, when its answer has been computed so that it may be inserted into the table. The following example illustrates this important but subtle point.

Example 4.2. Consider the simple logic program below:

$q\ X\ :-\ .$
$p\ Y\ :-\ q\ Y.$
$r\ Z\ Z\ :-\ .$

Let us consider a tabled evaluation of the following query using an arbitrary, but not fixed, computation rule:

$\#\ :-\ p\ X,\ r\ X\ a.$

Suppose the literal $p\ X$ is selected in the first resolution step. Then we obtain the new goal:

$\#\ :-\ q\ X,\ r\ X\ a.$

A computation rule with a fixed requirement would permit only the literal $q\ X$ to be selected at the subsequent resolution step. However, as there is no fixed requirement on the current rule, let us select the literal $r\ X\ a$ for resolution, which produces the following resolvent:

$\#\ :-\ q\ a.$

The resolution of this literal results in a refutation with the answer substitution $\{X \mapsto a\}$ which is too specific to be an answer substitution for $p\ X$.

On the other hand, a fixed computation rule would permit the selection of only $q\ X$ since it was introduced at the previous resolution step. Resolving this literal results immediately in a refutation of it and, implicitly, in a refutation of $p\ X$ with the identity substitution as its answer substitution. Therefore, we are able to insert $p\ X$ into the table as a genuine answer for $p\ X$. $\diamondsuit$

Technically, by ensuring that a computation rule adopts the fixed requirement, we can apply the notion of a 'subrefutation', defined in the following section, which allows us to identify precisely the answers for each resolved literal that can then be inserted into the table.

4.1.4 OLD-resolution

In this section, the original definition of OLD-resolution (Ordered selection using Linear resolution for Definite clauses), due to (Tamaki & Sato 1986), is generalised to use any arbitrary but fixed computation rule which lends more flexibility in the selection of literals than with the left-to-right computation rule. The modification to OLD-resolution, presented below, is couched simply as an SLD-resolution using the left-to-right computation rule (Definition 2.7) but where the order of the literals introduced at each resolution step is permuted; the permutation satisfies the arbitrary requirement of the computation rule since it dictates the order in which literals will be selected for resolution at subsequent steps. Moreover, by ensuring that the permutation of these literals is placed at the front of the resolvent, the fixed requirement of the computation rule is also achieved.

In other words, the modification to OLD-resolution makes explicit the restricted selection of subgoals—subgoals not introduced at the previous resolution step may only be selected once the current derivation is complete—yet permits any newly introduced literal to be selected at the next resolution step. The generalised form of OLD-resolution defined below differs from the original one by (Tamaki & Sato 1986) in that the order of introduced subgoals may be permuted.

Definition 4.1 (OLD-resolution). Let $C = A :- B_1, \ldots, B_m$, for $m \geq 0$, be a definite clause and $G = \# :- A_1, \ldots, A_n$ a goal. Moreover, suppose that A_1 and A are unifiable with most general unifier ϕ. Then, the selected literal is A_1 and the *OLD-resolvent* of C and G is the goal

$$G' = (\# :- Q, A_2, \ldots, A_n)[\phi]$$

where Q is some permutation of the literals $B_1, \ldots, B_m$. The substitution ϕ is called the *substitution of the OLD-resolution.* $\diamondsuit$

The following definitions related to OLD-resolution are almost identical to those for SLD-resolution which were reviewed in Section 2.3.3. For example, an OLD-derivation is analogous to an SLD-derivation from Definition 2.8 in that an OLD-derivation is a sequence of triples that records the following for each OLD-resolution step: the current OLD-resolvent; the program clause used to resolve against the selected literal in the current resolvent; and the substitution of the OLD-resolution. The only difference between an OLD-derivation and an SLD-derivation is that the initial goal in an OLD-derivation can be a permutation of the actual goal, corresponding to the order in which literals are selected by the arbitrary computation rule.

An OLD-refutation is exactly an SLD-refutation, given earlier in Definition 2.9. Moreover, it is easy to see that OLD-resolution using a left-to-right computation rule is precisely SLD-resolution (Definition 2.7). Also, the search tree formed from an OLD-resolution, i.e., an OLD-tree, is simply an SLD-tree from Definition 2.10. Again, the soundness and completeness of OLD-resolution follow directly from that of standard SLD-resolution (Hogger 1990, Lloyd 1987, Apt & van Emden 1982).

Despite the similarities of OLD-resolution to SLD-resolution, the notion of an *OLD-subrefutation* is particular to OLD-resolution. An OLD-subrefutation is a manifestation of the fixed requirement of a computation rule and allows the tabled evaluation of a query to be visualised as the resolution of each subgoal in a goal independently. Subrefutations make it possible to determine the correct answers for each subgoal, which can then be inserted into the table; we elaborate on this point in the following section. The definition of an OLD-subrefutation is due to (Tamaki & Sato 1986) and is repeated below.

Definition 4.2 (OLD-subrefutation). Let D be an OLD-derivation and T a triple in D with goal $G = \# :- A_1, \ldots, A_n$, for $n \geq 1$. Furthermore, let T' be a descendant of T in D with goal $(\# :- A_{k+1}, \ldots, A_n)[\phi]$, for $k < n$ and ϕ some substitution. Clearly, the path from T to T' can be considered as a refutation of $\# :- A_1, \ldots, A_k$ by neglecting the (instantiated versions of) literals $A_{k+1}, \ldots, A_n$ in each intermediate triple. Such a refutation is called a *subrefutation of* $\# :- A_1, \ldots, A_k$. When $k = 1$ the refutation is called a *unit subrefutation*. $\diamondsuit$

A unit subrefutation allows the refutation of a particular subgoal in a goal to be considered in isolation from the remaining subgoals in the goal. With an eye towards tabling the answers of a literal, it is the answer substitutions obtained from unit subrefutations that are used to populate the table with 'answer literals'. To explain, suppose a node in an OLD-tree is labelled with $G = \# :- A_1, \ldots, A_n$ and has a descendant $G' = (\# :- A_2, \ldots, A_n)[\phi]$, for some substitution ϕ. Then, according to Definition 4.2, the path between these two nodes constitutes a unit subrefutation of $\# :- A_1$.

Moreover, the literal $A_1[\phi]$ is an answer literal for A_1 and should, therefore, be inserted in the table. Conceptually, any bindings determined for variables in A_1 by the unit subrefutation can be viewed as being 'withheld' from the remaining subgoals $A_2, \ldots, A_n$ in G until the refutation of A_1 is complete since none of $A_2, \ldots, A_n$ have been selected for resolution. The notion of a subrefutation is central in the formulation of OLDT-resolution which we review in the following section.

4.1.5 OLDT-resolution

OLDT-resolution (OLD-resolution with Tabling) was originally introduced by (Tamaki & Sato 1986), and later reconsidered by (Warren 1992), to eliminate an incompleteness of Prolog caused by evaluating an infinite number of calls to the same literal during a derivation. Tabling is able to do so by avoiding the redundant resolution of literals that have already been resolved by obtaining their solutions from the table.

In the previous section, we modified the original definition of OLD-resolution to use any arbitrary but fixed computation rule, as opposed to the left-to-right one. In the current section, we permeate this modification through the original definition of OLDT-resolution and make the routine revisions to the completeness proof of OLDT-resolution which was originally formulated by (Tamaki & Sato 1986) for the left-to-right computation rule. We begin by reviewing the standard definitions of an answer literal and a table, which have been given previously by (Warren 1992, Tamaki & Sato 1986).

Definition 4.3 (Answer literal). For some literal A, suppose that $\# :- A$ has an OLD-refutation with answer substitution ϕ. We call $A[\phi]$ an *answer literal* (or just *answer*) for A. $\diamondsuit$

Definition 4.4 (Table). A *table* is a finite mapping taking literals to sets of their answer literals. A *tabled* predicate is one whose answers will be place in the table, otherwise it is a *non-tabled* predicate. Furthermore, suppose N is a node in an OLD-tree, labelled with a goal G. Node N is called *active* if the selected literal in G is a tabled predicate, or *interior* if it is non-tabled. $\diamondsuit$

OLDT-resolution can be viewed as the growth of a forest of OLD-trees. Each OLD-tree in the forest is rooted at a node labelled with a goal containing only one subgoal. For each tabled literal A, only one OLD-tree may exist with a variant of A as its root. The root node is called the *answer* node for A. Only answer and interior nodes are resolved with clauses from the program; any other variants of an answer node are resolved with the answer literals computed by the answer node. The definition of OLDT-resolution, due to (Tamaki & Sato 1986), formalises this process and is repeated below.

Definition 4.5 (OLDT-resolution). Let P be a program, F a forest of OLD-trees, and N a node in some tree in F labelled with $G = \# :- A_1, \ldots, A_n$, for $n \geq 0$. Furthermore, let T be a table. An OLDT-resolution step consists of one of the following actions:

Answer registration Suppose that $n = 0$ and that the root of the tree containing N is labelled with $\# :- B$, for some literal B. Then the path from $\# :- B$ to N is a refutation of B. Let the answer literal be B' which is added to the answers for B in T unless a variant of B' already exists there.

Program clause resolution Suppose that N is either an *interior* or *answer* node. Perform a previously untried OLD-resolution step on the goal G.

Answer clause resolution Suppose that N is *active* and its selected literal A_1 appears as the root of an OLD-tree in F. Then resolve against A_1 a previously untried answer of A_1 from T.

Sprout Suppose that N is active and its selected literal A_1 does not appear as the root of an OLD-tree in F. Then create a new OLD-tree with root A_1.

Completion Suppose N is the root of a tree that has been *completely evaluated*. Then identify N as a *complete* node.

$\diamondsuit$

The following notion of an answer node being completely evaluated was used in the definition of OLDT-resolution and is standard from (Warren 1992). Essentially, a subgoal is completely evaluated if all possible operations have been performed on its nodes and has, therefore, computed all its possible answers.

Definition 4.6 (Completely evaluated). Let P be a program, F a forest of OLD-trees, T a table, and R a tree in F whose root is labelled $\# :- \ A$. We say that R is *completely evaluated* if it satisfies the following conditions: (1) for each clause in P that unifies with A, there is a child from the root corresponding to the resolution step; (2) for each active node N in R with selected literal B then, for each answer B' for B in T, there is a child from N corresponding to the resolution of $B' :-$ and B; and (3) for each interior node N in R with selected literal B then, for each clause in P that unifies with B, there is a child from N corresponding to the resolution step. ♢

Intuitively, it is not difficult to see that OLDT-resolution is a correct resolution strategy: for each derivation path in an OLDT-resolution, there will be a corresponding derivation in an OLD-resolution constructed by replacing each answer clause resolution step of a literal with its entire tree. Conversely, any path in an OLD-resolution will correspond to a path in an OLDT-resolution, the only difference being that repeated OLD-derivations will appear only once in the OLDT-resolution. The following lemma characterises this behaviour and forms the basis of the soundness and completeness conditions for OLDT-resolution. The lemma and its proof are originally due to (Tamaki & Sato 1986) although the versions below are adapted to take account of the modified form of OLD-resolution from Definition 4.1.

Lemma 4.1. *Let P be a program, F a forest of OLD-trees, $G = \# :- \ A_1, \ldots, A_n$, for $n \geq 1$, a goal, and r an OLD-subrefutation of G. Suppose that N is a node in a tree of F labelled with $G' = \# :- \ B_1, \ldots, B_m$, for $m \geq n$, such that each B_i, for $1 \leq i \leq n$, is a variant of A_i. Then there exists an OLDT-subrefutation of G' in F.*

Proof. The proof is by induction on the length of the refutation r, i.e., the number of resolution steps in r. For r of length 1, then r is a refutation of $\# :-$ which is always satisfied. For the length of $r > 1$, we consider whether N is an active or answer node.

Case 1: N is an active node. There must exist an answer node N' in F labelled with $\# :- \ B'_1$, a variant of B_1. Now, consider the initial portion r_1 of r corresponding to the subrefutation of $\# :- \ A_1$. Since the length of $r_1 < r$, and A_1, B_1 and B'_1 are variants, the induction hypothesis dictates that we have an OLDT-subrefutation of B'_1, with answer B''_1, from node N'. Therefore, by *answer registration*, B''_1 appears in the table.

Suppose A''_1 is the answer of r_1. Then $A''_1 :-$ and $\# :- \ A_1, \ldots, A_n$ have an OLD-resolvent $\# :- \ A'_2, \ldots, A'_n$. Moreover, since B''_1 and $B_1, \ldots, B_n$ are respective instances of A''_1 and $A_1, \ldots, A_n$, we obtain, by *answer clause resolution*, the node M labelled with the variant OLD-resolvent $\# :- \ B'_2, \ldots, B'_n$. Therefore, we can construct the OLDT-subrefutation of $\# :- B_1$ from N to M in F.

The remaining portion r_2 of r is a OLDT-subrefutation of $A'_2, \ldots, A'_n$. Again, the length of $r_2 < r$ and, by the induction hypothesis, we have the OLDT-subrefutation s of $B'_2, \ldots, B'_n$ in F from node M. The path from N to M, followed by s, constitutes the required subrefutation of $B_1, \ldots, B_n$.

Case 2: N is other than an active node. Let $C = A :- \ Q$ be a definite clause, such that Q is some permutation of the literals $\{L_1, \ldots, L_k\}$ for $k \geq 0$, and A_1 and A are unifiable. Then the first child node in r is the OLD-resolvent $\# :- \ L'_1, \ldots, L'_k, A'_2, \ldots, A'_n$ of C and G which has a subrefutation r', i.e., the remaining subrefutation of r. The label of N, then, is OLDT-resolvable using *program clause resolution*, producing a node M

labelled with the variant goal $\# :- L''_1, \ldots, L''_k, B'_2, \ldots, B'_n$. By the induction hypothesis, we have a subrefutation s of $\# :- L''_1, \ldots, L''_k, B'_2, \ldots, B'_n$ from node M in F. The path from N to M, followed by s, constitutes the subrefutation of $B_1, \ldots, B_n$. $\diamondsuit$

The completeness of OLDT-resolution is a direct consequence of the completeness of OLD-resolution: for any OLD-refutation of a goal G, Lemma 4.1 states that an equivalent OLDT-refutation of G must also exist in the OLDT-tree of G. The soundness and completeness results of OLDT-resolution have been presented previously by (Tamaki & Sato 1986).

4.2 Fairness in tabled evaluations

So far in this chapter, we have developed a basic extension to the existing work on tabled evaluation so that we may deploy more flexible computation rules to overcome the limitations of the left-to-right one. The problem exists, however, of determining which literal to select for resolution so that the computation will terminate whenever any other tabling system would do so also; recall from the desiderata of a compositional language from Chapter 2 that this termination property is essential to permit programs to be written without regard for the language's evaluation order.

In the current section, we address this outstanding issue by introducing a novel computation strategy, called *fair OLDT-resolution*, which evaluates each literal in a goal *simultaneously* until the first completely evaluated (or terminating) literal is discovered. By doing so, fair OLDT-resolution overcomes the poor termination behaviour of the left-to-right computation rule and increases the class of terminating tabled programs beyond that of existing tabling systems, like XSB (Sagonas et al. 1994).

A trade-off is made, however, in fair OLDT-resolution between achieving better termination behaviour than in existing tabling systems versus the development of subgoals that do not contribute meaningfully to the resolution. Nevertheless, we demonstrate in this section that fair OLDT-resolution can provide better practical efficiency than fair SLD-resolution when executing compositional programs. (In the subsequent chapter, we rectify this trade-off by developing an evaluation mechanism for logic programs that employs a static analysis to indicate in advance whether the resolution of a given literal will terminate.)

4.2.1 Fair OLDT-resolution

The development of a subgoal in OLDT-resolution can be viewed as a sequential process that may spawn child processes where each child process is a new OLD-tree. From the description of OLDT-resolution in Section 4.1.5, such child processes are selected for development by the computation rule (via the *sprout* component of Definition 4.5). The left-to-right computation rule develops each child process *eagerly*, i.e., as soon as it is created, since the subgoal selected by the sprouting transformation is fixed.

However, a fair selection of a subgoal's child processes can be imposed such that a particular child is not further developed until its parent process has sprouted all of its remaining child processes. In effect, the computation rule is changed from depth-first to breadth-first. A simple alteration to the sprouting transformation of Definition 4.5 suffices to permit the fair development of OLD-trees.

Definition 4.7 (Fair OLDT-resolution). Let P be a program, F a forest of OLD-trees, and N a node in some tree in F labelled with $G = \# :- A_1, \ldots, A_n$, for $n \geq 1$. Fair OLDT-resolution is identical to OLDT-resolution (Definition 4.5) in all but the following case:

Sprout Suppose that N is *active*. Then, for each $1 \leq i \leq n$, add to F a new OLD-tree with root A_i if such an OLD-tree does not already exist in F.

Moreover, every node N in the forest F is selected for OLDT-resolution after a finite number of steps. ♢

The behaviour of fair OLDT-resolution is characterised by the following novel lemma which provides an analogous result for fair OLDT-resolution as Lemma 3.3 does for fair SLD-resolution. The lemma states that the fair OLDT-resolution of a query is guaranteed to terminate whenever any other OLDT-resolution for the query terminates also. In other words, this result ensures that the termination behaviour of a compositional program when executed using fair OLDT-resolution is independent of the manner in which it was composed from other ones.

Lemma 4.2. *For a program P and a goal G, if there exists an infinite fair OLDT-tree for $P \cup \{G\}$ then every OLDT-tree for $P \cup \{G\}$ is infinite.*

Proof. Let $D = G_1, G_2, \ldots,$ be an infinite, fair OLDT-derivation in the fair OLDT-tree for $P \cup \{G\}$, and let R be some arbitrary but fixed computation rule. It suffices to show that D can be transformed into some other infinite OLDT-derivation that uses R as its computation rule. The proof is identical to that of Lemma 3.3 except for the following simple consideration: any literal A in a goal is effectively selected in a fair OLDT-resolution only if it forms the root of a unit subrefutation, since each literal is developed simultaneously until the first unit subrefutation is discovered. Now, as unit subrefutations are finite, we are able to apply the switching lemma as desired and the proof of Lemma 3.3 applies. ♢

Adopting fair OLDT-resolution in a compositional language can overcome the often poor efficiency behaviour of fair SLD-resolution experienced when executing programs. The main reason for the improved efficiency is that redundant computation, which is prevalent in fair SLD-resolution, is eliminated by tabled evaluation. Example 3.3, from the previous chapter, illustrated the adverse effect of redundant computation during the fair SLD-resolution of the program

$$\begin{array}{l} \textit{frontier}\ (\textit{tip}\ A)\ (A : \textit{nil})\ :-\ . \\ \textit{frontier}\ (\textit{bin}\ S\ T)\ Z\ :- \\ \quad \textit{frontier}\ S\ (A : X),\ \textit{frontier}\ T\ (B : Y),\ \textit{append}\ (A : X)\ (B : Y)\ Z. \end{array}$$

and the query

$$\# :- \textit{frontier}\ T\ \text{``}\textit{abcde}\text{''}.$$

The redundancy in this case is due to the vast number of repeated calls to variants of *frontier* $T\ (A : X)$. In this case, fair OLDT-resolution avoids the redundant calls—effectively pruning them from the OLD-tree—and, thus, restores efficient execution. This desirable behaviour of fair OLDT-resolution is also observable during the execution of more elaborate compositional programs as discussed further in Chapter 6.

Fair OLDT-resolution also addresses the problem found in SLD-resolution of entering cyclic derivations when evaluating left-recursive programs. For example, in the case of Example 3.2, the fair OLDT-resolution of

$$\# \ {:}- \ \ \textit{close succ } 0 \ A.$$

reports the single solution $A = 0$ and then terminates whereas the fair SLD-resolution of this query loops infinitely.

4.2.2 Efficiency considerations in fair OLDT-resolution

In an OLDT-resolution, active nodes can be selected for resolution independently of their corresponding answer node. As a result, there exists an intrinsic asynchrony between the generation of answers by an answer node and the consumption of answers by an active node. Moreover, answers can be consumed by an active node in many different ways: the particular method is prescribed by the *scheduling strategy* (Freire, Swift & Warren 1996) and the choice of scheduling strategy can greatly influence the efficiency of a tabled evaluation. The scheduling strategy adopted in OLDT-resolution is analogous to the search strategy in SLD-resolution. We now discuss the scheduling strategy preferable in an implementation of fair OLDT-resolution, like the one presented in Appendix B.

Definition 4.7 above describes fair OLDT-resolution as the growth of a forest of OLD-trees such that the construction of a particular OLD-tree does not proceed indefinitely while other OLD-trees are waiting. However, by developing the OLD-tree for each subgoal in a goal—rather than, say, the leftmost one only—many irrelevant answers will be computed for some subgoals, e.g., the infinite ones. In other words, returning the answers discovered for a subgoal before it is completely evaluated may introduce many OLD-trees into the forest that are destined to fail yet, owing to fairness, must be developed nevertheless. Consequently, much unnecessary computation can be avoided by delaying the return of answers from a subgoal to a goal until the subgoal is completely evaluated. The scheduling strategy which behaves in this manner is a variant of *local scheduling*, introduced by (Freire et al. 1996), and is a natural companion to the breadth-first computation rule of fair OLDT-resolution.

A second phenomenon arises in any implementation of fair OLDT-resolution which requires careful consideration: the breadth-first computation rule of fair OLDT-resolution sprouts every subgoal in a goal, resulting in a forest that often contains more OLD-trees than one created by, say, a depth-first rule. Moreover, as answers are returned from a completely evaluated subgoal to a goal, the remaining subgoals in that goal become instantiated. As a result, some answer nodes in the forest may no longer have any corresponding active nodes. In other words, some answer nodes may become 'disconnected' from the OLDT-resolution of the original goal and it makes little sense to continue to develop them in an actual implementation. The following example illustrates this subtle point.

Example 4.3. Consider again the following query from Example 4.1:

$$\# \ {:}- \ \ \textit{add} \ M \ N \ K, \ \textit{add} \ M \ N \ 50.$$

This time, let us proceed with its evaluation using fair OLDT-resolution. Initially, the OLD-trees for both subgoals, i.e, *add* M N K and *add* M N 50, are sprouted. However, only the latter subgoal terminates and, once it becomes completely evaluated, its answers can be supplied to the remaining subgoal *add* M N K. Each answer updates this literal to produce a sequence of new ones, like # :− *add* 0 50 K for example. At this point, the OLD-tree in the forest rooted at *add* M N K becomes disconnected from each new goal of the original query and is, therefore, no longer relevant to the computation. ♢

The two implementation points above are discussed further in Appendix B which details an implementation of fair OLDT-resolution using local scheduling.

4.3 The limitations of fair OLDT-resolution

We saw in the previous section that fair OLDT-resolution can perform significantly better that fair SLD-resolution when executing compositional programs owing to the elimination of redundant computation. Despite this advantage in efficiency, fair OLDT-resolution does not exhibit the same termination behaviour as fair SLD-resolution. The reason for this is simple: finding a terminating proof of a goal may depend critically on a literal introduced by another subgoal or on the instance of a subgoal in the current goal. However, in OLDT-resolution the selection of literals is restricted by the arbitrary but fixed computation rule. In particular, the fixed requirement means that the selection of subsequent literals is confined to those introduced at the previous resolution step; the siblings of a subgoal may only be selected for resolution once the subgoal itself is completely evaluated. The following example illustrates this point.

Example 4.4. Consider the quicksort program shown below.

$$\begin{array}{lll} \textit{partition}\ A\ \textit{nil}\ \textit{nil}\ \textit{nil} & :- & . \\ \textit{partition}\ A\ (B:X)\ Y\ (B:Z) & :- & \textit{lt}\ A\ B,\ \textit{partition}\ A\ X\ Y\ Z. \\ \textit{partition}\ A\ (B:X)\ (B:Y)\ Z & :- & \textit{geq}\ A\ B,\ \textit{partition}\ A\ X\ Y\ Z. \end{array}$$

$\textit{qsort}\ \textit{nil}\ \textit{nil}\ :-\ .$
$\textit{qsort}\ (A:X)\ Y\ :-\ \textit{partition}\ A\ X\ X_1\ X_2,\ \textit{qsort}\ X_1\ Y_1,\ \textit{qsort}\ X_2\ Y_2,$
$\quad \textit{append}\ Y_1\ (A:Y_2)\ Y.$

where the predicates *lt* and *geq* are the operations $<$ and $\geq$ on Peano natural numbers, respectively. The query

$$\#\ :-\ \textit{qsort}\ (A:2:\textit{nil})\ (1:Y). \tag{4.6}$$

terminates under fair SLD-resolution with the single answer

$A = 1$
$Y = 2 : \textit{nil}$;
no

but fails to terminate under fair OLDT-resolution. The reason for this is apparent on examination of the first few OLDT-resolution steps. After one step, query (4.6) produces the new goal

$$\begin{aligned}\# \ :- \ \ & \mathit{partition}\ A\ (2 : \mathit{nil})\ X_1\ X_2,\ \mathit{qsort}\ X_1\ Y_1,\ \mathit{qsort}\ X_2\ Y_2, \\ & \mathit{append}\ Y_1\ (A : Y_2)\ (1 : Y).\end{aligned} \tag{4.7}$$

Each of the four subgoals in (4.7) has an infinite OLDT-tree and, therefore, the OLDT-resolution of (4.6) is consequently infinite. However, fair SLD-resolution of *partition* A (2 : *nil*) X_1 X_2 in (4.7) with the two unifiable clauses for *partition* produces the goals

$$\begin{aligned}\# \ :- \ \ & \mathit{qsort}\ X_1\ Y_1,\ \mathit{qsort}\ (2 : Z')\ Y_2,\ \mathit{append}\ Y_1\ (A : Y_2)\ (1 : Y), \\ & \mathit{lt}\ A\ 2,\ \mathit{partition}\ A\ \mathit{nil}\ X_1\ Z'.\end{aligned} \tag{4.8}$$

and

$$\begin{aligned}\# \ :- \ \ & \mathit{qsort}\ (2 : Z')\ Y_1,\ \mathit{qsort}\ X_1\ Y_2,\ \mathit{append}\ Y_1\ (A : Y_2)\ (1 : Y), \\ & \mathit{geq}\ A\ 2,\ \mathit{partition}\ A\ \mathit{nil}\ Z'\ X_2.\end{aligned} \tag{4.9}$$

Under fair SLD-resolution, the former goal terminates successfully whilst the latter finitely fails. Therefore, fairness coupled with the interaction of subgoals, i.e., the instantiation of common variables during resolution steps, is crucial for the termination of queries (4.8) and (4.9). ♢

The advantage fair SLD-resolution has over fair OLDT-resolution is that literals introduced by a resolution step join the goal directly and the substitution of the resolution is applied over every subgoal in the goal. OLDT-resolution, on the other hand, attempts to completely evaluate each individual subgoal before an answer is applied over the original goal. By doing so, literals that allow a computation to terminate can be overlooked. In other words, OLDT-resolution is complete with respect to finite success but not to finite failure. Moreover, characterising the precise class of programs for which OLDT-resolution does terminate has so far proven difficult and is a potential area of future investigation, as discussed in Section 7.2.

Nevertheless, fair OLDT-resolution terminates at least as often as 'regular' OLDT-resolution and more often than Prolog, although in both cases fair OLDT-resolution may take more resolution steps than either other strategy. Indeed, the number of OLD-resolution steps taken during the fair OLDT-resolution of a goal hinges upon the number of distinct literals in the forest of OLD-trees: each distinct literal forms an answer node in the forest and is computed using program clause resolution, whereas variant literals take their solutions from the table. Therefore, the more distinct literals there are in a resolution, the larger the forest of OLD-trees and, hence, the greater the number of resolutions that must be performed. The following example illustrates the difficulties fair OLDT-resolution can encounter when many distinct literals exist in a program.

Example 4.5. Consider the program:

$$\begin{aligned}&\mathit{mem}\ A\ (A : X)\ :-\ . \\ &\mathit{mem}\ A\ (B : X)\ :-\ \mathit{mem}\ A\ X.\end{aligned}$$

$$\mathit{explode}\ X\ :-\ \mathit{explode}\ (a\ X),\ \mathit{explode}\ (b\ X).$$

The constructors *a* and *b* are used to ensure that the recursive calls to *explode* are distinct from one another; if the calls were not distinct then tabling would avoid their redundant computation. Now, consider the following query:

$$\# \;:-\; mem\ a\ \text{``}bbbbbbbbbbbbbbbbbbbbbbbbbbbbb\text{''},\ explode\ a.$$

The efficiency behaviour of this query under fair OLDT-resolution is exponential in the length of the list since each recursive call to *explode* is distinct and, therefore, sprouts two new distinct OLD-trees. ◇

One solution to the problem of developing useless branches in Example 4.5 is to spot that calls to *explode* do not terminate and, therefore, postpone them in favour of the calls to *mem* which do terminate. By doing so, the size of the search space in this case becomes linear in the length of the list by avoiding the useless branches in the search tree that would otherwise have to be examined. In the following chapter, we pursue this point at more length and describe a computation rule that selects a literal in a goal depending upon whether its resolution terminates.

4.4 Summary

In the previous chapter, we saw that the fair SLD-resolution of a query will terminate should any other SLD-resolution of it terminate also. However, the efficiency of fair SLD-resolution using the breadth-first computation rule is often prohibitive in practice, owing primarily to the redundant computation of many literals in a goal. Furthermore, SLD-resolution using any computation rule can fail to terminate for left-recursive programs.

Tabled evaluation of logic programs, originally introduced by (Tamaki & Sato 1986) and reconsidered by (Warren 1992), ostensibly addresses these two limitations of fair SLD-resolution by eliminating redundant computation and avoiding the cycles that are intrinsic to left-recursive programs. However, existing tabling systems, like XSB (Sagonas et al. 1994), invariably employ the left-to-right computation rule and, therefore, inherit the same poor termination properties of Prolog when executing compositional programs.

In this chapter, we introduced the new concept of fair tabled evaluation to overcome the limitations of the left-to-right computation rule. We began by adapting OLD-resolution (Tamaki & Sato 1986) in a straightforward way to use any arbitrary but fixed computation rule which provided the flexibility required for our novel notion of fairness in tabled evaluation. Next, we made straightforward alterations to the existing definition of OLDT-resolution (Warren 1992, Tamaki & Sato 1986) and to its completeness proof. We then defined fair OLDT-resolution and showed that it terminates for a query whenever any other OLDT-resolution terminates for it. We also demonstrated that fair OLDT-resolution enhanced the execution efficiency and termination behaviour of a number of compositional programs.

However, the price paid for the improved efficiency behaviour over fair SLD-resolution is that some compositional programs that terminate under fair SLD-resolution no longer do so under fair OLDT-resolution. Furthermore, although fair OLDT-resolution is guaranteed to terminate as often as Prolog and XSB, and more often in practice, identifying the precise class of programs for which it terminates is difficult to characterise: we have been unable to do so at at this time and further investigation into this point would be a valuable area of future work.

Chapter 5

Prioritised Fair SLD-Resolution

Fair OLDT-resolution was introduced in the previous chapter to eliminate the redundant subcomputations that caused fair SLD-resolution to execute compositional programs inefficiently. However, unlike fair SLD-resolution, we saw that fair OLDT-resolution is incomplete with respect to finite failure. Furthermore, fair OLDT-resolution can itself suffer from adverse efficiency behaviour since, owing to its fairness condition, it is obliged to develop OLD-trees that do not meaningfully contribute to the computation of a query.

In this chapter, we address these two issues by formulating a novel resolution strategy, termed *prioritised fair SLD-resolution* (or *prioritised resolution* in short), which prioritises the selection of literals in a goal according to whether their computation is known to terminate. Prioritised resolution reinstates the desirable termination property of fair SLD-resolution—the fair SLD-resolution of a query is guaranteed to terminate whenever any other SLD-resolution strategy terminates for it also—yet simultaneously enhances its efficiency behaviour. We begin by defining a new computation rule, followed by a new algorithm which automatically generates a call set for the prioritised resolution of a program.

A crucial component of this algorithm is a termination test that detects termination of queries to logic programs. We review an existing termination test (Lindenstrauss & Sagiv 1997) and present a slight modification to it to determine termination of programs with respect to the new computation rule of prioritised resolution, rather than the left-to-right one. Furthermore, we describe an extension to the test to allow higher-order predicates to appear in programs, which were ignored in the original presentation. Although termination analysis has recently been used to verify the correctness of logic programs (Speirs et al. 1997), its application in this chapter to improve the efficiency of SLD-resolution is new.

5.1 Introduction

Recall from Section 3.5 that one problem with SLDF-resolution (SLD-resolution with Floundering) is that the method of automatically determining its call set is heuristic in nature. In many cases, the computed call set is a poor approximation of those subgoals that have finite SLDF-trees. Worse still, the computed set is often 'unsafe', i.e., an SLDF-resolution using an unsafe call set may terminate for some goal but may loop infinitely for an instantiation of it. The primary cause of such behaviour is the incremental nature of coroutining in the sense that, once a resolution step is performed, the computation rule is free to select any new callable literal in the resolvent. As a result, it is difficult to predict whether a query will execute efficiently and certainly whether it will terminate.

We can overcome these limitations by generalising the coroutining computation rule of Definition 3.8 to a new one that selects a literal in a goal only if its SLD-tree, as opposed to its SLDF-tree, is finite with respect to the computation rule. Then, since the resolution of the selected literal will terminate, it may be resolved away completely before considering any other subgoal in the goal. Essentially, the only difference between the new computation rule and the one in Definition 3.8 is the representation of the call set.

Of course, the main difficulty in describing such a computation rule rests in the determination of whether a given literal terminates: since definite clause logic is Turing complete, and the Halting Problem is undecidable, determining whether the proof of a literal terminates is also undecidable. However, in logic programs, the only cause of non-termination is infinite recursion and, for this special case, it is possible to detect termination automatically for a large, but so far uncharacterised, class of programs (Lindenstrauss & Sagiv 1997, Codish & Taboch 1997, Plümer 1991, Sagiv 1991).

The most important task in prioritised resolution is to describe an algorithm that automatically determines a safe call set for a program. The actual representation of a callable literal is guided by the representation of a literal during termination analysis since the call set will comprise precisely such literals. Termination analysis requires a comprehensive examination of the shape of a literal's arguments since it strives to determine termination for all literals whose arguments share the same structural patterns. In particular, the proof of termination for a literal A extends to *all* those literals that share the same canonical structure as A, irrespective of their actual form. An abstraction function, called a *norm*, has been used in termination analyses to determine whether or not two terms share the same canonical structure. The following section reviews the notion of a norm and paves the way towards the representation of the call set or prioritised resolution, introduced in Section 5.3.

5.2 Symbolic norms and norm functions

In order to uncover a suitable representation for members of the call set in prioritised resolution, we must examine the way in which termination is shown for a literal. Termination analysis (Lindenstrauss & Sagiv 1997, Codish & Taboch 1997, Plümer 1991, Sagiv 1991) has been formulated for classes of literals that share the same canonical structure, with respect to a measure called a *norm* (Dershowitz 1982). As detailed in this section, norms provide an account of the essential characteristics of a term's structure. Consequently, norms can be used to determine whether an actual literal shares the same essential structure as another. Moreover, norms provide an indication of the relative size of terms with the same canonical structure and, as shown later, can be compared using a well-founded order.

Existing termination analyses attempt to show that every derivation in an SLD-resolution is finite. This is invariably achieved by ensuring that the size of at least one of a literal's arguments, with respect to some norm, strictly decreases at each recursive call in a derivation towards a terminating condition which is ensured by the well-founded order on norms. The rigorous formulation of such a termination test is the topic of Section 5.6

which culminates in Theorem 5.1 (Sagiv 1991). The theorem provides a necessary and sufficient condition under which logic programs may be shown to terminate. Below, we review the standard definitions of symbolic norms and norm functions (Lindenstrauss & Sagiv 1997) which are used pervasively in this chapter.

Definition 5.1 (Symbolic norms). A *symbolic norm* is a linear integer expression of the form $a_0 + a_1X_1 + \cdots + a_nX_n$, where $n \geq 0$ and, for $0 \leq i \leq n$, each a_i is an integer and each X_i is a distinct variable. $\diamondsuit$

Definition 5.2 (Norm functions). Let T be a term. A *norm function* $\|\_\|$, or just a *norm*, maps terms to symbolic norms and is defined as:

$$\|T\| = \begin{cases} T, & \text{if } T \text{ is a variable} \\ n + \sum_{i=1}^{k} m_i \times \|T_i\|, & \text{if } T = f\ T_1 \ldots T_k, \text{ for } k \geq 0 \end{cases}$$

where f is a constructor and, for $0 \leq i \leq k$, T_i are terms, and n and m_i are non-negative integers that depend only on f. $\diamondsuit$

The following example defines the term-size norm which is occurs frequently in termination analysis (Lindenstrauss & Sagiv 1997, Plümer 1991).

Example 5.1 (Term-size norm). The *term-size* norm is obtained from Definition 5.2 by setting n to be the arity of the constructor f, and each m_i to be 1, as shown below:

$$\|T\| = \begin{cases} T, & \text{if } T \text{ is a variable} \\ n + \sum_{i=1}^{n} \|T_i\|, & \text{if } T = f\ T_1 \ldots T_n \end{cases}$$

The term-size norm of a term is an integer if only if the term is ground. $\diamondsuit$

The constants n and each m_i in Definition 5.2 lend generality to the construction of symbolic norms. However, we have found that taking $n = 0$ for constructors of arity 0, and $n = 1$ otherwise, and each $m_i = 1$, gives the same termination results in practice as using any other values. Indeed, these findings have been recently corroborated by another implementation of termination analysis (Speirs et al. 1997).

The symbolic norm of a term essentially provides an abstract interpretation of its structure in a canonical way. In particular, symbolic norms can be used to determine whether one term exhibits a comparable form to another term. The significance of this is that termination analysis is able to prove termination for entire classes of literals, in particular those whose arguments share the same canonical structure as determined by some norm. So, given some norm, two terms have the same essential structure if they are both *instantiated enough* with respect to that norm; the definition is due to (Lindenstrauss & Sagiv 1997) and is repeated below.

Definition 5.3 (Instantiated enough). A term T is *instantiated enough* with respect to some norm $\|\_\|$ if its symbolic norm, $\|T\|$, is an integer. $\diamondsuit$

The above notion of a term being instantiated enough takes into account the fact that variables may occur in it. This is important since, during a resolution, a variable in a term may become instantiated to any other term and, therefore, markedly affect the overall structure of the term that it appears in. However, depending on the norm used to describe the canonical structure of terms, variables in certain positions in a term may or may not be considered significant. The most important fact regarding a term being instantiated enough with respect to some norm is that its symbolic norm is invariant under substitution. In other words, for a term T that is instantiated enough (with respect to some norm) and for any substitution ϕ, it is the case that $\| T \| = \| T[\phi] \|$. As demonstrated later, it is precisely this property that allows the call set of prioritised resolution to be safe. The following example defines the commonly used list-size norm and serves to demonstrate how the choice of norm affects whether a term is instantiated enough or not.

Example 5.2 (List-size norm). The *list-size* norm is defined as:

$$\| T \| = \begin{cases} T, & \text{if } T \text{ is a variable} \\ 0, & \text{if } T = \mathit{nil} \\ 1 + \|X\|, & \text{if } T = A : X \end{cases}$$

The importance of the list-size norm is that any *finite* list is instantiated enough, irrespective of whether its elements are instantiated or not. For example, both $a : b : c : \mathit{nil}$ and $A : B : C : \mathit{nil}$ are instantiated enough, with symbolic norm 3, using the list-size norm but only the former list is instantiated enough, with symbolic norm 6, using the term-size norm. ♢

5.3 Instantiation patterns

The previous section reviewed norms as functions taking terms to symbolic norms and explained how norms can be used to determine whether two terms have a comparable canonical structure: if two terms are instantiated enough with respect to some norm then they share the same essential form. In this section, we review the notion of an *instantiation pattern* which is an abstraction of a literal that indicates whether or not its arguments are instantiated enough with respect to a norm. Critically, the proof of termination of a literal can be performed on its instantiation pattern rather than on its actual form. Therefore, the result of termination analysis for an instantiation pattern extends to *all* those actual literals which share that instantiation pattern.

The definitions of instantiation patterns and the abstraction function that transforms a literal into its instantiation pattern, with respect to some norm, are given by (Lindenstrauss & Sagiv 1997) and are repeated below.

Definition 5.4 (Instantiation patterns). Let P be a program and p be a predicate in P with arity n, for $n \geq 0$. Furthermore, let ie and nie be special terms that may not occur in P. An *instantiation pattern* for p is a literal of the form $p\ T_1\ \ldots\ T_n$ where, for each $1 \leq i \leq n$, the term T_i is either ie or nie. ♢

The special terms ie and nie in an instantiation pattern are used to represent arguments that are respectively instantiated enough or not instantiated enough, for some given norm. The abstraction function α, taking literals to their instantiation patterns with respect to some norm, is defined next.

Definition 5.5. Let A be a literal and $\|\_\|$ a norm. The abstraction function α, taking literals to instantiation patterns, is defined as follows:

$$\alpha(A) = \begin{cases} p\ \alpha(T_1)\ \ldots\ \alpha(T_n), & \text{if } A = p\ T_1\ \ldots\ T_n \text{ where } p \text{ is a predicate symbol} \\ ie, & \text{if } A \text{ is a term instantiated enough w.r.t. } \|\_\| \\ nie, & \text{otherwise} \end{cases}$$

where $T_1, \ldots, T_n$ are terms, for $n \geq 0$. $\diamondsuit$

Example 5.3. Consider calculating $\alpha(\mathit{append}\ X\ (A : \mathit{nil})\ Y)$ with respect to the list-size norm. We proceed by applying the first equation of Definition 5.5 to produce the new literal $\mathit{append}\ \alpha(X)\ \alpha(A : \mathit{nil})\ \alpha(Y)$. The first and third subexpressions in this term evaluate to *nie* via the third equation of Definition 5.5 since a variable is always not instantiated enough with respect to the list-size norm. However, the term $A : \mathit{nil}$ is instantiated enough with respect to the list-size norm and, therefore, $\alpha(A : \mathit{nil})$ evaluates to *ie*. Hence we have that $\alpha(\mathit{append}\ X\ (A : \mathit{nil})\ Y) = \mathit{append}\ \mathit{nie}\ \mathit{ie}\ \mathit{nie}$.

Alternatively, let us compute $\alpha(\mathit{append}\ X\ (A : \mathit{nil})\ Y)$ with respect to the term-size norm. The application of the first equation of Definition 5.5 results once more in the literal $\mathit{append}\ \alpha(X)\ \alpha(A : \mathit{nil})\ \alpha(Y)$ and, again, the first and third subexpressions are not instantiated enough, this time with respect to the term-size norm. Furthermore, the term $A : \mathit{nil}$ is not instantiated enough with respect to the term-size norm which means that $\alpha(A : \mathit{nil}) = \mathit{nie}$. Therefore, we have that $\alpha(\mathit{append}\ X\ (A : \mathit{nil})\ Y) = \mathit{append}\ \mathit{nie}\ \mathit{nie}\ \mathit{nie}$ for the term-size norm. $\diamondsuit$

One last useful concept is the notion of one instantiation pattern subsuming another. Essentially, one instantiation pattern subsumes another if the former has at least the same structure as the latter, and the definition, due to (Lindenstrauss & Sagiv 1997), is as follows.

Definition 5.6 (Subsuming instantiation patterns). Let p be a predicate, and A and B be instantiation patterns for p. Then A *subsumes* B if, for every argument position in B that is *ie*, the corresponding argument position in A is also. $\diamondsuit$

As formalised in the following section, the call set in prioritised fair SLD-resolution is a set of instantiation patterns, and a literal in a goal is callable if its instantiation pattern is contained in the call set. Moreover, a call set of instantiation patterns is always finite for a finite program even though it describes an infinite set of actual literals whose resolution terminates.

5.4 Prioritised fair SLD-resolution

In this section, we introduce the novel resolution strategy *prioritised fair SLD-resolution*, or *prioritised resolution* for brevity, which prioritises the selection of literals in a goal according to whether or not their resolution terminates. Prioritised resolution is an instance of the fair SLD-resolution formulated in Chapter 3 and, consequently, inherits the desirable termination property of fair SLD-resolution, namely, that if the resolution of a query terminates using some computation rule, then the prioritised fair SLD-resolution of that query also terminates. As we shall see shortly, however, prioritised resolution also overcomes the efficiency problems of fair SLD-resolution that were detailed in Section 3.4. In the following discussion, we borrow terminology from coroutining, reviewed in Section 3.5.

To institute prioritised fair SLD-resolution, a novel computation rule is envisaged as follows: given a goal and a set of callable literals, the rule selects the leftmost callable literal in the goal. In the case where the goal is floundered, i.e., no known terminating subgoal exists in it, a fair resolution step using the breadth-first computation rule of Definition 3.4 is performed. The latter case ensures that fairness is maintained and, therefore, a finite SLD-tree is constructed, if at all possible.

The criterion for a literal to be callable is that its prioritised SLD-resolution terminates without the need for any fair resolution steps. Thus, inefficient fair resolution steps are eliminated during the prioritised resolution of a callable literal and it is precisely this fact that makes prioritised resolution more efficient than fair SLD-resolution (Definition 3.4). Termination analysis, presented later in Section 5.6, is used to determine whether an instantiation pattern terminates without the need for fair resolution steps. We begin the presentation of prioritised resolution by providing the definition of the call set, below.

Definition 5.7 (Call set). A *call set* $\mathbb{C}$ is a set of instantiation patterns (from Definition 5.4). $\diamondsuit$

From this definition, a literal is callable if its instantiation pattern is included in the call set of the program.

Definition 5.8 (Callable literals). Let A be a literal, $\mathbb{C}$ be a call set, and α be the transformation function from Definition 5.5. Then A is *callable* with respect to $\mathbb{C}$ if and only if $\alpha(A) \in \mathbb{C}$. $\diamondsuit$

The description of the computation rule for prioritised resolution is similar to the one for coroutining (from Definition 3.8) but instead uses the two previous definitions where appropriate. Finally, prioritised fair SLD-resolution can be defined as follows.

Definition 5.9 (Prioritised fair SLD-resolution). Let R be a computation rule and let $\mathbb{C}$ be a set of instantiation patterns. Furthermore, let $C = A :- B_1, \ldots, B_m$, for $m \geq 0$, be a definite clause and $G = \# :- A_1, \ldots, A_n$, for $n \geq 1$, a goal. There are two cases to consider:

1. Suppose the selected literal is $R_{\mathbb{C}}(G) = A_i$, for $1 \leq i \leq n$, and that A_i and A are unifiable with most general unifier ϕ. The *SLD-resolvent* of C and G is the goal
$$G' = (\# :- B_1, \ldots, B_m, A_1, \ldots, A_{i-1}, A_{i+1}, \ldots, A_n)[\phi]$$
2. Suppose G is floundered but that A_1 and A are unifiable with most general unifier ϕ. Then the selected literal is A_1 and the SLD-resolvent of C and G is the goal
$$G' = (\# :- A_2, \ldots, A_n, B_1, \ldots, B_m)[\phi]$$

The substitution ϕ is called the *substitution of the SLD-resolution.* $\diamondsuit$

Prioritised fair SLD-resolution exhibits more predictable efficiency behaviour than that of fair SLD-resolution using the breadth-first computation rule since, whenever possible, prioritised resolution will select a terminating literal for evaluation over an infinite one. Consequently, many useless branches in the SLD-tree that would finitely-fail under fair SLD-resolution are completely avoided by prioritised resolution. The practical benefits of executing compositional programs using prioritised resolution are examined in detail in Chapter 6.

Furthermore, prioritised resolution is suitable for the application of tabling techniques, such as those presented in Chapter 4, to eliminate the redundant evaluation of callable literals. Therefore, prioritised resolution has the potential to increase further its efficiency gains over fair SLD-resolution. This point is pursued in more detail in Section 7.2.

5.5 Generating a call set automatically

In this section, we describe a novel algorithm—depicted in Figure 5.1—that automatically constructs a call set of a program for use in prioritised resolution. The algorithm commences, for a program P and some norm function, by using *patterns* P to construct the set $\mathbb{C}_0$ of every possible instantiation pattern for each predicate defined in P. The job of $\mathbb{C}$ is to accumulate the instantiation patterns of $\mathbb{C}_0$ whose prioritised resolution terminate without performing any fair resolution steps.

On each iteration of the outer loop, a set $\mathbb{C}'$ identical to $\mathbb{C}_0$ is created. Then, for each instantiation pattern $A \in \mathbb{C}'$, the following checks are performed. Firstly, the function call *subsumed* $(A, \mathbb{C})$ checks to see if A is subsumed by any instantiation pattern in the call set $\mathbb{C}$, according to Definition 5.6. If so, then A is guaranteed to terminate. Otherwise, the second check is carried out: the function *finite* tests whether the instantiation pattern A has a finite prioritised resolution, with respect to the call set $\mathbb{C} \cup \{A\}$, without performing any fair resolution steps. If either of the two above checks succeed, then A is removed from $\mathbb{C}_0$ and added to $\mathbb{C}$. The variable *finished* is assigned the value *false* since existing instantiation patterns in $\mathbb{C}_0$ may be proved terminating with respect to the expanded set $\mathbb{C}$.

The execution of *generate* always terminates for a program that contains a finite number of clauses since, in this case, the inner loop iterates over a finite set and, on each iteration, either decreases the size of $\mathbb{C}_0$ or maintains that *finished* $=$ *true*. The function *generate* terminates when every instantiation pattern in $\mathbb{C}_0$ cannot be shown to terminate. Neglecting the function *finite* for the time being, the efficiency behaviour of *generate* P is exponential in the maximal arity of the predicates in P since this factor dominates the cardinality of $\mathbb{C}_0$.

All that remains is the realisation of the function *finite*, i.e., the determination of whether the prioritised resolution of literals with a particular instantiation pattern terminate without performing any fair resolution steps. Such tests exist for the left-to-right computation rule (Lindenstrauss & Sagiv 1997, Codish & Taboch 1997, Sagiv 1991) and, in the following section, we extend the test of (Lindenstrauss & Sagiv 1997) in a straightforward manner to use the computation rule of prioritised resolution instead of the left-to-right one.

```
$generate\ P =$
  let $\mathbb{C}_0 := patterns\ P$ and $\mathbb{C} := \{\,\}$ and $finished := false$
  while not $finished$ do
    let $\mathbb{C}' = \mathbb{C}_0$
    $finished := true$
    while $\mathbb{C}' \neq \{\,\}$ do
      let $A \in \mathbb{C}'$ and $\mathbb{C}' := \mathbb{C}' - \{A\}$
      if $subsumed\ (A, \mathbb{C})$ or $finite\ (P, A, \mathbb{C}) \longrightarrow$
        $\mathbb{C}_0 := \mathbb{C}_0 - \{A\};\ \mathbb{C} := \mathbb{C} \cup \{A\};\ finished := false$
      fi
    done
  done
  return $\mathbb{C}$
```

Figure 5.1: The algorithm for determining a set of callable literals.

5.6 Termination analysis

In the logic programming community, much research has taken place over the past decade to detect automatically whether a given goal will terminate for a given program (Lindenstrauss & Sagiv 1997, Codish & Taboch 1997, Sagiv 1991, Plümer 1991). However, all of the algorithms developed test termination with respect to the left-to-right computation rule of Prolog (Definition 2.7). In this section, we review the termination test introduced by (Lindenstrauss & Sagiv 1997) and present a straightforward modification to it to allow the termination of literals to be determined for the computation rule of prioritised resolution, rather than for the left-to-right one. The adapted test is a vital component of the algorithm in Figure 5.1 which constructs a program's call set automatically for prioritised resolution. Thus, for the first time, termination analysis is used to improve the execution efficiency of logic programs.

The existing test is augmented with the notion of a call set as well as the novel computation rule of prioritised resolution, both introduced in Section 5.4. Moreover, in Section 5.7, the termination test is adapted to cope with the occurrence of higher-order predicates in programs—pervasive in compositional programs—which was neglected in the original presentation of the test (Lindenstrauss & Sagiv 1997).

Termination analysis attempts to show that the SLD-tree for a literal contains no infinite branches. Broadly speaking, the instantiation pattern of a literal, with respect to some norm, is used to to generate an approximation of the actual SLD-tree. Each derivation is shown to terminate by demonstrating that the size of each recursive call to a predicate strictly decreases in size, with respect to some well-founded order. The size of a term is determined by an appropriate norm function and the ordering is defined by the comparison of symbolic norms. Only arguments that are instantiated enough are considered in termination analysis since only they have a constant symbolic norm and can therefore be meaningfully compared. The choice of norm used in termination analysis, then, is crucial: an inappropriate norm may fail to identify some arguments of a literal as instantiated enough when another norm would not. Consequently, a proof of termination for a literal may be impossible. The following example illustrates this point.

Example 5.4. Consider $\mathit{append}\ (1 : A : 3 : \mathit{nil})\ X\ (B : 2 : 3 : 4 : \mathit{nil})$. Both the first and third arguments of the literal are instantiated enough with respect to the list-size norm. However, neither argument is instantiated enough with respect to the term-size norm. In this latter case, termination analysis would be unable to prove that the literal terminates. $\diamondsuit$

Since an arbitrary number of conceivable norms exist, it would be desirable to discover a method of automatically deriving the most suitable norm to use in a termination analysis, i.e., the norm that permits the greatest number of literals to be proven to terminate. To date, we have been unable to formalise such a process, although one possible direction of future research in this area is discussed in Section 7.2. In the rest of this section, we consider proving termination of programs with respect to the term-size norm of Example 5.1.

5.6.1 An informal example of termination analysis

To motivate the process of termination analysis, consider the following predicate that determines if one list is a sublist of another list:

$$\mathit{sublist}\ X\ Y\ \ {:}-\ \ \mathit{append}\ U\ X\ V,\ \mathit{append}\ V\ W\ Y.$$

Therefore, X is a sublist of Y if there exists three lists U, V, and W such that $U \mathbin{+\!\!\!+} X = V$ and $V \mathbin{+\!\!\!+} W = Y$. Let us informally demonstrate how the following query can be shown to terminate under prioritised fair SLD-resolution with respect to the term-size norm:

$$\#\ \ {:}-\ \ \mathit{sublist}\ X\ (a : b : c : \mathit{nil}). \tag{5.1}$$

The instantiation pattern for query (5.1) is *sublist nie ie*, where the symbolic norm of $a : b : c : \mathit{nil}$ is 6 with respect to the term-size norm. A portion of the prioritised fair SLD-tree for (5.1) is depicted in Figure 5.2, where the selected literal at each resolution step is underlined.

Termination analysis approximates the shape of the actual SLD-tree by replicating the resolution step made at each node. In this way, portions of a derivation are generated until a recursive call to a literal on the path is discovered, or all such non-recursive portions have been examined. At each recursive call, it is possible to check whether the call has decreased in size since its initial appearance by comparing the symbolic norm of each appropriate argument.

The approximated tree usually contains far fewer nodes than the actual SLD-tree since recursive calls are terminal nodes in the analysis. Moreover, since each literal is represented by its instantiation pattern, the termination test does not perform explicit unification of literals in an abstract resolution step. To explain, consider performing a simulated resolution step for an abstract literal and a program clause: the formal

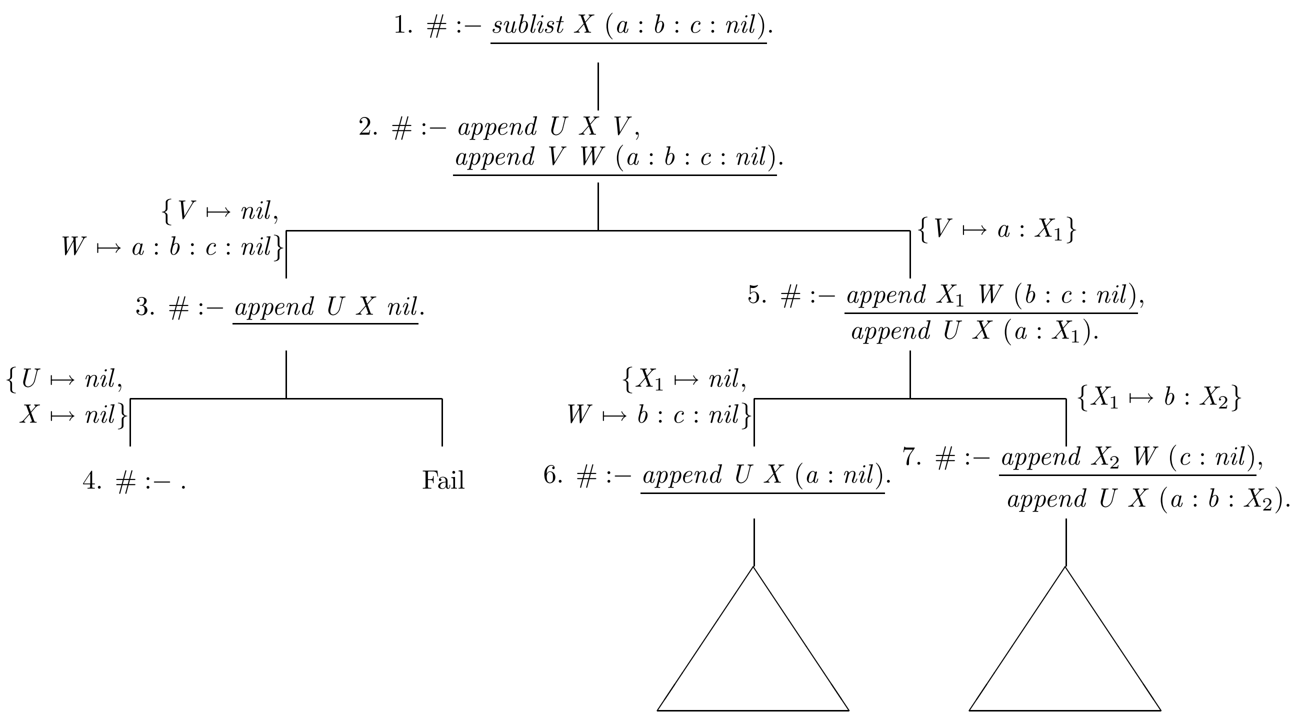

1. # :− sublist X (a : b : c : nil).
2. # :− append U X V, append V W (a : b : c : nil).
{V ↦ nil, W ↦ a : b : c : nil}
{V ↦ a : X1}
3. # :− append U X nil.
5. # :− append X1 W (b : c : nil), append U X (a : X1).
{U ↦ nil, X ↦ nil}
{X1 ↦ nil, W ↦ b : c : nil}
{X1 ↦ b : X2}
4. # :− .
Fail
6. # :− append U X (a : nil).
7. # :− append X2 W (c : nil), append U X (a : b : X2).

Figure 5.2: The prioritised fair SLD-tree for # :− *sublist* X $(a : b : c : nil)$.

parameters of the clause are instantiated enough if the corresponding arguments of the abstract literal are instantiated enough. Then, for each subgoal in the body of the clause, if all the variables contained in each of its arguments are instantiated enough then that argument becomes instantiated enough. These new subgoals are then used to create more abstract resolutions, following the actual resolution performed in the real SLD-tree.

So, in the SLD-tree of Figure 5.2, node 1 generates node 2 via the sole clause of *sublist*. In the abstract resolution step for this clause and the instantiation pattern *sublist nie ie*, only the final argument in the subgoal *append* V W $(a : b : c : nil)$ is instantiated enough with symbolic norm 6. Furthermore, this literal is selected for resolution in node 2 since it is callable. Let us show why this is the case. The instantiation pattern for the selected literal is *append nie nie ie*. In node 2, the resolution of it with the second clause for *append* produces node 5, introducing the next selected literal *append* X_1 W $(b : c : nil)$, i.e., a recursive call to *append*. Again, this literal has only its third argument instantiated enough, but this time with symbolic norm 4 with respect to the term-size norm. Now, since the third argument in the recursive call has a smaller norm than the original and there is only one recursive clause for *append* in the program, we deduce that all calls to *append* terminate that have an instantiated enough third argument with respect to the term-size norm.

The knowledge that the selected literal in node 2 terminates leaves us with the proof of whether the consequently instantiated version of *append* U X V terminates. The concern here is to determine whether any of the variables in this term will have become instantiated enough (with respect to the term-size norm) after the successful resolution of *append* V W $(a : b : c : nil)$. This information is gleaned from a preliminary stage of termination analysis, called *instantiation analysis*, that determines the possible instantiation patterns of literals in the success set of a predicate. Instantiation analysis is detailed in Section 5.6.2 but, for the time being, let us assume that both V and W become instantiated enough.

Therefore, the final argument of *append* U X V becomes instantiated enough. The size of V follows from constraints that exist between the sizes of the arguments of *append*. These constraints—established during another preliminary stage of termination analysis, called *constraint analysis*—provide vital information regarding the relative size of arguments and, consequently, help establish that recursive calls decrease in size. In the current example, however, such information is not necessary. Indeed, constraint analysis is an optional component of termination analysis but, more often than not, is essential to prove termination for a query. Returning to the example, the remaining subgoal has the instantiation pattern *append nie nie ie*. However, such patterns have just been proven to terminate. Therefore, we deduce that calls to *sublist* that have their second argument instantiated enough with respect to the term-size norm terminate since every branch in the abstract tree is finite.

The motivating example above provides an informal description of termination analysis for logic programs; we will return to this example later in the chapter and present a rigorous termination analysis once the necessary machinery has been introduced. The critical benefit of termination analysis is that termination is determined for instantiation patterns of literals with respect to some norm. Therefore, the result of termination analysis holds for *all* literals that share the same instantiation pattern. In the following section, the process of instantiation analysis is detailed which identifies all those arguments of a literal that become instantiated enough after its successful resolution.

5.6.2 Instantiation analysis

For a given predicate and a norm, instantiation analysis determines every possible instantiation pattern of the literals that are contained in the success set of the predicate, with respect to the norm. So, for the *append* predicate and the term-size norm, instantiation analysis would discover that calls to *append* where only the third argument is instantiated enough result in the other two arguments being instantiated enough on termination. Alternatively, calls to *append* where only the first argument is instantiated enough result in the remaining two arguments being insufficiently instantiated with respect to the term-size norm.

In this section, we describe the process of instantiation analysis, originally introduced by (Lindenstrauss & Sagiv 1997), which is a relatively simple application of abstract interpretation of logic programs, due to (Cousot & Cousot 1992). In the following discussion, we use the terminology introduced in Section 5.3, in particular the abstraction function α from Definition 5.5. In addition, from the standard definition of the Herbrand base of a program (Lloyd 1987), we define the *extended* Herbrand base of a program which differs from the standard one only in that it also contains literals that include variables.

The instantiation analysis of a program is computed by way of a bottom-up abstract interpretation. For a program P and its extended Herbrand base B, the effect of the instantiation analysis of P is described as an immediate consequence operator $\mathrm{T}_P : \mathsf{P}B \to \mathsf{P}B$. The behaviour of the function T_P is defined as follows: for a set $S \subseteq B$, $\mathrm{T}_P(S)$ is the set of literals in B that are heads of the clauses in P whose body literals all appear in S. The least fixed-point of T_P therefore comprises those literals in B that have an SLD-refutation with the identity substitution as the computed answer, i.e., that are logical consequences of P. Although this set may be infinite, we can appeal to the Fixed-Point Abstraction Theorem (Cousot & Cousot 1992) to help us compute the finite set of its instantiation patterns.

The Fixed-Point Abstraction Theorem is a standard result in logic programming which essentially permits the abstract interpretation of the least fixed-point of a function to be computed instead as the least fixed-point of an abstract interpretation of the function's domain. A number of rigorous conditions must be met by the function and the abstraction operator, in our case the functions T_P and α, respectively; the details of showing that the instantiation analysis adheres to these conditions can be found in (Lindenstrauss & Sagiv 1996). In our case, the Fixed-Point Abstraction Theorem dictates that, if a literal A is a logical consequence of a program P, then the abstract interpretation of that literal, i.e., $\alpha(A)$, is contained in the minimal model of a program P', where P' is an abstract interpretation of P. More specifically, we have that $\alpha(\mathrm{lfp}(\mathrm{T}_P)) \subseteq \mathrm{lfp}(\mathrm{T}_{P'})$ where 'lfp' is the least fixed-point operator.

The first step in describing the algorithm of instantiation analysis is to provide the function that transforms a program P into the abstract program P' as follows: for each clause C in P having n distinct variables, we create all 2^n instances of C, obtained by either substituting the special token '*ie*' for each variable or leaving the variable unchanged. For each new clause C', we apply α to each literal in C', resulting in a ground clause. The whole process produces a program P' that consists solely of ground clauses and, hence, its minimal model, i.e., the set of all literals that are true consequences of P', can be computed bottom-up in a finite number of steps.

The following example illustrates how to compute the minimal model of a program.

Example 5.5. Consider the *append* predicate whose definition is repeated below.

$$\begin{array}{l} \textit{append nil } Y\ Y\ \ {:-}\ \ . \\ \textit{append } (A : X)\ Y\ (A : Z)\ \ {:-}\ \ \textit{append } X\ Y\ Z. \end{array}$$

Let us construct the abstract representations of the first clause according to the method described above, with respect to the term-size norm. The abstraction process results in the following two new clauses obtained by substituting the special term *ie* for the variable Y and by leaving Y unchanged.

$$\begin{array}{l} \textit{append nil } Y\ Y\ \ {:-}\ \ . \\ \textit{append nil ie ie}\ \ {:-}\ \ . \end{array}$$

To each of these new clauses we apply the operator α, from Definition 5.5, which respectively produces the following two ground abstract clauses:

$$\begin{array}{l} \textit{append ie nie nie}\ \ {:-}\ \ . \\ \textit{append ie ie ie}\ \ {:-}\ \ . \end{array}$$

The next step is to translate the second clause of *append*, which contains four distinct variables, into its $2^4 = 16$ abstract clauses. For reasons of brevity, let us detail the construction of only one abstract clause: given the substitution $\{X \mapsto ie, Y \mapsto ie, Z \mapsto ie\}$, we obtain the intermediate clause

$$append\ (A : ie)\ ie\ (A : ie)\ \ :-\ \ append\ ie\ ie\ ie.$$

As before, we apply α to each literal in this clause which results in the following abstract clause:

$$append\ nie\ ie\ nie\ \ :-\ \ append\ ie\ ie\ ie.$$

Once we have obtained all the abstract clauses for the original program by following the method described above, we can then compute the minimal model of the abstract interpretation of *append*, starting with $\mathrm{T}_P(\{\,\})$ which results in a set that contains all the head literals of each fact in the abstract program. Specifically, in the current example we arrive at the set

$$\mathrm{T}_P(\{\,\}) = \{append\ ie\ nie\ nie, append\ ie\ ie\ ie\}.$$

We compute the next element in the least fixed point from this set as $\mathrm{T}_P(\mathrm{T}_P(\{\,\}))$, and continue the process until a fixed point is reached.

Ultimately, instantiation analysis of *append* produces the set

$$\{append\ ie\ ie\ ie, append\ ie\ nie\ nie, append\ nie\ ie\ nie, append\ nie\ nie\ nie\}$$

of instantiation patterns. Therefore, every literal in the success set of *append* must have one of these instantiation patterns.

The use of a different norm naturally results a different set of instantiation patterns. For example, the set $\{append\ ie\ ie\ ie, append\ ie\ nie\ nie\}$ is produced by using the list-size norm, instead. $\diamondsuit$

5.6.3 Mappings

The basic component of the termination test of (Lindenstrauss & Sagiv 1997) is a graph—historically known as a *mapping* (Sagiv 1991)—which is constructed for each clause (other than facts) in the program. A mapping is the abstract representation of a clause referred to in the motivating example of Section 5.6.1. Each node in a mapping corresponds to an argument of a literal in the clause. Arcs connect nodes and make explicit the size relationships between various arguments of the literals in the clause. In addition, a mapping identifies those arguments that are instantiated enough with respect to some norm. Essentially, mappings abstractly describe finite portions of the SLD-tree for a query. Termination analysis uses mappings to approximate the actual SLD-tree for a query and applies certain rules, which are formulated in this section, to the abstract tree to determine whether it is finite. The definitions contained in this section are standard from (Lindenstrauss & Sagiv 1997).

Definition 5.10 (Mappings). For a given norm and a clause C with a non-empty body, the *mapping* of C is a directed, weighted graph. A node in the graph corresponds to an argument position of a literal in C and is labelled with the corresponding argument. A node is coloured *black* if the label is instantiated enough with respect to the norm, or *white* otherwise. A zero-weight arc connects two nodes with equal symbolic norm. A weighted arc connects a pair of nodes such that the difference between their symbolic norms is a linear expression with (non-negative) non-positive coefficients and a (positive) negative constant term. ♢

Example 5.6. Consider again the *reverse* predicate whose definition is repeated below:

$$\begin{aligned}&\mathit{reverse}\ \mathit{nil}\ \mathit{nil}\ \ {:}-\ \ .\\&\mathit{reverse}\ (A:X)\ Y\ \ {:}-\ \ \mathit{reverse}\ X\ Z,\ \mathit{append}\ Z\ (A:\mathit{nil})\ Y.\end{aligned}$$

Using the term-size norm, the mapping for the second clause is

$$\begin{tikzcd}
\mathit{reverse} & \circ A{:}X \arrow[d, bend left, "2+A"] & \circ Y \arrow[drr, bend left, "0"] & \\
\mathit{reverse} & \circ X \arrow[u, bend left, "-(2+A)"] & \circ Z \arrow[dl, bend left, "0"] & \\
\mathit{append} & \circ Z \arrow[ur, bend left, "0"] & \circ A{:}\mathit{nil} & \circ Y \arrow[ull, bend left, "0"]
\end{tikzcd} \tag{5.2}$$

The head of the clause appears at the top of the mapping followed by successive literals in the body. Notice that there is no arc between the nodes labelled with $A : X$ and $A : \mathit{nil}$; their symbolic norms are respectively $2 + A + X$ and $2 + A$, and the difference between these two norms is X. The lack of a positive constant term in this norm indicates that we cannot be certain that the first term will always be strictly larger than the second, with respect to the term-size norm. For example, the variable X could eventually be bound to *nil* whose symbolic norm is 0. ♢

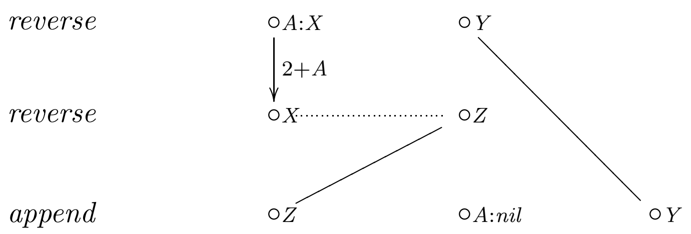
For brevity, we will often replace zero-weight arcs by unlabelled, undirected arcs and will omit negative-weight arcs, simply assuming their existence. Therefore, mapping (5.2) is equivalent to the mapping

$$\begin{tikzcd}
\mathit{reverse} & \circ A{:}X \arrow[d, "2+A"] & \circ Y \arrow[drr, dash] & \\
\mathit{reverse} & \circ X & \circ Z \arrow[dl, dash] & \\
\mathit{append} & \circ Z & \circ A{:}\mathit{nil} & \circ Y
\end{tikzcd} \tag{5.3}$$

which is easier on the eye than its counterpart.

Representing clauses as mappings captures concisely the relationships between the sizes of various arguments. Some of these relationships can be inferred simply from the symbolic norms of each node in a mapping but others can be derived in a more sophisticated fashion by scrutinising the clauses of a predicate. Indeed, the process of constraint analysis, detailed later in Section 5.6.4, provides such extra relationships, called *constraints*. A constraint between two nodes in a mapping is represented by an undirected arc if the nodes have the same symbolic norm, or by an arc of weight 1 if the symbolic norm of the first is greater than that of the second. Once constraints are known between nodes in a mapping, we can insert appropriate arcs to represent this knowledge. The following example illustrates this process.

Example 5.7. For the *reverse* predicate of Example 5.6, suppose we know that the length of the first list argument to *reverse* is equal to the length of its second list argument. We can represent this constraint by inserting an undirected arc (dotted, for emphasis) into mapping (5.3) between the arguments of the first subgoal of the clause as follows:



(5.4)

By doing so, we explicitly indicate that the variables X and Z have the same symbolic norm. ♢

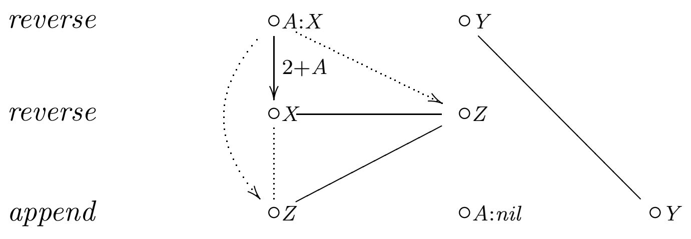
Constraint analysis provides crucial information about argument sizes in a clause without which proving termination of a literal may be impossible. Moreover, inserting constraints into a mapping often further elucidates the relationships between argument sizes since we may be able to infer more relationships from the extra information gleaned from the constraints. Technically, we can infer the existence of other arcs based on traversing *paths* in a mapping, which we define next.

Definition 5.11 (Paths). Let M be a mapping. A path between nodes u and v in M has *positive-weight* if the sum of the weights along the path is a linear expression in which the coefficients are non-negative and the constant term is positive. A path between nodes u and v in M has *zero-weight* if it contains no unlabelled arcs and the sum of the weights along the path is zero. ♢

Essentially, arcs are inferred in a similar vein to taking the transitive closure of a mapping but not identically since transitivity is predicated by the following rules.

Definition 5.12 (Inferred arcs). Let M be a mapping, and u and v be nodes in M. An undirected arc can be inferred between u and v if there is a zero-weight path between them. An unlabelled arc can be inferred from u to v if there is a positive-weight path between them. ♢

Example 5.8. For mapping (5.4), we can infer the new mapping



(5.5)

The inferred arcs are dotted for emphasis. ♢

The activity of inserting arcs into a mapping may introduce cycles into it. The existence of a positive-weight cycle in a mapping means that, for some norm and a node labelled with T, we have that $\|T\| < \|T\|$ which is impossible. Therefore, a mapping with a positive-weight cycle corresponds to a portion of the SLD-tree that cannot actually occur and we can disregard such mappings. We identify a mapping as *consistent* if it does not contain any positive-weight cycles.

Now that we have introduced the abstract representation of a clause, we introduce the notion of merging subgoals in a mapping. Merging two subgoals corresponds to their unification in the actual SLD-tree. The basic principle of merging two nodes in a mapping is that if either node is instantiated enough then both nodes become so after merging. The process of merging is defined below.

Definition 5.13 (Merging subgoals). Let P and Q be two subgoals in a mapping labelled with the same predicate. We can *merge* P and Q to form a new subgoal as follows: for each corresponding node in P and Q, we form a new node that is black if either of the two nodes are black, otherwise the new node is white. The arcs of the new subgoal are the union of the arcs between the nodes of P and Q. $\diamondsuit$

Merging subgoals essentially corresponds to the application of a substitution over literals in goal. Moreover, should a node be instantiated enough, we assume that all variables contained in the label of that node are instantiated enough, again in analogy with variables becoming bound in the actual SLD-tree. Therefore, once new nodes in a mapping become instantiated enough, other nodes in the mapping may consequently become instantiated enough; we can infer precisely which nodes are affected according to the following definition.

Definition 5.14 (Inferred instantiations). Let M be a mapping, and u and v be nodes in M. If u is black and there is a zero- or positive-weight path from u to v, then v is also black. A variable is instantiated enough if it appears in the label of a black node. Any node is black if all the variables in its symbolic norm are instantiated enough. $\diamondsuit$

In this section, we have introduced the abstract representation of a clause as a mapping and shown how the constraints between the sizes of arguments can be modelled as arcs in the mapping. In the following section, we show how mappings are used to infer constraints between the arguments of a predicate. This process of *constraint inference* is a precursor to the termination test and provides vital information that allows us to prove the termination of many literals.

5.6.4 Constraint analysis

Constraint analysis infers relationships between the size of arguments in a predicate, with respect to a norm, by examining a logic program in a bottom-up fashion. In this section, we review the process of constraint analysis introduced by (Lindenstrauss & Sagiv 1997). A constraint inferred by this process corresponds to an arc between arguments in the head literal of a mapping. Thus, we have *equality* constraints, corresponding to undirected arcs in the mapping, and *monotonicity* constraints, corresponding to directed arcs. Alternative forms of constraint analysis have been considered in the past (Codish & Taboch 1997, Sagiv 1991, Ullman & Gelder 1988) that use *linear inequalities* which can infer constraints that cannot be inferred by the method reviewed here. Conversely, the analysis in this section can infer constraints that cannot be inferred by linear inequalities.

The process of constraint analysis is optional in termination analysis but, without it, the proof of termination for many literals becomes impossible. A constraint for our purposes is as follows.

Definition 5.15 (Constraint pattern). A *constraint pattern*, or simply *constraint*, is the restriction of a mapping to the nodes of only one subgoal. Each node is white and the arcs are restricted to only those between the nodes of the subgoal. $\diamondsuit$

An example constraint for the predicate *append* is

$$\textit{append} \qquad\qquad \circ \qquad\qquad \circ \rule[0.5ex]{4em}{0.4pt} \circ \tag{5.6}$$

which indicates that the second and third arguments to *append* have equal size. Since a constraint is a subgoal, a constraint can be merged with another subgoal according to Definition 5.13.

The algorithm we review for constraint inference, which is due to (Lindenstrauss & Sagiv 1997), employs an abstract interpretation of programs similar to that used in the instantiation analysis of Section 5.6.2. This time, however, the abstraction function α takes a term $T \in B$ to the constraint pattern that can be inferred for T according to some norm. For example, using the term-size norm, α maps the literal *append nil* Y Y to the constraint (5.6) since the norms of the last two arguments are equal. We attempt to infer a new constraint for the predicate in the head of a mapping, using the method defined below.

Definition 5.16 (Constraint inference). Let $C = A :- B_1, \ldots, B_n$, for $n \geq 1$ be a clause. We derive a new constraint for the predicate of A as follows:

1. Let M be the mapping of C.
2. For each B_i, where $1 \leq i \leq n$, select a constraint inferred already for B_i. If none exists, C cannot be used to infer a new constraint. Otherwise, merge the constraint and the subgoal B_i in M according to Definition 5.13.
3. Infer all arcs according to Definition 5.12.

If M is consistent then the constraint pattern for A is inferred as a new constraint for the predicate of A. $\diamondsuit$

Suppose C is the set of all constraints and P is a program. Then the effect of a single inference step can be described as an immediate inference operator $\mathrm{I}_P : \mathsf{P}C \rightarrow \mathsf{P}C$ as follows: for a set of constraints $S \subseteq C$, the set $\mathrm{I}_P(S)$ contains those constraints inferred according to Definition 5.16 for the predicate of each constraint in S. Since I_P is both monotonic and continuous, and the set of abstract clauses is finite, the least fixed point of I_P can be computed iteratively in a finite number of steps. The Fixed-Point Abstraction Theorem gives us that, for each literal A that is a logical consequence of P, A satisfies at least one constraint in the fixed-point of I_P. That is, $\alpha(\mathrm{lfp}(\mathrm{T}_P)) \subseteq \mathrm{lfp}(\mathrm{I}_P)$ where T_P is the immediate consequence operator of Section 5.6.2 and 'lfp' is the least fixed-point operator. The details of fixed-point abstraction in constraint analysis are given by (Lindenstrauss & Sagiv 1996). The following example illustrates the inference of constraints for a program.

Example 5.9. Consider the inference of constraint patterns for the *append* predicate. We have that $\mathrm{I}_P(\{\,\})$ comprises the single constraint (5.6), obtained from the first (fact) clause. For the next ordinal of I_P, we take the basic mapping for the second clause of *append* and follow the process described in Definition 5.16. We end up with the following mapping, where the solid arcs are from the basic mapping and the dotted arcs are the inferred arcs.

$$\begin{tikzcd}
\textit{append} & \circ A{:}X \arrow[d, "2+A"] & \circ Y & \circ A{:}Z \arrow[l, dotted] \arrow[dl, dotted] \arrow[d, "2+A"] \\
\textit{append} & \circ X & \circ Y \arrow[u, dash] \arrow[r, dotted, dash] & \circ Z
\end{tikzcd}$$

Thus, we are able to derive a single additional constraint for *append*, namely

$$\begin{tikzcd}
\textit{append} & \circ & \circ & \circ \arrow[l]
\end{tikzcd}$$

which indicates that the third argument is larger than the second. ♢

The constraints inferred by the process described in this section are vital for the success of termination analysis. In the following section, we describe the termination analysis of (Lindenstrauss & Sagiv 1997) and its novel adaptation to test termination of programs using prioritised resolution (Definition 5.9).

5.6.5 The termination test

In this section, we adapt the termination test of (Lindenstrauss & Sagiv 1997) to use the computation rule of prioritised resolution, described in Definition 5.9, rather than the left-to-right one. The alterations to the existing test are relatively straightforward, requiring only the integration of the notion of a call set and the novel computation rule of prioritised resolution into the method of generating query-mapping pairs (Definition 5.20). Nevertheless, the extended termination test equips us with the power to implement the novel algorithm, presented earlier in Section 5.5, which automatically generates a call set of a program for use with prioritised resolution. The application of termination analysis in this way to improve the execution efficiency of resolution is unique to this thesis.

Termination analysis constructs an abstract version of the actual SLD-tree and tests whether every recursive call in the tree is finite. The crucial point of termination analysis is that a recursive call will be finite if certain arguments of it decrease in size: this observation is formalised and proved later in Theorem 5.1, which is originally due to (Sagiv 1991). We construct the abstract tree by mimicking the actual resolutions made in the SLD-trees. The crucial components in the abstract version are the selected literal and the relevant program clause. Therefore, an abstract resolution step is characterised by the abstract forms of both of these components. The abstract form of the program clause is a mapping (Definition 5.10) and a 'query pattern', defined next.

Definition 5.17 (Query pattern). A *query pattern* is the restriction of a mapping to the nodes of only one subgoal. The arcs of the query pattern are restricted to those between the nodes of the subgoal. ♢

Two query patterns are *identical* if they are labelled by the same predicate, corresponding nodes are the same colour and also have the same arcs between them. An instantiation pattern is obtained from a query pattern in the obvious manner: an argument in the instantiation pattern is *ie* if its counterpart in the query pattern is black, otherwise the argument is *nie*.

Let us how see how an actual resolution step is modelled abstractly using mappings. Consider a prioritised resolution step with a callable literal and a suitable program clause. The selected literal in the subsequent step is one of those introduced by the previous step, i.e., a subgoal in the body of the program clause used in the previous resolution. In order to construct an abstract portion of a derivation, we restrict our attention in an abstract resolution step to the head of a mapping and the subgoal in its body corresponding to the next selected literal. We refer to this restriction of a mapping as its 'summary', formalised in the next definition.

Definition 5.18 (Summary mapping). Let M be a consistent mapping of some program clause C. A *summary mapping* of M is a mapping that comprises only the following: the nodes corresponding to the head of C, called the *domain*; the nodes corresponding to one subgoal of C, called the *range*; and the arcs between these nodes in M. Moreover, we delete the label from each node, the weight from each non-zero arc, and also delete all non-zero arcs that connect nodes other than black ones. ♢

Example 5.10. The summary mapping of (5.5), with its range being the second subgoal in its body, is

$$\begin{tikzcd}
\textit{reverse} & \circ \arrow[d] & \circ \arrow[dr, dash] & \\
\textit{append} & \circ & \circ & \circ
\end{tikzcd} \tag{5.7}$$

The query pattern for the range of this summary is

$$\textit{append} \qquad \circ \qquad \circ \qquad \circ \tag{5.8}$$

The query pattern is bereft of arcs since no arcs appear in (5.7) between the nodes of *append*. ♢

A summary mapping isolates successive selected literals in a pair of resolution steps; the domain of the summary is the selected literal of the first resolution step and the range is the selected literal in the following step. The literal in the range, as in the actual SLD-resolution, is then 'unified' with an appropriate mapping. The repetition of this process allows us to generate an abstract version of the actual SLD-tree.

From a summary mapping, we define 'query-mapping pairs' that represent finite portions of actual SLD-derivations. Using a technique of 'composing' query-mapping pairs, the termination test generates all possible combinations of pairs that can be formed by the computation rule, and tests all recursive pairs for a strict decrease in size. Query-mapping pairs are defined as follows.

Definition 5.19 (Query-mapping pairs). A *query-mapping pair* (P, H) consists of a query pattern P and a summary mapping H such that the predicate labelling P is identical to the predicate labelling the domain of H. ♢

The range of a query-mapping pair, interpreted as a query pattern, is used to construct subsequent mappings. From these mappings, new query-mapping pairs are generated according to the computation rule of the resolution strategy; for example, (Lindenstrauss & Sagiv 1997) use the left-to-right computation rule. However, we are instead interested in generating query-mapping pairs for the computation rule of prioritised resolution.

So, given a query pattern for some predicate, we create a mapping for it using one of the predicate's program clauses. The head of the mapping and the query pattern are then merged and a constraint is added for each previously selected subgoal in the mapping, which are recorded in a sequence that makes explicit the prior order of selection. The key point of the sequence is that each previously selected literal in it is assumed to terminate and, therefore, to be a logical consequence of the program. As a result, we can merge with each such literal a subsuming instantiation pattern, created during instantiation analysis, as well as a constraint, created during constraint analysis. All extra arcs and bindings are then inferred. The final step is to select from the remaining subgoals in the clause the leftmost one whose binding pattern is contained in the call set. This process simulates the actual computation rule of prioritised resolution. A query-mapping pair is then created using the original query pattern, with the summary formed from the head of the mapping and the selected subgoal.

In order to use the computation rule of prioritised resolution in the termination test, we augment the original generation procedure presented by (Lindenstrauss & Sagiv 1997) with the notion of a call set and also a sequence that records the previously selected literals—the ones already known to terminate—in the mapping which results in the following algorithm.

Definition 5.20 (Generating query-mapping pairs). Let $C = A$:− $B_1, \ldots, B_m$, for $m \geq 1$, be a clause and Q a query pattern for the predicate of A. Furthermore, let $\mathbb{C}$ be a call set and S some (possibly empty) sequence of unique indices from $\{1, \ldots, m\}$. A query-mapping pair is created as follows:

1. Let M be the mapping for C.
2. Merge Q and A according to Definition 5.13.
3. For each $i \in S$, in turn:
   (a) select a constraint for the predicate of B_i and merge it with B_i in M according to Definition 5.13;
   (b) infer new arcs according to Definition 5.12;
   (c) select a subsuming instantiation pattern for B_i and blacken the nodes of B_i accordingly; and
   (d) infer new black nodes according to Definition 5.14.
4. Let B_j be the first subgoal in M such that $1 \leq j \leq m \wedge j \notin S$ and the instantiation pattern of B_j is in $\mathbb{C}$. Add j to the end of S.

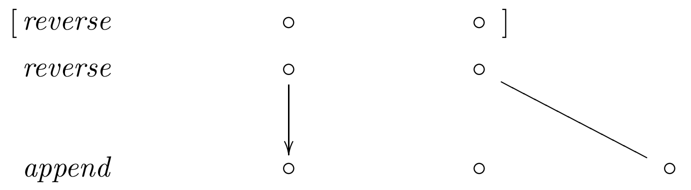
A query-mapping pair is then generated with the query pattern Q and the summary of M having B_j as its range. ♢

The important aspect of the method above is the sequence of indices that corresponds to the order in which literals are selected for resolution in the actual SLD-tree. Once a new query-mapping pair is created using the method of Definition 5.20, the query pattern of its range is used to construct more pairs in the same way. Through a process of *composition* of query-mapping pairs, defined shortly, the exploration of every branch in the actual SLD-tree is simulated. It is this process of composing all query-mapping pairs that determines whether a subgoal terminates. If composition implies that a subgoal does indeed terminate, then its index is added to the sequence of previously selected subgoals and the whole procedure repeats.

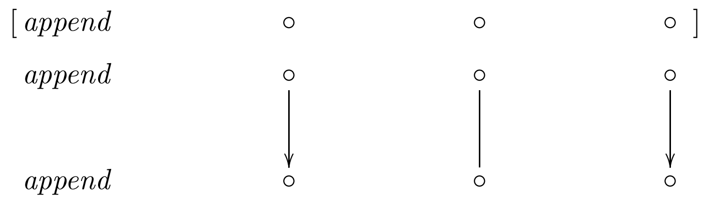
Thus, a generated query-mapping pair is composed with all appropriate pairs already discovered, and the composition of two pairs summarises that portion of a derivation in the actual SLD-tree by producing a new pair whose domain and range correspond to the first and last literals in the derivation. The essential importance of composition is that it elucidates the size relationships between the arguments of the first and last literals. We define the composition of query-mapping pairs next.

Definition 5.21 (Composition of query-mapping pairs). Let $S = (P, F)$ and $T = (Q, H)$ be query-mapping pairs. If the range of F and Q are identical, then the *composition* of S and T is a new query-mapping pair (P, K) obtained as follows. A new mapping K' is formed from F and H by merging the range of F with the domain of H, according to Definition 5.13. Finally, K is the summary mapping of K'. ♢

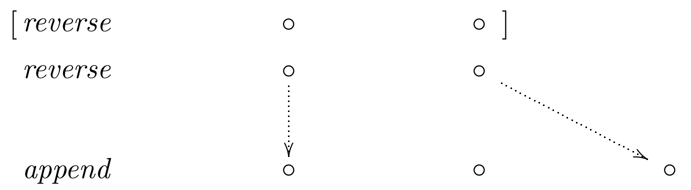
Example 5.11. In diagrams, the query pattern of a query-mapping pair is enclosed in parentheses. The composition of the query-mapping pair



and



is the new query-mapping pair



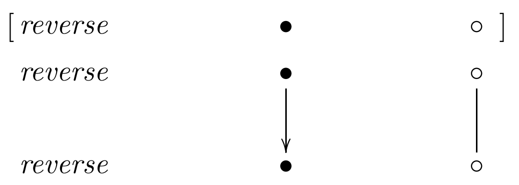
(5.9)

The dotted arcs in (5.9) follow from the composition but are deleted from the query-mapping pair as they do not connect black nodes (Definition 5.18). ♢

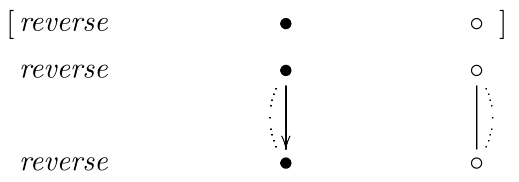
As mentioned earlier, the crux of the termination test is the generation of all possible query-mapping pairs for a query pattern. The composition of these pairs corresponds to the exploration of every branch in the actual SLD-tree. This process ends with pairs that have an identical range and domain which correspond to recursive calls in the actual derivation. For each recursive pair, we must ensure that the call in the range is strictly smaller than the call in the domain. We do so by creating a 'circular variant' for each recursive pair which indicates whether any of its arguments have decreased in size.

Definition 5.22 (Circular variant). Let $S = (P, H)$ be a query-mapping pair such that P and the query pattern of the range of H are identical. Then S has a *circular variant* (P, H') where H' is obtained from H by inserting a distinguished zero-weight arc, referred to as a *circular arc*, between corresponding nodes in the range and domain of H. ♢

Example 5.12. Given the query-mapping pair



the circular variant of this pair is



where the circular edges are dotted. ♢

The main technical component of the termination test is to determine that, for each circular variant derived for the program, it contains at least one forward positive cycle, the existence of which indicates a decrease in the size of the recursive call.

Definition 5.23 (Forward positive cycle). Let $S = (P, H)$ be the circular variant of some query-mapping pair. Then S has a *forward positive cycle* if H has a positive cycle such that each circular edge in the cycle is traversed from a node in the range to a node in the domain of H. ♢

The observation that each circular variant should possess at least one forward positive cycle corresponds to the SLD-tree having no infinite branches. The following theorem, due to (Sagiv 1991), establishes this result.

Theorem 5.1. *Let P be a program, H a goal, and S the set of all query-mapping pairs associated with the SLD-tree of $P \cup \{G\}$. If all circular variants of the query-mapping pairs in S have a forward positive cycle, then the SLD-tree for $P \cup \{G\}$ is finite.*

Proof. Suppose that every circular variant has a forward positive cycle, but there is an infinite branch in the SLD-tree for $P \cup \{G\}$. Then there must be an infinite sequence of selected literals $A_1, A_2, \ldots, A_i, \ldots$, along this infinite branch. For each $i \geq 1$, the head of some program clause is unified with A_i, and A_{i+1} is one of the subgoals of the clause (guaranteed by the computation rule). Let M_i be the mapping formed with A_i as its head. Then there is an infinite sequence of mappings $M_1, M_2, \ldots, M_i, \ldots$, corresponding to the infinite sequence of selected literals.

We can define an infinite sequence of query-mapping pairs $T_1, T_2, \ldots, T_i, \ldots$, as follows. For $i = 1$, $T_1 = (Q_1, H_1)$, where Q_1 is the initial query pattern and H_1 is the summary of M_1. For $i > 1$, $S_i = (Q_i, H_i)$ where Q_i is the query pattern of the range of M_{i-1} and H_i is the summary of M_i. Observe that each $T_i \in S$ and, for each $j > i$, the composition of the query-mapping pairs $T_i, T_{i+1}, \ldots, T_j$ is also in S. Now, since S is finite—any program can only have a finite number of query-mapping pairs—at least one query-mapping pair in $T_1, T_2, \ldots, T_i, \ldots$, call it T, must appear an infinite number of times in sequence. T is clearly a recursive query-mapping pair and, moreover, T must have a circular variant that contains a forward positive cycle. In this infinite composition of T, there exists a positive cycle showing that at least one argument of T must strictly decrease at each step. By induction, there are infinitely many subgoals in the composition of T, such that as we go from one to the next, the symbolic norm of at least one argument position become strictly smaller. This contradicts the well-foundeded property of the ordering on argument size, dictating that such an infinite composition cannot exist. This contradiction proves the theorem. $\diamondsuit$

At this point, all the machinery necessary to describe termination analysis for arbitrary literals has been introduced. An implementation of termination analysis can be found in Appendix C which provides a detailed view of all the important algorithms of this section. In the next section, we draw together the material of the current section by presenting a rigorous application of termination analysis to the motivating example given at the beginning of the chapter.

5.6.6 A formal example of termination analysis

Termination analysis attempts to determine whether a particular instantiation pattern of a predicate, with respect to some norm, terminates. In this section, the example of Section 5.6 is revisited, giving a formal termination analysis of the query

$$\# :- \ \textit{sublist}\ X\ (a : b : c : \textit{nil}).$$

We use the term-size norm and prioritised fair SLD-resolution in the example. The instantiation pattern of this query, with respect to the term-size norm, is *sublist nie ie* which corresponds to the query pattern *sublist* $\circ$ $\bullet$.

Termination analysis proceeds by approximating the SLD-tree of Figure 5.2 for the query pattern. Earlier, we noted that constraint analysis is an optional component of termination analysis and, for simplicity, let us ignore the use of constraints in the current example. The first resolution step involves the construction of the mapping for the query pattern and the subsequent generation of a query-mapping pair according to Definition 5.20. The following mapping is constructed by merging the query pattern with the head of the mapping for *sublist* and inferring new bindings, according to Definition 5.14:

$$\begin{tikzcd}
\textit{sublist} & \circ X \arrow[dr, no head] & \bullet Y \arrow[ddr, no head] & \\
\textit{append} & \circ U & \circ X & \circ V \arrow[dll, no head] \\
\textit{append} & \circ V & \circ W & \bullet Y
\end{tikzcd} \tag{5.10}$$

To construct the query-mapping pair, the subgoal in the body of (5.10) selected by prioritised resolution must be identified. To do so, a call set of instantiation patterns is necessary.

Therefore, it is advantageous to perform termination analysis following the topological ordering of the predicate dependency graph of the program. By doing so, each predicate in the body of a clause will have been previously analysed (other than those in the same strongly connected component of the dependency graph). In the current example, the call set will contain the instantation pattern *append nie nie ie* since it terminates. Let us demonstrate why this is so. The query pattern corresponding to the instantiation pattern is

$$\textit{append} \qquad \circ \qquad \circ \qquad \bullet \tag{5.11}$$

Again, the first task is to construct the mapping for *append* and merge it with (5.11):

$$\begin{tikzcd}
\textit{append} & \circ A{:}X \arrow[d, "2+A"] & \circ Y \arrow[d, no head] & \bullet A{:}Z \arrow[d, "2+A"] \\
\textit{append} & \circ X & \circ Y & \bullet Z
\end{tikzcd}$$

Since there is only one subgoal in the body of the mapping, the only query-mapping pair is the following one:

$$\begin{tikzcd}
{[}\,\textit{append} & \circ & \circ & \bullet\ {]} \\
\textit{append} & \circ & \circ \arrow[d, no head] & \bullet \arrow[d] \\
\textit{append} & \circ & \circ & \bullet
\end{tikzcd} \tag{5.12}$$

For each new query-mapping pair constructed, its circular variant (if it exists) is checked for at least one forward positive cycle (Definition 5.22). The pair (5.12) does indeed have a circular variant that contains a forward positive cycle. Query-mapping pairs with a circular variant are terminal in the generation of other pairs and so the pair (5.12) is sufficient to determine that all query patterns (5.11) terminate with respect to the term-size norm.

So, mapping (5.10) is revisited with the knowledge that its second subgoal terminates. At this point, the results of instantiation analysis are used to decide which arguments of the second subgoal have become instantiated enough after the successful resolution. The applicable instantiation patterns are those of (5.5), from Section 5.6.2. Since the third argument of the second subgoal is instantiated, the only subsuming instantiation pattern (Definition 5.6) is *append ie ie ie*. The arguments of the second subgoal in (5.10) are instantatiated appropriately and, after inferring new bindings, results in the mapping:

$$\begin{tikzcd}
\textit{sublist} & \circ X \arrow[dr, dash] & \bullet Y \arrow[ddr, dash] & \\
\textit{append} & \circ U & \circ X & \bullet V \arrow[dll, dash] \\
\textit{append} & \bullet V & \bullet W & \bullet Y
\end{tikzcd}$$

This time, the first subgoal must be selected for resolution since it is the only unconsidered one. The instantiation pattern for this subgoal is the same as before which is known to terminate. Therefore, termination is shown for the initial query using prioritised resolution with respect to the term-size norm.

5.7 Higher-order predicates in termination analysis

Higher-order predicates are a pervasive component of compositional programs. However, the original termination analysis introduced by (Lindenstrauss & Sagiv 1997) ignored the possibility of higher-order predicates appearing in programs. In this section, we show how to extend that termination analysis to cope with higher-order predicates; the modification is conceptually straightforward although it increases the complexity of an actual implementation of prioritised resolution. (An implementation of prioritised resolution that includes termination analysis for programs that may contain higher-order predicates can be obtained from the author.)

Let us illustrate the main difficulties of dealing with higher-order predicates in termination analysis using the following example. Consider the following parsing combinators from Section 1.1:

$$\begin{aligned}
&\textit{eq}\ A\ A\ :-\ .\\
&\textit{satisfy}\ P\ (A : X)\ V\ Y\ :-\ P\ A,\ \textit{succeed}\ A\ X\ V\ Y\ .\\
&\textit{literal}\ A\ X\ V\ Y\ :-\ \textit{satisfy}\ (\textit{eq}\ A)\ X\ V\ Y\ .
\end{aligned}$$

The clause for *satisfy* has the corresponding mapping

$$\begin{tikzcd}
\textit{satisfy} & \circ P & \circ A{:}X \arrow[dl, "2+X"'] \arrow[ddl, "2+X"] \arrow[dd, "2+A"] & \circ V \arrow[dd, dash] & \circ Y \arrow[dd, dash] \\
P & \circ A \arrow[d, dash] & & & \\
\textit{succeed} & \circ A & \circ X & \circ V & \circ Y
\end{tikzcd} \tag{5.13}$$

The variable P in mapping (5.13) occurs as the 'name' of a predicate and, in order to determine whether calls to *satisfy* terminate, the actual binding of P must be known. Consequently, termination analysis can only be performed on the unique 'first-order instance' of a higher-order predicate, i.e., where each higher-order variable is bound to a first-order term. For example, in the *literal* parser above, suppose we wish to discover whether the query pattern *literal* • • ◦ ◦ terminates. The analysis of this pattern relies on the termination of *satisfy* (*eq* •) • ◦ ◦ which, in turn, requires termination of the goal # :− *eq* • •, *succeed* • • ◦ ◦, that consists solely of first-order predicates. Thus, the termination analysis of *literal* is possible because the instance of *satisfy* in the body of the clause for *literal* is a first-order instance.

The implication of the above result on the programming language is that queries may not contain unbound higher-order variables. So far, however, the inability to evaluate such queries has not yet proved to be a significant restriction to the style of compositional programming presented in this thesis; for example, the programs presented throughout the thesis have made no use of such a facility. Nevertheless, it may be the case that the ability to determine the values of higher-order variables could prove to be valuable in some alternative notion of compositionality than the one advocated in this thesis. Of course, any such compositional language would need to adopt higher-order unification in its evaluation mechanism, as is the case with λProlog (Nadathur & Miller 1988). In such a case, though, it is difficult to see how termination analysis could be adapted to operate on goals that contain unbound higher-order variables. Integrating higher-order unification and prioritised resolution could, therefore, present an interesting area of future work; we remark further on this point in Section 7.2.

The implementation of prioritised resolution presented in Appendix C ignores the possibility of programs containing higher-order predicates, solely for reasons of simplicity. Fortunately, extending the implementation to cope with higher-order predicates, mainly affects the construction of the program's predicate dependency graph. In effect, a top-down process of partial evaluation is necessary to instantiate each higher-order variable in the first-order instance of a higher-order predicate. The main complication in the partial evaluation is determining for each predicate which of its arguments are higher-order and, therefore, require instantiation. A primitive typing mechanism, hidden from the programmer in the internals of the language, is sufficient for this purpose. Alternatively, a strong type system could be adopted for the language. (The interpreter referred to at the beginning of the section follows the latter approach.)

5.8 The limitations of prioritised fair SLD-resolution

The constraint analysis of Section 5.6.4 infers equality and monotonicity constraints for a program in a bottom-up style, and is implemented in this way in the interpreter of Appendix C. However, this approach is somewhat primitive in comparison to recent constraint analyses which instead solve linear inequalities between argument sizes of literals in a clause (Codish & Taboch 1997, Speirs et al. 1997). These new techniques are often able to infer a more comprehensive set of constraints for a program than the one of Section 5.6.4. Consequently, without the knowledge of these extra constraints, the termination test of Section 5.6 can find the termination proof of some literals impossible. Fortunately, the results of these new constraint analysis can be expressed straightforwardly in our termination test and we discuss how to do so in this section.

The constraint analysis of (Codish & Taboch 1997) is implemented using a standard constraint-solving library of a particular dialect of Prolog, called SICStus Prolog (SICS 1997). The use of this library simplifies the implementation and improves the efficiency of their termination analysis which evolved from the test of (Lindenstrauss & Sagiv 1997).

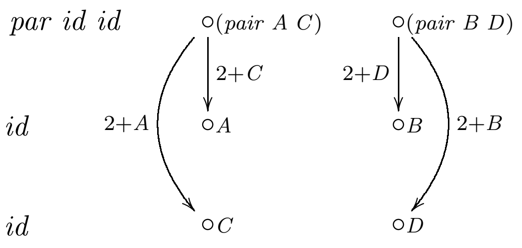
In order to illustrate the deficiencies of the bottom-up constraint analysis of Section 5.6.4, let us consider the following elementary Ruby program, familiar from Section 1.2:

id A A :– .
par R S (*pair A C*) (*pair B D*) :– *R A B*, *S C D*.

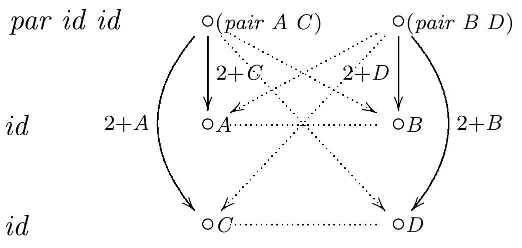
Consider determining the constraints for *par id id* using the method of Definition 5.16 and the term-size norm. Constraint analysis infers the following sole constraint for *id*:

$$\begin{tikzcd} id & \circ \arrow[r, dash] & \circ \end{tikzcd} \qquad (5.14)$$

To infer constraints for *par id id*, the mapping corresponding to its program clause is first constructed as follows:



Then, each occurrence of *id* in the body of the mapping above is merged with the constraint (5.14). The next step, according to Definition 5.16, is to infer all possible arcs which results in the following mapping where the inserted and inferred arcs are dotted for emphasis.



(5.15)

As there is no constraint in the head of the mapping, no constraint can be inferred for *par id id*.

An arc cannot be inferred between the arguments of the head since the path between the nodes labelled (*pair A C*), *A*, *B*, and (*pair B D*) has weight $(2 + C) + 0 - (2 + D) = C - D$. Since $C - D$ does not have all non-negative coefficients, a constraint cannot be inferred. However, the constraint between the second *id* subgoal in (5.15) explicitly states that $C = D$. Using this information would allow the weight of the path above to be simplified to 0 and permit the inference of an equality constraint between the arguments in the head of the mapping. On the other hand, the constraint analysis of (Codish & Taboch 1997) is able to infer constraints of this nature and, as we shall see in the following chapter, the inability to infer constraints like the one above can result in poor efficiency behaviour of prioritised resolution. The actual integration of alternative constraint analyses into our termination test has yet to be conducted and forms a desirable area of future work, as discussed in Section 7.2.

5.9 Summary

In this chapter, we introduced prioritised fair SLD-resolution to recover the desirable property of fair SLD-resolution, lost by fair OLDT-resolution, which ensures the termination of a query whenever any other resolution strategy would terminate for it. Furthermore, prioritised resolution aimed to improve the efficiency of executing compositional programs over that of fair SLD-resolution by prioritising the selection of literals in a goal according to whether their resolution terminates. Prioritised resolution also offers advantages over coroutining computation rules: firstly, the construction of the call set for a program in prioritised resolution is less heuristically based; and, secondly, prioritised resolution permits the prediction of whether a particular query will execute efficiently based on whether it can be proved to terminate.

We reviewed the definition of norm functions which we subsequently used to construct the members of the call set for prioritised resolution, namely instantiation patterns. We then formalised the novel computation strategy of prioritised fair SLD-resolution, followed by the presentation of an algorithm which automatically generates the call set for the prioritised resolution of a program. A pivotal component of this algorithm is automatic termination analysis which we use to determine whether the resolution of a literal terminates. We reviewed an existing termination analysis (Lindenstrauss & Sagiv 1997) and extended it to determine termination of programs with respect to the computation rule of prioritised resolution, rather than the left-to-right one. Furthermore, we extended that termination analysis to test termination for programs that may contain occurrences of higher-order predicates, which was neglected in the original presentation by (Lindenstrauss & Sagiv 1997).

The termination test presented in this chapter, however, contains a limitation that affects the performance of prioritised resolution when executing some compositional programs. The problem is caused by the particular incarnation of constraint analysis used since it is unable to infer all the constraints necessary to prove the termination of key predicates. However, alternative constraint analyses have recently been developed (Codish & Taboch 1997, Speirs et al. 1997) which are able to infer the missing constraints. The termination test of this chapter is amenable for use with these new constraint analyses and their integration would result in better performance of prioritised resolution.

Chapter 6

Evaluation of Compositional Logic Languages

This thesis has proposed a style of compositional logic programming based upon a higher-order syntax and the fair selection of subgoals in the resolution process. Furthermore, several prototypical implementations for such a language, written in Caml-light, are presented in Appendices A, B, and C. In this chapter, the performance of these implementations is evaluated for the execution of several example programs. The objective of the comparison is to identify the extent to which each interpreter satisfies the desiderata for a compositional language, as set out in Chapter 2; in particular, we aim to investigate the efficiency of each interpreter in terms of the number of resolutions steps taken to complete the execution of each program.

Naturally, the implementations presented in the appendices are far from optimal in terms of efficiency of execution: their primary roles are to elucidate algorithms and suggest appropriate data structures. Therefore, comparisons of the execution of the interpreters based on processor time consumed are somewhat unhelpful; rather, a comparison based on the number of resolution steps performed by each interpreter is favoured. Nevertheless, each interpreter does incur run-time overheads beyond those of any Prolog implementation and these are acknowledged in the cases where they become prohibitive to the execution of a query.

The four interpreters evaluated in this chapter vary only in the resolution strategy they adopt. The strategies, and their appropriate definitions elsewhere in the thesis, are as follows: SLD-resolution, i.e., Prolog, (Definition 2.7); fair SLD-resolution using the naive breadth-first computation rule (Definition 3.4); fair OLDT-resolution (Definition 4.5); and prioritised fair SLD-resolution (Definition 5.9). A graph of performance that charts the number of resolutions to the size of the input is provided for each example program.

Six programs—some of which are commonly used as performance benchmarks in the logic programming community—are featured in the comparison. The initial three programs considered are not explicitly compositional in nature but are concerned with compositionality since the order of evaluation of the literals in the body of their clauses is crucial for efficient computation. The final three programs are simple compositional ones that serve to illustrate the complexity of executing this style of program.

6.1 Reversing lists

The task of reversing a list, using the standard 'naive' algorithm, is a classic benchmark for logic programs in the literature. The program for reversing a list is repeated below for convenience.

$$\begin{aligned}&\textit{reverse nil nil} \;:\!-\; .\\&\textit{reverse}\ (A : X)\ Y \;:\!-\; \textit{reverse}\ X\ Z,\ \textit{append}\ Z\ (A : \textit{nil})\ Y.\end{aligned}$$

Figure 6.1 depicts the graph of performance of the four interpreters for calls to *reverse* that have their first argument bound to a ground list and their second argument free. The graph charts the number of resolution steps taken by each interpreter against the length of the list.

In the case of Prolog, we can see from the graph that, for lists of length n, calls to *reverse* perform $O(n^2)$ resolution steps. Furthermore, notice that the graphs for Prolog and prioritised resolution coincide exactly. The reason for this is that termination analysis succeeds for calls to *reverse* that have their first argument instantiated enough with respect to the term-size norm. Thus, prioritised resolution is able to evaluate such queries without making any fair resolution steps, as described in the previous chapter. By coincidence, the order of literals selected in the prioritised resolution is the same as in Prolog. On the other hand, the same calls to *reverse* using fair SLD-resolution with the breadth-first computation rule perform $O(n^4)$ resolutions. It appears from the graph that fair OLDT-resolution takes a constant factor of ten times as many resolutions as Prolog.

Consider, this time, calls to *reverse* with the argument bindings swapped; the performance of the interpreters is shown in Figure 6.2. The order of literals in the body of the second clause for *reverse* forces Prolog to loop infinitely and is consequently not represented in the graph. In this reversed direction of executing *reverse*, fair SLD-resolution takes a number of resolution steps proportional to the cube of the length of its second list. Although prioritised resolution is unable to determine that calls of the current nature terminate, the strategy nevertheless executes more efficiently than any of the other strategies. The reason for this is that the call set for prioritised resolution contains all the terminating instantiation patterns of *append*. So, each fair resolution step of *reverse* introduces a terminating call to *append* which is subsequently selected and, thus, restores efficient computation. OLDT-resolution performs only a constant factor of two worse than prioritised resolution.

6.2 The frontier of a binary tree

The predicate *frontier*, considered in Example 3.3, relates a binary tree to its frontier by way of a post-order traversal:

$$\begin{aligned}&\textit{frontier}\ (\textit{tip}\ A)\ (A : \textit{nil}) \;:\!-\; .\\&\textit{frontier}\ (\textit{bin}\ S\ T)\ Z \;:\!-\\&\quad \textit{frontier}\ S\ (A : X),\ \textit{frontier}\ T\ (B : Y),\ \textit{append}\ (A : X)\ (B : Y)\ Z.\end{aligned}$$

In that example, we saw that fair SLD-resolution constructed an impractically large search tree for literals of *frontier* whose first argument was free and whose second argument was

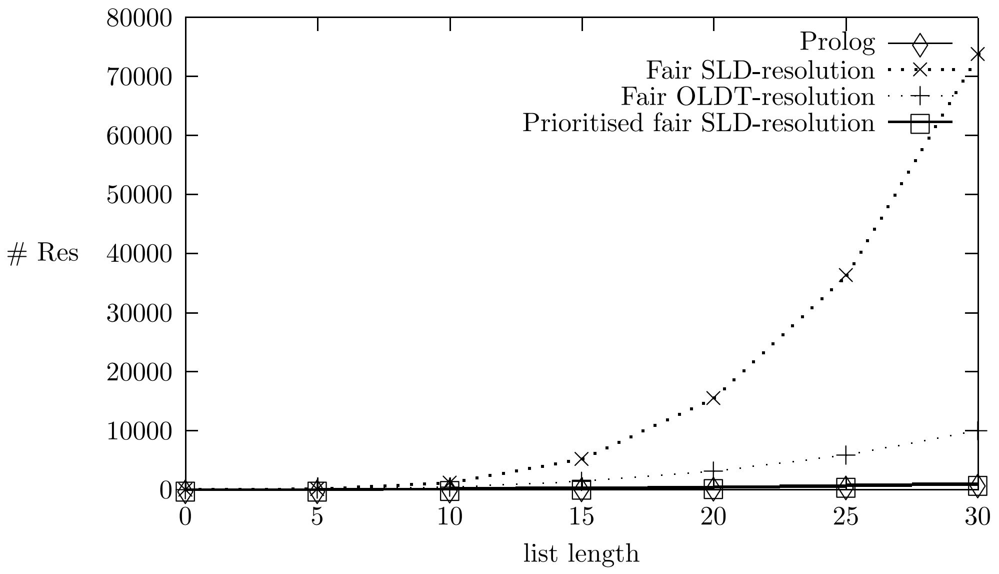


Figure 6.1: The execution of *reverse* with the first argument bound to a ground list and the second argument free.

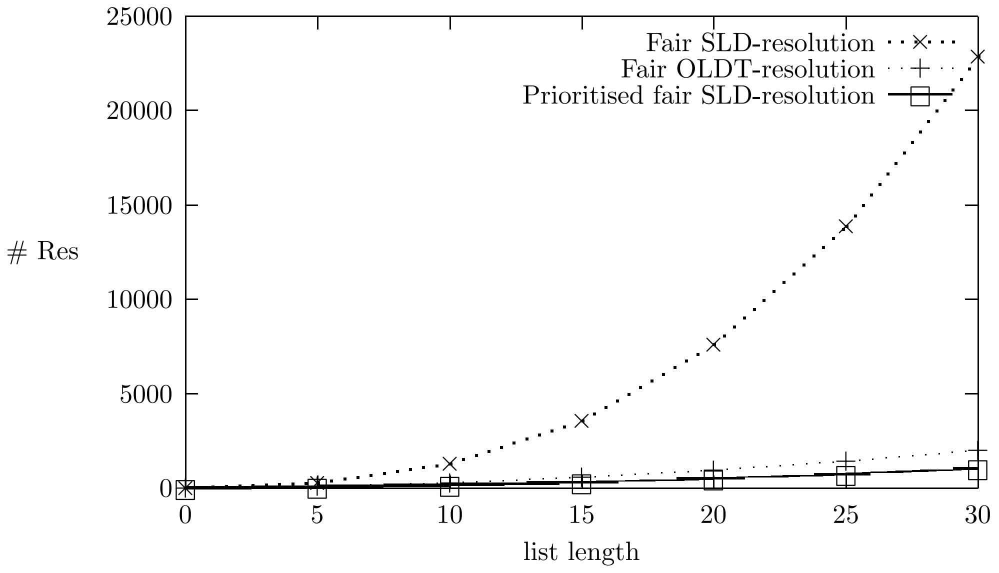


Figure 6.2: The execution of *reverse* with the first argument free and the second argument bound to a ground list.

bound. The redundant resolution of many instances of the same literal were responsible for this poor behaviour. In this section, we revisit the original example to identify how the other strategies deal with this situation.

The execution of calls to *frontier* with only the second argument bound to a ground term is illustrated in Figure 6.3. Fair SLD-resolution enumerates the first few binary trees before generating such a large search space that the computation is rendered infeasible in practical terms. The execution of Prolog and prioritised resolution are, again, identical even though calls of the current form are not contained in the call set of prioritised resolution. Given a frontier list of length $n \geq 1$, the Catalan numbers (Hilton & Pedersen 1991) predict the number of binary trees with that list as its frontier according to the following formula:

$$C(n) = \frac{\binom{2n}{n}}{n+1}.$$

Despite the exponential number of solutions, OLDT-resolution achieves huge efficiency gains over the other strategies by reusing tabled resolutions. Nevertheless, the implementation of tabling presented in Chapter 3 is naive—table access and the generation of solutions from the table takes time—incurring overheads that result in noticeably slow execution. In addition, OLDT-resolution reports answers for a query only once *every* answer has been computed.

Alternatively, reversing the bindings of the arguments to *frontier*, such that the first argument is a ground binary tree, results in the performance shown in Figure 6.4. Prolog suffers again from the ordering of subgoals in the body of the second clause for *frontier* and fails to terminate for all calls. Termination analysis determines that the current calls to *frontier* terminate, allowing prioritised resolution to proceed efficiently. Fair OLDT-resolution and prioritised resolution perform the same number of resolutions even though the selected literal in each resolution is different. Finally, OLDT-resolution performs slightly under double the number of resolutions of the prior two strategies.

6.3 Permuting lists

Permuting lists is another standard program for assessing termination behaviour of resolution strategies. The following predicate *perm* permutes a list:

$$\begin{array}{l} perm\ nil\ nil\ :-\ . \\ perm\ (A : X)\ Y\ :-\ perm\ X\ W,\ append\ U\ V\ W,\ append\ U\ (A : V)\ Y. \end{array}$$

A list of length n has $n!$ permutations. The performance of the interpreters using predicate *perm*, with its first argument bound to a ground list, is illustrated in Figure 6.5. Prolog and prioritised resolution share the same efficiency behaviour, with OLDT-resolution performing fractionally better, owing to the use of tabled resolutions. Fair SLD-resolution, on the other hand, performs twice as many resolutions as the other strategies.

In the other direction, where the second argument is bound to a ground list and the first is free, Prolog loops infinitely. Restoring termination in Prolog for such calls requires reversing the order of the literals in the body of the second clause for *perm*. However, rewriting programs according to the context in which they are used defeats

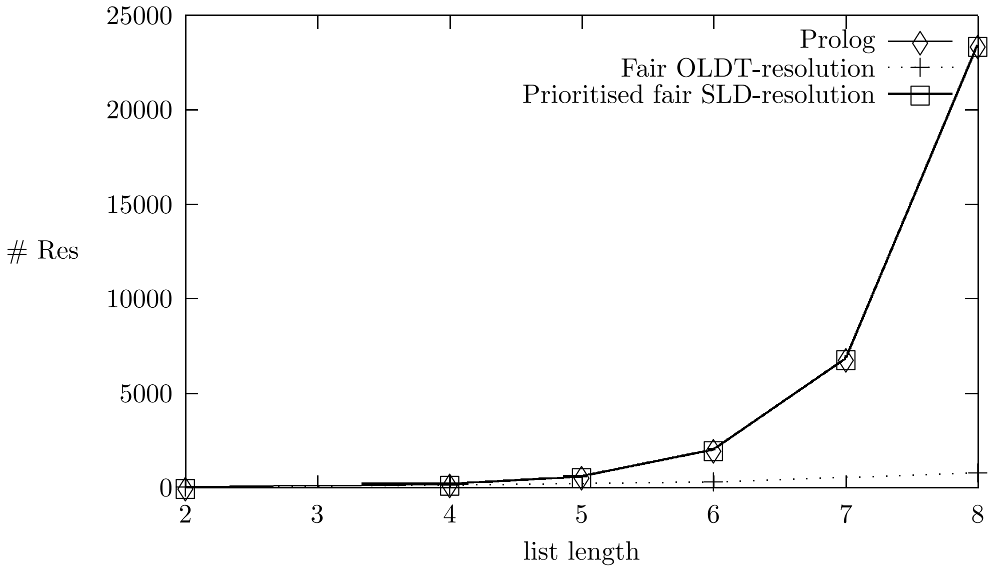


Figure 6.3: The execution of *frontier* with the first argument free and the second argument bound to a ground list.

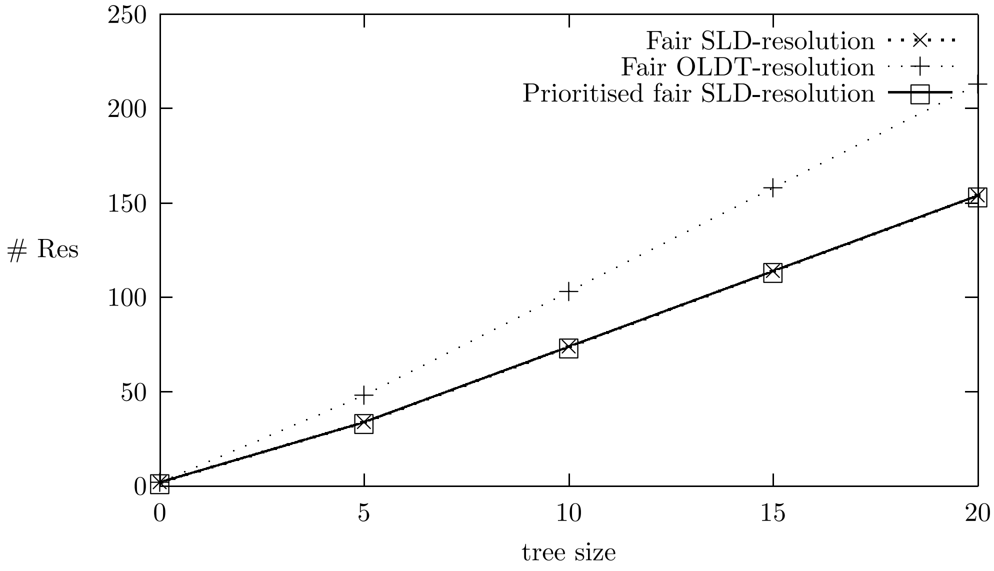


Figure 6.4: The execution of *frontier* with the first argument bound to a ground tree and the second argument free.

the purpose of declarative programming. The remaining interpreters have their efficiency behaviour documented in Figure 6.6. Prioritised resolution is unable to prove that these calls to *perm* terminate but, nevertheless, still performs the same number of resolutions that Prolog would, given the aforementioned redefinition of *perm*. Fair SLD-resolution quickly becomes intractable, whereas OLDT-resolution, as in the case for *frontier*, performs efficiently again owing to the benefits of tabling.

6.4 Sorting lists of natural numbers

The example in this section initially considers a compositional program to compare Peano natural numbers as a first step towards sorting lists of naturals. The Ruby relations that compare natural numbers—originally presented in Section 1.2—are used in the following definition of *insert* from Section 2.3.4:

$$\begin{array}{lll}
\mathit{insert}\ A\ \mathit{nil}\ (A : \mathit{nil}) & :- & . \\
\mathit{insert}\ A\ (B : Y)\ (A : B : Y) & :- & \mathit{leq}\ A\ B. \\
\mathit{insert}\ A\ (B : Y)\ (B : Z) & :- & \mathit{gt}\ A\ B,\ \mathit{insert}\ A\ Y\ Z.
\end{array}$$

Lists of natural numbers are sorted using the implementation of insertion sort below.

$$\mathit{isort}\ X\ Y\ :-\ \mathit{foldr}\ \mathit{insert}\ \mathit{nil}\ X\ Y\ .$$

The higher-order function *foldr* is defined as

$$\begin{array}{lll}
\mathit{foldr}\ F\ C\ \mathit{nil}\ C & :- & . \\
\mathit{foldr}\ F\ C\ (A : X)\ D & :- & F\ A\ B\ D,\ \mathit{foldr}\ F\ C\ X\ B.
\end{array}$$

A combinator approach to programming encourages reuse of code which is evident in the definition of *isort* above. Moreover, the declarative underpinnings of a compositional logic language frees the programmer from the often difficult job of ensuring a sequential ordering of program components. Indeed, in the case above, Prolog already fails to terminate for the expected finite calls to both *leq* and *gt*.

Consequently, Prolog does not terminate for any calls to *isort* for two reasons: the first is due to the use of the comparison relations *leq* and *gt* and the second is due to the ordering of the literals in the body of *foldr*. The major advantage of the novel resolution strategies presented in this thesis is that their efficiency behaviour is largely independent of subgoal order. In turn, the programmer is relieved from considering such operational details. Let us now consider the execution of calls to *isort*.

Fair SLD-resolution makes no practical headway in the execution of *isort* for any argument instantiations. The performance of the remaining two interpreters for calls having their first argument bound to ground, unsorted list of naturals is shown in Figure 6.7. The list comprised repetitions of naturals descending from 9. Termination analysis in prioritised resolution proves termination for all calls to *isort*, other than those with both arguments not instantiated enough with respect to the term-size norm, and calls of the current form execute efficiently. Despite this, OLDT-resolution performs better once again because of the benefits of tabling.

Indeed, the same is true, though even more markedly, in the opposite direction when the second argument bound to a sorted ground list (depicted in Figure 6.8). Calls of

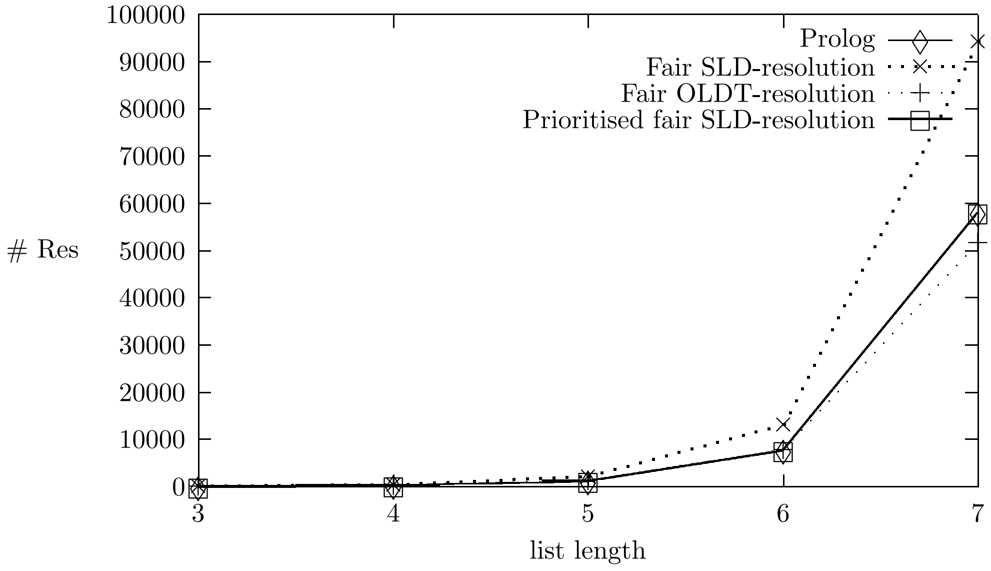


Figure 6.5: The execution of *perm* with the first argument bound to a ground list and the second argument free.

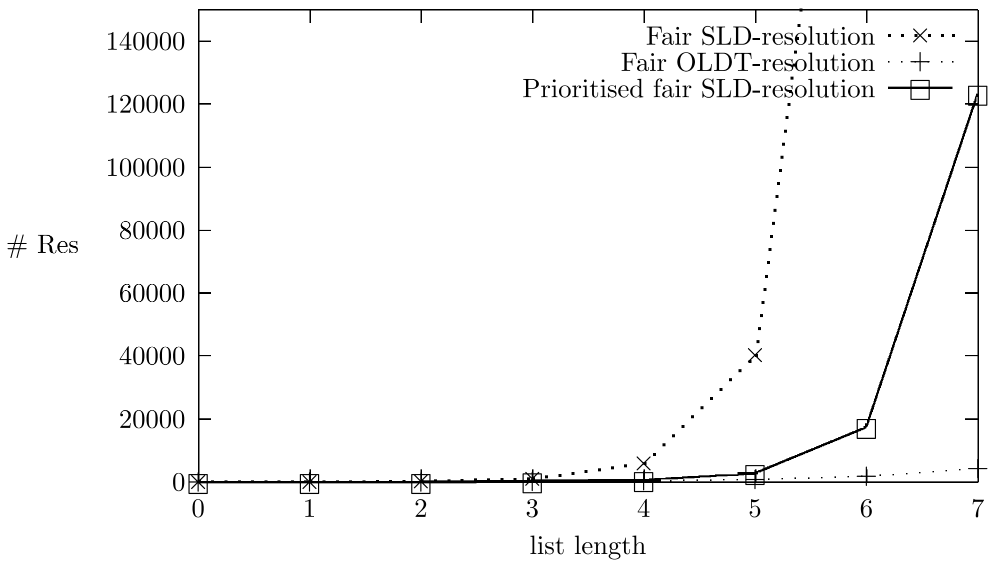


Figure 6.6: The execution of *perm* with the first argument free and the second argument bound to a ground list.

this form, with a sorted list of length n, have $n!$ solutions. Prioritised resolution, in this case, again performs in accordance with the factorial growth in the number of solutions, without the need for any fair resolution steps.

Moreover, consider the following 'mixed-mode' query

$$\# \;:-\; \mathit{isort}\;(4 : A : C : 6 : B : 5 : \mathit{nil})\;(1 : 2 : 3 : Y)\;. \tag{6.1}$$

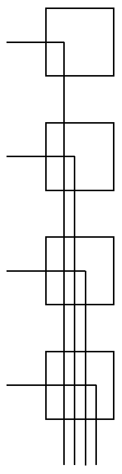
The answer substitutions to this query bind Y to the list $4 : 5 : 6 : \mathit{nil}$ and the variables A, B, and C to the appropriate permutations of 1, 2, and 3. Query (6.1) is executed efficiently under prioritised resolution, even though both arguments are not instantiated enough with respect to the term-size norm (from Example 5.1). The reason for this behaviour is that, at some stage in the resolution, subgoals are introduced that are callable and are thus executed efficiently. On the other hand, as explained in Section 4.3, OLDT-resolution fails to terminate for query (6.1), as does Prolog. Moreover, although fair SLD-resolution is guaranteed to construct a finite SLD-tree for (6.1), the tree is, in practice, so large that fair SLD-resolution is unable to solve it.

6.5 Reversing lists in Ruby

In this example, the performance of the interpreters is scrutinised when executing Ruby specifications. In particular, the task of reversing lists is revisited, this time using a more compositional program than before which is built solely from Ruby combinators. Using the Ruby combinators defined in Section 1.2, a list may be reversed using the program:

$$\mathit{rev}\;X\;Y\;:-\;\mathit{rdr}\;(\mathit{swap}\;\$\mathit{comp}\;\mathit{apr})\;(\mathit{pair}\;X\;\mathit{nil})\;Y\,.$$

An instance of *rev* as a Ruby circuit is shown below where each node represents the relation *swap* $*comp* *apr*, which simply reorders wires since *swap* and *apr* are Ruby primitives.



For calls to *rev* with the first argument bound to a ground list and the second argument free, Prolog does not terminate. However, the remaining three interpreters perform within a small constant factor of each other, as is shown in Figure 6.9. Prioritised resolution

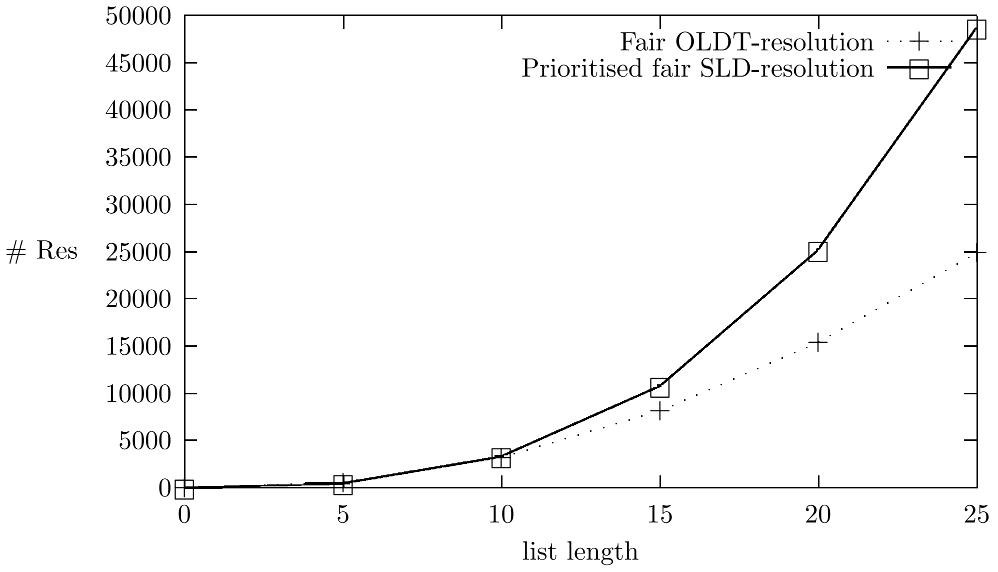


Figure 6.7: The execution of *isort* with the first argument bound to a ground list of natural numbers and the second argument free.

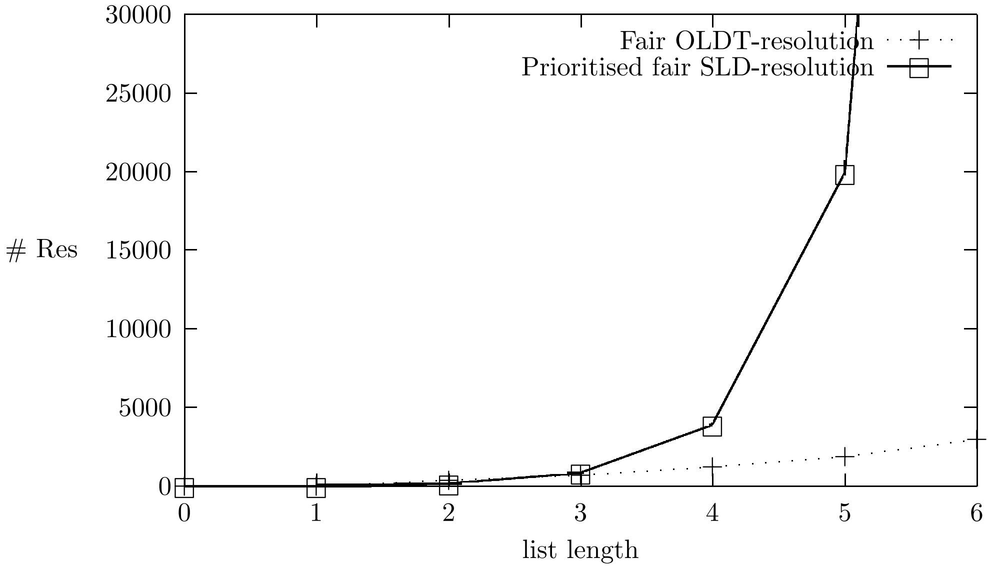


Figure 6.8: The execution of *isort* with the first argument free and the second argument bound to a ground list of natural numbers.

is unable to determine that any calls to *rev* terminate and, moreover, actually performs marginally more resolutions than both of the two other strategies.

With the bindings to arguments reversed, Prolog is still unable to terminate. The graph comparing the results of each of the other interpreters for queries of this form is illustrated in Figure 6.10. Here, it is clear that OLDT-resolution once again outstrips the other strategies in terms of resolutions performed.

In the previous examples, prioritised resolution performed efficiently in all directions since termination analysis generated a sufficiently comprehensive set of callable literals in each case. However, as witnessed here, prioritised resolution can regress back to the poor efficiency behaviour of fair SLD-resolution should its set of callable literals be insufficient. The disappointing performance of prioritised resolution in this example is due to the limitations of the implementation of termination analysis. In particular, constraint analysis is unable to infer constraints for a large proportion of predicates which makes their proof of termination impossible.

Moreover, the inability to prove termination for some predicates in a program often cascades down the predicate dependency graph which results in a large portion of predicates in a program having no terminating instantiation patterns. This deficiency with the termination analysis is fortunately not intrinsic: as outlined in Section 5.8, alternative constraint analyses are able to infer the constraints necessary to prove that calls to *rev* terminate. A prototype implementation of the termination analysis in Section 5.6 that permits the manual insertion of the constraints inferred by other constraint analyses verifies this hypothesis and restores efficient computation of *rev* using prioritised resolution.

6.6 Sorting lists of natural numbers in Ruby

A compositional style of logic programming relies heavily on the underlying computation strategy of the language to ensure efficient execution and termination of programs. Moreover, unravelling a hierarchy of combinators into a corresponding procedural logic program, amenable to execution in Prolog, can be a non-trivial undertaking. Indeed, this observation is corroborated by the fact that, even after reordering literals in the body of some clauses, Prolog is still unable to terminate for almost all Ruby programs.

The Ruby comparison operator *cmp* from Section 1.2, that sorts pairs of natural numbers is no exception to this poor behaviour of Prolog. The performance of each remaining interpreter when executing *cmp* is shown in Figure 6.11. There, it can be seen that each interpreter performs within a constant factor of one other. However, the execution of *sort*, again from Section 1.2, is not possible in practice for any permutation of argument instantiations when using any strategy other than fair OLDT-resolution. Fair OLDT-resolution is able to sort a ground list of naturals and also permute a sorted such list. Prioritised resolution of *sort* is unable to prove termination for any instantiation pattern of *sort* for the reasons given in Section 6.5.

6.7 Summary

In this chapter, several logic programs, ranging from standard ones to compositional ones, were executed in four different logic programming interpreters to assess the extent

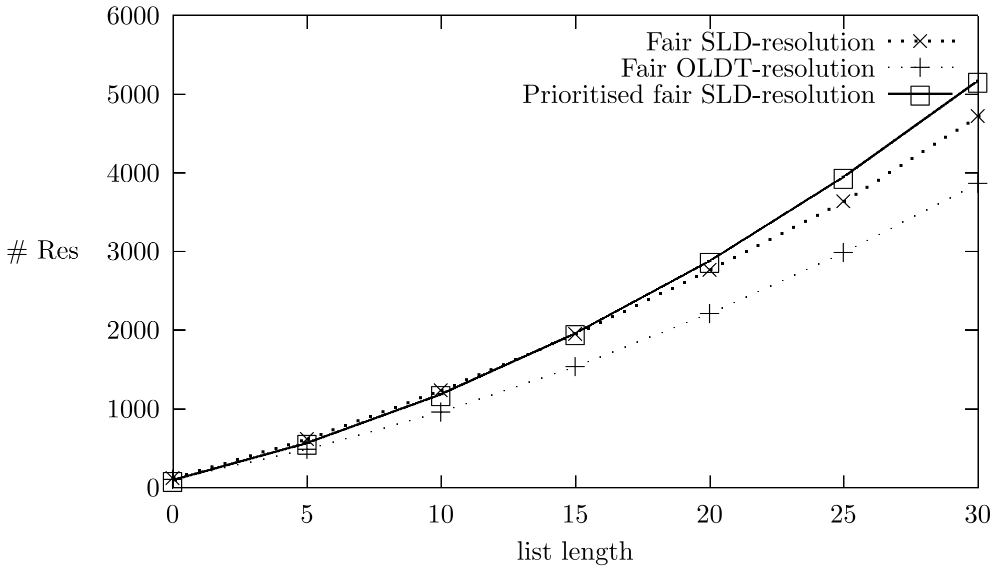


Figure 6.9: The execution of *rev* with the first argument free and the second argument bound to a ground list.

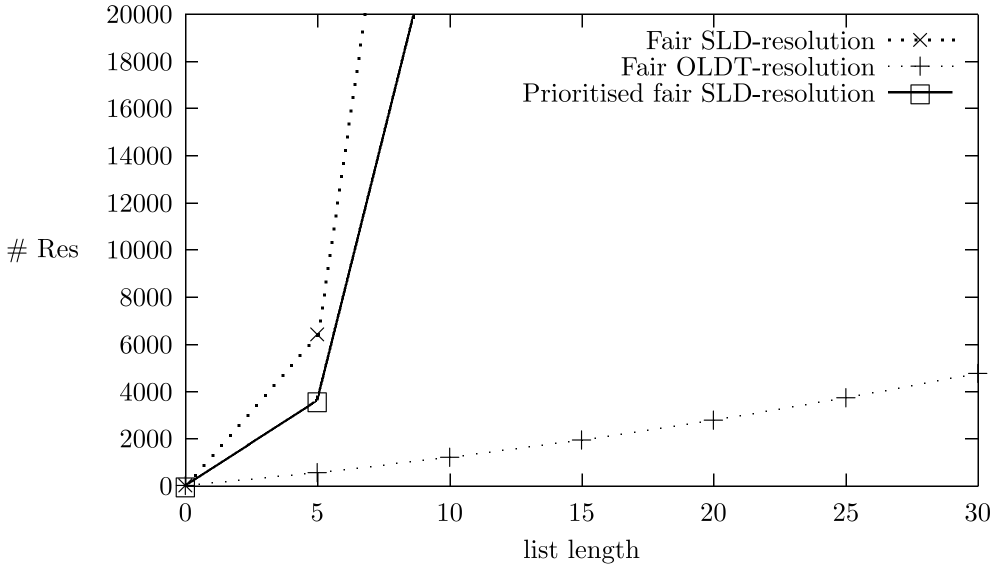


Figure 6.10: The execution of *rev* with the first argument bound to a ground list and the second argument free.

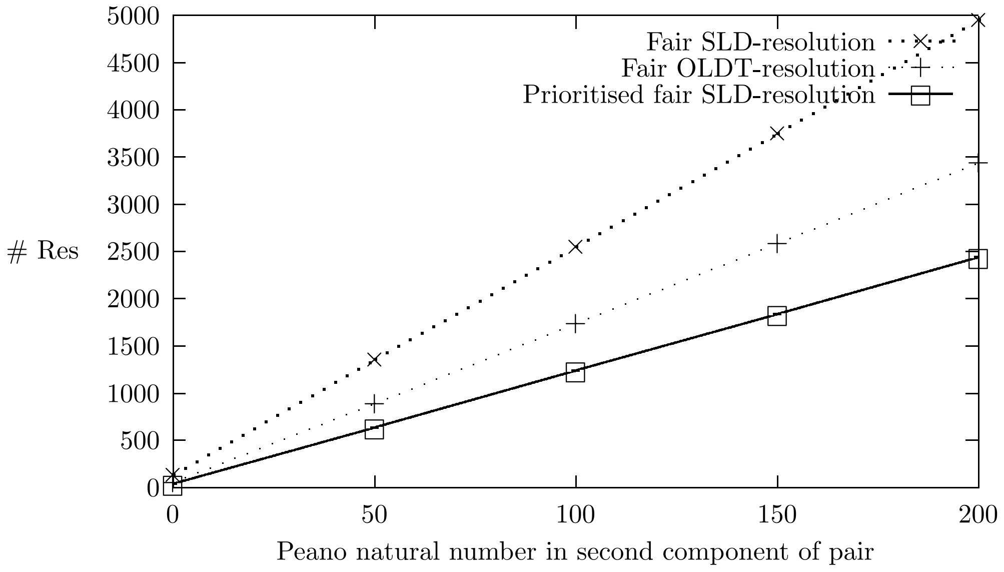


Figure 6.11: The execution of *cmp* with the first argument free and the second argument bound to a ground list.

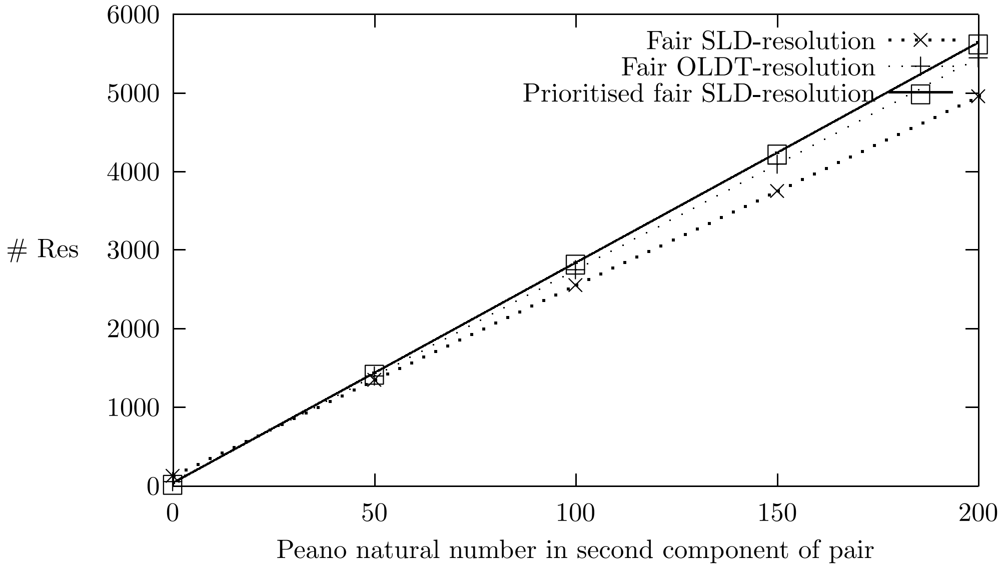


Figure 6.12: The execution of *cmp* with the first argument bound to a ground list and the second argument free.

to which each was able to support compositionality. Each interpreter employed a different operational strategy using one the following: SLD-resolution (Prolog), fair SLD-resolution using the breadth-first computation rule, fair OLDT-resolution, and prioritised fair SLD-resolution. The performance of each interpreter was compared against one another by the number of resolution steps taken to complete the evaluation of each example program.

Prolog is only able to execute carefully constructed programs that pander to the left-to-right computation rule. Consequently, compositional programs invariably fail to terminate in Prolog often without producing any solutions. The left-to-right computation rule effectively prohibits Prolog from executing most programs in more than one direction. Fair SLD-resolution partially overcomes these limitations by allowing the termination of simple programs in a variety of directions. However, fair SLD-resolution exhibits efficiency behaviour that is difficult to predict in practice and, consequently, identifying the programs it can execute in reasonable time is difficult. Indeed, more often than not, fair SLD-resolution using the breadth-first computation rule results in exponential execution time.

The two novel computation strategies presented in this thesis provide a better foundation for executing compositional programs than either Prolog or fair SLD-resolution. In the majority of the examples of this chapter, fair OLDT-resolution performs far fewer resolutions than any other strategy. The main reason for this efficiency is the elimination of redundant computation. The price paid for such desirable efficiency is the loss of the termination guarantees of fair SLD-resolution. Also, the local scheduling of answers in fair OLDT-resolution, as described in Section 4.2, means that solutions to a query are reported only once all of them are determined. Therefore, if an infinite number of solutions exists for a query then fair OLDT-resolution will be unable to report them.

Prioritised fair SLD-resolution provides the most promising basis for executing compositional programs: it restores the termination properties of fair SLD-resolution abandoned by fair OLDT-resolution and also executes more efficiently than fair SLD-resolution. Prioritised resolution relies on termination analysis for its efficiency behaviour since each callable literal will be executed efficiently. Even in cases when some literals are unable to be proved terminating, prioritised resolution still performs efficiently. The reason for this favourable behaviour is that floundered goals—those that do not contain any callable literals—can introduce callable literals after a fair resolution step and hence restore efficient computation. However, the reliance on termination analysis means that any of the limitations it may suffer from are subsequently reflected in prioritised resolution. Indeed, the termination analysis presented in Chapter 5 does have a few limitations as discussed in Section 5.7, the most detrimental being the constraint analysis it adopts. Fortunately, more comprehensive constraint analyses exist that allow termination analysis to prove the termination of many more programs. In turn, prioritised resolution is able to execute more efficiently.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

Relational program derivation has gathered momentum over the last decade with the development of many relational calculi. However, executing relational expressions in existing programming languages requires a high degree of care to ensure that they avoid entering an infinite loop. Consequently, prototyping specifications becomes a difficult undertaking, hindering an understanding of why some relational transformations lead to an efficient program and others do not.

This thesis has contributed a compositional style of logic programming, largely unavailable to programmers until now, in which relational specifications can be phrased naturally. Elements of functional programming style have been translated into the logic paradigm, consequently enriched with the extra expressiveness fostered by nondeterminism, logic variables, and program converses. In itself, compositionality facilitates greater code reuse and encourages more abstraction than currently practised in the logic community. Moreover, compositional programs are amenable to the formulation of results regarding their equivalence, allowing transformations to be safely applied to them. However, the compositional style of programming advocated in this thesis proves difficult to execute in existing logic languages.

The pivotal contributions of this thesis are the development of the language features necessary to support compositional logic programming: a higher-order framework that facilitates the description of relational specifications; and a fair resolution strategy that guarantees the termination of programs whenever possible. Moreover, this thesis provides several operational semantics, presented as interpreters, that permit the efficient, direct execution of relational specifications for the first time.

From the outset, this thesis identifies a fundamental distinction between 'programming' languages and 'specification' languages: the former providing efficient execution of carefully phrased algorithms, and the latter the execution of high-level specifications devoid of algorithmic instruction. It is precisely this latter category that enhances the intrinsic declarative nature of the logic paradigm and captures the essence of prototyping relational specifications without the consideration of evaluation order imposed by a language, often essential in languages like Prolog, Mercury, and λProlog. This thesis identifies a collection of desiderata for a compositional logic language, against which the suitability of existing languages for supporting compositionality can be gauged. Without such a study, making comparisons between incarnations of compositional languages is difficult.

For example, existing logic languages have, to some degree or another, addressed matters of syntax for the support of higher-order programming constructs, a necessity for compositionality. However, a compositional logic language requires greater powers of expression than a higher-order syntax alone. The majority of methods adopted in existing languages to support compositionality manifest themselves as little more than convenient syntactic notations. Indeed, the endeavour of becoming truly declarative requires a compositional language to cast off the shackles of a predetermined evaluation order: an observation corroborated by the fact that existing logic languages seldom permit the direct execution of compositional programs. The vital criterion for compositionality, lacking in all existing logic languages, is the notion of *fair* evaluation which is identified in this thesis as the critical requirement for supporting declarative, compositional programming.

The perpetrator of the poor behaviour of existing logic languages is the inflexible left-to-right computation rule pervasive in almost all logic languages including the functional logic ones. As is well-known amongst logic programmers (O'Keefe 1990, Clocksin & Mellish 1987), the left-to-right computation rule often necessitates multiple versions of the same predicate, depending on which of its arguments will be instantiated. However, rewriting a program according to the context in which it is used defies the very notion of compositionality.

It is for the above reason that existing languages are unsuitable for the compositional style of programming advocated in this thesis. Nevertheless, fairness alone is unable to institute compositionality in a programming language: without appropriate consideration, fair evaluation can be so inefficient that even logically elementary computations fail to terminate within acceptable time. The fair strategies presented in this thesis address this problem and restore acceptable efficiency behaviour to compositional programs.

We summarise the characteristics of the languages considered in this thesis in the following table.

Language	*Syntax*	*Fairness*	*Efficiency*
Prolog	×	×	✓
λProlog	✓	×	✓
Fair SLD	✓	✓	×
Fair OLDT	✓	✓×	✓×
Prioritised SLD	✓	✓	✓

In the table above, the columns are interpreted as follows: *Syntax* records whether the language allows higher-order programming constructs to be introduced without the need for extra-logical predicates; *Fairness* records whether a program executed in the language is guaranteed to terminate within the confines of SLD-resolution; and *Efficiency* records whether the execution efficiency of a program is simple to predict. The symbols '✓' and '×' in a column indicate that the language feature is present or absent, respectively. Both symbols appear in the *Fairness* column for fair OLDT-resolution since its notion of fairness does not encompass that of fair SLD-resolution but, nevertheless, its termination behaviour is independent of the way in which a program is composed. The appearance of both symbols in the *Fairness* column for fair OLDT-resolution results from the fact that it can be difficult to determine whether a given query will terminate when executed using fair OLDT-resolution (though we remark in the subsequent section about how this may be rectified).

Existing languages

The study of compositionality in this thesis began with the examination of existing languages according to a set of desiderata, deemed essential for compositional programming. The first desirable feature of a compositional language is that it should naturally embrace the expression of high-level specifications. This criterion also judges how declarative a language is, i.e., the degree of attention that a programmer must pay to the execution order of the language. Ideally, in a compositional language a programmer should be oblivious to any optimisations made during the execution of a program, but still retain confidence in the operational characteristics of the language. In particular, assurances must be made to the programmer about the termination properties of a program and the likelihood of its efficient execution. These desiderata for compositionality facilitated the following assessment of existing languages.

First-order logic programming, using Prolog, requires the use of extra-logical predicates, e.g., the *call* and *apply* predicates, to facilitate higher-order programming. In turn, the aesthetic benefits of specifications were lost during their conversion to programs. The use of *apply* in favour of *call* was suggested by (Naish 1996) since it provides a more faithful analogy to higher-order features found in functional languages.

When programming with combinators in an untyped logic language, it can often be difficult to spot errors caused by composing combinators of incompatible type, which commonly results in the failure of queries. On the other hand, strongly typed languages can detect all type errors in a program before it is executed. Therefore, augmenting a compositional language with a strong type system, either as an integral component or as an optional tool, is desirable from the point of view of developing error-free programs. However, the *apply* predicate cannot be given a strong type in, say, the type system of λProlog; ideally, the type of *apply* should be $(A \rightarrow B) \rightarrow A \rightarrow B$, permitting values of any kind to be returned. However, λProlog assigns the primitive 'return' type o to each predicate, as discussed in Section 2.4.3. Therefore, the only plausible type for *apply* in λProlog is $(A \rightarrow B \rightarrow o) \rightarrow A \rightarrow B \rightarrow o$. Then, for a predicate such as *add* of type $nat \rightarrow nat \rightarrow nat \rightarrow o$, literals like *apply add* $1\ P$ are ill-typed.

The cumbersome method of writing higher-order programs in Prolog can be attributed to the lack of appropriate theoretical foundations with respect to this style of programming. Alternatively, the higher-order logic language λProlog (Nadathur & Miller 1988) was designed with exactly such principles in mind. Consequently, λProlog allows specifications to be represented naturally as programs. However, the trade-off for naturality of programming is the adoption of higher-order unification which is an undecidable procedure. However, in Section 3.1, this thesis identified that the adoption of a curried representation of terms facilitated precisely the same higher-order constructions provided by the *call* primitive, but without requiring its use or that of higher-order unification. In addition, curried terms provide a much less obtrusive syntax than that of Prolog with no adverse impact on the underlying resolution strategy whatsoever.

In Chapter 2, a partial solution was offered in λProlog to rectify the shortcomings of the left-to-right computation rule confined to the domain of executing Ruby programs (Jones & Sheeran 1990). The trick was to encode two alternative definitions of sequential composition: the definitions being logically equivalent but with the order of the two conjuncts in their bodies reversed. Then a heuristic method selected the most appropriate definition of composition, based on the context in which it was used. Essentially, this procedure simulated the notion of fairness in an elementary way. Such heuristics, however, failed to generalise to arbitrary compositional programs. Moreover, the solution presented required extra-logical annotations to the program which affected its declarative nature. Furthermore, the extra-logical predicates used are unavailable in other logic languages.

Flexible computation rules

The comparison described in the previous section concluded that, although logic programming provides a promising starting point for executing compositional programs, the inflexible left-to-right computation rule requires attention. The logic programming community has acknowledged the limitations of a predetermined computation rule and suggested coroutining (Lüttringhaus-Kappel 1993, Naish 1992, Naish 1985), discussed in Section 3.5, that permits the dynamic selection of literals in a goal according to some criteria. The criterion adopted by (Lüttringhaus-Kappel 1993) is that literals can be selected if they have a finite SLDF-tree. The notion of a finite SLDF-tree, however, is weaker than that of a finite SLD-tree, and termination of an SLDF-resolution provides little guarantee about its completeness. In particular, it is possible for the SLDF-resolution of a goal to flounder when some SLD-resolution would find solutions for it.

Although coroutining often executes efficiently, it suffers from several important weaknesses within the domain of compositionality. Firstly, the syntactic construct of when-declarations (Definition 3.10), that provide the conditions under which literals in a goal may be selected, are somewhat limited. Indeed, all too often the SLDF-resolution of a literal will terminate only if at least one of its arguments is finite. However, when-declarations are inherently unable to represent such generalisations over the shape of terms.

Moreover, the algorithm used to generate when-declarations automatically for a program (Figure 3.5) is heuristic in nature. In particular, depth-bounds are used to restrict the size of terms in a when-declaration and also to determine whether such terms have a finite SLDF-tree. As a consequence, the computed set of when-declarations often poorly approximates the actual literals in a program that have finite SLDF-trees. Furthermore, the computed set is often unsafe in the sense that an SLDF-resolution using an unsafe call set may terminate for some goal but may flounder or loop for an instantiation of it. The above factors culminate in a computation strategy for which predicting whether a query will execute efficiently or not is difficult, in turn contravening a vital criterion for compositionality.

Fair SLD-resolution

Heuristic attempts to rectify the problem of non-termination caused by the left-to-right computation rule are unsatisfactory for at least two reasons. First and foremost, heuristics are unable to guarantee termination for arbitrary programs. Secondly, heuristic methods of altering the flow of control in a program, like that for λProlog presented in Section 2.4.3, tend not to generalise to other programs. Worse, the programmer is responsible in such cases for effecting changes to heuristic implementations which eradicates the principal benefit of declarative programming.

For these reasons, this thesis rediscovered fair SLD-resolution from the logic community as a method of ensuring termination of logic programs whenever any other SLD-resolution would terminate. The notion of fairness was originally introduced by (Lassez & Maher 1984) to provide a theoretical basis for SLDNF-resolution (Lloyd 1987). There, (Lassez & Maher 1984) showed that fair SLD-resolution was complete with respect to finite failure. In this thesis, that result was extended in Lemma 3.3 to prove that fair SLD-resolution is also complete with respect to finite success. The implication of this result is that fair SLD-resolution guarantees the termination of a query, i.e., the construction of a finite SLD-tree, whenever possible.

At this point, it is important to reiterate the difference between a 'computation rule' and a 'search strategy' in the resolution process. The computation rule selects literals in a goal for resolution and the search strategy selects clauses from the program to resolve against the selected literal. In standard logic programming parlance, 'breadth-first' and 'depth-first' search refer to the search strategy: the method of traversing the SLD-tree which is altered by varying the order in which clauses are chosen from the program. Crucially, whilst breadth-first search guarantees the enumeration of all possible solutions in an SLD-tree and depth-first search does not, it is the computation rule that determines the overall shape of the tree. In other words, only a change in computation rule can convert an infinite SLD-tree to a finite one. Indeed, the choice of search strategy is orthogonal to the choice of computation rule in resolution.

Nevertheless, some compositional programs are inherently infinite, e.g., those that exhibit unbounded nondeterminism, and the use of a fair computation rule is unable to translate these evaluations into terminating ones. Therefore, a complete search strategy, like breadth-first search, is essential in such cases to enumerate the variety of solutions. Despite this observation, depth-first search was chosen for the interpreters presented in the appendices since it exhibits better space efficiency that breadth-first search. Despite this decision, in Section 3.3 an alternative complete search strategy is discussed, called depth-first iterative deepening, that overcomes the space problems of breadth-first search.

The most elementary way to achieve fairness in resolution is through the use of a queue-based computation rule, rather than the left-to-right one. (A queue-based computation rule is sometimes called a breadth-first computation rule, though this overloading of notation should not be confused with a breadth-first selection strategy.) However, fair SLD-resolution using this computation rule often exhibits poor efficiency behaviour, rendering computations intractable in practical terms.

Previously, though entirely independently, (Janot 1991) proved a similar result regarding the termination properties of fair SLD-resolution and also investigated the implementation of fair SLD-resolution, albeit with a different emphasis than in this thesis. There, two versions of fair SLD-resolution are suggested in an effort to overcome the deficiencies of the breadth-first computation rule: the more efficient of the two being indexed fair SLD-resolution (Section 3.2). However, without a declarative style in mind, the solution is tailored to individual programs; each literal in a program requires explicit annotation with an integer index, whose actual value is the responsibility of the programmer. Moreover, empirical study of each program and its query is often essential to determine appropriate indices. Even then, the heuristic nature of selecting indices provides no guarantee of efficient execution. The main contribution of Janot's work is the evidence that new forms of fair SLD-resolution can be implemented efficiently in a particular Prolog abstract machine (Bekkers, Canet, Ridoux & Ungaro 1986) with only a small constant factor overhead.

Tabled evaluation of logic programs: fair OLDT-resolution

Over the last decade, proponents of tabled evaluation of logic programs have demonstrated that such systems, e.g., XSB (Sagonas et al. 1994), provide better termination behaviour than SLD-resolution (Chen et al. 1995, Warren 1992, Tamaki & Sato 1986). Indeed, a tabling system is guaranteed to terminate if an SLD-resolution, using the same computation rule, would make only finitely many different calls and generate only finitely many different answers. For example, tabling techniques overcome the classic non-termination of left-recursive predicates in logic programs. Despite this improved termination behaviour, tabling has been considered in the literature only for the left-to-right computation rule. Consequently, such tabling systems inherit all the problems prevalent in the earlier discussion.

Tabling is reformulated in this thesis for any 'arbitrary' but 'fixed' computation rule (Definition 4.1), allowing a tabled evaluation to be envisaged as the resolution of individual subgoals in a goal. An additional contribution made in this thesis is the generalisation of the new formulation of tabling to include a notion of fairness. The resulting tabling strategy, called fair OLDT-resolution (Definition 4.7), is more flexible than any existing tabling implementation. Indeed, fair OLDT-resolution will terminate for every program that either Prolog or XSB terminates for. Furthermore, fair OLDT-resolution terminates in practice more often than fair SLD-resolution by eliminating redundant subcomputations and, hence, restores efficient evaluation of compositional programs. Evidence of this performance enhancement was shown in Chapter 6 where the fair OLDT-resolution interpreter performed fewer resolutions, and terminated more frequently, than any other strategy.

The drawback of using an arbitrary but fixed computation rule is that the termination property of fair SLD-resolution is lost since the Switching Lemma—(Lloyd 1987) and Lemma 3.2—can no longer be applied arbitrarily. Nevertheless, the practical performance of fair OLDT-resolution, as outlined in Chapter 6, demonstrates that fair tabling provides a realistic operational basis for a compositional logic language.

Prioritised fair SLD-resolution

Although fair OLDT-resolution executes efficiently and terminates in practice for a number of compositional programs, its use of an arbitrary but fixed computation rule results in the loss of the desirable termination properties of fair SLD-resolution. These properties are reinstated by another contribution of this thesis: the marriage of coroutining and fair SLD-resolution to form prioritised fair SLD-resolution (Definition 5.9). Prioritised resolution shares a similar philosophy to coroutining in that literals in a goal are selected for resolution based on some property they satisfy. In the case of SLDF-resolution the property is that a literal has a finite SLDF-tree; the problems associated with this strategy were discussed earlier.

Alternatively, prioritised resolution selects literals in a goal if their SLD-resolution is known to terminate with respect to a particular arbitrary but fixed computation rule. The crucial component of prioritised resolution is the novel static analysis of a program to determine its set of terminating literals. The static analysis—presented in Chapter 5 with the algorithm depicted in Figure 5.1—makes critical use of termination analysis for logic programs. Termination analysis has been developed over the past decade but has been formulated only for the left-to-right computation rule.

The novel contribution of Chapter 5 is the adaptation of the termination analysis by (Lindenstrauss & Sagiv 1997) to employ the computation rule of prioritised resolution. Subsequently, the new termination test is used to generate the call set for prioritised resolution. Prioritised resolution improves on fair SLD-resolution by 'eagerly' evaluating literals whose resolution is known to terminate at the expense of any others. In the cases where a goal is floundered, a fair resolution step is performed using the breadth-first computation rule (Definition 3.4) which ensures that prioritised resolution is fair. Since termination analysis is undecidable, termination analysis may be unable to prove termination of an arbitrary literal, the same being true for the less sophisticated notion of termination adopted by coroutining. In the latter case, coroutining terminates unsuccessfully in such instances, whereas prioritised resolution may be able to transform a floundered goal using a fair resolution step into one that can be evaluated efficiently again. Prioritised resolution overcomes all the problems associated with coroutining that were discussed earlier.

7.2 Future work

Higher-order unification in compositional logic languages

In this thesis, we have proposed a style of compositional logic programming that allows the use of higher-order programming constructs but adopts first-order unification in the underlying operational model. The compositional programs presented throughout the thesis have all been executed within the confines of first-order unification: the use of higher-order unification was unnecessary. Although we have yet to discover a convincing example compositional program that makes essential use of higher-order unification, it may be the case that such a class of compositional programs exists. Therefore, an exploration of possible alternative notions of compositional logic programming may be desirable which could possibly uncover novel applications of the compositional programming paradigm. One clear starting point is to integrate a fair computation rule into a higher-order language, like λProlog.

Fair OLDT-resolution

In Chapter 6, we noted that the fair OLDT-resolution of a query is guaranteed to terminate if a corresponding SLD-resolution of the query makes only finitely many different calls and generates only finitely many different answers. Consequently, fair OLDT-resolution can permit this class of left-recursive programs to terminate whilst SLD-resolution cannot. Despite the advantage that fair OLDT-resolution has over fair SLD-resolution, fair OLDT-resolution is unable to terminate for other kinds of query for which fair SLD-resolution does.

In particular, the fair OLDT-resolution of a query is guaranteed to terminate only whenever the OLDT-resolution of it using any arbitrary but fixed computation rule (of which the left-to-right one is an example) would also terminate. Therefore, one deficiency with fair OLDT-resolution is determining how the class of programs for which it terminates relates to the class for which fair SLD-resolution terminates; the two classes clearly intersect, although the class for fair OLDT-resolution is not a strict subset of the class for fair SLD-resolution since fair OLDT-resolution is able to terminate for left-recursive programs that are infinite under fair SLD-resolution. As yet, we have been unable to define the boundaries between the two classes of programs and further exploration of this issue would be valuable.

Prioritised fair SLD-resolution

The implementation of prioritised resolution could be further developed beyond that of the prototype presented in Appendix C; for reasons associated with any prototype implementation, omissions and simplifications exist in it. One way to improve the execution efficiency of prioritised resolution would be to use the information gleaned during termination analysis to predetermine the order in which literals are selected in a goal. By doing so, the current run-time overhead observed in the interpreter of Appendix C caused by searching each goal for a terminating literal can be avoided. Thus, the overheads introduced by the dynamic computation rule may be eliminated.

The performance of prioritised resolution could potentially be improved further by adopting a new constraint analysis in the termination test of Chapter 5. New constraint analyses use linear inequality constraints and can often generate more comprehensive constraints than the constraint analysis of Section 5.6.4, which uses monotonicity constraints. Such additional constraints can help prove termination for literals that our analysis currently cannot. Fortunately, the termination analysis of Chapter 5 is able to represent such constraints as arcs in a mapping and the integration of new constraint analyses is a natural area of further research.

Another possible way to increase the efficiency of prioritised resolution is to employ alternative termination analyses in our algorithm of Section 5.5 which automatically generates the call set for a program. The description of the algorithm is essentially independent of the actual method of testing termination of literals and, recently, new termination analyses have been implemented that appear to be faster than the one of Chapter 5. However, such tests are currently formulated only for the left-to-right computation rule and it is unclear whether they can be adapted to use the computation rule of prioritised resolution. An investigation into this aspect would be desirable.

Termination analysis has recently been implemented in the Mercury programming language to verify the termination of both user programs and compiler generated optimisations (Speirs et al. 1997). However, in this thesis, termination analysis is used for the first time to improve the efficiency of programs by selecting literals whose evaluation terminates over those that do not. Therefore, as in the case for prioritised resolution, languages like Mercury may be able employ the new algorithm of Section 5.5 to find automatically a terminating order of literals in clauses and, consequently, improve the execution efficiency of programs. Examining the feasibility of such an integration would be a suitable area of further work.

Choice of norm in termination analysis

The description of termination analysis in Chapter 5 used the term-size norm to determine whether terms were instantiated enough. However, the use of the term-size norm can restrict the number of programs for which termination can be shown. For example, consider the following query:

$$\# \;{:}{-}\; \mathit{append}\ (1 : A : 3 : \mathit{nil})\ X\ (B : 2 : 3 : 4 : \mathit{nil})\ .$$

Neither of the three arguments to *append* are instantiated enough with respect to the term-size norm and, therefore, termination analysis would be unable to indicate that such queries terminate.

However, it is possible to prove that such queries do, in fact, terminate with respect to the list-size norm since it ensures all finite lists are instantiated enough irrespective of the nature of their elements. The problem with the term-size norm here is that it is too specific for *append* in the sense that it considers the instantiation state of each element in a list, which is irrelevant for the proof of termination of *append*.

Thus, one possible area of future work is to determine the 'most general' norm to use during the termination analysis of a predicate. Most general norms are characterised by the fact that if a predicate can be proven to terminate using a most general norm then it can be proven to terminate using all 'related' norms. One possible starting point is to try to infer automatically the most general norm of a data type from its actual declaration. An initial attempt to define the most general norm of a data type is given below.

Definition 7.1 (Most general norms). Let X be the declaration of a regular data type with components of the form $f\ \beta_1\ \dots\ \beta_n$, where f is a constructor, and $\beta_1, \dots, \beta_n$ are types, for $n \geq 0$. Then, the most general norm for the type X, written $\|\_\|_X$, contains an equation

$$\|f\ T_1\ \dots\ T_n\|_X = n + \sum_{i=1}^{n} \|T_i\|_{\beta_i}$$

for each component of X, where $T_1, \dots, T_n$ are terms. Moreover, the norm $\|\_\|_X$ contains the equation

$$\|Y\|_X = Y$$

where Y is an arbitrary variable. ♢

As an example, it is easy to see that the above rules define the most general norm for the list data type to be that of the list-size norm.

The concept of most general norms in termination analysis centres around using the inferred type of a predicate to guide the selection of the norm to use during its proof of termination. So, for example, if the type of *append* was inferred as *list* $\alpha \rightarrow$ *list* $\alpha \rightarrow$ *list* $\alpha \rightarrow o$, then the termination analysis of *append* would use the most general norm associated with the *list* data type during its termination proof. The details of these ideas remain to be worked out and, as yet, it is unclear whether unique most general norms always exist for regular data types.

Integrating tabled and prioritised resolution

As we explained in Chapter 4, OLDT-resolution can be viewed as a natural precursor to prioritised resolution: OLDT-resolution evaluates each subgoal simultaneously until the first completely evaluated, i.e., finite, one is discovered. Only then are its answer substitutions returned to the remaining subgoals. In other words, OLDT-resolution naively searches for terminating subgoals at run-time by evaluating all literals, even the infinite ones. On the other hand, prioritised resolution uses information gleaned from a compile-time termination analysis of the program to select at run-time a single subgoal whose resolution is known to terminate, rather than by exhaustive search as in the case of fair OLDT-resolution.

However, OLDT-resolution has the advantage of tabling the answers of each terminating literal and, consequently, eliminating their redundant evaluation. Prioritised resolution is, nevertheless, amenable to the integration of this form of tabling since each terminating literal can be tabled in the same way as in OLDT-resolution. The subtle reason for this is that no fair resolution steps need to be made during the prioritised resolution of a terminating literal—a point discussed in Section 5.4—which means that the computation rule is *fixed*. As explained in Chapter 4, this is the crucial requirement for a computation rule to allow tabled evaluation.

One remaining question is whether the additional termination property of OLDT-resolution, i.e., it terminates for all left-recursive programs that make only finitely many different calls and produce only finitely many different answers, can be integrated into prioritised resolution. The difficultly here is that termination analysis is currently designed only for SLD-resolution and it is unclear whether termination analysis can be modified to determine termination of programs that use tabled evaluation.

Applications

The design of a compositional logic programming language in this thesis was motivated by a desire to execute high-level relational specifications, like those presented by (Bird & de Moor 1996). Their relational calculus contains powerful primitive constructs like relational division which permits the expression of specifications that contain universal quantification, and 'ΛR' which converts a relation R to its isomorphic representation as a set-valued function. Recently, (Seres & Mu 2000) have presented an implementation of a subset of this calculus in Prolog; it uses the *call* and *bagof* extra-logical predicates and is carefully crafted to avoid non-termination. An intriguing area of further work would be to translate their implementation into the compositional language presented in this thesis and augment it, if possible, with the remaining operators of the calculus.

The definitions of these operators make essential use of relational negation. Capturing relational negation in a logic language requires the use of negation-as-failure (Lloyd 1987) which is commonly implemented by the extra-logical predicate 'not'. As stated at the outset, we have not considered this component of logic programming in this thesis since extending a logic language with negation-as-failure is straightforward; for example, the standard way to achieve this is to employ a *safe* computation rule (Lloyd 1987) to ensure that a negative literal is selected for resolution only if it is ground. (The interpreter presented in Appendix A has been trivially extended to implement negation-as-failure in this way.)

Once negation-as-failure is integrated into our compositional language, relational negation can be implemented as follows:

$$\mathit{negate}\ R\ A\ B\ \ {:-}\ \ \mathrm{not}\ (R\ A\ B).$$

The expressive power of the resulting compositional language can be exemplified by specifying the longest *up sequence* of a list. An up sequence of a list of natural numbers is a subsequence of the list that is sorted in ascending order. Assuming the definitions of the standard relational operators of composition, union, intersection, converse, and relational fold, we begin by implementing the relational division operator *over* in the following way:

$$\mathit{over}\ R\ S\ A\ B\ \ {:}{-}\ \ \mathit{negate}\ ((\mathit{negate}\ R)\ \$\mathit{comp}\ (\mathit{conv}\ S))\ A\ B.$$

The gist of the specification is to generate a subsequence of the list, ensure that it is ordered, and then select the minimum subsequence with respect to length. The higher-order predicate *min* performs the latter task and is defined below.

$$\mathit{min}\ R\ S\ A\ B\ \ {:}{-}\ \ (S\ \$\mathit{cap}\ (R\ \$\mathit{over}\ (\mathit{conv}\ S)))\ A\ B.$$

Using relational folds, we can implement two simple relations that respectively relate lists to their subsequences, and to their lengths:

$$\begin{array}{lll} \mathit{sub}\ X\ Y & {:}{-} & \mathit{fold}\ \mathit{nil}\ (\mathit{cons}\ \$\mathit{cup}\ \mathit{outr})\ X\ Y. \\ \mathit{length}\ A\ Y & {:}{-} & \mathit{fold}\ 0\ (\mathit{succ}\ \$\mathit{comp}\ \mathit{outr})\ A\ Y. \end{array}$$

Next, we define the preorder *len*, that relates lists to those ones shorter than it:

$$\mathit{len}\ X\ Y\ \ {:}{-}\ \ ((\mathit{conv}\ \mathit{length})\ \$\mathit{comp}\ (\mathit{leq}\ \$\mathit{comp}\ \mathit{length}))\ X\ Y.$$

The longest up sequence of a list can then be specified as

$$\mathit{lup}\ X\ Y\ \ {:}{-}\ \ \mathit{min}\ (\mathit{conv}\ \mathit{len})\ (\mathit{ordered}\ \$\mathit{comp}\ \mathit{sub})\ X\ Y.$$

where the relation *ordered* is the identity on ascendingly sorted lists. Therefore, we can query

$$\#\ \ {:}{-}\ \ \mathit{lup}\ X\ (4:5:2:3:5:6:2:7:\mathit{nil}).$$

which succeeds with the single answer $X = (2:3:5:6:7:\mathit{nil})$. Although specifications like *lup* contain such little algorithmic information that their execution is markedly inefficient, the ability to run such high-level specifications is extremely useful, especially for prototyping relational program transformations.

Appendix A

Implementing fair SLD-resolution

In this appendix, we present the implementation of a simple logic programming interpreter that uses fair SLD-resolution, as described in Section 3.2. This implementation serves as a precursor to the two interpreters developed in the subsequent appendices. The language used for all implementations in these appendices is Caml-light and we assume considerable familiarity with it in what follows. The main reasons for selecting Caml-light as the implementation language are that it comes equipped with extensive libraries that prove useful for interpeter writing, and it is sufficiently high-level enough to permit algorithms to be implemented without regard for low-level details.

The actual program code is presented for each interpreter in an effort to formalise the important algorithms and to provide a platform from which more efficient versions may be launched. The programs presented are primarily meant to be understood rather than executed and, consequently, many optimisations in both data representation and algorithms could be made to these programs. The implementation in this appendix commences with the introduction of terms and substitutions, followed by the unification algorithm. Each of these elements will be used pervasively in the later interpreters.

A.1 Terms

The first departure from traditional logic programming is the adoption of a curried form of terms, suggested in Definition 3.1. We allow variables to appear as the head of terms, in addition to constants, which allows us to dispense with the use of meta-level predicates to deal with calls to instantiated higher-order variables. The definition of terms in Caml-light is as follows:

```
type term =
    Const of string
  | Var of string
  | App of term * term ;;
```

Pleasingly, the change to a curried term syntax facilitates most of the desired higher-order programming constructs without complicating the details of unification and resolution.

A.2 Substitutions

Substitutions can be implemented directly from their abstract representation given in Definition 2.4, i.e., a substitution is a function mapping a variable to a term:

```
type substitution == string → term ;;
```

However, the alternative representation of a singleton substitution $\{X \mapsto t\}$—mapping the variable X to the term t and all other variables to themselves—as an association pair of type *string* * *term* will prove useful in the unification algorithm, presented in the next section. We construct the elementary substitution $\{X \mapsto t\}$ from the association pair (X, t) using the function *single* : *string* * *term* → *substitution* as follows.

```
let single (x,t) y = if x = y then t else Var y ;;
```

The identity substitution ι, presented in Section 2.3.1, maps all variables to themselves and is implemented by the function *idsub* : *substitution* as follows:

```
let idsub = Var ;;
```

So each (string) argument to *idsub* is wrapped up with the constructor *Var* to form a variable term.

For a given substitution ϕ, we implement the 'apply' operator $[\phi]$, of Definition 2.5, via the function *apply* : *substitution* → *term* → *term* by case analysis on the structure of terms:

```
let rec apply phi = function
      Const c → Const c
    | Var x → phi x
    | App (f,g) → App (apply phi f, apply phi g) ;;
```

We deviate slightly from Definition 2.5 since terms are curried: to determine the instance of an application of terms, we recursively apply the substitution over both subterms.

We use *compose* : *substitution* → *substitution* → *substitution* to implement the composition of two substitutions and its definition follows directly from Definition 2.5.

```
let compose phi sigma x = apply phi (sigma x) ;;
```

The abstract representation of a substitution as a function allows for a natural translation from specification to implementation.

The function *add* : *string* * *term* → *substitution* → *substitution* augments a substitution with a new binding for a variable:

```
let add (x,t) = compose (single (x,t)) ;;
```

Substitutions are constructed from the composition of many singleton substitutions and, therefore, the application of a substitution over a term can become an inefficient operation. Indeed, this representation of substitutions is undesirable other than for illustrative purposes. Nevertheless, we find the case for clarity of algorithms more convincing than for speed of execution.

A.3 Unification

The process of unifying two terms results in a substitution, called the most general unifier, if the terms are unifiable. In the case where the terms are not unifiable, we must indicate failure of their unification in some way. In terms of the Caml-light implementation, we raise the exception

```
exception No_unifier ;;
```

to signal the lack of a unifier for two terms.

Such a case arises when a variable is unified with a term in which it is itself contained. The so-called ‘occurs check’ identifies whether a particular variable is contained in some term and is implemented by the function *occurs* : *string* → *term* → *bool* using recursion over the structure of terms:

```
let rec occurs x = function
    Const c → false
  | Var y → x = y
  | App (f,g) → occurs x f || occurs x g ;;
```

Historically, most logic programming languages exclude the occurs check from the unification algorithm which may, in certain cases, introduce unsound proofs of programs in favour of their efficient execution. As always, we base our implementation decisions on theoretical rather than practical considerations and include the occurs check in the unification algorithm below.

We follow the abstract algorithm of Figure 2.1 to unify two terms, representing sets as lists in Caml-light. We make frequent use of the function *map_single* : *string* * *term* → (*term* * *term*) *list* → (*term* * *term*) *list* to apply a singleton substitution, represented as an association pair, over each pair of terms in the list we are unifying.

```
let map_single (x,t) =
  let phi = single (x,t) in
  map (fun (u,v) → (apply phi u, apply phi v)) ;;
```

The function *unify* : *term* → *term* → *substitution*—whose implementation is shown in Figure A.1—returns the most general unifier of the two terms, if it exists, otherwise the exception *No_unifier* is raised. We begin by creating the singleton list comprising the paired terms we wish to unify and follow the algorithm of Figure 2.1 almost identically. The main difference is that compound terms are replaced by the application of two terms; the unification of two such terms proceeds by unifying the corresponding components of the two applications.

A.4 Fair SLD-resolution

The proof of a goal proceeds by constructing an SLD-tree, searching it for refutations. A goal, in Caml-light, is a list of terms

```
type goal == term list ;;
```

```
let unify u v =
  let ps = ref [ (u,v) ] and p = ref (u,v) and phi = ref idsub in
  while !ps <> [] do
    p := hd !ps; ps := tl !ps;
    match !p with
      (Const a, Const b) →
        if a <> b then raise No_unifier
    | (App (f,g), App (h,k)) →
        ps := (f,h) :: (g,k) :: !ps
    | (Var x, Var y) →
        if x <> y then begin
          ps := map_single (x,Var y) !ps; phi := add (x,Var y) !phi
        end
    | (Var x, t) →
        if not occurs x t then begin
          ps := map_single (x,t) !ps; phi := add (x,t) !phi
        end else raise No_unifier
    | (t, Var x) →
        ps := (Var x, t) :: !ps
    | _ → raise No_unifier
  done;
  !phi ;;
```

Figure A.1: The implementation of the function *unify*.

and a refutation is an empty list, corresponding to the empty goal # :−. A program clause is a pair whose first component is the head of the clause, i.e., a term, and the second is the body of the clause, i.e., a goal. The type of a clause is as follows:

```
type clause == term * goal ;;
```

The function *clauses* : *term* → *clause list*, whose definition we omit, takes a term and returns the appropriate list of clauses for the predicate of the term.

As discussed in Section 2.3.3, we have a choice of how to construct the SLD-tree for a goal: the computation rule may select any literal from a goal and the search strategy may use any program clause to resolve with the selected literal. According to Definition 2.7, a resolution step involves matching the selected literal with a variant of a program clause. By variant, we mean that the logic variables contained in the clause must be distinct from all others in the SLD-tree. The function *fresh* : *clause* → *clause* performs this operation by canonically renaming all variables in a program clause:

```
let fresh (l,ls) =
  let freshen = apply (fun x → Var (x ^ string_of_int !count)) in
  incr count; (freshen l, map freshen ls) ;;
```

The global variable *count* : *int ref* generates a unique integer that is concatenated to the end of each variable and so ensures that each clause contains unique variable names.

The choice of search strategy dictates which parts of the SLD-tree are constructed first and, hence, schedules the nodes in the tree that are pending resolution. A depth-first search strategy can record the current state of an SLD-tree by maintaining only one node for each level in the tree. In fact, depth-first search can be organised simply using a stack of nodes where the head of the stack is the current, unexplored node in the SLD-tree. Recall from Definition 2.10 that a node in the SLD-tree is labelled with a goal, and followed by a number of branches to child nodes, where each branch is obtained by resolving the selected literal of the goal with an appropriate program clause and labelled with the substitution of the resolution.

Therefore, the current state of a node can be captured sufficiently from the current goal, the clauses remaining to be resolved with the selected literal, and the answer substitution of the derivation. We arrive at the following type definition for a node in an SLD-tree:

```
type SLD_node == goal * substitution * int
```

The data type invariant for a node is that a node $([\,],\ \mathit{theta},\ i)$ is a refutation with answer substitution *theta* whilst for a node $(l :: \mathit{ls},\ \mathit{theta},\ i)$, l is the selected literal and $i \geq 0$ is the number of program clauses already resolved against l in the resolution. Furthermore, if $\mathit{cls} = \mathit{clauses}\ l$ and $i < \mathit{list\_length}\ \mathit{cls}$, then the clause $\mathit{nth\_list}\ i\ \mathit{cls}$ is used next for resolution with l.

We utilise the Caml-light library module *stack* and define an SLD-tree as

```
type SLD_tree == SLD_node stack__t ;;
```

We construct an SLD-tree from a goal using $\mathit{mk\_SLD\_tree} : \mathit{goal} \rightarrow \mathit{SLD\_tree}$:

```
let mk_SLD_tree gs =
  let tree = stack__new () in stack__push (gs,idsub,0) tree;  tree ;;
```

The abstraction functions for an SLD-tree are defined simply from the standard stack operators:

```
let complete = stack__empty
and current = stack__pop
and sprout = stack__push
and replace = stack__push ;;
```

The function $\mathit{sprout} : \mathit{SLD\_node} \rightarrow \mathit{SLD\_tree} \rightarrow \mathit{unit}$ is used to schedule a child node in the SLD-tree. On the other hand, the function $\mathit{replace} : \mathit{SLD\_node} \rightarrow \mathit{SLD\_tree} \rightarrow \mathit{unit}$ is used to position a node at the current location of the SLD-tree such that, for a node N and a tree T, immediately after the operation *replace* N T, then $N = \mathit{current}\ T$. The provision of these functions aids the implementation of alternative search strategies by allowing nodes to be scheduled in different ways. For example, we could implement breadth-first search using the Caml-light *queue* library as follows:

```
let mk_SLD_tree gs =
  let tree = queue__new () in queue__add (gs,idsub,0) tree;  tree ;;

let complete = queue__empty
and current = queue__take
and sprout = queue__add
and replace = queue__addhead ;;
```

where *queue__addhead* adds an item to the head of a queue rather than to the tail.

Abstracting over the type of an SLD-tree allows us to change search strategy without altering the code for the resolution function *resolve* : *goal* → *unit*. The goal supplied to *resolve* is executed by repeatedly sprouting its SLD-tree by performing resolution steps at the current node. If a refutation is discovered, the corresponding answer substitution is printed by the function *report_solution* : *substitution* → *unit*. The definition of *resolve* is presented below.

```
let resolve gs' =
  let tree = mk_SLD_tree gs' in
  while not complete tree do
    let (gs,theta,i) = current tree in
    match gs with
      [] → report_solution theta
    | l :: ls →
        let cls = clauses l in
        if i < list_length cls then
          let (l',ls') = fresh (nth_list i cls) in
          replace (goal, theta, i+1) tree;
          try
            let phi = unify l l' in
            sprout (map (apply phi) (ls @ ls'), compose phi theta, 0) tree
          with No_unifier → ()
  done ;;
```

The computation rule can be altered by ammending the expression *ls* @ *ls'* in the body of *resolve*. Currently, new subgoals are introduced at the end of the pending goal, thus implementing a fair computation rule. Alternatively, the expression *ls'* @ *ls* implements a stack of subgoals and, therefore, mimics the left-to-right computation rule of Prolog.

Appendix B

Implementating fair OLDT-resolution

In Chapter 4, we defined fair OLDT-resolution and, although it does not guarantee the construction of a finite search tree, showed that it can improve upon the termination behaviour of fair SLD-resolution for a number of example compositional programs. However, at any stage in an OLDT-resolution there may be several possible transformations to apply to a node in a tree. Without directing the search, we could continue to generate trees whose answers are not required for the computation. Although fairness guarantees that no such tree will be developed indefinitely, this concept introduces a potential inefficiency when considering an implementation of fair resolution. In particular, some partially computed trees in fair resolution may become disconnected from the main OLDT-tree. In this appendix, we introduce data structures and algorithms that eliminate these problems. Although cycle detection is important in a tabled interpreter since more programs would terminate than in SLD-resolution, we will present a version here that does not employ cycle detection to keep the implementation simple.

B.1 Tables

As a consequence of active nodes depending on their corresponding answer nodes, we require an efficient way to access answer nodes and, moreover, to determine if a particular literal already appears as the root of an OLD-tree. We do this by creating a *table* that maps terms to values of some sort. In fact, tables are pervasive in the interpreter since they also facilitate quick lookup for terms. We specify the type of a table, and its associated operations, in the interface to the *table* module, given below.

```
type 'a t ;;

value new : unit → 'a t
and clear : 'a t → unit
and add : term → 'a → 'a t → unit
and find : term → 'a t → 'a ;;
```

We create a table using the function $new : unit \rightarrow {'a}\ t$, the polymorphic type $'a$ being the target of the table, i.e., the entry in the table associated with a term. Since table access and updates will occur frequently, we must ensure that the above operations are efficient. One efficient method of indexing terms is by using generalised *tries*, an idea suggested in (Ramakrishnan, Rao, Sagonas, Swift & Warren 1995). In this section, we study the implementation of tries of terms.

Tries are multiway branching trees, classically applied to the problem of lexicographic searching. The underpinning technique of tries is to fragment canonically a large data structure into smaller parts that can collectively be used to index the entire structure incrementally. So, in the case of dictionary words, each character of the word is used to index a branch in a node of the trie. With respect to logic programming terms, we can use each constant symbol and variable name to index branches. The fragments of the indexing data structure are sequenced using a list; consequently any data type having such an interpretation can form the index of a trie. We call the list representation of the data structure the *key* of the trie.

Our logic programming terms have a straightforward representation as a list: terms can be thought of themselves as trees and the inorder traversal of this tree can be used to convert terms to keys. We must, however, make a simple observation to discriminate completely between the various possible occurrences of constants and variables in a term. Since our terms are of a curried nature, the same constant or variable could quite possibly appear in a different term with a different number of arguments. To distinguish such occurrences, we augment each component of a term's key with the number of arguments it has. The function *flatten* : *int* → *term* → *term* * *int list* transforms a term into its unique representation as a list, the integer argument being the number of arguments of the given term.

```
let rec flatten n = function
    App (e1,e2) → flatten (n+1) e1 @ flatten 0 e2
  | e → [e,n] ;;
```

The function *unflatten* : *term* * *int list* → *term* satisfies the identity

$$unflatten \cdot flatten\ 0 = id$$

and can be defined straightforwardly, although we omit its definition here.

Owing to the fact that tries can share common portions of keys, a terminal node in a trie cannot simply be identified by a node that has no branches. A terminal node can appear anywhere in a trie and we must, therefore, explicitly distinguish them from other nodes in the trie. A node in a trie comprises: a *key* field; a *data* field (to record whether it is a terminal node); and a pointer to the next part in the trie. In fact, the data field allows us to use tries as tables since we can assign an arbitrary value to this field of a terminal node. Of course, since only some nodes in a trie terminate a key, we employ the *maybe* data type to distinguish terminal nodes from other nodes:

```
type 'a maybe =
    Nothing
  | Just of 'a ;;
```

A non-terminal node has its *data* field set to *Nothing*. The type of a trie is given by the mutually recursive data type

```
type ('a, 'b) node =
    { mutable key : 'a;
      mutable data : 'b maybe;
```

```
        mutable next : ('a, 'b) trie }

  and ('a, 'b) trie =
      { mutable trie : ('a, 'b) node list } ;;
```

parametrised by the type of the key, '*a*, and the type of the target entries, '*b*. We could employ data structures other than lists to represent the collection of nodes in a trie: representations like balanced binary trees or hash tables could allow more efficient access to the node associated with a particular key. We develop the following programs to abstract over the actual choice of data structure.

An empty trie is created using the function *new* : *unit* → ('*a*,'*b*) *trie*, defined as

```
  let new () = { trie = [] } ;;
```

and the function *mk_node* : '*a* → ('*a*,'*b*) *node* creates a new node with the given key:

```
  let mk_node k = { key = k; data = Nothing ; next = new () } ;;
```

A node is inserted into a trie using *insert* : ('*a*,'*b*) *node* → ('*a*,'*b*) *trie* → *unit*, whose side-effect is to update the list of nodes in the trie, defined as:

```
  let insert v ({ trie = vs } as trie) = trie.trie ← v :: vs ;;
```

The function *assoc* : '*a* → ('*a*, '*b*) *trie* → ('*a*, '*b*) *node* finds the node with the given key field in a trie:

```
  let assoc p { trie = vs } =
    let rec assoc' p = function
        [] → raise Not_found
      | u :: us → if p = u.key then u else assoc' p us
    in
    assoc' p vs ;;
```

If no such node exists, the build-in Caml exception *Not_found* is raised.

New keys are inserted into a trie, after being converted into their list representation, using the function *add* : '*a list* → '*b* → ('*a*,'*b*) *trie* → *unit*. We assume that the key we are inserting does not already occur in the trie:

```
  let rec add (k :: ks) item trie =
    let u =
      try assoc k trie with Not_found →
        let v = mk_node k in
        insert v trie;  v
    in
    if ks = [] then u.data ← Just item else add ks item u.next ;;
```

The function *find* : '*a list* → ('*a*,'*b*) *trie* → '*b* takes the list representation of a key, and returns its associated value in the trie. If the key does not exist in the trie, we raise the exception *Not_found*.

```
  let rec find (k :: ks) trie =
    let u = assoc k trie in
    if ks = [] then
      match u.data with
        Nothing → raise Not_found
      | Just item → item
    else find ks u.next ;;
```

Now we can implement the *table* module by the program

```
type 'a t == (term * int, 'a) trie ;;

let new = trie__new
and clear tbl = tbl.trie ← []
and add l = trie__add (flatten 0 l)
and find l = trie__find (flatten 0 l) ;;
```

As well as using tables to implement association mappings for terms, we will also use them to implement *sets* of terms; in particular, tables facilitate efficient membership tests for sets of terms. In this case, a set of terms is a table whose target is of type *bool*:

```
type term_set == bool table__t ;;
```

The function *mem* : *term* → *term_set* → *bool* tests membership of a set of terms and is defined below.

```
let mem l set = try find l set with Not_found → false ;;
```

In the following section, we describe the computer representation of OLD-trees.

B.2 OLD-trees

In this section, we present the implementation of the type of OLD-trees, and its associated operations, after discussing the requirements for the data type. Recall from Definition 4.5 that an OLD-tree is rooted at a node that contains a single literal and has a number of branches equal to the number of program clauses for that literal. A branch connects to a child node that contains a number of subgoals. Each subgoal in a child node can be considered the root of its own OLD-tree; from this, we can form the forest of OLD-trees that comprise an OLDT-tree.

The first aspect of an OLD-tree we consider is the representation of its children. As answers are returned to a goal, we create new subgoals to develop and so, in the representation of an OLD-tree, we must maintain the current state of the children of the tree. For example, consider the abstract representation of an OLD-tree depicted in Figure B.1; the completion of the subgoal *arc a Z* in node 2 provides two answer substitutions that update node 2 to nodes 3 and 5; we only represent the current, uncompleted children in an OLDT-tree. So, the current children of this tree are the nodes 1 and 5. Notice that since node 3 is completely evaluated and that node 2 has been updated, neither is represented as a current child.

Each child in an OLD-tree must record the substitution of the OLD-resolution, θ that annotates the arc connecting the child to its parent node. From an implementation point of view, it is useful for each OLD-tree to record the initial answer substitution χ of the current derivation, such that the entire computed answer thus far is $\theta \circ \chi$. The final pieces of information that an OLD-tree maintains are the number of program clauses tried for it and a list of the 'answers' discovered so far in its resolution, the type of which we will introduce later.

Another consideration for OLD-trees is the order in which to develop them. For this purpose, it is instructive to conceptualise a fair OLD-tree as a process that can spawn

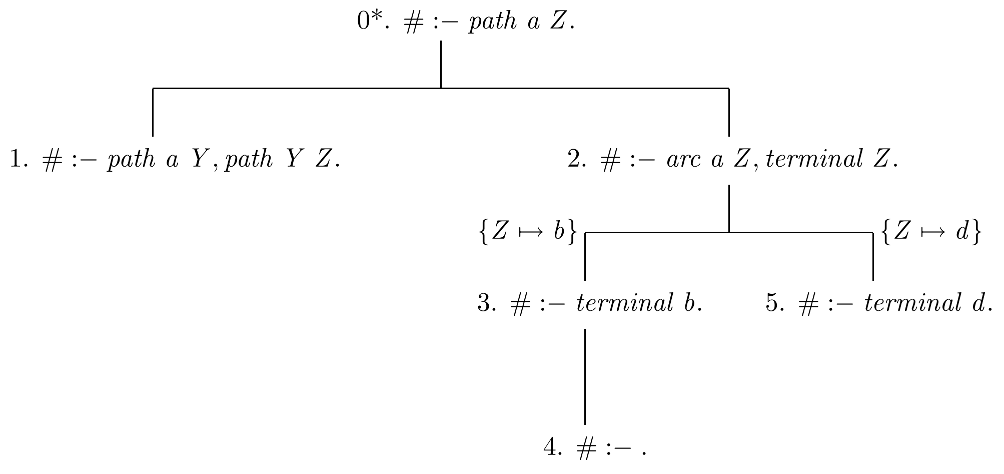


Figure B.1: An example OLD-tree.

child processes where each child process is itself an OLD-tree. We have a choice of how to develop these child processes; we can develop them *eagerly*, i.e., as soon as they are created, or, in the case of fair OLDT-resolution, we can impose a fair selection of child processes such that a particular child is not further developed until its parent process has sprouted *all* of its child processes. The latter procedure creates a 'frontier' of OLD-trees that can each be developed one resolution step at a time to maintain fairness.

Each leaf OLD-tree in the frontier requires the construction of a child using either program or answer clause resolution, depending on whether it is an answer or active node, respectively. Locating leaf nodes in the OLDT-tree is not as simple as one might perhaps imagine. Unfortunately, it is not enough simply to record the leaves in the expanding frontier since the frontier changes dynamically as leaves become completely evaluated and their answers are returned to goals. Rather, we must locate leaves by traversing the OLDT-tree in some fashion, e.g., by a depth-first traversal, examining each branch in the tree until we encounter a leaf. After performing a computation step, we backtrack to the last partially explored branch of the tree and attempt to find subsequent leaves.

As the forest of OLD-trees grows, so does the number of OLD-trees we must traverse until we find a leaf. Consequently, the traversal of the OLDT-tree can be quite time consuming if we have to visit each OLD-tree in the forest. Fortunately, it is not necessary to visit *all* the OLD-trees in the forest, only those that present a choice of paths to traverse. A path is a sequence of connected OLD-trees and is unique when the current child of each OLD-tree contains only one subgoal. Such a path is called a *spine* of the tree. The plan is to collect those OLD-trees in the forest that chart unique paths in the tree. This way, we can limit the number of OLD-trees we must visit in order to locate a leaf. To illustrate, consider the goal

$$\# \;:-\; \mathit{add}\ 100\ 100\ 200.$$

that produces the spine shown in Figure B.2. In this spine, each node forms the root of an OLD-tree. We can collect all these trees in a sequence such that its frontier leaf is the head of the sequence. In many derivations, the length of a spine can be considerable

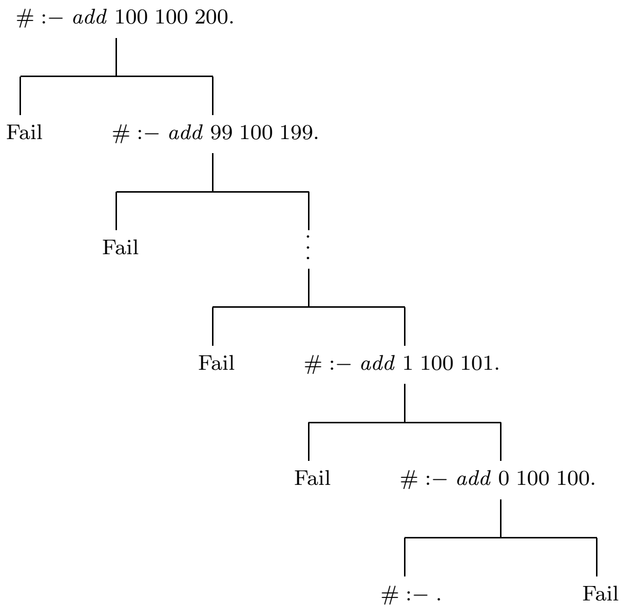


Figure B.2: The spine formed by # :− *add* 100 100 200.

and, as we shall see shortly, representing parts of the OLD-tree forest using spines allows us to access the current leaf in time independent of the length of the spine.

From all the requirements discussed above, we arrive at the following mutually recursive data type declaration for OLD-trees:

```
type old_tree =
    { label : term;
      answer : substitution;
      answers : answers;
      program_clause : int;
      mutable children : (forest list * substitution) list
    }

and forest =
    { first : old_tree;
      mutable spine : old_tree list
    } ;;
```

The data type invariant for a forest F, is that *F.spine* contains at least one OLD-tree, and that *F.first* is the the last OLD-tree in *F.spine*. Also, each OLD-tree in *F.spine* has only one subgoal in its current child. Finally, for a spine $[t_0; t_1; \ldots; t_n]$, we have the relationship, for $n \geq 0$ and each $0 \leq i < n$:

$$t_i = hd\ (fst\ t_{i+1}.children)$$

for consecutive OLD-trees in the spine.

An OLD-tree is constructed using *mk_tree* : *substitution* → *term* → *old_tree* which takes the answer substitution computed so far for the derivation and the root label of the tree. The function *mk_forest* : *old_tree* → *forest* generalises a leaf OLD-tree to a forest:

```
let mk_forest t = { root = t; spine = [t] } ;;
```

and the function *sprout* : *forest* → *unit* ensures that the given forest adheres to the data type invariant. Its definition is

```
let sprout forest =
  match children forest with
    ([forest'] ,_) :: _ → forest.spine ← forest'.spine @ forest.spine
  | _ → () ;;
```

The accessor function *children* : *forest* → (*forest list* * *substitution*) *list* returns the children of the last OLD-tree in the forest as

```
let children forest = (hd forest.spine).children ;;
```

and the function *current_goal* : *forest* → *forest list* returns the current goal of the last OLD-tree in the given forest

```
let current_goal forest =
  match children forest with
    [] → []
  | ch :: chrn → fst ch ;;
```

B.3 Variants of terms

A common operation on literals is to check whether it is a 'variant' of another, i.e., if they are identical up to the renaming of variables. We create a variant of a literal by replacing its variables with a canonical choice of identifier. The function *variant* : *term* → *term* performs this replacement. So, if we have two terms u and v that are identical up to the renaming of variables, then *variant* u = *variant* v. We retrieve the variables from a term by an inorder traversal of its abstract representation as a tree using the function *set_of_vars* : *term* → *string list*. Each variable in the list is associated with a canonical identifier using the function *new_vars* : *string list* → *string* * *string list*, defined as follows:

```
let new_vars ys =
  fst (it_list (fun (xs,n) x → ((x, "_V" ^ string_of_int n) :: xs, n + 1))
        ([],0) ys) ;;
```

Using this association list, we create the canonical representation of a term via the function *rename* : *string* * *string list* → *term* → *term*,

```
let rec rename xs = function
    Const c → Const c
  | Var x → Var (assoc x xs)
  | App (f,g)) → App (rename xs f, rename xs g) ;;
```

and define *variant* in terms of these functions

```
let variant t = rename (new_vars (set_of_vars t)) t ;;
```

B.4 Traversing OLDT-trees

An OLDT-tree is simply the forest of OLD-trees that develops from the root OLD-tree. The distinguished root for a query $\# :- A_1, \ldots, A_n$ is an OLD-tree with a special label and only one child node whose subgoals are $A_1, \ldots, A_n$. As mentioned earlier, we develop the OLDT-tree by visiting each leaf in the tree, performing a computation step on it. The only obligation that we must adhere to during a traversal is that, after a finite number of steps, every leaf must be selected for resolution so that we maintain fairness.

The manner in which this traversal is performed dictates how the OLDT-tree is generated. Naturally, many different methods of traversal are possible, like depth-first and breadth-first. Moreover, alternative traversal strategies, like bounded depth-first search, could employ heuristics to obtain good efficiency behaviour in practice. We delay the discussion of this point until later and opt for depth-first traversal of the tree.

Earlier, we stipulated that only one variant of a subgoal, called the answer node, should be developed using program clause resolution whilst all others, called active nodes, should perform answer clause resolution. Deciding on when active nodes can begin to perform answer clause resolution is one choice that influences the efficiency of OLDT-resolution. In our implementation, we allow active nodes to receive answers only once their corresponding answer node is completely evaluated, known as *local scheduling* of answers. Local scheduling helps restrict the size of the OLDT-tree by preventing the return of many meaningless answers to a goal. If an answer node depends on an active node, i.e., if there is a cycle in the OLD-tree, then the active node must receive answers, otherwise the answer node can never become completely evaluated. For example, this situation arose with node 1 in Figure 4.2.

We record the active nodes encountered during the traversal of an OLDT-tree using the global set *answer_nodes* : *term_set*:

```
let answer_nodes = table__new () ;;
```

In fact, we store variants of literals in the set to avoid distinguishing between literals identical up to the renaming of variables.

As we traverse the OLDT-tree, it can change dynamically as completely evaluated subgoals return their answer substitutions to a goal. As these substitutions are applied over a goal, new subgoals are created and the old subgoals are discarded. We can determine if the current child of a tree has been updated by the completion of a subgoal by checking to see if the number of the subgoals in a goal has decreased since the last visit to this goal. The *traverse* function performs a backtracking traversal of an OLD-tree, maintaining the state of the traversal on the stack *path* that contains the subgoals left to examine in the current traversal. The stack *path* : (*forest* * (*int* * *int*)) *stack__t* holds pairs of the form (*forest*, (*i*,*n*)), where

$$n = \mathit{list\_length}\ (\mathit{current\_child}\ \mathit{forest})$$

and $0 \leq i \leq n$. Moreover, for successive elements $(\mathit{forest}, (i, n))$ and $(\mathit{forest}', (i', n'))$ on *path*, we have that

$$\mathit{forest}' = \mathit{nth\_list}\ (i-1)\ (\mathit{current\_child}\ \mathit{forest}).$$

When $n = 0$ for *forest′*, then *hd forest.spine* is a leaf node. The memoisation of the number of subgoals of a child n is used to determine if the OLDT-tree has been dynamically

```
let traverse path root active_table =
  let traverse' path active_table =
    let current = ref (fst (peek path)) and found = ref false in
    while not !found do
      let (forest, (i,n)) = stack__pop path in
      let n' = list_length (current_goal forest) in
      if n = 0 then found := true
      else if n <> n' then stack__push (forest, (0,n')) path
      else if i < n then begin
        let child = nth_list i (current_goal forest) in
        let l = variant child.root.label in
        push (forest, (i+1,n)) path;
        try table__find l active_table; () with Not_found →
          table__add l true active_table;
          push (child, (0,list_length (current_goal child))) path
      end;
      current := forest
    done;
    ( (if empty path then Nothing else Just (peek path)), !current)
  in
  try traverse' path active_table with Empty →
    table__clear active_table;
    stack__push (root, (0, list_length (current_goal root))) path;
    traverse' path active_table ;;
```

Figure B.3: The definition of the function *traverse*.

updated by a completely evaluated literal; should this number decrease on backtracking, then a completely evaluated literal has updated the child and we can then recommence the examination of the new subgoals for leaf nodes.

The function *traverse* returns a leaf of the forest and the parent of the leaf if it has one. When there are no more leaves left to visit, the exception *stack__Empty* is raised. As we progress with the traversal, we record the selected literals, i.e., the answer nodes, in the table *answer_nodes* and skip any subsequent subgoals corresponding to active nodes. The definition of *traverse* is given in Figure B.3.

B.5 Answers

The pervasive operation in the tabled interpreter is the return of answers for a subgoal to a goal. Each OLD-tree records its own computed answers which, in turn, depend on the answers computed for each subgoal in the child of the tree. An OLDT-refutation can contain a considerable number of OLD-trees and so answers substitutions are proportionally comprised of long compositions of unifiers. Unfortunately, our current representation of substitutions as finite mappings becomes inefficient owing to the large number of applications we must perform as we return answers. Nevertheless, this is the only area in the implementation where performance suffers due to the representation of substitutions. Rather than adopting a new representation for substitutions universally, we instead employ a new data structure for answers in isolation.

Recall that the answer substitution for a refutation is the composition of all the substitutions of the OLD-resolutions along the refutation. Each OLD-tree corresponding to the resolutions along a derivation records its own substitution so we can adopt a representation of answer substitutions as a *trail* of lists of variable/term pairs, where each variable is explicitly associated with its binding term. A trail is implemented as a linked sequence of such lists, the link being made to next completed OLD-tree in the trail and a terminal node in a trail indicated with the *maybe* constructor *Nothing*. We define the following:

```
type answer == (string * term) list ;;

type trail == answer * old_tree maybe ;;
```

The answers for an OLD-tree are a list of such trails in the table. We could implement a trail more efficient using lookup tables instead of linked lists but, again, to keep the implementation descriptive rather than efficient, we will use the linked list implementation. One further requirement for answers is that we must be able to determine duplicate solutions, so answers also contain a set of the answer terms, themselves.

```
type answers =
  { list : trail list;
    set : bool table__t
  } ;;
```

The function *apply_answer* : *answer* → *term* → *term* takes an association list representation of a substitution and applies it over the given term, in analogy with the function *apply* : *substitution* → *term* → *term*. The definition of *apply_answer* is given below.

```
let rec apply_answer chis = function
      Const c → Const c
    | App (f,g) → App (apply_answer chis f, apply_answer chis g)
    | Var x → try assoc x chis with Not_found → Var x ;;
```

The job of *insert_answers* : *substitution* → *old_tree* → *bool* is to update the answers of the OLD-tree by inserting the substitution, if it is not a duplicate. The boolean valued result of *insert_answers* is *false* if the answer was a duplicate and *true* otherwise.

```
let insert_answer phi tree =
  let l = variant (apply phi tree.label) in
  try not table__find l tree.answers.set with Not_found →
    table__add l true tree.answers.set;
    tree.answers.list ←
        (map (fun x → (x, apply phi (Var x))) tree.vars, Nothing) :: tree.answers.list;
    true ;;
```

The subtle point with *insert_answer* is that the answer we insert is terminal in terms of the trail; such answers are immediately generated by successful resolution of facts, for example, and thus do not link to answers of child subgoals.

Once an OLD-tree has been completely evaluated, we place it in *complete_nodes* so that subsequent calls may be retrieved from the table. The answers will, however, be stored as a trail in the table and we must convert this into a list of substitutions. The function *gather_answers* : *old_tree* $\rightarrow$ *answer list* produces a list of all the alternative representation answer substitutions for an OLD-tree by traversing the trail and accumulating the answers.

```
let rec gather_answers tree = flat_map (get_answers tree) tree.answers.list

and get_answers tree = fun
  (chis, Nothing) → [ chis ]
| (chis, Just child) →
    let phiss = gather_answers child in
    let compose chis phis = map (fun (x,t) → (x,apply_answer phis t)) chis in
    map (fun phis → compose chis phis) phiss ;;
```

The function *compose* : *answer* $\rightarrow$ *answer* $\rightarrow$ *answer*, which is defined in *get_answers*, composes two answers and returns a tabled answer, restricted to the variables of *tree*. Although gathering the answer from a table is an efficient operation and, moreover, is only carried put once tabled answers are actually required, we could record the substitutions we derive to avoid having to recompute them. Such an extension to the data type of answers is simple but we do not implement this optimisation.

The function *tabled_answers* : *old_tree* $\rightarrow$ *string list* $\rightarrow$ *substitution list* takes a tree and collects all its answers. The slight complication is that the variables in the domain of the answers returned are those of the tabled tree. Naturally, we wish to apply the answers to literals in a goal with different variable names. We cope with this requirement by passing a list of variables to *tabled_answers* that replace the variables in the domain of each answer.

```
let tabled_answers tree xs =
  let mk_sub xs chis =
    list_it unify__add (combine (xs, map snd chis)) idsub
  in
  map (mk_sub xs) (fresh (gather_answers tree)) ;;
```

The child of an OLD-tree must be updated whenever answers are returned from one of its subgoals. Given three substitutions (ϕ, θ, χ), *mk_child* (ϕ, θ, χ) : *child list* $\rightarrow$ (*forest list* * *substitution*) *list* creates a new child as follows: the substitution ϕ is the returned solution which is applied over each subgoal in the goal, θ is the existing substitution of the OLD-resolution for the child, and χ is the computed answer thus far in the current derivation. Therefore, the new substitution of the OLD-resolution is $\phi \circ \theta$ and the answer of each new subgoal is the composition $\phi \circ \theta \circ \chi$.

```
let mk_child (phi,theta,chi) goal =
  let chi' = compose phi (compose theta chi) in
  (map (fun subgoal → mk_tree chi' (apply phi subgoal.root.label)) goal,
   compose phi theta) ;;
```

In the following section, we present the procedure that embodies fair OLDT-resolution.

B.6 OLDT-resolution

We use the function *mk_root* : *term list* → *forest* to create a special OLD-tree, called the root, from a given query.

```
let mk_root ls =
  let l = it_list (fun x y → App (x,Var y)) (Const "#") !vars in
  let tree = mk_tree idsub l in
  tree.children ← [ (map (fun l → mk_forest (mk_tree idsub l)) ls, idsub) ];
  let root = mk_forest tree in sprout root; root ;;
```

So, for a query $\# :- A_1, \ldots, A_n$, where $n \geq 1$, we create the OLD-tree labelled with the special term $\# :- X_1 \; \ldots \; X_m$, where the variables $X_1, \ldots, X_m$, for $m \geq 0$, are the distinct variables of $A_1, \ldots, A_n$. We create this 'dummy' term so that we can avoid reporting duplicate solutions, as we shall see shortly.

From the root, we repeatedly traverse the tree, locating leaves and performing either program clause resolution or answer clause resolution. The latter case arises when a variant of the label of the current leaf OLD-tree already occurs in the table *complete_nodes*. In this instance, we immediately return all answers from the table to the goal and traverse any new leaves. In the former case, the current leaf does not appear in the table and our action depends on the state of its current child. If it has no children, then we check to see if it is completely evaluated. If so, we can enter it in the table and update the forest by returning answers to the appropriate goal. If it is not completely evaluated, then we must perform a resolution step with a previously untried program clause. In the case where there are no literals in the child, then we have found a refutation and we report the solution if it is not a duplicate. In the final case, we ensure that the data type invariant for the forest is maintained by sprouting the current forest.

In the case where the current leaf OLD-tree has already been completely evaluated, we take its computed answers from the table and return them to its containing goal, removing that literal from the new goal and applying each answer substitution over the remaining subgoals. The action taken depends on whether the completed leaf OLD-tree is the only member of the spine, in which case, we must update the parent forest, or update the previous tree on the spine.

```
let return_answers completed_tree = function
    Nothing → ()
  | Just (forest, (i,_)) →
      let tree = hd forest.spine in
      let (goal,theta) :: chrn = tree.children in
      let (subgoal,goal') = (nth_list (i−1) goal, drop_nth (i−1) goal) in
      let phis = tabled_answers completed_tree (set_of_vars subgoal.root.label) in
      tree.children ←
          map (fun phi → mk_child (phi,theta,tree.answer) goal') phis  @ chrn;
      sprout forest ;;
```

The side-effect of *return_answers* is to update the head node of the parent spine, ensuring that the data type invariant for the forest is maintained.

In the case where the parent is another OLD-tree on the spine, we delete the head of the spine in addition to returning the solutions.

```
let update_spine forest =
  let (child, tree) = (nth_list 0 forest.spine, nth_list 1 forest.spine) in
  let ([ _ ],theta) :: chrn = tree.children in
  let thetas = map (fun x → (x, apply theta (Var x))) (set_of_vars tree.label) in
  forest.spine ← tl forest.spine;
  tree.answers.list ← (thetas, Just child) :: tree.answers.list;
  tree.children ← chrn ;;
```

The spine itself traces the path of the trail and the answer we insert in *tree* depends on the answers for *child*. Therefore, we create a link in the trail that will allow us to recover the entire answers for *tree* at a later date.

The function *resolve* : *term list* → *unit* takes a goal, converts it into the special root OLD-tree, and performs fair OLDT-resolution on it until completely evaluated. The condition for this is given by the function *completely_evaluated* : *old_tree* → *bool*, defined as

```
let completely_evaluated tree =
  let i = tree.program_clauses in
  tree.children = [] && i <> list_length (clauses tree.label) ;;
```

where the function *clauses* : *term* → *clause list* returns the program clauses from the database for the predicate of the given term.

We commence the resolution by creating the root OLD-tree for the query, the *path* stack, and the *active_nodes* table. We repeat the traversal of the forest of OLD-trees until the root OLD-tree is completely evaluated.

```
let resolve ls =
  let root = new_root ls
  and path = stack__new () and active_nodes = table__new () in
  while not completely_evaluated root.first do
    let (parent,forest) = traverse path root active_nodes
    and finished = ref false in
```

If the root OLD-tree is not completely evaluated, there must be at least one leaf node in the tree. Once we have located the first unexamined leaf in the inorder traversal, we may be able to perform several computation steps. We introduce another **while** loop that terminates when the current leaf OLD-tree has been fully processed. The first step in the loop is to check whether the leaf OLD-tree has already been evaluated.

```
    while not !finished do
      let leaf = hd forest.spine in
      let tabled = try Just (find (variant leaf.label) complete_table) with
        Not_found → Nothing in
```

If *leaf* has been previously evaluated, the action taken depends on whether its parent OLD-tree is contained on *forest.spine* or in the *parent* structure returned by *traverse*. The latter case arises when *forest.spine* contains only *leaf*. Here, *return_answers* returns all the answers from the completed OLD-tree to the goal containing *leaf* as a subgoal. This operation updates the goal, providing new subgoals to traverse. Therefore, we terminate the inner loop and traverse the new subgoals in the updated goal.

```
      if tabled <> Nothing && parent <> Nothing && singleton forest.spine
      then
        let (Just tabled_tree) = tabled in
        return_answers tabled_tree parent; finished := true
```

In the case where other OLD-trees occupy *forest.spine*, we remove *leaf* from the spine and return its tabled answers to its parent tree. Since the parent tree is on the spine, we know that there will only be one subgoal in its current goal, namely *leaf*. We re-use *return_answers*, ensuring that the invariant pertaining to the second argument is satisfied.

```
else if tabled <> Nothing && not singleton forest.spine then
  let (Just tabled_tree) = tabled in
  forest.spine ← tl forest.spine;
  return_answers tabled_tree (Just (forest,(1,1)))
```

Returning solutions in this case may result in a refutation for the spine, the effect of which is captured below. Therefore, we iterate round the loop again with *forest* updated.

Should *leaf* not be previously evaluated, we examine its current child. If *leaf* has no children then we begin by checking whether it is completely evaluated. If so, we add the label of *leaf* to *complete_nodes* and, again, distinguish the cases where *leaf* is the only member of *forest.spine* and where other trees exist on the spine. In the latter case, we use *update_spine* to ensure that the trail of answers for the spine is maintained correctly.

```
else match leaf.children with
  [] →
    if completely_evaluated leaf then begin
      table__add (variant leaf.label) leaf complete_nodes;
      if singleton forest.spine then begin
        return_answers leaf parent; finished := true
      end else update_spine forest
```

In the case of *leaf* having program clauses left to resolve with, we perform a resolution step with a previously untried clause. The function *new_clause* : *old_tree* → *clause* returns such a clause from the database.

```
    end else begin
      try
        let (l,ls) = new_clause leaf in
        let phi = unify leaf.label l in
        let chi = compose phi leaf.answer in
        leaf.children ← [ map (fun l → mk_tree chi (apply phi l)) ls, phi ]
      with No_unifier → ()
    end
```

A resolution step may result in the immediate subrefutation of a literal. This situation is identified by the current goal of *leaf* being empty. Here, we insert the substitution of the OLD-resolution into the answers of the OLD-tree and, furthermore, report the solution if appropriate. Explicitly, we report the solution if *leaf* is, in fact, the root OLD-tree and the solution is not a duplicate, or if the first OLD-tree on *forest.spine* is the root and the current answer of the OLD-refutation is not a duplicate.

```
| ([],theta) :: chrn →
    let inserted = insert_answer theta leaf in
    leaf.children ← chrn;
    if parent = Nothing then
      let chi = compose theta leaf.answer in
      if (leaf == root.first && inserted) || insert_answer chi root.first
      then report_solution chi
```

The definition of *report_solution* : *substitution* → *unit* is omitted.

The final case for the inner loop simply ensures that the data type invariant for *forest* is maintained.

```
      | (ls,_) :: chrn → sprout forest; finished := true
    done
  done ;;
```

This completes the implementation of OLDT-resolution.

Appendix C

Implementing prioritised fair SLD-resolution

In this appendix, we present an implemention of prioritised fair SLD-resolution. The implementation is intricate owing to the plethora of graph algorithms used and, therefore, it is instructive to present the actual algorithms in full. A similar structure to that of Chapter 5 is followed by first introducing symbolic norms and then implementing instantiation analysis. Following this, the pivotal data structure of a graph is given from which the definition of a mapping is obtained. Next, constraint analysis is implemented followed by the construction of query-mapping pairs. After all these components are in place, the implementation of the termination test ends the appendix.

C.1 Symbolic norms

A symbolic norm is a polynomial expression comprising an integer constant and a number of variables with integer coefficients. The size of a symbolic norm for a term is proportional to the number of distinct variables in that term. Since the number of distinct variables in a term is likely to be small, a symbolic norm is implemented as an association list of variable/coefficient pairs. A variable is a string and a coefficient is an integer. The type definition is as follows.

```
type symbolic_norm == (string * int) list ;;
```

The empty string is conveniently associated with the constant term since the empty variable cannot occur in a program. For example, the symbolic norm $2+X+3Y$ corresponds to the Caml association list [("",2) ; ("X",1) ; ("Y",3)].

The data type invariant of a symbolic norm is that pairs with a zero coefficient do no appear in the association list and each variable in the domain of the list occurs in only one pair. Consequently, the empty association list represents the symbolic norm 0. Since they are used regularly during the construction of arcs in a mapping, definitions of the symbolic norms 0 and 1 are given below to allow the abstraction over their actual representation.

```
let norm_zero = [] and norm_one = [ "",1 ] ;;
```

When traversing a path in a mapping, a common operation is to sum or subtract the symbolic norms of each node in the path. These simple arithmetic operations are implemented by the function *combine_norms* : $(int \rightarrow int \rightarrow int) \rightarrow symbolic\_norm \rightarrow$

symbolic_norm → *symbolic_norm*. The first argument to *combine_norms*, which may be **prefix** + or **prefix** −, allows the respective addition or subtraction of its two arguments. The data type invariant for symbolic norms is enforced by *combine_norms* such that each string occurs in only one pair in the list and that zero-valued associations are deleted.

```
let combine_norms f ms ns =
  let combine ms (s,n) =
    try let m = assoc s ms in m := f !m n; ms with
      Not_found → (s,ref (f 0 n)) :: ms
  in
  let add_ref = map (fun (s,n) → (s,ref n)) in
  let del_ref = map (fun (s,ref n) → (s,n)) in
  let rem_zeros = filter (fun (_,0) → false | _ → true) in
  rem_zeros (del_ref (it_list combine (add_ref ms) ns)) ;;
```

For reasons of efficiency, references play a crucial role in *combine_norms* to update a variable's coefficient without alteration to the structure of the list. The final step is to remove association pairs with zero coefficients from the list. Both of these steps ensure the data type invariant is maintained.

The function *mk_norm* : *string* * *int list* → *symbolic_norm* converts a list of variable/integer pairs to a symbolic norm. Although association lists are likely to be a reasonably efficient representation of symbolic norms, the provision of *mk_norm* facilitates the abstraction over the actual representation of symbolic norms. For example, an alternative association representation, like a hash table, could be adopted later to improve efficiency.

```
let mk_norm = combine_norms (prefix +) norm_zero ;;
```

The call to *combine_norms* in the body of *mk_norm* ensures that the data type invariant is satisfied for the newly created symbolic norm.

Common operations on a symbolic norm are those that test whether it is zero, positive, negative, or an integer constant, each of type *symbolic_norm* → *bool*. The following functions provide this information:

```
let zero_norm = prefix = norm_zero ;;
let integer_norm = fun [ "",_ ] → true | ns → zero_norm ns ;;

let cmp_norm cmp ns =
  not zero_norm ns && for_all (fun (s,n) → cmp n 0) ns &&
  exists (fun ("",_) → true | _ → false) ns ;;

let positive_norm = cmp_norm (prefix >)
and negative_norm = cmp_norm (prefix <) ;;
```

The functions above break the data abstraction for symbolic norms since equality is most efficiently defined in terms of the actual structure of the association list. Also, let us mention in passing that the comparisons made by *cmp_norm* are likely to be reasonably efficient since the Caml logical operators '&&' and '||', used in the definitions of the library functions *for_all* and *exists*, are non-strict.

Recall from Definition 5.2 that a *norm* is a function mapping terms to symbolic norms which corresponds to the following type synonym:

```
type norm == term → symbolic_norm ;;
```

The function *termsize_norm* : *norm* calculates the term-size norm, from Example 5.1, for a given term. Recall that the term-size norm associates a value to each constructor equal to the number of arguments it has; for an arbitrary term, the number of its arguments can be determined from its 'spine' representation which is obtained using the function *spine* : *term* → *term list*:

```
let rec spine = function
  App (f,g) → spine f @ [g]
| f → [f] ;;
```

The definition of *termsize_norm* is as follows:

```
let termsize_norm t =
  let rec normify (e :: es) =
    (match e with Const c → ("",list_length es) | Var x → (x,1)) ::
    flat_map (fun e → normify (spine e)) es
  in
  mk_norm (normify (spine t)) ;;
```

The function *ie* : *norm* → *term* → *bool* takes a symbolic norm and a term, and determines whether the term is instantiated enough with respect to the norm. From Definition 5.3, a term is instantiated enough if its norm is an integer:

```
let ie norm t = integer_norm (norm t) ;;
```

The functions above capture the main operations on norms that are used in the remainder of the section. Next, an implementation of instantiation analysis is presented.

C.2 Instantiation analysis

The programs of this section are introduced following the notation of its counterpart, Section 5.6.2. Before giving an implementation of the abstraction function α for terms, necessary type definitions are given to describe the abstract forms of terms and clauses. The first of these is instantiation patterns (Definition 5.4) which essentially provide an account of whether or not each argument in a term is instantiated enough.

```
type instantiation_pattern == bool list;;
```

The value *true* corresponds to an argument being instantiated enough, i.e., the special term *ie*, and *false* otherwise, i.e., the special term *nie*. Given two instantiation patterns, a useful operation is to determine whether the first pattern subsumes the second, given in Definition 5.6. The implementation of *subsumes* : *instantiation_pattern* → *instantation_pattern* → *bool* is as follows:

```
let subsumes ip ip' =
  let subsume = fun (false, true) → false | _ → true in
  for_all subsume (combine (ip,ip')) ;;
```

This function is used frequently in the implementation of termination analysis.

An abstract form of a literal, with respect to some norm, is implemented as a tuple whose first component is the predicate name and whose second is the instantiation pattern of its arguments.

```
type abs_term == string * instantiation_pattern ;;
```

Therefore, the tokens *ie* and *nie* from Definition 5.5 are replaced by *true* and *false*, respectively. The abstraction function α for terms is implemented by *alpha* : *norm* $\rightarrow$ *term* $\rightarrow$ *abs_term* which takes a norm and a literal, and converts the literal into its abstract form by applying the function *ie* to each of its arguments. The spine representation of the literal provides a simple method of accessing each argument.

```
let alpha norm t = let (Const p :: es) = spine t in (p, map (ie norm) es) ;;
```

The instantiation analysis of a predicate proceeds by generating every special instance of its program clauses. For a clause that contains n distinct variables, recall from Section 5.6.2 that 2^n instances of the clause are constructed by either substituting each variable with the special token *ie* or by leaving the variable unchanged. The function *instances* : *clause* $\rightarrow$ *clause list* builds a list of 2^n substitutions that performs this task; we use the special constant term *Const* "_ie_" (that cannot occur naturally in the program) to represent the abstract token *ie*. After creation of the list of substitutions, each one is applied over the original clause to create the abstract clauses. The definition of *instances* is given below.

```
let instances (l,ls) =
  let xs = nub (flat_map set_of_vars (l::ls)) in
  let phis xs =
    let ie_term = Const "_ie_" in
    let f x phis = phis @ (map (add (x, Const "_ie_")) phis) in
    list_it f xs [ idsub ]
  in
  map (fun phi → (apply phi l, map (apply phi) ls)) (phis xs) ;;
```

So, *instances* is used to create the special instances of a clause and, according to Section 5.6.2, each instance is transformed into an abstract clause by applying *alpha* to each literal in it. The type of an abstract clause is

```
type abs_clause == abs_term * abs_term list ;;
```

The function *abstract_cl* : *clause* $\rightarrow$ *abs_clause* takes a clause and transforms it into its abstract form, as follows:

```
let abstract_cl norm (l,ls) = (alpha norm l, map (alpha norm) ls) ;;
```

The following example demonstrates how the above programs correspond to Example 5.5.

Example C.1. Suppose the following instance of the second program clause of *append* has been created by the function *instances*.

$$append\ (A : ie)\ ie\ (A : ie)\ :-\ append\ ie\ ie\ ie.$$

The abstract clause for this instance is obtained by an application of *abstract_cl* producing

$$append\ nie\ ie\ nie\ :-\ append\ ie\ ie\ ie.$$

That is, (("*append*", [*false*; *true*; *false*]), [("*append*", [*true*; *true*; *true*])]) in the Caml-light implementation where the boolean values *true* and *false* correpond to the special tokens *ie* and *nie*, respectively. ♢

Instantiation analysis is a bottom-up process that examines the literals in the body of a clause in order to infer instantiation patterns for its head. Therefore, when analysing a given predicate, it is advantageous to perform instantiation analysis first on the predicates that preceed it in the topological ordering of the predicate dependency graph of the program. By doing so, each predicate in the body of a clause can be assumed to have been previously analysed. The definition of dependency analysis is omitted and we assume that instantiation analysis is performed according to this order.

During the instantiation analysis, the instantiation patterns inferred so far for a predicate are constantly accessed. For reasons of efficiency, this information is recorded in a hash table that takes predicates to their inferred instantiation patterns. Indeed, a substantial quantity of information will be inferred for each predicate during the course of termination analysis. So, rather than maintaining multiple incarnations of such tables, a single hash table is generalised to record all of a predicate's pertinent information. The following type definition stores this information:

```
type predicate_data =
  { mutable ips : instantiation_pattern list;
  } ;;
```

The field *ips* records the instantiation patterns determined for a predicate by instantiation analaysis.

The hash table *predicate_database* : (*string*, *predicate_data*) *hashtbl__t*, whose definition is given below, maps predicates to their associated data:

```
let predicate_database = hashtbl__new 997 ;;
```

The value 997 denotes the initial size of the hash table. The accessor functions for this table are

```
let find_ips = find_data (fun data → data.ips) ;;
let add_ips hashtbl c ips =
  let data = find hashtbl c in data.ips ← nub (data.ips @ ips) ;;
```

The *predicate_data* type will be enhanced with extra fields as the need arises later in the section.

Instantiation analysis first analyses the facts of a predicate—a fact contains no literals in its body—to determine the initial bottom-up instantiation patterns. Therefore, a predicate's abstract clauses are seperated into facts and the remaining clauses. The function *facts_cls* : *abs_clause list* → *abs_clause list* * *abs_clause list* performs this task:

```
let facts_cls =
  let oplus (cls1,cls2) cl =
    if fact cl then (cl :: cls1, cls2) else (cls1, cl :: cls2)
  in it_list oplus ([],[]) ;;
```

Given an abstract clause, the instantiation pattern of its head is inferred only if the pattern of each literal in its body has already been inferred. The function *analyse* : *abs_clause* $\rightarrow$ *instantiation_pattern list* takes a clause and returns the singleton list containing the instantiation pattern of its head if appropriate, otherwise it returns the empty list.

```
let analyse ((c,ip),ips) =
  if for_all (fun (c',ip') → mem ip' (find_ips predicate_database c')) ips
  then [ip] else [] ;;
```

The function *instantiation_analysis* : *norm* $\rightarrow$ *string* $\rightarrow$ *unit* takes a norm and the name of a predicate, and performs instantiation analysis of the predicate. The side effect of *instantiation_analysis* is to update the global hash table *predicate_database* with the computed instantiation patterns.

```
let instantiation_analysis norm c =
  let (facts,cls) = facts_cls (flat_map instances (clauses c)) in
  let abs_facts = map (abstract_cl norm) facts
  and abs_cls   = map (abstract_cl norm) cls in
  let (cls',pending) = (ref abs_cls,ref abs_cls) in
  add_ips predicate_database c (flat_map analyse abs_facts);
  while !pending <> [] do
    match analyse (hd !pending) with
      [] → pending := tl !pending
    | [ip] →
        add_ips predicate_database c [ip];
        cls' := exceptq (hd !pending) !cls';
        pending := !cls'
  done ;;
```

The instantiation patterns inferred for a predicate are retrieved frequently, and the hash table *predicate_database* provides constant time access to this information. In the next section, the fundamental graph data structure, forming the backbone of termination analysis, is presented.

C.3 Graphs

The pervasive data structure in termination analysis is the mapping (Definition 5.10). The majority of the computational effort expended during termination analysis involves the manipulation of mappings. Therefore, an efficient representation of a mapping must be selected, tailored to the particular requirements of termination analysis. In terms of computer representation, a mapping is essentially a *directed weighted graph*.

Abstractly, a directed, weighted graph G is a set of *vertices* V and a set of *arcs* A with three functions: *src* : $A \rightarrow V$ and *tgt* : $A \rightarrow V$ that identify the source and target of an arc, respectively; and *wgt* : $A \rightarrow \mathsf{N}$, where N is the set of natural numbers, which returns the weight of an arc. For each vertex $v \in V$, there is an associated set $A_v \subseteq A$ such that $A_v = \{a \mid a \in A \wedge src\ a = v\}$. The definition a directed, weighted graph can be reformulated to make explicit the connection between a vertex and the arcs that leave it. Given the same V and A, the set $\{(v, A_v) \mid v \in V\}$ of *adjacency pairs* is defined, each pair associating a vertex v with the arcs having v as their source. A graph is implemented as this set of adjacency pairs, using an efficient representation of the set.

The type of an arc in Caml is parametrised by both the type of the target and the type of the weight of the arc:

```
type ('a,'b) arc =
    { tgt : 'a;
      weight : 'b
    } ;;
```

Parametrising the data type in this manner provides the flexibility to adopt different representations of the target of the arc. For example, the target of an arc could be a unique identifier that facilitated efficient table lookup of the corresponding vertex, or a reference to the actual location in memory of the vertex. An arc is constructed by the function *mk_arc* : 'a → 'b → ('a,'b) *arc*:

```
let mk_arc id ns = { tgt = id; weight = ns } ;;
```

An adjacency pair is given by the type synonym

```
type ('a,'b) adj == 'a * 'b list ;;
```

again parametrised by the types of a vertex and an arc. The various components of an adjacency pair are accessed using the functions *adj_vertex* : ('a,'b) *adj* → 'a and *adj_list*: ('a,'b) *adj* → 'b, defined simply as

```
let adj_vertex = fst and adj_list = snd ;;
```

A graph is specified by functions *mk_graph* : ('a,'b) *adj list* → ('a,'b) *graph* and its inverse *graph_adjs* : ('a,'b) *graph* → ('a,'b) *adj list* such that the following identity holds:

$$mk\_graph \cdot graph\_adjs = id. \tag{C.1}$$

The function *graph_adj* : *int* → ('a,'b) *graph* → ('a,'b) *adj* selects an adjacency pair from a graph G according to the following specification:

$$graph\_adj\ n\ G = nth\_list\ n\ (graph\_adjs\ G). \tag{C.2}$$

The set of adjacency pairs is implemented as a *vector*—the Caml terminology for an array—of such pairs:

```
type ('a,'b) graph == ('a,'b) adj vect ;;
```

Conveniently, the specifications of (C.1) and (C.2) are easily satisfied by the following function definitions:

```
let mk_graph = vect_of_list
and graph_adjs = list_of_vect
and graph_adj n graph = graph.(n) ;;
```

The implementation of a graph as a vector suggests a refinement to the set of vertices $V = \{v_0, \ldots, v_{n-1}\}$ of a graph. The following set is constructed that associates a unique integer i with each vertex in V:

$$V' = \{(v_i, i) \mid v_i \in V \wedge 0 \leq i < n\},$$

Unsurprisingly, the following equivalence is identified:

$$v_i \equiv adj\_vertex\ (graph\_adj\ i\ G) \tag{C.3}$$

Operationally, a vertex can be looked-up in constant time using *graph_adj* and *adj_vertex*. In the following section, graphs are used to implement mappings.

C.4 Mappings

Before giving the type of a mapping, let us settle on the type of a node. Recall from Definition 5.10 that a node consists of a label (a term) and whether or not it is instantiated enough with respect to some norm. From an implementation point of view, recording the symbolic norm of the label and the name of the associated predicate is useful. The refinement to the graph data type, introduced in (C.3), associates a unique integer to each vertex to facilitate their efficient lookup. Consequently, a node also contains this identifier. The type definition of a node is as follows:

```
type node_id == int;;

type node =
    { id : node_id;
      name : term;
      label : term;
      norm : symbolic_norm;
      mutable bound : bool
    } ;;
```

A node is constructed by the function *mk_node* : *node_id* $\rightarrow$ *term* $\rightarrow$ *term* $\rightarrow$ *node*, defined below.

```
let mk_node k u t =
  { id = k;
    name = u;
    label = t;
    bound = ie termsize_norm t;
    norm = termsize_norm t
  } ;;
```

To simplify the implementation, the term-size norm is favoured in the remainder of the section. However, the use of other norms in termination analysis is discussed further in Section 7.2.

Abstractly, a mapping is a graph whose vertices are nodes and whose arcs have a norm as their weight and a node as their target. However, since each node is uniquely identified according to (C.3), the target of an arc in a mapping is the unique identifier of the actual node rather than the node itself:

```
type mapping_arc == (node_id, symbolic_norm) arc ;;
```

In addition to efficient access of vertices, using an integer identifier means that copying arcs becomes a matter of copying integers rather than nodes.

Each subgoal in the body of a clause has a corresponding list of nodes in its mapping representation. Specifically, consider a clause A :− $B_1, \ldots, B_n$, for $n \geq 1$ and suppose some subgoal $B_i = f\ t_1\ \ldots\ t_m$, where $m \geq 0$, f is a constant, and $t_1, \ldots, t_m$ are terms. The nodes corresponding to $t_1, \ldots, t_m$ form the subgoal B_i in the mapping. In the implementation, individual subgoals are accessed regularly to create query-mapping pairs, infer constraints, and so on. However, the vector implementation of a graph does not provide enough structure to record this information. Therefore, a mapping is augmented with a list *subgoals* : *node list list* such that, for some i and j, the node corresponding to argument t_j of B_i can be obtained by *nth_list* j (*nth_list* i *subgoals*).

So, a mapping is a graph coupled with the list *subgoals*. In fact, the type of *subgoals* can be refined to that of lists of node identifiers rather than nodes themselves. Then the type definition of a mapping becomes

```
type mapping =
    { adjs : (node, mapping_arc) graph;
      subgoals : (node_id list) list
    } ;;
```

The various parts of a mapping are accessed using the functions

```
let mapping_adjs mapping = graph_adjs mapping.adjs ;;
let mapping_adj n mapping = graph_adj n mapping.adjs ;;
let mapping_adj_id v = (adj_vertex v).id ;;
```

The type *subgoal* is identified as a list of adjacency pairs which corresponds to a subgoal in the mapping:

```
type subgoal == (node, mapping_arc) adj list ;;
```

The function *mapping_subgoal* : *int* → *mapping* → *subgoal* returns the appropriate subgoal in the mapping and is defined below.

```
let mapping_subgoal n mapping =
  map (fun k → mapping_adj k mapping) (nth_list n mapping.subgoals) ;;
```

The function *mk_mapping* : *clause* → *mapping* converts a clause into its representation as a mapping. The process is relatively intricate and we proceed towards its implementation in stages. First, the adjacency pairs for each subgoal in the clause must be created. The function *l2adjs* : *node_id* → *term* → *subgoal* converts a literal into a list of adjacency pairs, where each pair initially has an empty adjacency list. The first argument to *l2adjs* is the initial identifier from which to label the successive nodes of the literal. The definition of *l2adjs* is given below.

```
let l2adjs n l =
  let (e :: es) = spine l in
  fst (list_it (fun e' (adjs,i) → (mk_adj (mk_node i e e') :: adjs, i − 1))
         es ([], n + list_length es − 1)) ;;
```

Given a clause $A :- B_1, \ldots, B_n$, the next step is to construct the sequence of subgoals $[A, B_1, \ldots, B_n]$, converting each one into a list of adjacency pairs using *l2adjs*. The complication is to ensure that the identifier for each node satisfies the invariant for a graph. The function *mk_adjs* : *term list* → *subgoal list* performs this transformation iteratively.

```
let mk_adjs ls =
  fst (it_list (fun (adjss,n) l →
    let adjs = l2adjs n l in (adjss @ [adjs], n + (list_length adjs)))
         ([],0) ls) ;;
```

The adjacency list of each pair is initially empty and must be updated with the appropriate arcs (Definition 5.10). The function *connect* : (*node*, *mapping_arc*) *adj list* → (*node*, *mapping_arc*) *adj list* takes the list of adjacency pairs, examining each one in turn, and inserts arcs into any node that has a comparable norm to any other node.

```
let connect adjs =
  let visit adjs adj =
    new_adj (adj_vertex adj)
      (flat_map (fun adj' →
        let (w,v) = (adj_vertex adj, adj_vertex adj') in
        let ns = combine_norms (prefix −) w.norm v.norm in
        if mapping_adj_id adj' <> mapping_adj_id adj && accept_norm ns
        then [ mk_arc v.id ns ] else [])
         adjs)
  in
  map (visit adjs) adjs;;
```

The function *mk_mapping* : *clause* → *mapping* builds a mapping by translating the clause into a list of subgoals, from which the list *subgoals* can be obtained. The next step is to make the graph from the adjacency pairs:

```
let mk_mapping (l,ls) =
  let adjs = mk_adjs (l :: ls) in
  let kss = map (map mapping_adj_id) adjs in
  new_mapping kss (mk_graph (connect (concat adjs))) ;;
```

In the remainder of this section, implementations are presented for the various operations on mappings, like inferring new arcs and bindings. Many of these operations share a similar traversal pattern of a mapping: each node is visited in turn and its adjacency list is traversed with some operation performed of the target nodes. Therefore, generic algorithms are given when possible.

The function *insert_arc* : *node_id* → *mapping_arc* → *mapping* → *unit* inserts an arc into the identified adjacency pair of the mapping. The arc is only inserted if it does not already exist in the adjacency list. The side-effect of *insert_arc* is to update physically the mapping argument.

```
let insert_arc k arc mapping =
  let adj = mapping_adj k mapping in
  let duplicate_arc = exists (fun arc' → arc.tgt = arc'.tgt) (adj_list adj) in
  if not exists (fun arc' → arc.tgt = arc'.tgt) (adj_list adj) then
    update_adj k (mapping.adjs) (new_adj (adj_vertex adj) (arc::adj_list adj)) ;;
```

The generic way of traversing a mapping is to visit each adjacency pair in sequence and, for each pair, traverse each arc in the adjacency list, performing some operation on the target node of the arc. The function *do_mapping f* : *mapping* → *unit* implements the generic traversal above, applying the function *f* : *mapping_arc list ref* → *mapping* → *node_id* → *mapping_arc* → *unit* to each applicable node. The first argument to *f* is a reference to the list of arcs yet to be examined; the second argument is the mapping being traversed; the third argument is the identifier of the current node being visited; and the final argument is the arc currently being examining. The definition of *do_mapping* is given below.

```
let do_mapping f mapping =
  do_list (fun root →
    let pending = ref (adj_list root)
    and visited = ref [mapping_adj_id root] in
    while !pending <> [] do
```

```
      let arc = hd !pending in
      pending := tl !pending;
      if not mem arc.tgt !visited then begin
        visited := arc.tgt :: !visited;
        f pending mapping (mapping_adj_id root) arc
      end
    done) (mapping_adjs mapping) ;;
```

The function *do_mapping* physically updates its mapping argument and is used to infer arcs and bindings for a mapping, and also to test whether a positive-weight cycle exists in a mapping.

Inferring arcs is similar to taking the transitive closure of a mapping except Definition 5.12 determines whether two nodes are connected. In particular, an arc can be inserted between two nodes connected by either a zero-weight path if no non-zero weight arc exists in the path, or a positive-weight path. Given a source node and an arc that leaves it, the target of the arc is first considered. For each arc that leaves that node, an arc is inferred from the source node to the new target node, if appropriate. An inferred arc must itself be traversed so the pending list is reset on each iteration of the loop; the function *do_mapping* ensures that revisiting nodes is avoided. The definition of *infer_arcs* is given below.

```
let infer_arc pending mapping src arc =
  do_list (fun arc' →
    if arc'.tgt <> src then
      let ns = combine_norms (prefix +) arc.weight arc'.weight in
      let new_arc = mk_arc arc'.tgt ns in
      let unlabelled_arc = positive_norm arc.weight || positive_norm arc'.weight in
      if (zero_norm ns && not unlabelled_arc) || positive_norm ns then
        insert_arc src new_arc mapping)
    (adj_list (mapping_adj arc.tgt mapping));
  pending := adj_list (mapping_adj src mapping) ;;
```

We use *do_mapping* to define the function *infer_arcs* : *mapping* → *unit* that infers all possible arcs in a mapping:

```
let infer_arcs = do_mapping infer_arc ;;
```

Given a node in a mapping and an arc that has that node as its source, the following function is used to infer whether either the source or target node is instantiated enough, according to Definition 5.14:

```
let infer_binding _ mapping src arc =
  let src_node = adj_vertex (mapping_adj src mapping)
  and tgt_node = adj_vertex (mapping_adj arc.tgt mapping) in
  if src_node.bound || tgt_node.bound then
    if zero_norm arc.weight then
      (src_node.bound ← true; tgt_node.bound ← true)
    else if positive_norm arc.weight then
      tgt_node.bound ← true ;;
```

An application of *do_mapping* allows all bindings in a mapping to be inferred.

```
let infer_bindings = do_mapping infer_binding ;;
```

In a similar vein, a subgoal in a mapping may be instantiated with some instantiation pattern. The function *instantiate* : *instantiation_pattern* → *int* → *mapping* → *unit* takes the index of a subgoal in a mapping and carries out this operation.

```
let instantiate ip k mapping =
  let instantiate_node = fun (true, node) → node.bound ← true | _ → () in
  let nodes = map adj_vertex (mapping_subgoal k mapping) in
  do_list_combine instantiate_node (ip, nodes); infer_bindings mapping ;;
```

The final operation is to determine whether or not a mapping is consistent, i.e., contains a positive-weight cycle. Such mappings cannot occur in the SLD-tree and, therefore, such mappings can be discarded. We make use of Caml-light exceptions to abort further computation in the event of a positive-weight cycle being discovered in a mapping. We declare the exception

```
exception Inconsistent ;;
```

For a mapping with all inferred arcs, a cycle exists in the mapping if, from any node, the target of an arc leaving that node itself contains an arc to the original node. Moreover, the cycle has a positive weight if the sum of the weights of the two arcs in question are of positive weight.

```
let positive_cycle _ mapping src arc =
  do_list (fun arc' →
    if arc'.tgt = src &&
      positive_norm (combine_norms (prefix +) arc.weight arc'.weight)
    then raise Inconsistent)
    (adj_list (mapping_adj arc.tgt mapping)) ;;
```

The function *consistent* : *mapping* → *bool* then determines whether a given mapping contains a positive cycle:

```
let consistent mapping =
  try do_mapping positive_cycle mapping; true with Inconsistent → false ;;
```

In this section, the machinery to create and manipulate mappings according to the important definitions of Section 5.6.3 has been presented. Mappings are pervasive in the remainder of the implementation and the functions above are used frequently in the remainder of the implementation. In the following section, mappings are used to infer the constraints between arguments in a predicate.

C.5 Constraint analysis

Constraint analysis is an involved process that is implemented using backtracking. The difficulty stems from computing the fixed-point of the immediate consequence operator in Section 5.6.4. Therefore, as new constraints are inferred, previous clauses must be reconsidered since any new constraint may allow further ones to be inferred from the old clauses. As usual, we begin by introducing the relevant data types.

Following Definition 5.15, a constraint is a subgoal with arcs restricted to only those between the nodes of the subgoal:

```
type constraint == subgoal ;;
```

The data type invariant for a constraint is enforced by restricting the arcs that appear in an adjacency list to those with particular targets. The function *restrict_adjs* : *bool* → *mapping* → *node_id list* → *subgoal* → *subgoal* carries out this task. The boolean argument to *restrict_adjs* indicates whether positive-weight arcs that do not connect two black nodes should be deleted, a requirement for constructing a summary mapping (Definition 5.18). The remaining arguments are: the mapping whose adjacency pairs are being restricted; a list of node identifiers that are allowable targets of arcs; and the list of adjacency pairs to be restricted. The definition of *restrict_adjs* is given below.

```
let restrict_adjs constraint mapping ks =
  map (fun adj →
    let arcs =
      it_list (fun arcs arc →
        let tgt_node = adj_vertex (mapping_adj arc.tgt mapping) in
        let src_bound = (adj_vertex adj).bound in
        if mem arc.tgt ks &&
          (zero_norm arc.weight ||
          positive_norm arc.weight && (src_bound || constraint))
        then
          (if zero_norm arc.weight then arc else mk_arc arc.tgt norm_one) ::
          arcs
        else arcs)
        [] (adj_list adj)
    in new_adj (adj_vertex adj) arcs) ;;
```

A common operation is to merge a constraint with a particular subgoal in a mapping, as discussed in Definition 5.13. Each adjacency pair in the constraint is merged with the corresponding adjacency pair in the subgoal, ensuring that arcs in the constraint correspond to arcs in the subgoal. In particular, the node identifiers of arcs in the constraint may not match those in the subgoal. The function *merge_adj* : *mapping* → (*node_id* * *node_id*) *list* → (*node_id*, *mapping_arc*) *adj* * (*node_id*, *mapping_arc*) *adj* → *unit* updates the given mapping by merging the two adjacency pairs. The second argument is a list of node identifier pairs that associates the identifiers of the nodes in the constraint to their corresponding nodes in the subgoal.

```
let merge_adj mapping new_ids (adj1,adj2) =
  let (node1, node2) = (adj_vertex adj1, adj_vertex adj2) in
  node2.bound ← node2.bound || node1.bound;
  do_list (fun arc →
    let arc' = mk_arc (assoc arc.tgt new_ids) arc.weight in
    insert_arc (mapping_adj_id adj2) arc' mapping)
    (adj_list adj1) ;;
```

The function *merge* : *subgoal* → *int* → *mapping* → *unit* utilises the function *merge_adj* to merge the given subgoal with the *k*th subgoal of the mapping. The mapping is physically updated by *merge*.

```
let merge subgoal k mapping =
  let subgoal' = mapping_subgoal k mapping in
  let new_ids =
    zipWith (fun (adj1,adj2) → (mapping_adj_id adj1, mapping_adj_id adj2))
      (subgoal,subgoal')
  in
  do_list_combine (merge_adj mapping new_ids) (subgoal,subgoal');
  infer_arcs mapping; infer_bindings mapping ;;
```

The function *mk_constraint* : *mapping* → *constraint* extracts the constraint, if any, from the head of the given mapping.

```
let mk_constraint mapping =
  let mapping' = copy_mapping mapping in
  let adjs = mapping_subgoal 0 mapping' in
  let ks = map mapping_adj_id adjs in
  do_list (fun adj → let node = adj_vertex adj in node.bound ← false) adjs;
  restrict_adjs true mapping' ks adjs ;;
```

A copy the mapping is made since each node is reset to 'white', i.e., not instantiated enough, following the procedure of Definition 5.15. The efficiency obtained in the implementation by using destructive updates is traded against the fact that a mutable structure cannot be safely shared since any changes are reflected in every reference to it. The boolean argument to *restrict_adjs* is *true* to keep arcs that do not connect pairs of black nodes.

At this point, the algorithm to compute the constraints for a given predicate can be presented. The complication is that all clauses for a predicate must be retried for each new constraint inferred, ensuring that at least one recursive occurrence of the predicate in the body of a clause receives the new constraint. This enterprise ensures that redundant computation is avoided. The function *recursive_subgoals* takes a predicate and one of its mappings, and returns the mapping coupled with a pair of integer lists; the first list gives the index of recursive calls to the predicate, and the second the remaining subgoals.

```
let recursive_subgoals c mapping =
  let (js_ks,_) =
    it_list (fun ((js,ks),n) subgoal →
      if c = subgoal_name (mapping_subgoal n mapping)
      then ((n::js,ks), n+1)
      else ((js,n::ks), n+1))  (([],[]),1) (tl mapping.subgoals)
  in
  (mapping, js_ks) ;;
```

Recall from Section 5.6.4 that the first step of constraint analysis is to infer constraints from the facts of a predicate; facts have no literals in their body and the immediate consequence operator begins with the empty set of constraints. The constraints are stored in the *predicate_database* hash table, by augmenting the type with

```
mutable cns : constraint list;
```

and the accessors

```
let find_cns = find cns ;;
let add_cns table c cns =
  let data = hashtbl__find table c in data.cns ← data.cns @ cns ;;
```

We initialise *predicate_database* for a given predicate using the function

```
let initialise_cns c =
  let facts = filter fact (clauses c) in
  let cns = map (fun cl → mk_constraint (mk_mapping cl)) facts in
  add_cns predicate_database c cns ;;
```

Every permutation of constraints is inserted into the subgoals of a mapping. In each case, if the resulting mapping is consistent, then the constraint for its head is inferred as a new constraint. Constraint analysis maintains a stack of mappings that require a constraint to be inserted for each subgoal. Formally, the stack contains tuples, each tuple having four components: (1) the current mapping; (2) the index of the current subgoal that requires insertion of a constraint; (3) the indices of the subgoals pending insertion of a constraint; and (4) the constraints yet to be inserted for the selected subgoal. For a stack *s* of tuples above, the function *push_constraints s* : *constraint list* → *mapping* * (*int list* * *int list*) → *unit* pushes onto *s* a mapping paired with the appropriate indices, such that the properties above are satisfied. In particular, if the mapping has no recursive subgoals, then the newly selected subgoal receives its constraints from *predicate_database*. Otherwise, for each recursive subgoal, the mapping is pushed onto *s* with that subgoal selected and using constraints from the first argument rather than from the table. By doing so, redundant computation of constraints is avoided. The definition of *push_constraints* follows.

```
let push_constraints stack new_cns = function
    (mapping, ([],k :: ks)) →
      let u = subgoal_name (mapping_subgoal k mapping) in
      stack__push
        (copy_mapping mapping,k,ks,find_cns predicate_database u) stack
  | (mapping, (js,ks)) →
      do_list (fun j →
        stack__push (copy_mapping mapping,j,except j (js@ks),new_cns) stack)
        js ;;
```

The function *constraint_analysis* : *string* → *unit*, shown in Figure C.1, computes the constraints for a predicate. The side-effect of *constraint_analysis* is to update the hash table *predicate_database* with the constraints inferred for the given predicate. The algorithm terminates when no new constraints are inferred for a predicate using the current constraints.

C.6 Query-mapping pairs

Recall from Definition 5.19 that a query-mapping pair consists of a query pattern and a summary mapping. Let us begin by first giving the type of a query pattern, from Definition 5.17:

```
type query_pattern == subgoal ;;
```

The function *mk_qp* : *int* → *mapping* → *query_pattern* takes the index of a subgoal and a mapping, and creates a query pattern for that subgoal.

```
let mk_qp k mapping =
  let subgoal = mapping_subgoal k mapping in
  restrict_adjs false mapping (map mapping_adj_id subgoal) subgoal ;;
```

```
let constraint_analysis c =
  let mappings = map (recursive_subgoals c) (find_mappings predicate_database c) in
  let aug_graphs = ref [] and new_cns = ref [] and finished = ref false in
  initialise_cns c;
  new_cns := find_cns predicate_database c;
  while not !finished do
    let pending = stack__new () in
    do_list (push_constraints pending !new_cns) mappings;
    new_cns := [];
    while not empty pending do
      match stack__pop pending with
        (_,_,_,[]) → ()
      | (mapping, k, ks, cn :: cns) →
          stack__push (copy_mapping mapping, k, ks, cns) pending;
          merge cn k mapping;
          if consistent mapping then
            match ks with
              [] →
                let new_cn = mk_constraint mapping in
                if not empty_cn new_cn && not exists (eq_cn new_cn)
                    (!new_cns @ find_cns predicate_database c)
                then new_cns := new_cn :: !new_cns
            | j :: js →
                let p = subgoal_name (mapping_subgoal j mapping) in
                stack__push
                  (copy_mapping mapping,j,js,find_cns predicate_database p) pending
    done;
    if !new_cns=[] then finished := true else add_cns predicate_database c !new_cns
  done ;;
```

Figure C.1: The function *constraint_analysis*.

The type of a query-mapping pair is as follows:

```
type qmpair =
   { query : query_pattern ;
     mapping : mapping
   } ;;
```

The two components of a query-mapping pair are accessed by the functions *qm_query* : *qmpair* → *query_pattern* and *qm_mapping* : *qmpair* → *mapping*. The mapping component of a query-mapping pair has both a range and domain, as defined in Definition 5.19. Each of these is accessed from the mapping using the functions *domain* : *mapping* → *subgoal* and *range* : *mapping* → *subgoal*. The implementations of these functions are given below.

```
let qm_query qm = qm.query and qm_mapping qm = qm.mapping ;;
let domain = mapping_subgoal 0 and range = mapping_subgoal 1 ;;
```

According to Definition 5.20, a query pattern is used to generate new mappings from which query-mapping pairs can be constructed. The function *generate_mappings* : *query_pattern* → *mapping list* performs the first of these tasks.

```
let generate_mappings qp =
  flat_map (fun mapping →
    let mapping' = copy_mapping mapping in
    try merge qp 0 mapping'; [ mapping' ] with Inconsistent → [])
    (find_mappings predicate_database (subgoal_name qp)) ;;
```

Each mapping returned by *generate_mappings* is consistent. For each such mapping, a query-mapping pair is constructed from it by forming the summary of its head and one subgoal. The function *mk_qmpair* : *int* → *mapping* → *qmpair* takes the index of a subgoal in the given mapping and returns a query-mapping pair by restricting the nodes of the mapping to those of only the head and indicated subgoal. The arcs from these nodes are restricted to only those between these nodes. Furthermore, non-zero arcs that connect nodes other than black ones deleted. The definition of *mk_qmpair* follows.

```
let mk_qmpair qp n mapping =
  let mapping' = copy_mapping mapping in
  let (d,r) = (mapping_subgoal 0 mapping', mapping_subgoal n mapping') in
  let ks = concat (map (map mapping_adj_id) [d; r]) in
  let m = list_length d in
  let n = mapping_adj_id (hd r) − m in
  let new_adjs = map (update_id (prefix −) m n) in
  let restrict = restrict_adjs false mapping' ks in
  new_qmpair qp (new_adjs (restrict d)) (new_adjs (restrict r)) ;;
```

The complication with *mk_qmpair* is that the range of the query-mapping pair, produced by *restrict_adjs*, cannot be converted to a mapping straight way. The reason is that, for some node v in the subgoal, *adj_vertex* (*nth_list* v.*id* *adjs*) $\neq v$ since the node identifiers in the range are unlikely to be valid. To reinstate the data type invariant for the mapping, the nodes in the range must have the correct identifiers and the targets of all arcs must be consistent with this. The arithmetic juggling is carried out by *update_id* : *node_id* → *int* → (*node*, *mapping_arc*) *adj* → (*node*, *mapping_arc*) *adj* which takes a node identifier i, an integer n, and an adjacency pair *adj*, and returns a new adjacency pair in which the node and the target of each arc in the adjacency list, with an identifier strictly greater than i, have n subtracted from it. This results in the definition:

```
let update_id f m n adj =
  let new_index m' = f m' n in
  let new_node =
    let node = adj_vertex adj in
    if node.id > m then begin
      let node' = mk_node (new_index node.id) node.name node.label in
      node'.bound ← node.bound;  node'
    end else node
  in
  let arcs = map (fun arc →
    if arc.tgt > m then mk_arc (new_index arc.tgt) arc.weight else arc)
      (adj_list adj) in
  new_adj new_node arcs ;;
```

As outlined Section 5.6, the composition of query-mapping pairs is the fundamental aspect of termination analysis. Two query-mapping pairs are composable (Definition 5.21) if the range of the first is identical to the query pattern of the second. This condition is embodied in the function *composable* : *qmpair* → *qmpair* → *bool*.

```
let composable qm1 qm2 =
  eq_qp (mk_qp 1 (qm_mapping qm1)) (qm_query qm2) ;;
```

The function *eq_qp* : *query_pattern* → *query_pattern* → *bool* tests equality of two query patterns and is defined simply as *eq_qp* = *eq_cn*.

Given two composable pairs, *compose_qm* : *qmpair* → *qmpair* → *qmpair* computes their composition. The idea behind *compose_qm* is to construct a new mapping by 'gluing' the two mapping parts of the query-mapping pairs together: zero-weight arcs are inserted between corresponding nodes in the range of the first pair and domain of the second. The function *glue* : *mapping* → *subgoal* * *subgoal* → *unit* carries out this task.

```
let glue mapping =
  do_list_combine (fun (adj1,adj2) →
    let (node1, node2) = (adj_vertex adj1, adj_vertex adj2) in
    insert_arc node1.id (mk_arc node2.id norm_zero) mapping;
    insert_arc node2.id (mk_arc node1.id norm_zero) mapping) ;;
```

The main complication when constructing the new mapping is to ensure that the data type invariant is maintained; the identifiers of the nodes and arcs in the second mapping will require updating. The function *compose_mappings* : *mapping* → *mapping* → *mapping* relabels all the nodes and arcs in the second mapping with appropriate identifiers:

```
let compose_mappings M1 M2 =
  let n = list_length (mapping_adjs M1) in
  let update_adj = update_id (prefix +) (−1) n in
  let adjss = map (map (fun k → update_adj (mapping_adj k M2))) M2.subgoals in
  let kss = M1.subgoals @ map (map mapping_adj_id) adjss in
  let M = new_mapping kss (mk_graph (mapping_adjs M1 @ concat adjss)) in
  glue M (mapping_subgoal 1 M, mapping_subgoal 2 M);
  infer_arcs M; infer_bindings M; M ;;
```

If the resulting mapping is consistent then a new query-mapping pair is generated with the last subgoal as its range. If the mapping is inconsistent, an exception is raised.

```
let compose_qm qm qm' =
  let (qm,qm') = (copy_qmpair qm, copy_qmpair qm') in
  let mapping = compose_mappings (qm_mapping qm) (qm_mapping qm') in
  if consistent mapping then
    mk_qmpair (qm_query qm) 3 mapping
  else raise Inconsistent ;;
```

The remaining operation is to check whether a query-mapping pair has a circular variant (Definition 5.22) and, if so, whether the variant has a forward positive cycle. The following exception is introduced to indicate that the circular variant does not have a forward positive cycle.

```
exception Non_termination of qmpair ;;
```

A variant is checked for a forward positive cycle using the following algorithm: zero-weight arcs are inserted between corresponding arguments in the variant and then all arcs are inferred. Naturally, the variant will likely be inconsistent so no check for consistency is made. The function *check_variant* : *qmpair* → *qmpair* is the identity on all query-mapping pairs other than those that have a circular variant without a forward positive cycle. In this case, the exception *Non_termination* is raised.

```
let check_variant qm =
  if eq_qp (qm_query qm) (mk_qp 1 (qm_mapping qm)) then
    let qm' = copy_qmpair qm in
    let M' = qm_mapping qm' in
    glue M' (domain M', range M');
    infer_arcs M';
    if forward_cycle qm' then qm else raise (Non_termination qm)
  else qm ;;
```

So, the variant has a forward positive cycle if it contains a positive-weight arc from the domain to the range, with no similar arc in the opposite direction. The following function *forward_cycle* : *qmpair* → *bool* makes this determination.

```
let forward_cycle qm =
  let M = qm_mapping qm in
  let positive_arc mapping src tgt =
    exists (fun arc → arc.tgt = tgt && positive_norm arc.weight)
      (adj_list (mapping_adj src M))
  in
  exists (fun (adj_d,adj_r) →
    let (node1,node2) = (adj_vertex adj_d, adj_vertex adj_r) in
    positive_arc M node1.id node2.id && not positive_arc M node2.id node1.id)
    (combine (domain M, range M)) ;;
```

All that remains is to provide an implementation of the termination test itself in the following section.

C.7 Termination analysis

In Chapter 5, we adapted the termination analysis of (Lindenstrauss & Sagiv 1997) to use a new computation rule and described its novel application to the automatic generation of a call set for prioritised fair SLD-resolution (Definition 5.9). In this appendix, we provide algorithms to implement the adapted termination test.

When performing termination analysis, it is desirable to follow the topological ordering of the predicate dependency graph of a program. By doing so, we can assume that each predicate in the body of a clause will have been previously analysed. Again, we update the *predicate_database* table, this time with the two fields

```
mutable tips : instantiation_pattern list;
mutable analysed : bool;
```

The job of *tips* is to record the terminating instantiation patterns and *analysed* to record whether the predicate has been analysed for all possible instantiation patterns.

The first stage in implementing the new termination test is to find a method of mimicking of the computation rule used in the actual prioritised SLD-resolution. The following few functions perform this duty. The function *terminates* : *subgoal* → *bool* takes a subgoal, and determines whether its instantiation pattern is known to terminate, using the information in *predicate_database*.

```
let terminates subgoal =
  mem (subgoal_ip subgoal) (find_tips predicate_database (subgoal_name subgoal)) ;;
```

The function *select_terminating* : *mapping* → *int list* → *int maybe* takes a mapping and a list of indices of subgoals not yet selected in the termination analysis of the mapping. The result of *select_terminating* is an index from its second argument of the first subgoal with a known terminating instantiation pattern. If no such literal can be selected, the result is *Nothing*.

```
let rec select_terminating mapping = function
    [] → Nothing
  | j :: js →
      if terminates (mapping_subgoal j mapping) then Just j
      else select_terminating mapping js ;;
```

The function *select_subgoal* : *mapping* * *int list* → *int maybe* takes a mapping and a list of indices of subgoals already considered in the analysis, and proceeds as follows. We find the indices *js* of subgoals yet to be considered. Using this list, we select the leftmost, known terminating subgoal according to *select_terminating*. If no such literal exists, we select the leftmost literal of a predicate that has yet to be completely analysed since the current termination pattern of it may terminate. This case deals with predicates that appear in the same strongly connected component of the predicate dependency graph. Finally, if we cannot select such a literal then we assume that the goal is floundered.

```
let select_subgoal (mapping,ks) =
  let js = subtract (enum 1 (list_length mapping.subgoals − 1)) ks in
  match select_terminating mapping js with
    Just k → Just (k,true)
  | Nothing →
      match select_leftmost mapping js with
        Nothing → Nothing
      | Just k → Just (k,false) ;;
```

The function *enum* : *int* → *int* → *int list* takes two integers m and n, and returns the list $[m, \ldots, n]$. We select the leftmost unexplored literal using the following function.

```
let rec select_leftmost mapping = function
    [] → Nothing
  | k :: ks →
      let u = subgoal_name (mapping_subgoal k mapping) in
      if not find_analysed predicate_database u then Just k
      else select_leftmost mapping ks ;;
```

Termination analysis maintains a stack of mappings from which to generate query-mapping pairs, corresponding to the exploration of each branch in the actual SLD-tree. The stack contains tuples, each with the following four components: (1) *mapping* :

mapping, the current mapping being considered in the termination test; (2) *js* : *int list*, the indices of the subgoals in *mapping* already identified as terminating; (3) *selected* : (*int* * *bool*) *maybe*, such that if *selected* is the value *Just* (*k,terminates*), then the currently selected subgoal is *mapping_subgoal k mapping*, and *terminates* is *true* if the subgoal is already known to terminate; and (4) *ips* : *instantiation_pattern list*, the untried instantiation patterns for the selected subgoal.

For a stack *s* of tuples above, *push_mappings s* : *mapping* * *int list* → *unit* pushes a mapping with its already analysed subgoals onto *s* such that the properties above are satisfied.

```
let push_mapping stack (mapping,qp,ks) =
  match select_subgoal (mapping,ks) with
    Nothing → stack__push (copy_mapping mapping,qp,ks,Nothing,[],[]) stack
  | (Just (k,terminates)) as selected →
      let subgoal = mapping_subgoal k mapping in
      let u = subgoal_name subgoal in
      let ips = subsuming_ips u subgoal in
      let cns = find_cns predicate_database u in
      stack__push (copy_mapping mapping,qp,ks,selected,cns,ips) stack ;;
```

Moreover, each constraint for the selected subgoal is inserted into the mapping before pushing it onto *s*; if there are no constraints—constraint analysis is an optional part of termination analysis—then the mapping itself is pushed onto the stack.

The crux of termination analysis is to generate all query-mapping pairs for a query and check that those with circular variants contain a forward positive cycle. Consequently, the majority of computation involves composing existing query-mapping pairs with new ones. In terms of efficiency, we maintain two hash tables, each taking predicates to query-mapping pairs; one table maps predicates to pairs whose query pattern is labelled by that predicate, and the second to pairs whose range is so labelled. Both tables are required to facilitate constant time access to the current set of query-mapping pairs since every possible composition must be considered. The sundry functions to access the tables are trivial and their definitions are omitted.

The function *termination_test* : *query_pattern* → *bool*, shown in Figure C.2, tries to determine whether the query pattern terminates. The test begins by generating all the initial mappings for the query pattern, storing them on the *pending* stack. The algorithm continues until *pending* is empty or a circular variant is discovered that does not contain a forward positive cycle. On each iteration of the loop, the selected literal of the current mapping is considered: for each of its subsuming binding patterns, the mapping is instantiated appropriately and placed onto *pending*, unless the selected literal is the last remaining one.

If the selected literal is not known to terminate, then the query-mapping pair is constructed for the mapping and the selected literal. If the resulting pair is new, further mappings are generated from its range query pattern. The function *generate_qm* takes the stack, the two query-mapping pair hash tables, and the new query-mapping pair, and

```
let termination_test qp' =
  let qm_domain = hashtbl__new 997 and qm_range = hashtbl__new 997
  and okay = ref true and pending = stack__new () in
  do_list (fun mapping → push_mapping pending (mapping,qp',[]))
    (generate_mappings qp');
  while not empty pending && !okay do
    let (mapping,qp,ks,selected,cns,ips) = stack__pop pending in
    match selected with
      Nothing → okay := false
    | Just (k,terminates) →
        let subgoal = mapping_subgoal k mapping in
        let u = subgoal_name subgoal in
        begin match ips with
          [] →
            if cns <> [] then
              let ips = subsuming_ips u subgoal in
              stack__push (mapping,qp,ks,selected,tl cns,ips) pending

        | _ →
            stack__push (copy_mapping mapping,qp,ks,selected,cns,tl ips) pending;
            if list_length (k::ks) <> (list_length mapping.subgoals − 1) then
              let aug_mapping = copy_mapping mapping in
              try
                if cns <> [] then merge (hd cns) k aug_mapping;
                instantiate (hd ips) k aug_mapping;
                push_mapping pending (aug_mapping, qp, k::ks)
              with Inconsistent → ()
        end;
        if not terminates then
          let new_qm = mk_qmpair qp k mapping in
          match generate_qm pending qm_domain qm_range new_qm with
            Nothing → ()
          | Just qm → okay := compose_all_qms qm_domain qm_range qm
  done;
  !okay ;;
```

Figure C.2: The function *termination_test*.

implements the above tasks.

```
let generate_qm stack qm_domain qm_range qm =
  if mem_qm qm_domain qm then begin
    Nothing
  end else
    let new_qp = mk_qp 1 (qm_mapping qm) in
    add_qm qm_domain qm_range qm;
    if not mem_qp qm_domain new_qp then begin
      do_list (fun mapping → push_mapping stack (mapping,new_qp,[]))
        (generate_mappings new_qp);
    end;
    Just qm ;;
```

A new query-mapping pair is composed with all existing query-mapping pairs and any circular variants are checked for forward positive cycles. The function *compose_all_qms* takes the two tables and a query-mapping pair, and performs all applicable compositions of pairs.

```
let compose_all_qms qm_domain qm_range qm1 =
  let new_qms = stack__new () and okay = ref true in
  stack__push qm1 new_qms;
  while not empty new_qms && !okay do
    let qm2 = stack__pop new_qms and pending = stack__new () in
    stack__push ([qm2], find_qp qm_domain (mk_qp 1 (qm_mapping qm2))) pending;
    stack__push (find_qp qm_range (qm_query qm2), [qm2]) pending;
    while not empty pending && !okay do
      match stack__pop pending with
        ([],_) → ()
      | (_,[]) → ()
      | (qm :: qms, qm' :: qms') →
          stack__push (if qms = [] then ([qm],qms') else (qms,[qm'])) pending;
          if composable qm qm' then
            try
              let new_qm = check_variant (compose_qm qm qm') in
              if not mem_qm qm_domain new_qm then begin
                add_qm qm_domain qm_range new_qm;
                stack__push new_qm new_qms;
              end
            with
              Inconsistent → ()
            | (Non_termination qm) → okay := false;
    done
  done;
  !okay ;;
```

Any newly created pairs are added to the tables and then each pair is composed in the same fashion. The intricate point is that each new query-mapping pair is added immediately to the tables and is, therefore, guaranteed to be composed with all any remaining new pairs at a later date. This subtlety is a product of performing both left and right compositions in *compose_all_qms*. The implementation of termination analysis is now complete.

Bibliography

Aarts, C. J., Backhouse, R. C., Hoogendijk, P., Voermans, E. & Van der Woude, J. C. S. P. (1992), A relational theory of datatypes, Available via anonymous ftp from `ftp://ftp.win.tue.nl` in directory `pub/math.prog.construction`.

Apt, K. R. & Etalle, S. (1993), On the unification free Prolog programs, *in* 'Proceedings of the Conference on Mathematical Foundations of Computer Science', Vol. 711 of *Lecture Notes in Computer Science*, Springer-Verlag.

Apt, K. R. & van Emden, M. H. (1982), 'Contributions to the theory of logic programming', *Journal of the ACM* **29**(3), 841–862.

Backhouse, R. C. & Hoogendijk, P. F. (1993), Elements of a relational theory of datatypes, *in* B. Möller, H. Partsch & S. Schuman, eds, 'Formal Program Development', Vol. 755 of *Lecture Notes in Computer Science*, pp. 7–42.

Backus, J. (1985), From function level semantics to program transformations and optimization, *in* H. Ehrig, C. Floyd, M. Nivat & J. Thatcher, eds, 'Mathematical foundations of software development, Vol. 1', Vol. 185 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 60–91.

Bekkers, Y., Canet, B., Ridoux, O. & Ungaro, O. (1986), MALI: A memory with a real-time garbage collector for implementing logic programming languages, *in* 'Proceedings of the Third Symposium on Logic Programming', IEEE, Salt-Lake City.

Bellia, M. & Levi, G. (1986), 'The relation between logic and functional languages: a survey', *Journal of Logic Programming* **3**(3), 317–236.

Bellia, M., Bugliesi, M. & Occhiuto, M. E. (1990), Combinatory forms for equational programming: Instances, unification and narrowing, *in* A. Arnold, ed., '15th Colloquium on Trees in Algebra and Programming', Vol. 431 of *Lecture Notes in Computer Science*, Springer-Verlag.

Berghammer, R. & Von Karger, B. (1995), Formal derivation of CSP programs from temporal specifications, *in* 'Mathematics of Program Construction', Vol. 947 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 180–196.

Bird, R. S. (1980), 'Tabulation techniques of recursive programs', *Computing Surveys* **12**(4), 3–17.

Bird, R. S. (1987), An introduction to the theory of lists, *in* M. Broy, ed., 'Logic of Programming and Calculi of Discrete Design', Vol. 36 of *NATO ASI Series F*, Springer-Verlag, pp. 3–42.

Bird, R. S. & de Moor, O. (1994), Relational program derivation and context-free language recognition, *in* A. Roscoe, ed., 'A Classical Mind: Essays dedicated to C.A.R. Hoare', Prentice Hall International.

Bird, R. S. & de Moor, O. (1996), *Algebra of Programming*, Prentice-Hall International.

Bol, R. N. (1991), Loop Checking in Logic Programming, PhD thesis, CWI Amsterdam.

Cattrall, D. M. (1993), The Design and Implementation of a Relational Programming Language, PhD thesis, University of York.

Chen, W., Swift, T. & Warren, D. S. (1995), 'Efficient top-down computation of queries under the well-founded semantics', *Journal of the ACM* **13**(1), 20–74.

Church, A. (1940), 'A formulation of the simple theory of types', *Journal of Symbolic Logic* **5**, 56–68.

Clocksin, W. & Mellish, C. (1987), *Programming in Prolog*, third edn, Springer-Verlag.

Codish, M. & Taboch, C. (1997), A semantic basis for termination analysis of logic programs and its realization using symbolic norm constraints, *in* J. Hanus, M. Heering & K. Meinke, eds, 'Proceedings of the Sixth International Joint Conference on Algebraic and Logic Programming', Vol. 1290 of *Lecture Notes in Computer Science*, Springer-Verlag, Southampton, pp. 31–45.

Collins, G. & Hogg, J. (1997), The circuit that was too lazy to fail, *in* 'Proceedings of the Glasgow Functional Programming Workshop'.

Colmerauer, A., Kanoui, H., Pasero, P. & Roussel, P. (1973), Un systeme de communication homme-machine en francais, Rapport, Groupe d'Intelligence Artificielle, Universite d'Aix-Marseille II, Marseille, France.

Cousot, P. & Cousot, R. (1992), 'Abstract interpretation and application to logic programs', *Journal of Logic Programming* **13**, 103–179.

Darlington, J., Field, A., Harrison, P., Kelly, P., Sharp, D., Wu, Q. & While, R. (1993), Parallel programming using skeleton functions, *in* 'Parallel Architectures and Languages Europe', Springer-Verlag.

Dershowitz, N. (1982), 'Orderings for term-rewriting systems', *Theoretical Computer Science* **17**(3), 279–301.

Freire, J., Swift, T. & Warren, D. S. (1996), Beyond depth-first: Improving tabled logic programs through alternative scheduling strategies, *in* 'Proceedings of the eighth International Symposium on PLILP', Springer-Verlag, pp. 243–258.

Gegg-Harrison, T. S. (1995), Representing logic programming schemata in λProlog, *in* L. Sterling, ed., 'Proceedings of the Twelfth International Conference on Logic Programming', MIT Press, pp. 467–481.

Hallgren, T. & Carlsson, M. (1995), Programming with Fudgets, *in* 'International Spring School on Advanced Functional Programming Techniques, Båstad, Sweden', Vol. 925 of *Lecture Notes in Computer Science*, Springer-Verlag.

Hamfelt, A. & Nilsson, J. F. (1996), Declarative logic programming with primitive recursive relations on lists, *in* M. Maher, ed., 'Proceedings of the Joint International Conference and Symposium on Logic Programming', MIT Press, pp. 230–243.

Hanus, M. (1994), 'The integration of functions into logic programming: From theory to practice', *Journal of Logic Programming* **19**(20), 583–628.

Hanus, M., Kuchen, H. & Moreno-Navarro, J. J. (1995), Curry: a truly functional logic language, *in* 'Proceedings of the ILPS'95 post-conference workshop: visions for the future of logic programming', Portland, pp. 95–107.

Hilton, P. & Pedersen, J. (1991), 'Catalan numbers, their generalisation, and their uses', *Math. Intelligencer* **13**(2), 64–75.

Hogger, C. J. (1990), *Essentials of Logic Programming*, Oxford University Press.

Huet, G. (1973), 'The undecidability of unification in third-order logic', *Information and Control* **22**, 257–267.

Huet, G. (1975), 'A unification algorithm for typed λ-calculus', *Theoretical Computer Science* **1**(1), 27–57.

Hughes, J. (1995), The design of a pretty-printing library, *in* 'International Spring School on Advanced Functional Programming Techniques, Båstad, Sweden', Vol. 925 of *Lecture Notes in Computer Science*, Springer-Verlag.

Hutton, G. (1992*a*), Between Functions and Relations in Calculating Programs, Research report fp-93-5, Department of Computer Science, Glasgow University.

Hutton, G. (1992*b*), 'Higher-order functions for parsing', *Journal of Functional Programming* **2**(3), 323–343.

Hutton, G. (1993), *The Ruby Interpreter*, Chalmers University of Technology, Sweden.

Janot, S. (1991), Règles d'Evaluation Equitables en Programmation Logique, PhD thesis, Laboratoire d'Informatique Fondamentale de Lille.

Jones, G. & Sheeran, M. (1990), Circuit design in Ruby, *in* J. Staunstrup, ed., 'Formal Methods for VLSI Design', Elsevier Science Publications, pp. 13–70.

Jones, G. & Sheeran, M. (1993), Designing arithmetic circuits by refinement in Ruby, *in* R. S. Bird, C. C. Morgan & J. C. P. Woodcock, eds, 'Mathematics of Program Construction (2nd international conference, Oxford, UK, June/July 1992)', Vol. 669 of *Lecture Notes in Computer Science*, pp. 208–232.

Korf, R. E. (1985), 'Depth-first iterative deepening: An optimal admissible tree search', *Artificial Intelligence* **27**(1), 97–109.

Lassez, J. L. & Maher, M. J. (1984), 'Closures and fairness in the semantics of programming logic', *Theoretical Computer Science* **29**(1–2), 167–184.

Lindenstrauss, N. & Sagiv, Y. (1996), Checking termination of queries to logic programs, Available from `http://www.cs.huji.ac.il/ naomil`.

Lindenstrauss, N. & Sagiv, Y. (1997), Automatic termination analysis of logic programs, *in* L. Naish, ed., 'Proceedings of the fourteenth International Conference on Logic Programming', pp. 63–77.

Lipton, J. & Broome, P. (1994), Combinatory logic programming, *in* P. van Hentenryck, ed., 'Proceedings of the Eleventh International Logic Programming Symposium', MIT Press.

Lloyd, J. W. (1987), *Foundations of Logic Programming*, second edn, Springer-Verlag.

Lloyd, J. W. (1995), Declarative programming in Escher, Technical Report CSTR-95-013, Department of Computer Science, University of Bristol.

Lüttringhaus-Kappel, S. (1992), Laziness in Logic Programming, PhD thesis, Universität Bonn.

Lüttringhaus-Kappel, S. (1993), Control generation for logic programs, *in* D. S. Warren, ed., 'Proceedings of the Tenth International Conference on Logic Programming', MIT Press, Budapest, pp. 478–495.

MacLennan, B. (1983), Relational programming, Technical Report Technical Report NPS52-83-012, Naval Postgraduate School, Monterey, CA93943.

McPhee, R. & de Moor, O. (1996), Compositional logic programming, *in* M. Chakravarty, Y. Guo & T. Ida, eds, 'Proceedings of the JICSLP'96 post-conference workshop: Multi-paradigm logic programming', Report 96-28, Technische Universität Berlin, pp. 115–127.

Möller, B. (1993), Derivation of graph and pointer algorithms, *in* B. Möller, H. Partsch & S. Schuman, eds, 'Formal Program Development', Vol. 755 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 123–160.

Moreno-Navarro, J. & Roderiguez-Artalejo, M. (1992), 'Logic programming with functions and predicates: The language Babel', *Journal of Logic Programming* **12**(3), 191–223.

Nadathur, G. & Miller, D. (1988), An overview of λProlog, *in* R. Kowalski & K. A. Bowen, eds, 'Fifth International Logic Programming Conference', MIT Press, Seattle, U.S.A., pp. 810–827.

Nadathur, G. & Miller, D. (1998), Higher-order logic programming, *in* 'Handbook of Logics for Artificial Intelligence and Logic Programming', Vol. 5, Clarendon Press, Oxford, pp. 499–590.

Naish, L. (1985), 'Automating control of logic programs', *Journal of Logic Programming* **2**(3), 167–183.

Naish, L. (1992), Coroutining and the construction of termination logic programs, Technical Report Technical Report 92/5, University of Melbourne, Department of Computer Science.

Naish, L. (1996), Higher-order logic programming in Prolog, *in* M. Chakravarty, Y. Guo & T. Ida, eds, 'Proceedings of the JICSLP'96 post-conference workshop: Multi-paradigm logic programming', Report 96-28, Technische Universität Berlin.

Nilsson, J. F. & Hamfelt, A. (1995), Constructing logic programs with higher-order predicates, *in* M. Alpuente & M. I. Sessa, eds, 'Proceedings of GULP-PRODE 1995, the Joint Conference on Declarative Programming', Università Degli Studi di Salerno, pp. 307–312.

O'Keefe, R. A. (1990), *The Craft of Prolog*, MIT Press.

Paterson, M. S. & Wegman, M. M. (1978), 'Linear unification', *Journal of Computer and System Sciences* **16**(2), 158–167.

Plümer, L. (1991), Automatic termination proofs for logic programs operating on non-ground terms, *in* 'Proceedings of the International Conference on Logic Programming', MIT Press.

Prehofer, C. (1994), Higher-order narrowing, *in* 'Proceedings of the Ninth Annual IEEE Symposium on Logic in Computer Science', IEEE Computer Society Press, pp. 507–516.

Qian, Z. (1994), Higher-order equational logic programming, *in* 'Proceedings of the 21st ACM Symposium on Principles of Programming Languages', ACM Press, Portland, U.S.A., pp. 254–267.

Ramakrishnan, I., Rao, P., Sagonas, K., Swift, T. & Warren, D. S. (1995), Efficient tabling mechanisms for logic programs, *in* 'Proceedings of the International Conference on Logic Programming', pp. 697–711.

Reddy, U. (1987), Functional logic languages part I, *in* 'Proceedings of a Workshop on Graph Reduction', Lecture Notes in Computer Science 279, Springer-Verlag, pp. 401–425.

Sagiv, Y. (1991), A termination test for logic programs, *in* 'Proceedings of the International Logic Programming Symposium', MIT Press, pp. 518–532.

Sagonas, K., Swift, T. & Warren, D. S. (1994), XSB as an efficient deductive database engine, *in* 'Proceedings of the ACM SIGMOD International Conference on the Management of Data', Minnesota, pp. 442–453.

Schmidt, G. & Ströhlein, T. (1988), *Relationen und Grafen*, Springer-Verlag.

Seres, S. & Mu, S.-C. (2000), Optimisation problems in logic programming: an algebraic approach, *in* 'Proceedings of Logic Programming and Software Engineering 2000', London, U.K.

Seres, S. & Spivey, M. (1999), Embedding Prolog into Haskell, *in* 'Proceedings of HASKELL 1999', Paris, France.

Sheeran, M. (1989), Describing hardware algorithms in Ruby, *in* A. David, ed., 'IFIP WG 10.1 workshop on Concepts and Characteristics of Declarative Systems, Budapest 1988', North-Holland.

Sheeran, M. & Jones, G. (1987), Relations + higher-order functions = hardware descriptions, *in* W. E. Proebster & H. Reiner, eds, 'Proceedings of the IEEE Comp Euro 87: VLSI and Computers', Hamburg, pp. 303–306.

SICS (1997), SICStus Prolog user manual, release 3.6, Swedish Institute of Computer Science.

Somogyi, Z., Henderson, F. J. & Conway, T. (1995), Mercury: an efficient purely declarative logic programming language, *in* 'Proceedings of the Australian Computer Science Conference', Glenelg, Australia, pp. 499–512.

Speirs, C., Somogyi, Z. & Søndergaard, H. (1997), Termination analysis for Mercury, *in* P. V. Henternryck, ed., 'Static Analysis: Proceedings of the Fourth International Symposium', Vol. 1302 of *Lecture Notes in Computer Science*, Springer.

Spivey, M. (1996), *An introduction to logic programming through Prolog*, Prentice-Hall.

Stickel, M. E. (1988), 'A Prolog technology theorem prover: Implementation by an extended Prolog compiler', *Journal of Automated Reasoning* **4**, 353–380.

Tamaki, H. & Sato, T. (1986), OLD resolution with tabulation, *in* 'Proceedings of the third International Conference on Logic Programming', Vol. 225 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 84–98.

Thom, J. A. & Zobel, J. (1987), NU-Prolog reference manual, Technical report, Melbourne University.

Ullman, J. D. & Gelder, A. V. (1988), 'Efficient tests for top-down termination of logical rules', *Journal of the ACM* **35**(2), 345–373.

Wadler, P. (1985), How to replace failure by a list of successes, *in* J.-P. Jouannaud, ed., 'Functional Programming Languages and Computer Architecture', Vol. 201 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 113–128.

Warren, D. (1982), 'Higher-order extensions to Prolog: Are they needed?', *Machine Intelligence* **10**, 441–454.

Warren, D. S. (1992), 'Memoing for logic programs with applications to abstract interpretation and partial deduction', *Communications of the ACM.*