The Algebra of Logic Programming

Silvija Seres

Wolfson College

Hilary Term 2001

Submitted in partial fulfilment of the requirements for the degree of Doctor of Philosophy in Computation

Oxford University Computing Laboratory
Programming Research Group

Abstract

At present, the field of declarative programming is split into two main areas based on different formalisms; namely, functional programming, which is based on lambda calculus, and logic programming, which is based on first-order logic. There are currently several language proposals for integrating the expressiveness of these two models of computation. In this thesis we work towards an integration of the methodology from the two research areas. To this end, we propose an algebraic approach to reasoning about logic programs, corresponding to the approach taken in functional programming.

In the first half of the thesis we develop and discuss a framework which forms the basis for our algebraic analysis and transformation methods. The framework is based on an *embedding* of definite logic programs into lazy functional programs in Haskell, such that both the declarative and the operational semantics of the logic programs are preserved.

In spite of its conciseness and apparent simplicity, the embedding proves to have many interesting properties and it gives rise to an algebraic semantics of logic programming. It also allows us to reason about logic programs in a simple calculational style, using rewriting and the algebraic laws of combinators. In the embedding, the meaning of a logic program arises compositionally from the meaning of its constituent subprograms and the combinators that connect them.

In the second half of the thesis we explore applications of the embedding to the algebraic transformation of logic programs. A series of examples covers simple program derivations, where our techniques simplify some of the current techniques. Another set of examples explores applications of the more advanced program development techniques from the Algebra of Programming by Bird and de Moor [18], where we expand the techniques currently available for logic program derivation and optimisation.

To my parents, Sandor and Erzsebet.

And the end of all our exploring
Will be to arrive where we started
And know the place for the first time.
T. S. Elliot, Four Quartets

Acknowledgements

First, I would like to express a deeply felt gratitude to Mike Spivey, my supervisor. Although I was a different kind of student from the ones he usually advises, he guided me in a most friendly, competent, and patient way. The thesis presented here is truly joint work. His guidance and encouragement throughout the course of this thesis, and particularly during the writing-up process, have been invaluable; his criticisms always constructive and his logic always clear. I could not have wished for a kinder supervisor.

This thesis would not have happened without Tony Hoare's support; I am deeply grateful for the encouragement with which he welcomed me to Oxford, for the far-sightedness with which he guided me in the early stages of my doctoral work, and most of all for the compassion and concern with which he helped me resolve several dilemmas regarding my future.

I would like to thank Jeremy Gibbons for the patience and cheerfulness with which he answered my questions on algebra of programming, and Oege de Moor and Richard Bird for their interest in my work and helpful suggestions for looking at my results in a different way. My sincere thanks go also to Ole-Johan Dahl and Herman Ruge Jervell, for first giving me the confidence to apply to Oxford, and for their encouragement during my time here.

I have benefited enormously from being at the remarkable Compaq Systems Research Centre in Palo Alto for one most enjoyable and instructive summer. I thank all the scientists at SRC for letting me work with them, and most of all to Greg Nelson, for showing me how excellent and uncompromising industrial research can be. I also very much appreciated Krzysztof Apt's hospitability, guidance and inspiration on several occasions. A dear friend, Edsger W. Dijkstra has taught me, in words and by his actions, the rôle of work in both scientific and (not necessarily disjointly) social contexts. His kindness was crucial for my coming to Oxford. Thank-you also to Ria and Jill, for your friendliness and your help; both have been much appreciated.

I am especially indebted to the Research Council of Norway, for funding my doctoral scholarship, and to the Norwegian Computing Centre and the ORS committee for co-funding my fees. The Norwegian Computing Centre provided for me not only financially, but also as an office-away-from-office during my time in Norway. I would especially like to thank Ole-Henrik, Jon, Naci and Dalip for their support regarding my doctorate. All my three Oxford colleges – Wolfson, Oriel, and Magdalen – have helped me make the most of my life in beautiful Oxford.

Many people have contributed less directly to the writing of this thesis, and their support over the past three years has been very important to me. I want to thank Ben, Paul, Christie, Yorck, and most of all Karina, for being my good friends as well as my lab-mates, and to Shin and Rick for their good co-operation. I will miss office 002 and its inhabitants. Thank-you to my always supportive Comlab colleagues, my rowing crews and my Wolfson friends as well, and particularly to Nenagh, Dave, Anne and Guilaine for the warmth of their friendship throughout my time in Oxford.

A big thank-you goes to my much-missed family and friends at home, both in Norway and Yugoslavia, for keeping in touch while I was away and for making it so easy to fit back in when I visited home.

My gratitude to my husband Andreas is immense. His support – intellectual, technical and emotional – over the last three years has enriched my thesis and my life as well. I owe so very much to his unselfish love and patience.

Finally, I want to express my appreciation of my father Sandor, my mother Erzsebet, my sister Gabi, and my grandmother Manci. Their letters and phone calls cheered me up all along the way. I thank them for that constant contact, for their interest, for their belief, and for making me feel that they were always there for me, despite the physical distance between us.

For all this kindness and support I feel I have been truly blessed.

Contents

Chapter 1

Aims and Results

In this chapter we position our work and outline the contents of the thesis. We explain our motivation, which is to apply program transformation techniques from functional programming in a logic programming setting, and we point out the main influences. We also list the main results of the thesis.

1.1 Motivation

The most important characteristic of declarative languages is that they allow the programmer to express programs in terms of *what* is to be computed, rather than *how* to arrive step by step at the desired result. In theory, the declarative programmer does not need to concern himself with particular execution mechanisms of the language, or the underlying hardware. The compiler or interpreter of the given language is responsible for turning the declarative statements into sequences, or even concurrent streams, of imperative commands.

In contrast to imperative programming, the execution models of declarative programs are rooted in mathematics: program statements can be viewed as sentences in some logic, for example, equational or Horn clause logic, and program computation as a deduction in that logic. This mathematical basis offers a considerable, but not yet fully realised, potential for simplification of formal reasoning about programs and program development.

An important step towards popularising the declarative paradigm is to unify the ideas, the expressiveness, and the main methodological results, from various partitions of this field. The two main groups of declarative languages are functional and logic programming. These languages have important stylistic differences, and traditionally they employ different programming methodologies regarding the specification, synthesis, transformation, debugging and verification of programs. Research in functional programming has to a larger extent been focused on equational reasoning, whereas in logic programming more effort has been put into improving the efficiency of programs through non-declarative features.

However, these differences are not as deep as they appear. In the recent words of Robinson [115]:

> It has been a source of weakness in declarative programming that there have been two major paradigms needlessly pitted against each other, competing in the same marketplace of ideas. The challenge is to end the segregation and merge the two. There is in any case, at bottom, only one idea.

The languages from different sub-paradigms of declarative programming have different mathematical models of computation, but all of these can be subsumed within a unified system based on narrowing. This is being successfully done in several major projects, including the integrated functional logic programming languages Curry [60], Escher [78], Babel [90], Mercury [129], and others. This research mainly focuses on unifying the *expressiveness* of the two underlying programming models; however, it is also important to unify the underlying *methodology*, that is, their approach to program analysis, transformation, parallelisation and programming environments. This unification is the motivation for this thesis; we aim to explore the possibility of a transfer of the techniques for program transformation from functional to logic programming.

We have chosen to focus in particular on program transformation methodology because, in spite of the similarities between the two programming styles, their current approaches to this area are very dissimilar. The standard approaches to logic program transformation are based on transformational laws which preserve the declarative semantics. However, there are several important questions that an approach based purely on the declarative semantics cannot answer: termination, efficiency, and therefore, the guiding of the derivation of a more efficient program. For example, of two programs with an equivalent declarative semantics, only one of the programs might terminate under a given evaluation strategy. We contend that another approach, based on equational reasoning as found in functional program transformation, is also appropriate for program transformation of logic programs.

The particular approach that we have taken is inspired by the following two sources. On an abstract level, the Unifying Theories of Hoare and He [65] use algebraic descriptions to classify and compare different programming paradigms. Such a description consists of a set of laws, which may be used for syntactical, rather than semantical, reasoning about programs. Such reasoning is also much used in functional programming, so the framework of Unifying Theories motivated our attempt to transfer some of the standard algebraic methods used in the functional programming style over to the other declarative style.

The second source, the Algebra of Programming of Bird and de Moor [18], explores the problem of functional program transformation by mathematical calculation, both in order to derive individual programs and to study programming principles (such as algorithm design) in general. They classify many important examples of transformation as instances of more general transformational strategies, valid for entire families of functional programs, based on higher-order functions, and parametric in the data structures. Because of the similarity of logic and functional programming, many of the examples from the Algebra of Programming are also well-known and have been explored in the setting of logic programming. We were motivated to compare the existing transformational techniques to the ones originating from functional programming.

1.2 Methodology

The standard techniques for program transformation of functional programs are based on algebraic reasoning: equational rewriting is used in conjunction with definitions of functions and equational laws that capture the properties of the generic recursion operators.

To help in applying this algebraic technique to the logic paradigm, we have designed and implemented a theoretical tool in which a pure logic program is translated into a lazy functional program, so that a logic programming language is embedded in the functional language. This embedding can be viewed as an executable denotational semantics for the logic program, and it serves both as a platform and a combinator library for writing logic programs in a functional setting.

More concretely, the embedding is an implementation of the first-order logic operators which appear in the completed form of the logic program. Using the completed form of a logic program facilitates the use of equational reasoning, which simplifies program transformation. It also emphasises the similarities between the logic and functional programming paradigms. Furthermore, it singles out the four basic operators (&, $\|$, $\doteq$ and $\exists$) which are necessary for the expressiveness of pure logic programs. By focusing on each of these separately, we gain a *compositional* understanding of logic programming.

We have three consistent interpretations for the completed version of the logic program: *declarative*, where the meaning of the four operators arises from first-order logic and from a theory of equality; *operational*, as given by our embedding in Haskell, where the meaning of the four operators arises from their functional definition; and finally, *algebraic*, where the meaning of the four operators arises from a set of algebraic equations. The declarative and the operational semantics of the predicates in the embedding can be proved consistent with the standard declarative and operational readings of a logic program. The algebraic semantics is useful not only for program transformation but also for a better understanding of the nature of predicates as used in logic programming.

Finally, the embedding provides an expressive programming tool, since it makes available in a single framework the most useful facets of both declarative styles: namely, nested expressions, types, higher-order functions and lazy evaluation from functional programming; and logical variables, partial data structures and built-in search from logic programming. We argue that it could well serve also for the implementation of other extensions of the logic programming model, for example typed logic programming, constraint programming, concurrent logic programming or higher-order logic programming.

1.3 Results

Below we list the main contributions of this thesis:

Implementation of the embedding. We provide an implementation of the embedding of pure logic programs into Haskell. The embedding is based on atomic predicates *true*, *false*, and equality-predicates, together with the four operators &, $\|$, *not*, and $\exists$. As described above, it is a concise yet expressive tool which provides a framework for the compositional study of logic programs; we use it in the remainder of the thesis in for several theoretical and practical applications.

Algebraic semantics of logic programs. The relationship between the declarative meaning of the basic operators of the embedding, and their operational meaning that arises from LD-resolution (the standard operational reading of pure Prolog programs), is imperfect. If the completed program is to have the same *computational* meaning as the original logic program, only some of the usual laws of Boolean or Cylindric Algebra should hold for these logic operators. By recognising which properties are shared by their meanings as given by first-order logic and by LD-resolution, we can distill the algebraic description of a logic program.

Analysis of different search strategies. A closer inspection of the algebraic laws shows that the primitive scheduling operators of logic programming obey a rich set of laws, and that some of these laws can be found in the categorical theory of monads. We describe implementations of three search strategies for logic programming, and present the mathematical framework with which we explore and express the relationships between the three models.

Simulation of LD-resolution. The algebraic laws for the operators of the embedding give rise to a specification which is useful independently from their implementation. In addition to the point above, we use this specification to prove that the answers computed by the embedding are the same as those computed by LD-resolution, even though the embedding does not use LD-resolution as the computation mechanism. The soundness and completeness of the embedding follow from this result.

Treatment of recursive predicates. The embedding can also be used to identify certain universal healthiness properties that all predicates must satisfy. We use these properties to relate the embedding to the standard declarative semantics of logic programs, and to argue that recursive predicate definitions have unique solutions.

Equational program transformation. The algebraic laws allow the application of two important aspects of program transformation: we may use equational reasoning, and we may compare the computational behaviour of the two programs. We transfer some of the successful program transformation techniques from functional to logic programming, and show examples where the combination of the use of higher-order functions and predicates results in simpler techniques and programs.

To some extent, the use of our library of functions will a give functional programmer a small taste of the power of a *functional logic language*. But current functional logic languages are much more powerful; they embody both rewriting and resolution and thereby result in a functional language with the capability to solve arbitrary linear constraints for the values of variables. The list of languages that have been proposed in an attempt to incorporate the expressive power of both functional and logic paradigms is long and impressive [13, 59]. Our research goal is different from the one set by these projects. They aspire to build an efficient language that can offer programmers the most useful features of both worlds; to achieve this additional expressiveness they have to adopt somewhat complicated semantics. Our present goal is a perspicuous declarative and operational semantics for the embedding, rather than maximal expressiveness. Nevertheless, the extension of our embedding to incorporate both narrowing and residuation in its operational semantics seems feasible and we suggest it as a subject for our further work.

The emphasis on the similarity between the evaluation mechanisms in logic and functional programs has several benefits. First, the compositional operational semantics of a logic program in Haskell is not only a useful theoretical tool for logic programmers, it also gives functional programmers easy access to understanding the operational semantics of logic programs. Second, bringing the two programming styles into a common setting encourages a transfer of methods between the two. For example, higher-order techniques are readily available, and because of the similarities between functional programs and logic programs in the embedding, it is obvious that these techniques are necessary for capturing general program transformation and derivation techniques. Finally, this approach has potential for a simple implementation of an efficient functional logic programming language, even including parallelism and constraints "for free".

Some material from this thesis has been published in refereed conferences and journals. The embedding and the algebraic laws have been described in [134] and [121], and the analysis of different search strategies has been published [123] and [133]. The techniques and examples involving general program transformation were discussed in [122], and the advanced transformation techniques were presented in [120].

1.4 Thesis outline

The rest of this thesis is organised as follows:

In Chapter 2 we describe the relevant background material, including an overview of logic and functional programming.

In Chapter 3 we describe an embedding of logic programming in Haskell. We present an execution example and discuss how the embedding can be altered to encompass different search strategies.

In Chapter 4 we describe the algebraic laws for each of the four basic operators of the embedding.

In Chapter 5 we give two alternative implementations of the embedding, one corresponding to a breadth-first search model of logic programming, and one to a model which permits any search strategy.

In Chapter 6 we present a categorical analysis of the three resulting search models, presenting structure preserving maps between them, and proving that the list of laws from Chapter 4 is in a sense complete.

In Chapter 7 we prove that the embedding, and consequently the derived algebraic semantics, is correct with respect to LD-resolution, the usual procedural reading of logic programming languages.

In Chapter 8 we explore the algebraic properties of predicates and the denotational semantics of recursively defined predicates. We show that, in any fair model of the embedding, all definable recursive predicates have a unique fixpoint.

In Chapter 9 we show how the algebraic laws can be used for transformation of logic program. The examples deal with the emulation of the standard unfold/fold technique, accumulating parameters and the improvement of generate-and-test based programs.

In Chapter 10 we show how the approach used in Chapter 9 can be applied to optimisation problems: we derive efficient algorithms for problems of string edit, minimal tardiness and 0-1 knapsack.

In Chapter 11 we present our conclusions and compare our work to the main results in related areas. Finally, we point out directions for further research.

For completeness, we present the full code for the embedding in Appendix A, the code for the examples from Chapter 10 in Appendix B.

Chapter 2

Basic Notions

This chapter outlines the basic concepts from logic and functional programming that are used in this thesis; it sketches the main ideas of functional logic programming, and concludes with a comparison of these paradigms.

2.1 Substitutions and unifications

We begin by defining the set $T(F, X)$ of *terms*. The set $F = \{f_1, f_2, \dots\}$ contains function symbols, each with a given arity. We require that F contains at least one constant. A countably infinite set $X = \{x_1, x_2, \dots\}$ contains variables. Then $T(F, X)$ denotes the set of all terms $\{t_1, t_2 \dots\}$ built from F and X in the following inductive manner:

- a variable x_i is a term,
- if f_i is a n-ary term and $t_1, \dots, t_n$ are terms,
  then $f_i(t_1, \dots, t_n)$ is a term.

$\mathit{Var}(t)$ denotes the set of variables occuring in a term t. A *ground* term is a term t without variables, that is, such that $\mathit{Var}(t) = \{\}$.

A *substitution* is a function from variable symbols to terms in which all but a finite number of variable symbols are mapped to themselves. Substitutions can therefore be presented as a finite set of variable/term pairs:

$$\{x_1/t_1, \ldots, x_n/t_n\},$$

in which the x_i are distinct variable symbols and no t_i is the same as the corresponding x_i. A pair x_i/t_i is called a *binding* for x_i.

Given a substitution $\{x_1/t_1, \ldots, x_n/t_n\}$, the set $\{x_1, \ldots, x_n\}$ is called the *domain* of the substitution. If X is a set of variables, $\theta|X$ denotes the substitution obtained by the restriction of the domain of θ to the set X. If $n = 0$ the substitution is called an *empty* substitution, and is denoted by ε. A substitution that is a 1-1 and maps its domain to another set of variables, is called a *renaming*. Two substitutions are *equal* if they are equal as functions; equivalently, they are equal if they have the same domain and act in the same way on each element of this domain.

A substitution σ can be extended to a function from terms to terms, where a term is mapped into a term obtained from t and σ by *simultaneously* replacing in t every variable which has a binding in σ with the corresponding term defined by the binding. Thus the obtained term is denoted as $t\sigma$ and is called an *instance* of t. If σ is a renaming, then $t\sigma$ is called a *variant* of t.

The *composition* of two substitutions uniquely identifies a substitution. If σ and θ are substitutions and x a variable, then their composition, denoted $\sigma\theta$, is defined by:

$$(\sigma\theta)(x) = (\sigma(x))\theta,$$

and consequently, for any term t, composition satisfies $(t\sigma)\theta = t(\sigma\theta)$. Composition is associative, that is, $(\sigma\theta)\eta = \sigma(\theta\eta)$. Furthermore, the empty substitution ε is a left and right identity. Thus substitutions form a monoid under composition.

A substitution σ is *more general* than another substitution θ, or σ *subsumes* θ, if there exists a substitution η such that $\theta = \sigma\eta$; we write $\sigma \sqsubseteq \theta$. Later we shall operate on sets of substitutions, and we will need a subsumption ordering for them as well: given two sets of substitutions S and S', we say that S subsumes S' (written $S \sqsubseteq_* S'$) iff for every $\sigma' \in S'$ there exists a more general $\sigma \in S$:

$$S \sqsubseteq_* S' \Leftrightarrow (\forall \sigma' \in S'. (\exists \sigma \in S.\ \sigma \sqsubseteq \sigma')).$$

A *unifier* of two terms t_1 and t_2 is a substitution σ such that $t_1\sigma = t_2\sigma$. A pair of terms may have no unifier or several, but if they do have a unifier, then it can be shown that there exists a unique (up to variable renaming) *most general unifier*, abbreviated as *mgu*. Given two terms, if σ is any unifier and μ is their mgu, then $\sigma \sqsubseteq \mu$. The most general unifier μ subsumes every other unifier σ, and it can be though of as a substitution which makes two terms equal without any unnecessary bindings. An mgu μ of two terms is *strong* if for any unifier σ of these terms, $\sigma = \mu\sigma$ holds. In particular, $\mu = \mu\mu$.

The problem of determining whether two terms are unifiable is solved by providing an algorithm that terminates with failure if the terms are not unifiable, and that otherwise produces one of their most general unifiers, in fact a strong one. The first unification algorithm is due to Robinson [113]; another well-known unification algorithm is by Martelli and Montanari [83], and we present this one because it suits our framework better. It is this basic unification algorithm that we shall be using in our implementation. It works on unifying a finite set of pairs of terms, written as a set of equations:

$$\{t_1 = t'_1, \ldots, t_n = t'_n\}.$$

A unifier for this set makes each equation true. The algorithm chooses non-deterministically an equation from the set of equations below and performs the associated action. We stress that x_i stands for a variable.

(1)	$f_i(t_1, \ldots, t_n) = f_i(t'_1, \ldots, t'_n)$	Replace by $t_1 = t'_1, \ldots, t_n = t'_n$.
(2)	$f_i(t_1, \ldots, t_n) = f_j(t'_1, \ldots, t'_m)$, where $i \neq j$	Halt with failure.
(3)	$x_i = x_i$	Delete the equation.
(4)	$t_i = x_i$, where t_i is not a variable	Replace by equation $x_i = t_i$.
(5)	$x_i = t_j$, where t_j is not a variable and x_i does not occur in t_j	Perform the substitution $\{x_i/t_j\}$ on all other equations.
(6)	$x_i = t_j$ where $x_i \in \mathit{Var}(t_j)$, and $x_i \neq t_j$	Halt with failure.

This algorithm terminates when no action can be performed or when failure arises. In case of success, by changing in the final set of equations $=$ to $/$, we obtain the desired mgu. The unification of other syntactic constructions, such as atoms, is based on this unification algorithm for terms.

This and the next section are compiled and adapted from a tutorial by Apt in [4] (for the most part) and the books by Apt, Lloyd and Spivey [2, 76, 131].

2.2 Logic programming

Let $R = \{r_1, r_2, \dots\}$ be a set of relation symbols with given arities, including the predefined constant relations *true* and *false*. Then $A = \{p_1, p_2, \dots\}$ denotes the set of atomic formulae, or simply *atoms*, where a p_i is defined as a n-ary relation from R applied to an n-tuple of terms from $T(F, X)$, that is, $r_i(t_1, \dots, t_n)$.

A *query*, denoted as q_i, is a finite sequence $p_1, \dots, p_n$ of atoms. A *clause*, denoted as c_i, is a construct $p_i \leftarrow q_i$. A logic *program*, denoted as P, is a finite set of clauses.

A query $p_1, \dots, p_n$ is interpreted as the conjunction of all the literals within it, and the empty query is equivalent to *true* and traditionally denoted by $\Box$. A clause $p_i \leftarrow q_i$ is interpreted as an implication from q_i to p_i, where the atom p_i is referred to as the *head* and the query q_i as the *body* of the clause. A clause $p_i \leftarrow$ with an empty body is called a *unit* clause, and its meaning is that p_i holds unconditionally.

Given a program P, we regard the set of all clauses that mention the relation r in their heads as the *definition* of r in P. For example, let the relation $append(x_1, x_2, x_3)$ mean that x_1 appended with x_2 results in the list x_3. Let $[\,]$ stand for the empty list and $[x|y]$ for the list containing x as the head and y as the tail. Then the predicate *append* can be defined by the following two clauses:

$$append([\,], x_1, x_1) \leftarrow .$$
$$append([x_1|x_2], x_3, [x_1|x_4]) \leftarrow append(x_2, x_3, x_4).$$

The clauses as defined as above are usually called *definite* clauses, while in general clauses may allow more than one atom in the head of the clause. In this thesis we consider only definite clauses.

All the variables occuring in a clause are implicitly governed by a universal quantifier. Equivalently, we may say that those variables that occur in the body but not in the head are governed by an implicit existential quantification whose scope is the body. We extend *Var* to also denote the set of variables for the syntactic constructions defined in this section.

To execute a logic program one has to give it an initial query $p_1, ..., p_n$. The standard computational mechanism, *SLD-resolution*, finds instances of a query $p_1, ..., p_n$ such that all of $p_1\sigma, ..., p_n\sigma$ are implied by the program, thus providing a constructive proof of $(\exists\vec{x}.\, p_1 \wedge \ldots \wedge p_n)$. This is often explained by saying that SLD-resolution proves that the clause $\mathit{false} \leftarrow p_1, ..., p_n$ is inconsistent with the program, in the sense that the clause $\mathit{false} \leftarrow$ is derivable. This proof technique is referred to as proof by *refutation*.

For example, the query $\mathit{append}([1], y, [1, 2])$ results in one answer $\{y/[2]\}$, while the query $\mathit{append}(x, [2], z)$ results in an infinite list of answer substitutions $\{x/[\,], z/[2]\}$, $\{(x/[x_1], z/[x_1, 2])\}$, $\ldots$ The basic step of this computation is an SLD-resolution step $\underset{\text{SLD}}{\longrightarrow}$ as defined below.

Queries are always executed relative to some input substitution; initially, this substitution is ε. In standard descriptions of SLD-resolution, the input substitution is implicit and is automatically applied to the variables of the query. On the other hand, in our functional presentation of the embedding, as in the description of Apt in [4], these substitutions are explicit. Given a query q and a substitution θ, we denote by the pair $\langle q; \theta\rangle$ the query q in the environment θ, that is, the variables of q interpreted as in θ. We assume that, if q is non-empty, one atom is selected in it. Consider a program P. We define the relation $\langle q; \theta\rangle \underset{\text{SLD}}{\longrightarrow} \langle q'; \theta'\rangle$ as follows:

Consider a pair $\langle q_1, p, q_3; \theta\rangle$, with p the selected atom, and a clause c from P. Let $h \leftarrow q_2$ be a variant of c, variable disjoint with $\langle q_1, p, q_3; \theta\rangle$, that is, such that:

$$\mathit{Var}(h \leftarrow q_2) \cap (\,\mathit{Var}(q_1, p, q_3) \cup \mathit{Var}(\theta)) = \emptyset.$$

Suppose that $p\theta$ and h unify with the mgu η. Then the pair $\langle q_1, q_2, q_3; \theta\eta\rangle$ is the *SLD-resolvent* of $\langle q_1, p, q_3; \theta\rangle$ and c, and we write :

$$\langle q_1, p, q_3; \theta\rangle \underset{\text{SLD}}{\longrightarrow} \langle q_1, q_2, q_3; \theta\eta\rangle.$$

We obtain *SLD-derivations* by repeating the $\underset{\text{SLD}}{\longrightarrow}$ steps. An SLD-derivation of a query q in program P is a maximal sequence of pairs $\langle q_i; \theta_i\rangle$ such that $i \geq 0$ and for all $j \geq 0$ we have $\langle q_j; \theta_j\rangle \underset{\text{SLD}}{\longrightarrow} \langle q_{j+1}; \theta_{j+1}\rangle$.

And SLD-resolution is called *successful* if it is finite and the query in the last pair is empty, and it is called *failed* if it is finite and the query in the last pair is non-empty. If we derive $\langle q; \varepsilon\rangle \underset{\text{SLD}}{\longrightarrow}^* \langle \Box; \tau\rangle$, we call $q\tau$ a *computed instance* of q with respect to P, and we call $\tau | \mathit{Var}(q)$ a *computed answer substitution* for q with respect to P.

In Chapter 7 we prove the adequacy of our embedding, and a crucial part of this argument is based on the correctness of SLD-resolution. To be more precise about "correctness" here, we need to be formal about the meaning of our syntactic entities such as terms and relations.

The meaning of terms arises from an *algebra*. Given a language of terms $T(F, X)$, an algebra J for $T(F, X)$ consist of a non-empty domain D and an assignment to each n-ary function symbol f in F a mapping f_J from D^n to D. A state ρ over the domain D assigns to each variable x an element from D and and to each non-variable term t an element $\rho(t)$ from D, defined by induction:

$$\rho(f(t_1, \dots, t_n)) = f_J(\rho(t_1), \dots, \rho(t_n)).$$

The meaning of relation symbols arises from an *interpretation*. Given a set of relations based on $T(F, X)$ and a set of relation symbols R, an interpretation I extends an algebra J by an assignment, for each n-ary relation symbol r in R, of a subset r_I of D^n.

Given an interpretation I, the relation $\models_\rho$ between an interpretation I and an atom, query or clause formalises our earlier informal description of their meaning. If $r(t_1, \dots, t_n)$ is an atom, then $I \models_\rho r(t_1, \dots, t_n)$ iff $(\rho(t_1), \dots, \rho(t_n)) \in r_I$. If $p_1, \dots, p_n$ is a query, then $I \models_\rho p_1, \dots, p_n$ iff $I \models_\rho p_i$ for all $i \in [1, n]$.

If $p_i \leftarrow q_i$ is a clause, then $I \vDash_\rho p_i \leftarrow q_i$ iff $I \vDash_\rho q_i \Rightarrow I \vDash_\rho p_i$. Finally, if $c_1, \ldots , c_n$ is program, then $I \vDash_\rho c_1, \ldots , c_n$ iff $I \vDash_\rho c_i$ for all $i \in [1, n]$.

Models are used to define the concept of a logical consequence, and consequently, the declarative notion of an answer. Let E denote an atom, clause or query. If for all states ρ we have $I \vDash_\rho E$, then we write $I \vDash E$. Such an interpretation I gives every ground instance of the expression E the value *true*, and it is called a *model* for E. Given a program P and a query q, we say that q is a semantic consequence of P, written $P \vDash q$, if every model of P is also a model of q. Which brings us to the declarative counterpart of a computed answer: if $P \vDash q\theta$, then $\theta | \mathit{Var}(q)$ is called a *correct answer substitution* of q and $q\theta$ is called a *correct instance* of q.

The following two theorems justify the use of SLD-resolution, and later also the use of our embedding:

Theorem 2.1
SLD-resolution is sound: Consider a program P and a query q. Then, every computed instance of q is a correct instance of q.

Theorem 2.2
SLD-resolution is complete: Consider a program P and a query q. For every correct instance of q, there exists a computed instance of q that is equal or more general.

In Chapter 8 we relate our embedding to the standard declarative reading of logic programs, and one particular type of interpretations is central for this reading and for SLD-resolution. Resolution is fundamentally based on unification, and this choice is motivated by the syntactically-based concept of Herbrand models. For a ground expression E, a Herbrand model is an interpretation H, such that the domain D_H of H consists of all ground atoms $p(t_1, \ldots , t_n)$, where each t_i denotes a ground term constructed using the constants and function symbols in E, and the state ρ of H assigns a truth value to some subset of this domain, such that the expression E is true. For any non-ground E, a ground instantiation $G(E)$ is obtained by replacing all variables in E by terms from D_H, and any Herbrand model of $G(E)$ is also a Herbrand model of E.

Given a program P, most Herbrand models of P are larger than strictly necessary to satisfy P, but the set of all these models has a unique least element called the *Least Herbrand Model*, $\mathcal{H}_P$, which is the intersection of all the Herbrand models of P. This is the intended interpretation of a logic program, and is taken to be its declarative meaning. It can be shown that $\mathcal{H}_P$ is precisely the set of ground queries that succeed by resolution with the clauses of P.

Thus, resolution uses the idea of unification to guide the choice of substitutions for universally quantified variables, and finds all the instances of the query which are in the Least Herbrand Model of the program. The unifiability requirement acts as an implicit filter for the selection of the relevant instances, and moreover represents its selection very economically through the device of the mgu.

In Chapter 7 we shall need the concept of an SLD-tree, which is a tree with the query at the root, depicting a computation from that root, with subsequent resolvents as adjacent nodes. Each edge in an SLD-tree signifies a resolution step, and the tree contains enough information to determine exactly which answers, if any, are computed in response to the root node. Different selection rules determine different SLD-trees, but all those trees agree upon the computed answer set; this property is customarily referred to as the *independence of the selection rule*.

An SLD-tree will typically contain a number of branches and hence a number of distinct computations, making the execution process non-deterministic, and hence giving rise to a need for a search strategy. In Chapter 5 we analyse the different search strategies which determine the manner of the exploration of the SLD-tree. The standard search strategy, used by Prolog, is top-down, depth-first search (dfs) with backtracking. This strategy is not guaranteed to produce every correct answer in a finite time; in particular, in case of infinitely deep search trees the exploration of an infinite branch will defer indefinitely the exploration of some alternative capable of yielding a correct answer. In contrast, breadth-first search (bfs) would in all cases ensure effective completeness in that every correct answer would be discovered after some finite time, but can then continue to explore the infinite branches indefinitely. However, if the search tree is finite, dfs is optimal in memory utilisation while ensuring completeness.

As we show later on, our embedding is parametric in the search rule. We have three implementations that share the same algebraic properties, where one traverses the SLD-tree in a depth-first manner, using backtracking, another that traverses it in a breadth-first manner, and a third one that can traverse the tree using any given search strategy. However, we are not able to vary the selection rule. All the implementations use the *left-most selection rule*, the same one as used by Prolog. Following Apt in [2], we refer to this computational model as *LD-resolution*, standing for SLD-resolution with the leftmost selection rule. To emphasise this restriction, we use the corresponding terminology throughout the thesis: by an *LD-resolvent*, *LD-tree*, *LD-resolution* and *LD-derivation* we mean respectively an SLD-resolvent, an SLD-tree, an SLD-resolution and an SLD-derivation with respect to the leftmost selection rule.

Computation in pure Prolog is obtained by imposing certain restrictions on the SLD-resolution for the sake of efficiency; the fixed selection rule is one of these restrictions. Another restriction, which we also set in our embedding, is that the clauses are tried in the order in which they appear in the program text, so the program is viewed as a sequence of clauses rather than a set.

Because Prolog always uses depth-first search, the computation of a Prolog program does not necessarily correspond to an LD-tree. The *Prolog tree* of a logic program is a subtree of the LD-tree which consists of the nodes that will be generated by the depth-first search; that is, all the branches to the right of the first infinite branch are pruned out. As we show later, our depth-first model of the embedding corresponds to the Prolog-tree of the corresponding program. The other two models correspond to the LD-tree of the same program. Since a Prolog tree of a program is a subset of its LD-tree, in the rest of the thesis we will for simplicity consider LD-trees, and mention explicitly when some branches become unreachable due to depth-first search.

The concept of *predicates* plays a central role in this thesis, and our use of this term differs from the standard nomenclature of Prolog. In Prolog terminology, “predicate” is synonymous with “relation symbol”, and “definition” of a predicate r is the set of all clauses in a program that have r in their heads. In Prolog, as in logic programming, the syntactic notion of a definition, in its own right, does not have an attributed meaning. In our setting, a “definition” is used as a composite function, where the components correspond to the basic constructs of Prolog, and these functions behave as *semantic objects that we can analyse and reason about*. In our search for a name, we found the word “definition” too vague and has a syntactic connotation; the term “atom” misleading, because our relation definitions are not necessarily atomic; the term “query” unsatisfactory, because, even though in our equational setting, a definition is semantically equivalent to a “query”, this term implies a call to evaluation, while we want to emphasise the *compositional meaning* of these objects.

In the end, for a lack of a better word, we have decided to settle for the word “*predicate*”, which we use as the Haskell object, a function, which is the result of an embedding of a Prolog definition in Haskell. It is the meaning of such predicates, both operational, compositional and denotational, that this thesis concerns itself with, and the applications of these semantics. We also use the word “*relation*” to mean a Haskell function which expects a tuple of terms, and returns a predicate as defined above. We return to this discussion in the next chapter.

Our embedding can also compute a restricted version of negation: we allow predicates in a query to be negated, but only guarantee correct behaviour if they are ground. Computationally, we implement this as “finite failure”, which is an extension of LD-resolution that enables one to express, via a call *not p*, that all attempts to prove p shall fail in a finite time. The reason for restricting the computation to ground negation is that with the standard LD-resolution, negation can lead to unsoundness when it appears in non-ground queries; in addition, it results in incompleteness when applied to activated non-ground calls.

For the sake of efficiency and the ease of programming, Prolog has several features that do not correspond with the declarative nature of logic programming. Negation is one of these features; there are other, such as *cut*, *assert*, and *retract*. In this thesis we do not deal with such non-declarative features, although an embedding of such features in a functional setting might well constitute a topic for our future investigation. Since there is no scope for confusion, by “Prolog” we shall henceforth mean “pure Prolog”.

However, as discussed earlier, we do not restrict the search strategy to depth-first search, so we do not only embed pure Prolog, but a general pure logic programming language that computes by means of LD-resolution. It is an interesting point that the embedding can be made parametric in the search rule without major changes, but not in the selection rule. The reason for this is that any non-leftmost variation of the selection rule would need to alter the stack-like discipline that our embedding obeys. It would involve a more complex interaction between the conjuncts of predicates, so some sort of AND-parallelism would be needed. This is an area for our further work, and could possibly be connected to the work of McPhee in [85].

2.3 Functional programming

We now proceed to describe the basic concepts from pure functional programming that are crucial for understanding of our embedding and equational proofs. However, some knowledge of lazy functional programming is assumed throughout the thesis, and for additional reading the reader is referred to [17] by Bird.

A functional program is a collection of function definitions. Such definitions are often expressed as a set of guarded recursion equations, each defining a relationship between inputs and outputs. For example, the Haskell program that corresponds to the predicate *append* from the previous section is:

$$\begin{aligned}&\mathit{append}\ [\,]\ xs = xs\\ &\mathit{append}\ (x : xs)\ ys = x : (\mathit{append}\ xs\ ys)\end{aligned}$$

Functional application associates to the left, so *append* $xs\ ys$ is equivalent to $(\mathit{append}\ xs)\ ys$. The definition above is *curried*, so the two arguments can be provided one at a time, each time resulting in a function requiring one less argument. Lists are constructed from the constructor [ ], representing the empty list, and constructor (:), representing the addition of an element to the front of the list. The syntax $[1, 2, 3]$ is used to denote the list $1 : 2 : 3 : [\,]$.

The *composition* of two functions f and g is denoted by $f \cdot g$ and defined as:

$$(f \cdot g)\ x = f\ (g\ x).$$

Functions like $(\cdot)$ which are written using infix notation are called *operators*. Composition is an associative function. It is also a *higher-order* function, because it takes functions as its input, and returns a new function as its result. Higher-order functions are extremely useful for capturing "patterns" of computation, and we base our implementation and program transformation on their use. They have the effect of removing the recursion from a program, and replace reasoning by induction with equational reasoning.

Haskell has *strong typing*: the only expressions regarded as well-formed are those that can be assigned a type, according to a certain discipline. Such typing is useful for our embedding: the types are associated with operations, and they give us much assistance in error detection and it steers us into a certain discipline of thought.

An expression in a functional program is evaluated by reducing it to its simplest equivalent form according to the rewrite rules given by the function definitions of the program. If it cannot be reduced further, the expression is in a canonical or *normal form*. For some well-typed expressions, the process of reduction may never stop; for example, the *Integer* typed expression *infinity*:

$$\mathit{infinity} = \mathit{infinity} + 1,$$

never produces an answer, that is, it is not clear what value it corresponds to. Such expressions are identified with a special value $\bot$, denoting the undefined value of any type. In a lazy functional language, an application of a function to an expression of value $\bot$ does not necessarily result in another undefined value. This result depend on the *strictness* of the given function in the argument where $\bot$ appears. A function f is *strict* if $f\ \bot = \bot$. When a function returns $\bot$, we shall also say that it *diverges*. For example, given a function *multiply*:

$$\mathit{multiply}\ (x, y) = \mathit{if}\ x == 0\ \mathit{then}\ 0\ \ \mathit{else}\ x * y,$$

in the expression *multiply* $(0, \mathit{infinity})$ has value 0, while *multiply* $(\mathit{infinity}, 0)$ has value $\bot$, that is, this computation diverges. Lazy functional languages also allow computation on *infinite data structures*, such as infinite lists, trees etc. For example, the form [1..] is a Haskell shorthand for the infinite list of all consecutive integers starting from 1. Finally, lazy functional programs may also operate on *partial* lists, such as the list $1 : 2 : 3 : \bot$. In fact, the infinite list [1..] may be viewed as the limit of the infinite sequence of partial lists:

$$\bot, 1 : \bot, 1 : 2 : \bot, 1 : 2 : 3 : \bot, \ldots$$

The declarative meaning of functional programs is typically given by some version of (typed or untyped) λ-calculus, which is then given a denotational semantics. The two most important ways of reducing expressions in λ-calculus are *applicative-order* and *normal-order evaluation*. The first one will evaluate all the argument expressions before they are passed to the function's body, while the second passes the argument expressions unevaluated into the function, and delays the evaluation until they are required. The implementation of the second strategy, where graph-reduction is used to ensure that each expression is evaluated at most once, is referred to as *lazy* evaluation. Under lazy evaluation a function f might return a value even though some of its arguments cause an error or fail to terminate. If a function has a head normal form, it will be found under lazy evaluation, and the evaluation will require no more (and possibly fewer) steps than eager evaluation.

The program transformation technique used in chapters 9 and 10 depends on the principle of referential transparency: an expression can always be substituted for its value everywhere that it occurs. Any equality that holds in the program can be used in this manner. The principle of *extensionality* means that two functions are equal if they give equal results for equal arguments. By appealing to extensionality, we can prove that $f = g$ by proving that $fx = gx$ for all x. Depending on the definition of f and g, sometimes we may prove $f = g$ directly. The two techniques are called *pointwise* and *point-free* styles of proof, and both will appear in this thesis.

The higher-order function *map* applies a unary function to each element of a list. This function is *polymorphic*, as it accepts a list of any type as input; the only constraint is that the type of the input function and the list elements must agree. If the polymorphic list type is denoted by $[a]$, and a and b denote type variables, the type of *map* is:

$$map :: (a \rightarrow b) \rightarrow [a] \rightarrow [b].$$

The function *map* has a number of useful algebraic identities. The first two laws below (2.1, 2.2) express the fact that it is a functor, and they are consequently called functor laws. The function *concat* concatenates a list of lists into one long list, and the third law (2.3) expresses the fact that *concat* is a natural transformation. It is called the naturality condition for *concat*. The fourth law (2.4) is an instance of the book-keeping law. These point-free laws will be used in our program transformation proofs later.

$$map\ id = id, \tag{2.1}$$

$$map(f \cdot g) = (map\ f) \cdot (map\ g), \tag{2.2}$$

$$map\ f \cdot concat = concat \cdot map\ (map\ f), \tag{2.3}$$

$$concat \cdot concat = concat \cdot map\ concat. \tag{2.4}$$

The first laws (2.1) states that applying the identity function of every element of the list leaves the list unchanged. The second law states that applying g to each element of a list, and consequently applying f to each element of the result, is the same as applying $f \cdot g$ to the input list. Law (2.3) says that one can either concatenate a list of lists and apply f to each element of the result, or apply *map* to every sub-list and then concatenate the result. Law (2.4) says that flattening a list of lists of lists into one long list can be done by concatenation inside-out or outside-in.

Many recursive functions on lists can be defined in terms of *foldr*. This higher-order function applies a binary function to a list in a right-associative manner, reducing the list to a single value:

$$foldr :: (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b$$

$$foldr\ f\ e\ [\,] = e \tag{2.5}$$

$$foldr\ f\ e\ (x : xs) = f\ x\ (foldr\ f\ e\ xs) \tag{2.6}$$

There exists also an alternative fold operator, *foldl*, which groups the parentheses from the left. There exist three *duality theorems* which clarify the relationship between *foldr* and *foldl*, giving conditions under which one may be replaced by the other; we only list the first and the simplest one. Given an associative binary operator $\oplus$ with unit e, and a finite list xs, we have:

$$\mathit{foldr}\ (\oplus)\ e\ xs = \mathit{foldl}\ (\oplus)\ e\ xs. \tag{2.7}$$

The *fold-fusion theorems* (2.8) and (2.9) deal respectively with *foldr* and *foldl*. These two theorems have several important corollaries which are much used in program transformation. They hold under the following conditions: given functions f and g and terms a and b, such that f is strict, $f\ a = b$, and respectively, for all x and y:

$$f\ (g\ x\ y) = h\ x\ (f\ y) \qquad \text{and} \qquad f\ (g\ x\ y) = h\ (f\ x)\ y.$$

If these conditions are satisfied, we have:

$$f \cdot \mathit{foldr}\ g\ xs = \mathit{foldr}\ h\ ys, \tag{2.8}$$

$$f \cdot \mathit{foldl}\ g\ xs = \mathit{foldl}\ h\ ys. \tag{2.9}$$

These laws turn out to be essential transformation rules in program development. The patterns in these laws about folds over lists are patterns which stress the similarities with other type constructors, such as binary trees and their operations, rather than what is special to lists. Our embedding makes it simple to apply laws like these in a logic programming setting, since we can use them without having to worry about higher-order unification and other closely related matters.

We shall also make use of list comprehension notation for Haskell. It takes the form $[e|Q]$, where e is an expression and Q is a qualifier, which is a possibly empty sequence of generators and guards, separated by commas. A generator takes the form $x \leftarrow xs$, where x is a variable or a tuple of variables and xs is a list-valued expression. A guard is a boolean-valued expression.

Finally, we shall need to operate on a particular notion of a *state* in our embedding. In next chapter we show how to translate, in a *compositional* manner, predicate definitions from pure Prolog into functions in Haskell.

This composition allows us to express, among others, a conjunction of two predicates. In this case, predicate functions need to be evaluated relative to the result of a predicate which proceeds them in the conjunction. This result, or the state, or the environment of the computation, is represented as a *substitution*, and it is the input to which the given predicate is applied. The result of the function application is a collection of new, alternative, environments which are consistent with the input substitutions, and which satisfy the original predicate. The predicate functions thus behave as purely functional *state transformers*.

Chapter 3

The Embedding

In this chapter we describe a simple framework for a *compositional* translation, or embedding, of logic programs into functional ones. We also show an execution example, and begin an analysis of different search strategies.

3.1 Syntax of the embedding

As described in Chapter 1, the embedding that is the core of this thesis is compositional; this makes it very different in nature from the standard interpreters of logic programs in a functional setting. In an interpreter, logic programs are represented as closed syntactic objects, and the representation of the input program is an element of a free datatype, subject to analysis by pattern matching and recursion. However, the evaluation of the program does not necessarily proceed by recursion over the structure of this datatype – it may involve a computation of the resolvents, which are not compositional parts of the input program.

In our embedding, each definition in the object logic program becomes a separate entity in Haskell, rather than one monolithic value. The Haskell entities corresponding to definitions of the logic program are called *predicates*, and as we show later, they are implemented as functions. These functions can be combined by higher-order combinators, which correspond to the basic constructs from the logic program. So, a predicate is built inductively from other predicates using the basic combinators. One can say that the compositional approach yields the recursive evaluation in the embedding, and it also yields a compositional, that is, denotational, semantics of the original logic program.

In the proposed embedding of pure Prolog into a lazy functional language, we aim to give rules that allow any Prolog predicate to be translated into a function with the same meaning. To this end, we introduce two data types *Term* and *Predicate* into our functional language, together with the following four operations:

$$\begin{aligned}
&(\&), (\|) : \mathit{Predicate} \to \mathit{Predicate} \to \mathit{Predicate},\\
&(\doteq) : \mathit{Term} \to \mathit{Term} \to \mathit{Predicate},\\
&\mathit{exists} : (\mathit{Term} \to \mathit{Predicate}) \to \mathit{Predicate},
\end{aligned}$$

where the *Term* argument of *exists* always holds a variable. The intention is that the operators & and $\|$ denote conjunction and disjunction of predicates, $\doteq$ forms a predicate expressing the equality of two terms, and the operation *exists* expresses existential quantification. In terms of logic programs, we will use & to join literals of a clause, $\|$ to join bodies of clauses, $\doteq$ to express the primitive unification operation, and *exists* to introduce fresh local variables in a clause. We shall often abbreviate the expression *exists* $(\lambda x.\ r\ x)$ by the form $(\exists x.\ r\ x)$ in this thesis, although the longer form shows how the expression can be written in any lazy functional language that has λ-expressions. We shall also write $(\exists x_1, x_2.\ r(x_1, x_2))$ for $(\exists x_1.\ (\exists x_2.\ r(x_1, x_2)))$.

The four operations &, $\|$, $\doteq$, and *exists* suffice to translate any pure logic program, provided we are prepared to exchange pattern matching for explicit equations, to bind local variables with explicit quantifiers, and to gather all the clauses defining a predicate into a single equation. These steps can be carried out systematically, and could easily be automated. The result is that the implicational form of the original logic program is replaced an equational form, such that each relation definition from the logic program becomes a function definition in Haskell.

This transformation is a well known technique in logic programming. It is called the *completion* process of predicates, and was first proposed in 1987 by Clark [26] in order to deal with the semantics of negation in definite logic programs. Using the same process, Clark translates logic programs to first-order formulas, while we translate them to Haskell functions. Given an appropriate equality theory, it can be shown that this process preserves the atomic consequences for the original predicate, that is, its declarative reading. The syntactic differences are minimal, and simply a matter of Haskell syntax; the interesting difference is that, by having an underlying operational semantics of equational term-rewriting, we may also express and emphasise the operational semantics of the syntactic entities that this translation introduces in Haskell; namely, the combinators &, $\|$, $\doteq$, and $\exists$.

Clark's completion of a logic program P consists of an application of the following five steps. Here x_i denotes a fresh variable, $\bar{x}$ and $\bar{y}$ denote variable sequences, and **true**, **false**, $\wedge$, $\vee$, $\exists$ and $\leftrightarrow$ denote the standard first-order logic primitives and operators, while F stands for a first-order formula. For each relation r in the program P:

1. Remove patterns from heads: replace each clause $r(t_1, \ldots, t_n) \leftarrow q$ in P by the formula $r(x_1, \ldots, x_n) \leftarrow x_1 = t_1 \wedge \ldots x_n = t_n \wedge q$.
2. Quantify local variables: replace each formula $r(\bar{x}) \leftarrow q$ obtained in the previous step by $r(\bar{x}) \leftarrow \exists \bar{y}.\ q$, where $\bar{y}$ is the sequence of variables that occur in q but not in $\bar{x}$; that is, the list of variables from the original clause.
3. Introduce explicit disjunction: replace all formulas $r(\bar{x}) \leftarrow F_1, \ldots,$ $r(\bar{x}) \leftarrow F_n$ obtained in the previous step, such that have the relation r on the left hand side, with the single formula $r(\bar{x}) \leftarrow F_1 \vee \ldots \vee F_n$. If $F_1 \vee \ldots \vee F_n$ is empty, replace it by **true**.
4. Introduce explicit failure: add a formula $r(\bar{x}) \leftarrow$**false** for each relation r not appearing in the head of any clause in P.
5. Replace implications by equivalences: replace each formula $r(\bar{x}) \leftarrow F$ with $\forall \bar{x}.\ (r(\bar{x}) \leftrightarrow F)$.

According to Clark, this first-order logic predicate is equivalent to the set of substitutions, represented in an equational form, computed by SLD-resolution for the original logic program. In a sense, Clark represents the

SLD-tree for the original logic program in a first-order logic form. However, the standard logic operators which he introduces in the process do not have the same operational behaviour as their implicit counterpart from logic programming. The left-to-right selection rule and the top-down sequencing of clauses do not agree with the commutativity of $\wedge$ and $\vee$; the computation of answers, both in Prolog and in our embedding, does not return a set but some implementable representation of it, such as a sequence. Therefore, we loose additional Boolean properties, such as the left distributivity of $\wedge$ through $\vee$, and so on. These differences are discussed in detail in Chapter 4.

To arrive at an embedding in Haskell, we need to translate and implement the syntax of first-order logic in Haskell. We introduce six new functions: &, $\|$, $\doteq$, $\exists$, *true* and *false*, and use them to replace, respectively, $\wedge$, $\vee$, $=$, $\exists$, **true** and **false**. We take care to implement them in such a way that their behaviour corresponds to LD-resolution, in a compositional way.

We also need to represent Prolog terms, such as lists, in Haskell. For type technical reasons discussed later, we cannot simply use terms in Haskell. So, we use *nil* for the value of type *Term* that represents the empty list, and we write *cons* for the function on terms that corresponds to the Prolog list constructor [|]. We assume the following order of precedence on the operators, from highest to lowest: $\doteq, \&, \|, \exists$.

As an example, we take the well-known program for *append*:

$$\begin{aligned}&\mathit{append}([\,], c, c).\\ &\mathit{append}([a|b], d, [a|c]) \leftarrow \mathit{append}(b, d, c).\end{aligned}$$

As a first step, we perform *homogenisation*: we remove any patterns and repeated variables from the head of each clause, replacing them by explicit equations written at the start of the body. These equations are computed by unification in Prolog.

$$\begin{aligned}&\mathit{append}(x, y, z) \leftarrow x = [\,],\ y = z.\\ &\mathit{append}(x, y, z) \leftarrow x = [a|b],\ z = [a|c],\ \mathit{append}(b, y, c).\end{aligned}$$

Now the head of each clause contains only a list of distinct variables, and renaming can ensure that these lists of variables are the same in every clause.

Further steps of the translation consist of joining the clause bodies with the $\|$ operator and the literals in a clause with the & operator, existentially quantifying any variables that appear in the body but not in the head of a clause, and using our notation for lists:

$$\begin{aligned}
&\mathit{append}(x, y, z) = \\
&\qquad (x \doteq [\,] \;\&\; y \doteq z) \\
&\quad \| \; (\exists a, b, c.\; x \doteq [a|b]) \;\&\; z \doteq [a|c] \;\&\; \mathit{append}(b, y, c)).
\end{aligned} \tag{3.1}$$

The function *append* defined by this recursive equation has the type:

$$\mathit{append} :: \; (\mathit{Term}, \mathit{Term}, \mathit{Term}) \rightarrow \mathit{Predicate}.$$

Although we have shown how the translation from Prolog to Haskell can be carried out in a way that preserves syntactic well-formedness – so that, given operators with the appropriate types, we obtain a valid Haskell program – we have given no implementation of these operators. Also, despite the fact that we can superficially read the Haskell program as a logical formula, with a meaning equivalent to the original program, we have done nothing so far to ensure that the Haskell program (read as a Haskell program) has any semantic relation to the Prolog program.

However, the relationship between the Prolog predicate and the Haskell function extends beyond their declarative semantics. The next section contends that the *procedural* reading of the Prolog predicate is also preserved through the implementation of the functions & and $\|$.

3.2 Implementation of the embedding

Before we begin describing the implementation, we stress once again that our implementation is *not* the same as building an interpreter. We do not extend the base functional language; rather, we implement *in* the language a set of functions designed to support unification, resolution and search. Also, this is a simple and necessarily inefficient implementation; in Chapter 11 we briefly consider the prospects for a more efficient implementation.

It should also be stressed that the implementation presented in this section is just one of the several possible implementations of &, $\|$, $\doteq$ and $\exists$ such that the declarative and procedure reading of the corresponding logic program are preserved. This is the implementation that conforms to the depth-first search of Prolog. Later, we shall be able to give alternative definitions that correspond to breadth-first search, or other search strategies based on the search tree of the program.

We now proceed to give definitions to the type of predicates and to the four basic operations. The key idea is that each predicate is a function that takes an *answer*, representing the state of knowledge about the values of variables at the time the predicate is solved, and produces a lazy stream of answers, each corresponding to a solution of the predicate that is consistent with the input. This approach is similar to that taken by Wadler [141] and by Jones and Mycroft [70]. An unsatisfiable query results in an empty stream, and a query with infinitely many answers results in an infinite stream.[1]

$$\textbf{type}\ \mathit{Predicate} = \mathit{Answer} \to \mathit{Stream}\ \mathit{Answer}.$$

An answer is (in principle) just a substitution. However, it will be necessary in the implementation of *exists* to generate fresh variables not in used before in the computation. We provide for this by augmenting the substitution with a counter that tracks the number of variables that have been used so far, so that a fresh variable can be generated at any stage by incrementing the counter.

A substitution is represented as a list of (variable, term) pairs:

$$\textbf{type}\ \mathit{Answer} = (\mathit{Subst}, \mathit{Int}),$$
$$\textbf{type}\ \mathit{Subst} = \mathit{List}\ (\mathit{Var}, \mathit{Term}),$$

where the data-type *Term* is a straightforward implementation of Prolog's term type, and the data-type *Var* is a disjunctive union of user provided variable names, which are strings, and automatically generated variables,

[1]For clarity, we use the type constructor *Stream* to denote possibly infinite streams, and *List* to denote finite lists. In a lazy functional language, these two concepts share the same implementation.

which are represented by integers:

$$\textbf{data}\ \mathit{Term} = \mathit{Func}\ \mathit{Fname}\ (\mathit{List}\ \mathit{Term}) \mid \mathit{Var}\ \mathit{Vname},$$
$$\textbf{type}\ \mathit{Fname} = \mathit{String},$$
$$\textbf{data}\ \mathit{Vname} = \mathit{Name}\ \mathit{String} \mid \mathit{Auto}\ \mathit{Int}.$$

The explicit use of the constructor *List* in the definition of *Func* above highlights that all these structures are intended to be finite; they are also meant to be total. Constants are functions with arity 0, in other words they are given empty argument lists.

For example, the Prolog list $[a, b]$ can be represented in the embedding as *Func* "*cons*" [*Func* "*a*" [ ], *Func* "*cons*" [...]]. With the use of the simple auxiliary functions *cons*, *atom* and *nil* the same list can be written more conveniently as the expression *cons* (*atom* "*a*") (*cons* (*atom* "*b*") *nil*) in Haskell. In our examples, we shall further shorten this to read just $[a, b]$, and abbreviate variables *Var*(*Name* "*x*") as x and *Var*(*Auto* i) as v_i.

Returning to the type of the function *append*, when provided with the following triple of terms, the function call:

$$\mathit{append}([1], [2], x)$$

returns a *Predicate*. This, in turn, is a function which expects an answer, and returns a stream of answers. The usual initial *Answer* is a pair consisting simply of the empty substitution and the zero variable counter, $([\,], 0)$. When applied to this argument, the predicate above returns the same set of solutions, the singleton $\{x/[1, 2]\}$, as would the corresponding Prolog query.

We can now give definitions for the four operators. The operators & and $\|$ act as predicate combinators. The $\|$ operator simply concatenates the streams of answers returned by its two operands:

$$(\|) :: \mathit{Predicate} \rightarrow \mathit{Predicate} \rightarrow \mathit{Predicate}$$
$$(p_1 \parallel p_2)\ x = p_1\ x \mathbin{+\!\!+} p_2\ x. \tag{3.2}$$

This definition implies that the answers are returned in a left-to-right order as in Prolog. If the left-hand argument of $\|$ is unsuccessful and returns an empty answer stream, it corresponds to a finite but unsuccessful branch of the search tree in Prolog and backtracking is simulated by evaluating the right-hand argument.

For the & operator, we start the evaluation of $p_1 \,\&\, p_2$ by applying the predicate p_1 to the incoming answer; this produces a stream of answers, to each of which we apply the predicate p_2. Finally, we concatenate the resulting stream of streams into a single stream:

$$\begin{aligned}&(\&) :: \mathit{Predicate} \rightarrow \mathit{Predicate} \rightarrow \mathit{Predicate}\\ &p_1 \,\&\, p_2 = \mathit{concat} \cdot \mathit{map}\ p_2 \cdot p_1.\end{aligned} \tag{3.3}$$

Because of Haskell's lazy evaluation, the function p_1 returns answers only when they are needed by the function p_2. This corresponds with the backtracking behaviour of Prolog, where the predicate $p_1 \,\&\, p_2$ is implemented by enumerating the answers of p_1 one at a time and filtering them with the predicate p_2. Infinite lists of answers in Prolog are again modelled with infinite streams.

We can also define primitive predicates *true* and *false*, one corresponding to immediate success and the other to immediate failure:

$$\begin{aligned}&\mathit{true} :: \mathit{Predicate} &&\mathit{false} :: \mathit{Predicate}\\ &\mathit{true}\ x = [x]. &&\mathit{false}\ x = [\ ].\end{aligned} \tag{3.4}$$

The pattern matching of Prolog is implemented by the operator $\doteq$. It is defined in terms of a function *unify* which implements the standard algorithm for a unification of two terms, as described in Chapter 2, relative to a given input substitution. The type of *unify* is:

$$\mathit{unify} :: \mathit{Subst} \rightarrow (\mathit{Term}, \mathit{Term}) \rightarrow \mathit{List\ Subst}.$$

The use of *List* in the return type of *unify* accounts for the fact that the unification might fail. More precisely, the result of $\mathit{unify}\ \sigma\ (t_1, t_2)$ is either $[\mu]$, where μ is a strong mgu of t_1 and t_2 with respect to σ, that is, $\mu = \sigma\eta$ where $\eta = \mathrm{mgu}(t_1\sigma, t_2\sigma)$, or $\mathit{unify}\ \sigma\ (t_1, t_2) = [\ ]$ if $t_1\sigma$ and $t_2\sigma$ are not unifiable. The coding of *unify* is otherwise routine and therefore omitted here, but can be found in Appendix A.

The $\doteq$ operator is just a wrapper around *unify* that passes on the counter for fresh variables. It is implemented by means of a list comprehension, computing the list of all (s', n), where s' is produced by the generator *unify* s (t_1, t_2):

$$
\begin{aligned}
&(\doteq) :: \mathit{Term} \to \mathit{Term} \to \mathit{Predicate} \\
&(t_1 \doteq t_2)\ (s, n) = [(s', n) \mid s' \leftarrow \mathit{unify}\ s\ (t_1, t_2)]
\end{aligned}
\tag{3.5}
$$

We shall call predicates which are built only using the $\doteq$ operator *equations*.

As stated before, the predicate $(\exists x.\ r\ x)$ is our shorthand notation for the Haskell term $\mathit{exists}(\lambda x.\ r\ x)$. The function *exists* is responsible for allocating fresh names for all the local (or existentially quantified) variables in the predicates. This is necessary in order to guarantee that the computed answer is the most general result. It is defined as the following higher-order function:

$$
\begin{aligned}
&\mathit{exists} :: (\mathit{Term} \to \mathit{Predicate}) \to \mathit{Predicate} \\
&\mathit{exists}\ \mathit{lp}\ (s, n) = \mathit{lp}\ (\mathit{makevar}\ n)\ (s, n+1),
\end{aligned}
\tag{3.6}
$$

where *makevar* n returns a term representing the n'th generated variable: *Var* (*Auto* n). For example, the result of *makevar* 6 is *Var* (*Auto* 6), which we shall henceforth abbreviate to v_6.

The slightly convoluted flow of information in the implementation of *exists* may be clarified by a small example. The argument *lp* of *exists* will be a function that expects a variable, such as $(\lambda x \to \mathit{append}(t_1, x, t_2))$. We apply this function to a newly-invented variable $v_n = \mathit{makevar}\ n$ to obtain the predicate $\mathit{append}(t_1, v_n, t_2)$, and finally apply this predicate to the answer $(s, n + 1)$, in which all variables up to the n'th are marked as used.

Finally, the function *solve* evaluates the main query. It simply applies its argument p, the predicate of the query, to the initial answer $([\,], 0)$, and converts the resulting stream of answers to a stream of strings, containing only the input variables:

$$
\begin{aligned}
&\mathit{solve} :: \mathit{Predicate} \to \mathit{Stream\ String} \\
&\mathit{solve}\ p = \mathit{map\ print}\ (p\ ([\,], \mathit{0}))
\end{aligned}
\tag{3.7}
$$

where *print* is a function that converts an answer to a string, pruning it to show only the values of named variables *Name x* from the query. This is the point where all the internally generated variables are filtered out in our present implementation. It may appear that a possibly cleaner solution is to let the ∃ operator do this filtering task before it returns. However, we can not do this in general, because local variables might be needed in the final answer, for example in under-determined predicate calls. This problem is discussed in detail in Section 4.2.

The operators presented so far are the "core" operators of the embedding. One can go further, in allowing a predicate operator *not*, which mimics finite-failure negation:

$$\begin{aligned}
&\mathit{not} :: \mathit{Predicate} \to \mathit{Predicate} \\
&\mathit{not}\ p\ (\sigma, n) = [(\sigma, n)],\ \ \mathit{if}\ \ p(\sigma, n) = [\ ] \\
&\qquad\qquad\quad\ = [\ ],\ \ \mathit{otherwise}.
\end{aligned} \tag{3.8}$$

We discuss negation further in Chapter 7. Now we proceed to clarify the computation of the embedding through a simple example.

3.3 Execution example

As described, queries in the embedding are solved by the function *solve*, yielding a stream of solutions. The lazy evaluation of Haskell ensures that these solutions are computed and printed one at a time. Take for example:

$$\mathit{solve}\ (\mathit{append}\ (a, b, [1, 2, 3])),$$

where, as stated before, the term x abbreviates $\mathit{Var}(\mathit{Name}\ x)$, the list $[x, y]$ abbreviates $\mathit{cons}(x, \mathit{cons}(y, \mathit{nil}))$, and a numeral such as 1 stands for the term *Func* "1" [ ]. We shall also use $[x|xs]$ as shorthand for $\mathit{cons}(x, xs)$, and [ ] as shorthand for the empty list *nil*.

The computation commences by *solve* providing the default answer $([\ ], 0)$.

The predicate $\mathit{append}(x, y, z)$ is defined by two branches joined by $\|$. Since the operator $\|$ in the depth-first model computes as a concatenation of two lazy streams, the system will evaluate, in turn, each of the $\|$ branches on the input $([\,],0)$:

$$\begin{aligned}
&\mathit{append}\,(a,b,[1,2,3])\ ([\,],0)\\
&= \qquad (a \doteq \mathit{nil}\ \&\ b \doteq [1,2,3])\ ([\,],0)\\
&\quad +\!\!+\ (\exists x,y,z.\ a \doteq [x|y]\ \&\ [1,2,3] \doteq [x|y]\ \&\ \mathit{append}(y,b,z))\ ([\,],0)).
\end{aligned}$$

We now look at each of these two branches in turn.

In the first branch, the & operator is applied next. By the definition of &, *map* applies the equation $b \doteq [1,2,3]$, to each of the answers in the stream resulting from the computation of the equation $a \doteq \mathit{nil}$ on the given input.

Expressed as a rewriting, the computation is as follows:

$$\begin{aligned}
&\mathit{concat}\ (\mathit{map}\ (b \doteq [1,2,3])\ ((a \doteq \mathit{nil})\ ([\,],0))\ )\\
&= \mathit{concat}\ (\mathit{map}\ (b \doteq [1,2,3])\ [([(a,\mathit{nil})],0)]\ )\\
&= \mathit{concat}\ [(b \doteq [1,2,3])\ ([(a,\mathit{nil})],0)]\\
&= \mathit{concat}\ [[([(a,\mathit{nil}),(b,[1,2,3])],0)]]\\
&= [([(a,\mathit{nil}),(b,[1,2,3])],0)].
\end{aligned}$$

In this rewriting, first the $\doteq$ operator unifies the variable a with the constant *nil*, relative to the empty input substitution $[\,]$, resulting in the singleton answer list $[([a/\mathit{nil}],0)]$. Then, *map* applies the equation $b \doteq [1,2,3]$ to the only element of this list, resulting in the singleton $[[([a/\mathit{nil},\ b/[1,2,3]],0)]]$. This result is next flattened by *concat*.

Because of laziness, in the computation of $\mathit{map}\ p_2\ (p_1(\sigma))$, the answers in the stream $p_1(\sigma)$ are returned in response to *map* needing a next element to compute on. When p_1 is an equation, as in the computation above, *map* only gets a finite list as an input; but in other examples $p_1(\sigma)$ may well compute an infinite stream, in which case *map* computes ad infinitum.

The first element in the stream returned by *append* is thus computed. Finally, the lazy evaluation of *solve* filters out the input variables a and b in the composed substitution and prints the first answer: $\{a/\mathit{nil}, b/[1,2,3]\}$.

With this, the first branch of $\|$ terminates, and the computation of the second branch of $append(a, b, [1, 2, 3])$ begins:

$$(\exists x, y, z.\ a \doteq [x|y]\ \&\ [1, 2, 3] \doteq [x|z]\ \&\ append(y, b, z))\ ([\,], 0).$$

The $\exists$ operator (which here is a shorthand for three consecutive applications of $\exists$) produces three fresh automatic variables v_0, v_1, v_2, and we get:

$$(a \doteq [v_0|v_1]\ \&\ [1, 2, 3] \doteq [v_0|v_2]\ \&\ append(v_1, b, v_2))\ ([\,], 3),$$

where the last part of the input pair indicates that three variables have been generated so far. The first two predicates in this conjunction are equations, and thus have only single solutions, respectively $\{a/[v_0|v_1]\}$ and $\{v_0/1, v_2/[2, 3]\}$. These solutions are computed in a few steps similar to the ones described before, and after composing the substitution for v_0, we arrive to a new call to *append*, as given below:

$$append(v_1, b, v_2)\ ([a/[1, v_1],\ v_0/1,\ v_2/[2, 3]],\ 3). \qquad (*)$$

This call to *append* again rewrites to two $\|$ branches; the first one is:

$$(v_1 \doteq nil\ \&\ b \doteq v_2)\ ([a/[1, v_1],\ v_0/1,\ v_2/[2, 3]],\ 3)$$

After a few more steps the substitution components v_1/nil and $b/[2, 3]$ are added to the state, and the second answer, containing the combined substitution values for a and b, is returned: $\{a/[1], b/[2, 3]\}$. The remaining two answers, $\{a/[1, 2], b/[2, 3]\}$, and $\{a/[1, 2, 3], b = [\ ]\}$, follow in a similar manner from further expansion of the second $\|$ branch of the call $(*)$.

At some later point in the computation, the $\doteq$ operator fails to find any unifier relative to the input substitution. It then returns the empty stream, and since it is a part of an & expression, the empty stream will be input to *map* and *concat*, so they terminate. Finally, since this is the last part of the topmost $\|$ branch, the $+\!\!+$ performed by the outermost $\|$ also terminates, and so does *solve*.

In Chapter 7 we show that the operators $\doteq$, $\exists$, & and $\|$ behave in a way that actually simulates the effect of LD-resolution. This result rests on the algebraic properties of the operators that we present in Chapter 4.

3.4 Alternative search strategies

Our implementation of $\|$, together with the laziness of Haskell, causes the search for answers to behave like a depth-first search in Prolog: when computing $p_1\ x \mathbin{+\!\!+} p_2\ x$, all the answers corresponding to the $p_1\ x$ part of the search tree are returned before the other part is explored. Similarly, our implementation of &, through the fact that in $p_1 \,\&\, p_2$ the answers returned by p_1 become the input of p_2, reflects the left-to-right selection rule of Prolog.

An interesting question arises about how much our implementation would need to change to accommodate other search strategies or selection rules. At this stage we have deliberately chosen to simulate Prolog, so neither our search strategy nor our selection rule are fair; a fair search strategy would share the computation effort more evenly between the two branches of the computation of $\|$, and a fair selection rule would allow one to chose the literals in a different order. In this section we demonstrate that a fair search strategy can be achieved with minimal re-coding, but explain why a fair selection rule would require much more extensive changes to our system.

Regarding varying the search strategy, one possible solution (suggested by McPhee and de Moor in [86]) is to *interleave* the streams of answers to $\|$, taking one answer from each stream in turn. A function *twiddle* that interleaves two lists can be defined as follows:

$$\begin{aligned}
&\mathit{twiddle} \ :: \ [a] \to [a] \to [a]\\
&\mathit{twiddle}\ [\,]\ \mathit{ys} = \mathit{ys}\\
&\mathit{twiddle}\ (x : \mathit{xs})\ \mathit{ys} = x : (\mathit{twiddle}\ \mathit{ys}\ \mathit{xs}).
\end{aligned}$$

The operators $\|$ and & can be redefined by replacing $\mathbin{+\!\!+}$ with *twiddle* and recalling that $\mathit{concat} = \mathit{foldr}\ (\mathbin{+\!\!+})\ [\,]$:

$$\begin{aligned}
&(p_1 \parallel p_2)\ x = \mathit{twiddle}\ (p_1\ x)\ (p_2\ x)\\
&(p_1 \,\&\, p_2)\ x = \mathit{foldr}\ \mathit{twiddle}\ [\,] \cdot \mathit{map}\ p_2 \cdot p_1.
\end{aligned}$$

This implementation of & produces in a finite time solutions of p_2 that are based on later solutions $\sigma_2, \sigma_3, \ldots$ returned by p_1, even if the first such solution σ_1 produces an infinite stream of answers from p_2. Laziness is important here in ensuring that at each stage no more work is devoted to solving $p_2(\sigma_i)$ for some input answer σ_i than is needed to produce the next answer. Thus even if $p_2(\sigma_1)$ produces an infinite stream of answers, effort can move between computing successive elements of other streams $p_2(\sigma_i)$. On the other hand, the original implementation of & produces all solutions of p_2 that are based on the first solution σ_1 produced by p_1 before producing any that are based on the second solution σ_2 from p_1.

Note that this implementation of operators does *not* give breadth-first search of the search tree; it deals with infinite success but not with infinite failure. Even in the interleaved implementation, the first element of the answer list has to be computed before we can switch branches; if this takes an infinite number of steps the other branch will never be reached.

To implement breadth-first search in the embedding, the *Predicate* datatype needs to be changed. It is no longer adequate to return a single, flat stream of answers; this model is not refined enough to take into account the number of *computation steps* needed to produce a single answer. That is, we obtain just the answers, with no indication of what they cost to compute. Worse still, an infinitely failed computation has value $\perp$, and can never be distinguished from a computation that will produce answers in the future.

In a model that does contain information about the cost of each answer, the key idea is to let *Predicate* return a stream of lists of answers, where each list represents the answers reached at the same level of the search tree. The lists of answers with the same cost are always finite since in a finite program there is only a finite number of nodes at each level of the search tree.

In this model, each successive list of answers in the stream contains the answers with the same computational "cost". The cost of an answer increases with every resolution step in its computation. This can be captured by adding a new function *step* in the definition of predicates, where *step* is used as a wrapper around each function definition. For example:

$$\begin{aligned}
&\mathit{append}(x, y, z) = \\
&\qquad \mathit{step}\ ((x \doteq [\,]\ \&\ y \doteq z) \\
&\qquad\qquad \parallel (\exists a, b, c.\ x \doteq [a|b]\ \&\ z \doteq [a|c]\ \&\ \mathit{append}(b, y, c))).
\end{aligned}$$

The only change is the application of *step* to the whole right-hand side.

In the depth-first model, *step* is the identity function on predicates, but in the breadth-first model it is defined as follows:

$$\begin{aligned} &\mathit{step} :: \mathit{Predicate} \rightarrow \mathit{Predicate} \\ &\mathit{step}\ p\ x = [\,] : (p\ x). \end{aligned} \tag{3.9}$$

Thus, in the stream returned by *step p*, there are no answers of cost 0, and for each n, the answers of *step p* with cost $n+1$ are the same as the answers of p that have cost n. This is recorded by the fact that they are elements of the $(n+1)$'st list in the answer stream to the query *step p*. Further, the implementations of the *Predicate* combinators $\|$ and & need to be changed so that they no longer operate on streams but on streams of lists. They must preserve the cost information that is embedded in the input lists.

To implement *both* depth-first search and breadth-first search in the embedding, the model has to be further refined. It is not sufficient to implement predicates as functions returning streams of answer lists; they have to operate on lists of trees. Again, the implementation of operators $\|$ and &, and of the function *step*, needs to be adjusted to the new type of predicates. We postpone the presentation of these two models, and the definitions involved, to Chapter 5.

In spite of the changes to *Predicate* type, the implementations of the operators $\|$ and & in each of the three models remains strikingly concise and similar. Their closely parallel definitions hint at a deeper algebraic structure, and in fact the definitions are all instances of the so-called Kleisli construction from category theory. Even greater similarities between the three models exist, and we give a more detailed study of the relation between the three in Chapter 6.

Concerning varying the selection rule, much greater changes are required. As noted above, laziness of Haskell and the implementation of & enforce a left-to-right selection discipline. To override this, we would need a sophisticated interaction between the two conjuncts corresponding to AND-parallelism. Referring to Kowalski's formula "*logic programming = logic + control*", it is interesting that the search element of control is so much more susceptible to varying than the selection element. This is because $\|$ parallelism requires no communication between processes, unlike & parallelism.

We now defer the discussion about the search models until later, and turn our attention to the algebraic properties of the operators described in this chapter. These properties are important, because our goal is to reason about logic programs written in our embedding, using properties of operators that are shared by all alternative definitions of the operators, not one single set of definitions. Also, an important aspect of algebraic program transformation is its compositional nature, and the algebraic properties of the operators are a vital part of this compositionality.

Chapter 4

Algebraic Semantics

The operators &, $\|$, $\doteq$ and $\exists$ enjoy many algebraic properties as a consequence of their definitions. This chapter lists a number of common laws and proves their validity for the embedding. It lists only those laws that are valid for both predicate calculus and for our embedding. The laws presented here are expressed as equations, and by equations we mean extensional equality between Haskell programs; in other words, these equalities mean that given the same input, the two programs return the same output.

4.1 Laws regarding & and $\|$

One of the strengths of the embedding is that all the algebraic properties of the operators are simple to prove, using equational reasoning. The laws follow directly from definition of the operators and from the standard properties of Haskell's list operators $+\!\!+$, *map*, *concat*, and functional composition. Most of these standard properties are listed in Chapter 2 as laws (2.1–2.4), and the remaining ones can be found in standard texts on functional programming, for example, in [17] by Bird.

There is an abstract approach to proving some of the properties listed here, through the use of category theory. Given our implementation, it is a well known fact that *map*, *concat* and *true* form a structure that category theory calls a *monad*, in particular the Stream monad. The composition operator

& can be obtained from this monad by a standard construction called *Kleisli composition*. Consequently, & must be associative with unit element *true*.

However, here we wish to show that these properties of the logic programming primitives & and *true*, and several others regarding also $\|$ and *false*, can be proved with no reference to category theory. As an example, we can prove the associativity of & by the following point-free rewriting:

$$
\begin{aligned}
& (p_1 \mathbin{\&} p_2) \mathbin{\&} p_3 \\
&= \mathit{concat} \cdot \mathit{map}\ p_3 \cdot \mathit{concat} \cdot \mathit{map}\ p_2 \cdot p_1 && \text{by defn. of } \& \\
&= \mathit{concat} \cdot \mathit{concat} \cdot \mathit{map}\ (\mathit{map}\ p_3) \cdot \mathit{map}\ p_2 \cdot p_1 && \text{by (2.3)} \\
&= \mathit{concat} \cdot \mathit{map}\ \mathit{concat} \cdot \mathit{map}\ (\mathit{map}\ p_3) \cdot \mathit{map}\ p_2 \cdot p_1 && \text{by (2.4)} \\
&= \mathit{concat} \cdot \mathit{map}\ (\mathit{concat} \cdot \mathit{map}\ p_3 \cdot p_2) \cdot p_1 && \text{by (2.2)} \\
&= p_1 \mathbin{\&} (p_2 \mathbin{\&} p_3). && \text{by defn. of } \&
\end{aligned}
$$

The proofs of the following properties are at least as elementary as this. The predicate *false* is a left zero for &, $\mathit{false} \mathbin{\&} p = \mathit{false}$. However, this operator is strict in its left argument, so *false* is not a right zero. If $p(\sigma) = \bot$, we get:

$$
\begin{aligned}
& (p \mathbin{\&} \mathit{false})(\sigma) \\
&= \mathit{concat}\ (\mathit{map}\ \mathit{false}\ p(\sigma)) && \text{by defn. of } \& \\
&= \mathit{concat}\ (\mathit{map}\ \mathit{false}\ \bot) && \text{since } p(\sigma) = \bot \\
&= \mathit{concat}\ \bot && \mathit{map} \text{ strict} \\
&= \bot && \mathit{concat} \text{ strict}
\end{aligned}
$$

This corresponds to the feature of Prolog that $\mathit{false} \mathbin{\&} p$ has the same behaviour as *false*, but $p \mathbin{\&} \mathit{false}$ may fail infinitely if p does.

Further, owing to the properties of $+\!\!+$ and [ ], the $\|$ operator is associative and has *false* as a left and right identity. For example:

$$
\begin{aligned}
& (p_1 \mathbin{\|} \mathit{false})(\sigma) \\
&= p_1(\sigma) +\!\!+ \mathit{false}(\sigma) && \text{by defn. of } \| \\
&= p_1(\sigma) +\!\!+ [\,] && \text{by defn. of } \mathit{false} \\
&= p_1(\sigma) && \text{by defn. of } +\!\!+
\end{aligned}
$$

Other identities that are satisfied by the connectives of propositional logic are not shared by our operators because, in our stream-based implementation, answers are produced in a definite order and with definite multiplicity. This behaviour mirrors the operational behaviour of Prolog. For example, the $\|$ operator is not idempotent, because $\mathit{true} \parallel \mathit{true}$ produces its input answer twice as an output, but *true* itself produces only one answer. The & operator also fails to be idempotent, because the predicate

$$(\mathit{true} \parallel \mathit{true}) \mathbin{\&} (\mathit{true} \parallel \mathit{true})$$

produces the same answer four times rather than just twice. We might also expect for & to distribute over $\|$ from the left, that is,

$$p_1 \mathbin{\&} (p_2 \parallel p_3) = (p_1 \mathbin{\&} p_2) \parallel (p_1 \mathbin{\&} p_3),$$

but this is not the case. For a counterexample, take for p_1 the predicate $x \doteq a \parallel x \doteq b$, for p_2 the predicate $y \doteq c$, and for p_3 the predicate $y \doteq d$. Then the left-hand side of the above equation produces the four answers $\{x/a, y/c\}$; $\{x/a, y/d\}$; $\{x/b, y/c\}$; $\{x/b, y/d\}$ in that order, but the right-hand side produces the same answers in the order $\{x/a, y/c\}$; $\{x/b, y/c\}$; $\{x/a, y/d\}$; $\{x/b, y/d\}$. However, as proved in Chapter 7 for law 7.1, this property is true when p_1 is determinate, that is, when it returns at most one answer.

On the other hand, the other distributive law,

$$(p_1 \parallel p_2) \mathbin{\&} p_3 = (p_1 \mathbin{\&} p_3) \parallel (p_2 \mathbin{\&} p_3),$$

does hold, and it is vitally important to the unfolding steps of program transformation. The simple proof depends on the fact that both *map r* and *concat* are homomorphisms with respect to $+\!\!+$:

$$\begin{aligned}
&((p_1 \parallel p_2) \mathbin{\&} p_3)\ \sigma \\
&= \mathit{concat}\ (\mathit{map}\ p_3\ (p_1\ \sigma +\!\!+ p_2\ \sigma)) && \text{by defn. of } \|, \& \\
&= \mathit{concat}\ (\mathit{map}\ p_3\ (p_1\ \sigma) +\!\!+ \mathit{map}\ p_3\ (p_2\ \sigma)) && \mathit{map}/+\!\!+ \\
&= \mathit{concat}\ (\mathit{map}\ p_3\ (p_1\ \sigma)) +\!\!+ \mathit{concat}\ (\mathit{map}\ p_3\ (p_2\ \sigma)) && \mathit{concat}/+\!\!+ \\
&= ((p_1 \mathbin{\&} p_3) \parallel (p_2 \mathbin{\&} p_3))\ \sigma. && \text{by defn. of } \&
\end{aligned}$$

It might be seen as a weakness of our approach based on the implementation of the embedding that these properties must be expressed in terms of the weak notion of a *predicate* definable in terms of our operators, when an interpreter for Prolog written in Haskell would allow us to formulate and prove them as an inductive property of *program texts*. We believe that this is a price well worth paying for the simplicity and the clear declarative and operational semantics of our embedding.

In summary, the algebraic laws regarding the operators & and $\|$ cover the part of the embedding that has to do with searching:

$$(p_1 \mathbin{\&} p_2) \mathbin{\&} p_3 = p_1 \mathbin{\&} (p_2 \mathbin{\&} p_3), \tag{4.1}$$
$$p \mathbin{\&} \mathit{true} = \mathit{true} \mathbin{\&} p = p, \tag{4.2}$$
$$\mathit{false} \mathbin{\&} p = \mathit{false}, \tag{4.3}$$
$$(p_1 \parallel p_2) \parallel p_3 = p_1 \parallel (p_2 \parallel p_3), \tag{4.4}$$
$$p \parallel \mathit{false} = \mathit{false} \parallel p = p, \tag{4.5}$$
$$(p_1 \parallel p_2) \mathbin{\&} p_3 = (p_1 \mathbin{\&} p_3) \parallel (p_2 \mathbin{\&} p_3). \tag{4.6}$$

It is worth noting that the laws that are absent from the list tell us as much about the computational side of logic programming as the ones present: the & and $\|$ encountered here and in standard logic programming have a declarative reading in terms of conjunction and disjunction in predicate logic, but there are significant differences between the algebraic properties of our operators and those of the corresponding operators in logic. Another interesting aspect of this list of laws is that it can be proved to be complete, in the sense that all the search models, as formally defined in Chapters 5 and 6, will share exactly these six algebraic laws regarding & and $\|$. We return to this proof in Chapter 5, and now we consider the algebraic properties of the remaining operators.

4.2 Laws regarding $\exists$

The algebraic properties of $\exists$ are important for program transformation because they allow local variables to be introduced and eliminated. As before, we wish these laws to preserve operational equivalence between predicates, that is, equality between the streams of answers that are returned. Unfortunately, in its strict sense such equality does not permit some desirable laws for $\exists$, such as the reordering of the bound variables in $(\exists x, y.\ r(x, y))$ and $(\exists y, x.\ r(x, y))$, because the variables generated during execution of the two programs would be named differently, and these names may appear in answer substitutions.

To avoid referring to equality "up to variable renaming", we will assume that, upon termination, each $\exists$ predicate transforms its answers into a *standard form* in which the irrelevant differences in local variables are neutralised. Below we outline such a transformation, to show that it is implementable; however, since the differences are only a minor syntactic issue and we do not wish to complicate the formalities of proofs about our implementation unnecessarily, we choose not to use this standardisation in our implementation.

It may appear that the simplest solution would be to let each $\exists$, after its termination, remove the variable that it introduces. The reason that we can not do this is that some invented variables might appear in the final answer.

Since equality of substitutions is extensional, the substitution components may be combined, and the *ordering* in which the substitutions occur in the answer is irrelevant. This is the main idea behind the standard form of an answer: to rename the local variables in a consistent manner which does not affect the value of the substitution.

To simplify the description of this standardisation of answers, we shall once more resort to the *append* example. Let us consider the predicate call $append([1|a], b, c)\ ([\ ], 0)$. According to the definition of *append* (3.1), and by the same reasoning as presented in Chapter 3, this call will eventually rewrite, among others, to the call:

$$(\exists x, y, z.(v_1 \doteq [x|y]\ \&\ v_2 \doteq [x|z]\ \&\ append(y, b, z)) \qquad (*)$$
$$([(a, v_1), (c, [v_0|v_2]), (v_0, 1)],\ 3).$$

We now consider a computation of one answer to this $\exists$ predicate (actually, three nested $\exists$ predicates), and describe one possible strategy for a standardisation of these answers. According to its definition, the $\exists$ in $(*)$ inserts fresh variables v_3, v_4 and v_5 for the bound x, y and z. After some more rewriting, the first answer to $(*)$ is returned:

$$\{v_0/1,\ a/[v_3|nil],\ c/[1|[v_3|v_5]],\ v_1/[v_3|nil],\ v_2/[v_3|v_5],\ v_4/nil,\ b/v_5\}.$$

We now want to compute the standard form of this answer. The question, then, is which variables may be renamed? The names of the global variables, such as a, must remain unchanged. Some local variables, produced by an outer $\exists$, such as v_0, v_1 and v_2, are relevant for further computation, because they may appear in still unresolved predicates, so they can not change either. That leaves for renaming only the variables introduced by the terminated $\exists$ calls – that is, v_3, v_4, and v_5, in this case.

One possible strategy for renaming involves the use of a separate alphabet for variable names that can be changed. For example, this alphabet may consist of names w_i, where the indices i are kept "compact" and for each new w_i, i will be the next unused index for w's. Renaming v_3, v_4, and v_5 to w_0, w_1, and w_2, we get:

$$\{v_0/1,\ a/[w_0|nil],\ c/[1|[w_0|w_2]],\ v_1/[w_0|nil],\ v_2/[w_0|w_2],\ w_1/nil,\ b/w_2\}.$$

In any answer all variables must appear either with a single occurrence in the domain, or they occur *one or more* times in the range of the substitution. If a w_i appears in the domain, it is not needed in the answer, and it may be removed. The remaining bindings (x_j, t_j) can be arranged according to an ordering on the variables x_j. Because we are using strong mgu's, this ordering will be unique for all equivalent answers. Then, all the indexes i to the w_i are reshuffled so that the first occurrence of each w_i variable happens in an increasing order of i's, and the later occurrences of w_i are renamed consistently with the first one. This results in a standard form of the answer. Returning to our example, the standard form of the answer above is:

$$\{a/[w_0|nil],\ b/w_1,\ c/[1|[w_0|w_1]],\ v_0/1,\ v_1/[w_0|nil],\ v_2/[w_0|w_1]\}$$

As it happens, in our example this answer is not used in further computation, so the bindings for v_0,v_1 and v_2 are not used, but other examples could have more conjuncts after *append* in $(*)$, which might refer to v_0, v_1 and v_2.

We now proceed to list the laws for $\exists$; they allow us to identify the conditions under which we can eliminate or rename the bound variables in predicates. If the standardisation is performed, all the laws listed below preserve operational equivalence. Their proofs are based on properties of Haskell's λ-expressions and on our implementation of $\exists$.

We have previously introduced the expression $(\exists x.\ p)$ as a shorthand for the Haskell expression $\mathit{exists}(\lambda x.\ p)$, where the function *exists* simply provides a fresh variable name and applies the λ-expression to it. This notation is convenient for writing embedded predicates and their transformations. However, for the proofs of $\exists$ properties, the λ-notation in Haskell and the language rules provide a concise framework. Therefore, in this section we will reason about the laws in Haskell notation, and at the end of this section we list the laws in the shorthand notation which we use in the rest of the thesis.

One more notational comment: in the laws below, we sometimes need to express that some variable x appears in predicate p; we can use our convention for denoting relations with r and write $r\ x$ for such p. As before, our relation r will typically be written as a Haskell λ-abstraction.

We can always consistently rename the bound variables in predicates, simply because bound variables can be renamed in Haskell, and *exists* is just a wrapper around a Haskell λ-expression. So, we have:

$$\mathit{exists}(\lambda x.\ r\ x) = \mathit{exists}(\lambda y.\ r\ y). \tag{4.7}$$

For example, in an expression $\mathit{exists}(\lambda x.\ r\ x)$, the expression r can be instantiated to $\lambda w.\ \mathit{append}(w, [3], [2, 3])$. Then we know that $\mathit{exists}(\lambda x.\ r\ x)$ rewrites to $\mathit{exists}(\lambda x.\ \mathit{append}(x, [3], [2, 3]))$, where x appears in the bound predicate. By the same reasoning, the right hand side of the equation would rewrite to $\mathit{exists}(\lambda y.\ \mathit{append}(y, [3], [2, 3]))$. These two expressions are equivalent in Haskell.

The equation (4.7) does not say anything specific about the behaviour of the $\exists$ operator in the embedding, since it holds for any λ-expression. The remaining laws depend on the properties of the $\exists$ operator as we have implemented it. For example, the law (4.8) states that the function *exists* applied to a constant function yields the same constant:

$$exists(\lambda x.\ p \mathbin{\&} r\ x) = p \mathbin{\&} exists(\lambda x.\ r\ x). \tag{4.8}$$

In program transformation this equation is used to remove from the scope of $\exists$ any parts of the predicate that do not mention the bound variable; these parts would typically already have been used in combination with the Leibnitz rule (4.28) or the rule for substitutions of equals for equals (4.12) to pass the parameters. The proof is trivial. Let x' denote the fresh variable name that *exists* supplies; we then have:

$$\begin{aligned}
& exists(\lambda x.\ (p \mathbin{\&} r\ x)) && \\
&= (\lambda x.\ p \mathbin{\&} r\ x)\ x' && \text{by } (3.6)\\
&= p \mathbin{\&} r\ x' && \beta\text{-reduction}\\
&= p \mathbin{\&} exists(\lambda x.\ r\ x). && \text{by } (3.6)
\end{aligned}$$

The proofs of the following two equations are similar. Equation (4.9) is used to eliminate parts of predicates that can not contribute to the computation of the answers. Equation (4.10) states that $\exists$ distributes through $\|$. This is a much used equality in program transformation because it helps us break apart larger predicates, and is true even without the standardisation of answers.

$$exists(\lambda x.\ x \doteq t) = true, \tag{4.9}$$

$$exists(\lambda x.\ (r_1\ x \parallel r_2\ x)) = exists(\lambda x.\ r_1\ x) \parallel exists(\lambda x.\ r_2\ x). \tag{4.10}$$

Equation (4.9) holds because, if the variable x_1 is fresh, $x_1 \doteq t$ must be unifiable. Equation (4.10) holds because in the definition of $\|$ both conjuncts are applied to the same input answer, so *exists* will supply the same fresh variable to both disjuncts.

The equation (4.11) states that two nested applications of $\exists$ can be exchanged, and is true because the sets of answers computed by $r\ x_1\ x_2$ and $r\ x_2\ x_1$ are equal when reduced to the standard form, if x_1 and x_2 are fresh. Some of the fresh variables will be numbered differently in the result, but upon renumbering of the local variables, the results would be the same. So we have:

$$\mathit{exists}(\lambda x.\ \mathit{exists}(\lambda y.\ r\ x\ y)) \ = \ \mathit{exists}(\lambda y.\ \mathit{exists}(\lambda x.\ r\ x\ y)). \tag{4.11}$$

The next law (4.12) is a special case of the one-point rule and is used for parameter passing. Intuitively, once the & operator has passed on the substitutions resulting from $x \doteq u$ to further computation, this subpredicate is superfluous. The condition in this equation is that x and u must be unifiable, that is, if $x \notin \mathit{Var}(u)$, we have:

$$\mathit{exists}(\lambda x.\ (x \doteq u \mathbin{\&} rx)) = ru. \tag{4.12}$$

The main step in the proof below is based on the law (4.28) about the substitution of equals for equals; we justify this law in the next section.

$$\begin{aligned}
&\mathit{exists}(\lambda x.\ x \doteq u \mathbin{\&} r\ x) && \\
&= x' \doteq u \mathbin{\&} r\ x' && \text{by (3.6)}\\
&= x' \doteq u \mathbin{\&} r\ u && \text{by (4.28)}\\
&= \mathit{exists}(\lambda x.\ x \doteq u) \mathbin{\&} r\ u && \text{by (3.6)}\\
&= \mathit{true} \mathbin{\&} r\ u && \text{by (4.9)}\\
&= r\ u. && \text{by (4.2)}
\end{aligned}$$

In summary, and in shorthand notation, we have the following algebraic laws regarding the operator $\exists$:

$$\begin{aligned}
(\exists x.\ r\ x) \ &= \ (\exists\ y.\ r\ y), && (4.13)\\
(\exists x.\ p) \ &= \ p, && (4.14)\\
(\exists x.\ p \mathbin{\&} r\ x) \ &= \ p \mathbin{\&} (\exists x.\ r\ x), && (4.15)\\
(\exists x.\ x \doteq t) \ &= \ \mathit{true}, && (4.16)\\
(\exists x.\ r_1\ x \parallel r_2\ x) \ &= \ (\exists x.\ r_1\ x) \parallel \exists x.\ r_2\ x), && (4.17)\\
(\exists x_1.\ (\exists x_2.\ r\ x_1\ x_2)) \ &= \ (\exists x_2.\ (\exists x_1.\ r\ x_1\ x_2)), && (4.18)\\
(\exists x.\ x \doteq t \ \mathbin{\&} \ r\ x) \ &= \ r\ t. && (4.19)
\end{aligned}$$

This list of laws is very similar to the laws given for Cylindric Algebra [62] of Henkin, Monk and Tarski, or Regular Logic [48] of Freyd and Schedrov, which extend Boolean Algebra by adding axioms to deal with existential quantifiers and equality, and allow an alternative presentation of predicate logic. However, not all of those laws hold, for the same reason as for & and $\|$: the operational semantics of logic programming reproduces the declarative semantics of predicate logic in a close but imperfect manner.

Two additional laws (4.20) and (4.21) are much used in program transformation to split predicates; both deal with scoping of $\exists$. The first follows trivially from two applications of the law (4.8). The second follows from the distributivity equation (4.10) and the equation for eliminating the unwanted quantifiers (4.8).

$$(\exists x_1.\ (\exists x_2.\ r_1\ x_1 \mathbin{\&} r_2\ x_2)) = (\exists x_1.\ r_1\ x_1) \mathbin{\&} (\exists x_2.\ r_2\ x_2), \tag{4.20}$$

$$(\exists x_1.\ (\exists x_2.\ r_1\ x_1 \parallel r_2\ x_2)) = (\exists x_1.\ r_1\ x_1) \parallel (\exists x_2.\ r_2\ x_2), \tag{4.21}$$

These equalities require rewriting to the standard forms, that is, they hold only up to variable renaming.

4.3 Laws regarding $\doteq$

From a model-theoretic point of view, it is a well known result that the completed form of a predicate has the same models as the original predicate, provided that the relation "=" is interpreted as identity, where $t_1 = t_2$ is interpreted to be true if t_1 and t_2 are interpreted as the same element of the domain. In our notation, equality between terms is denoted by $\doteq$. The following *free equality axioms* capture the prescribed behaviour of equality in an algebraic manner:

$$f(t_1, ..., t_n) \doteq f(t'_1, ..., t'_n) \ \equiv\ t_1 \doteq t'_1 \mathbin{\&} ... \mathbin{\&} t_n \doteq t'_n, \tag{4.22}$$

for each n-ary function f

$$f(t_1, ..., t_n) \doteq g(t'_1, ..., t'_m) \ \equiv\ \mathit{false}, \tag{4.23}$$

for each n-ary function f and m-ary function g such that $f \neq g$

$$x \doteq t \ \equiv\ \mathit{false}, \tag{4.24}$$

for each variable x and term t such that $x \in \mathit{Var}(t)$

$$t \doteq t \;\equiv\; \mathit{true}. \tag{4.25}$$

$$t_1 \doteq t_2 \;\equiv\; t_2 \doteq t_1. \tag{4.26}$$

$$t_1 \doteq t_3 \;\equiv\; (\exists t_2.\ t_1 \doteq t_2 \mathbin{\&} t_2 \doteq t_3). \tag{4.27}$$

Here and in the rest of this section, to avoid confusion with $\doteq$, we use $\equiv$ to denote equality between two Haskell terms. These laws recognisably characterise the rationale of the unification algorithm, as described in Chapter 2. Law (4.22) corresponds to the conditions under which two functional terms are unifiable, where the two main function symbols are equal and all the corresponding components are unifiable. Law (4.23) deals with non-unifiability due to a mismatch between function symbols, while law (4.24) deals with non-unifiability due to the failure of the occurs-check. The laws (4.25–4.27) are the standard axioms for equality. The laws (4.22) and (4.25) overlap for the case of functional terms, but we include the second one because of variables.

It is easy to see that these laws are obeyed in our implementation of $\doteq$, since $\doteq$ is implemented as a wrapper around unification, which is implemented according to the standard unification algorithm described in Chapter 2. The only additional action is that a counter for fresh variables is passed along with the computed substitution, but this counter is not affected by the computation of $\doteq$, so the interesting properties only concern the answer substitutions.

Also the Leibnitz law about substitution of equals for equals holds. It is a consequence of the implementation of & and a healthiness property which will be proved later. This property states that predicates do not differentiate between arguments which have the same value under the input substitution. We have:

$$t_1 \doteq t_2 \mathbin{\&} r\ t_1 \;\equiv\; t_1 \doteq t_2 \mathbin{\&} r\ t_2. \tag{4.28}$$

This is true because, in each case the first conjunct computes a substitution in which t_1 is unified with t_2. The & operator then passes this substitution as an input to predicates $r\ t_1$ and $r\ t_2$, and due to the property mentioned above, these predicates do not differentiate between the arguments t_1 and t_2, when applied to a substitution where these two are unified.

This last law is particularly useful for program transformation, because it is used to simulate the passing of computed substitutions between the conjuncts; it reproduces the effect of a computation of unifications and a subsequent application of the remaining predicates to this input.

The algebraic laws presented in this section are shown valid for the implementation of embedding that computes by depth-first search. The rest of the thesis extends this result in several ways. First we introduce implementations of the embedding for alternative search strategies, and use these in a categorical setting to analyse the completeness of this set of laws, and their genericity. Then we use this list of laws to argue for the correctness of the embedding, by simulating LD-resolution. Finally, we use the laws to conduct program transformation.

Chapter 5

Different Search Strategies

The basic implementation of embedding uses streams to mimic the standard depth-first strategy of Prolog. We now explore alternative implementations for two other search strategies: a model where the search tree is traversed in a breadth-first manner, and a general model which allows any traversal.

5.1 Breadth-first search

In a model that allows breadth-first search, we need to maintain the information about the computational cost for each answer. The cost of an answer is measured by the number of resolution steps required in its computation. To record this information, we let predicates in our breadth-first model return a stream of bags, or a *matrix*, of answers; each successive bag of answers in the stream contains the answers with the same computational 'cost'. In order to present simpler definitions of predicate operators, where we are not distracted by questions of the exact depth at which a predicate terminates, we let this stream of bags be infinite for all predicate calls.

The underlying implementation of bags and lists is same in a functional language, but we use the *Bag* type constructor to stress the semantical difference, and we use bag equality for all our algebraic laws. Since each bag represents the finite number of answers reached at the same depth, of the search tree, and since there are only a finite number of branches in each node in the search tree, all the bags must be finite. In such case, the bag equality is always computable. The type of *Predicate* is thus:

$$\textbf{type } \mathit{Predicate} = \mathit{Answer} \to \mathit{Matrix}\ \mathit{Answer},$$
$$\textbf{type } \mathit{Matrix}\ a = \mathit{Stream}\ (\mathit{Bag}\ a).$$

The implementations of the operators $\|$ and & need to be adapted to work on matrices and to propagate the cost information for answers.

The $\|$ operator simply zips the two matrices into a single one, by taking the bag-union $\uplus$ of all the bags of answers with the same cost, and returning a single stream of these new bags. The function *zipwith* does this pairwise union; it requires that its input lists are of the same length, but this is ensured since in the breadth-first model the streams $p_1(\sigma)$ and $p_2(\sigma)$ are infinite for any σ. So the implementation of $\|$ in the breadth-first model is:

$$(p_1 \parallel p_2)\ x = \mathit{zipwith}\ (\uplus)\ (p_1\ x)\ (p_2\ x). \tag{5.1}$$

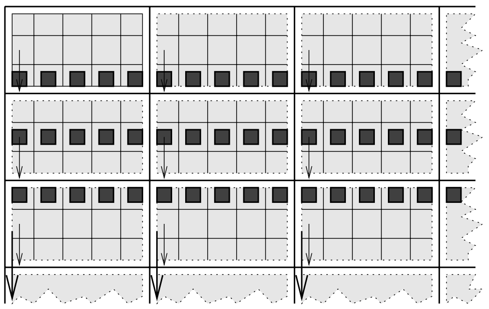
The implementation of & is harder, because for each answer it has to add the costs of its component computations. The idea is first to compute the answers to $p_1(\sigma)$, then map p_2 on each answer σ' in the resulting matrix by the matrix-map function *mmap*, and then to flatten the resulting matrix of matrices to a single matrix of answers according to the cost. This flattening is done by the *shuffle* function as explained below. The &-operator is thus:

$$p_1 \mathbin{\&} p_2 = \mathit{shuffle} \cdot \mathit{mmap}\ p_2 \cdot p_1. \tag{5.2}$$

We now try to illustrate this definition of &. Let, for the sake of brevity, S stand for for streams and L for finite lists, and recall that our bags are implemented as finite lists.

The function *mmap* is simply a composition of *map* with itself. A matrix is a stream of bags, so we write *SL*. The result of *mmap* $p_2 \cdot p_1$ is then of type *SLSL*. It can be visualised as a matrix of matrices, where each element of the outer matrix corresponds to a single answer of $p_1(\sigma)$. The first row in this matrix contains all the answers to $p_1(\sigma)$ with cost zero, the next row contains all the answers to $p_1(\sigma)$ with cost one etc. Each such answer σ' is used as an input to p_2 and consequently gives rise to a new stream of lists of answers, which are represented by the elements of the inner matrices. The first row of the resulting sub-matrix contains the answers to $p_2(\sigma')$ with cost zero and so on.

The cost of each answer to $(p_1 \mathbin{\&} p_2)(\sigma)$ is obtained by combining the costs of the two computations. For example, the answers of $\mathit{mmap}\ p_2 \cdot p_1(\sigma)$ with cost 2 are marked in the drawing below:



The rows of both the main matrix and the sub-matrices are finite, while the columns of both are infinite.

The function *shuffle* collects all the answers marked in the drawing, and all the other corresponding answers, into lists. It returns a stream of bags *SL*, where the first bag contains all the answers with combined cost zero, the next bag contains the answers with cost one, and so on.

Thus, *shuffle* is given an *SLSL* of answers, and it has to return an *SL*. Two auxiliary functions are required to do this: *diag* and *transpose*:

$$\mathit{diag} :: \mathit{Stream}\ (\mathit{Stream}\ a) \rightarrow \mathit{Stream}\ (\mathit{List}\ a)$$
$$\mathit{diag}\ \mathit{xss} = [[(\mathit{xss}\ !\ i)\ !\ (n - i) \mid i \leftarrow [0..n]] \mid n \leftarrow [0..]],$$

$$\mathit{transpose} :: \mathit{List}\ (\mathit{Stream}\ a) \rightarrow \mathit{Stream}\ (\mathit{List}\ a)$$
$$\mathit{transpose}\ \mathit{xss} = [[\mathit{xs}\ !\ n \mid \mathit{xs} \leftarrow [\mathit{xss}]] \mid n \leftarrow [0..]].$$

A stream of streams is converted to a stream of lists by *diag*, by collecting one element from each stream in a list, following the finite diagonal. So, in *diag*, if $\mathit{xss}(i, j)$ denotes the j^{th} element in the i^{th} list of *xss*, the first resulting list contains only the element $\mathit{xss}(0, 0)$, the second list contains elements

$xss(1,0)$ and $xss(0,1)$, and so on. Thus, *diag* can be used to combine the costs of answers, but first we need to convert the input *SLSL* to form which can be processed by *diag*; this is done by *transpose*. This function converts a list of streams to a stream of lists, by collecting all the first elements of each stream in the first list, and so on. Since there is a finite number of streams in the input, each resulting list must be finite.

Given *diag* and *transpose*, the function *shuffle* can be implemented as follows. The input to *shuffle* is of type *SLSL*. The application of *map transpose* swaps the middle *LS* to an *SL*, and gives *SSLL*. Then the application of *diag* converts the outermost *SS* to *SL* and returns *SLLL*. This can now be used as input to *map* (*concat* · *concat*) which flattens the three innermost levels of lists into a single list, and returns *SL*:

$$\mathit{shuffle} = \mathit{map}\ (\mathit{concat} \cdot \mathit{concat}) \cdot \mathit{diag} \cdot \mathit{map}\ \mathit{transpose}.$$

From the definitions of these functions it can be proved by structural induction on the matrices that they enjoy the following algebraic properties:

$$\mathit{mmap}\ (f \cdot g) = (\mathit{mmap}\ f) \cdot (\mathit{mmap}\ g), \tag{5.3}$$
$$\mathit{mmap}\ f \cdot \mathit{shuffle} = \mathit{shuffle} \cdot \mathit{mmap}\ (\mathit{mmap}\ f), \tag{5.4}$$
$$\mathit{shuffle} \cdot \mathit{mmap}\ \mathit{shuffle} = \mathit{shuffle} \cdot \mathit{shuffle}, \tag{5.5}$$
$$\mathit{zipwith}\ f\ (\mathit{zipwith}\ f\ l_1\ l_2)\ l_3 = \mathit{zipwith}\ f\ l_1\ (\mathit{zipwith}\ f\ l_2\ l_3), \tag{5.6}$$
$$\mathit{mmap}\ f(\mathit{zipwith}\ g\ l_1\ l_2) = \mathit{zipwith}\ g\ (\mathit{mmap}\ f\ l_1)(\mathit{mmap}\ f\ l_2), \tag{5.7}$$
$$\mathit{shuffle} \cdot \mathit{zipwith}\ (\uplus)\ l_1\ l_2 = \mathit{zipwith}\ (\uplus)(\mathit{shuffle}\ l_1)(\mathit{shuffle}\ l_2). \tag{5.8}$$

The law (5.6) requires that f is associative, and the law (5.7) holds only if f and g "commute" in the sense that $f(g\ a\ b) = g\ (f\ a)\ (f\ b)$. For these equalities to hold it is necessary to interpret the equality sign in the laws as equality of *streams of bags* rather than a stream equality. Since all the bags are finite, such equalities are computable.

The predicate *true* has to be lifted to matrices, where $\mathit{true}(\sigma)$ returns the infinite stream with the bag containing σ as its only answer at level 0, and empty bags at all other levels. The predicate *false* in the matrix model has no answers at any level so for any input answer it returns an infinite stream of empty streams $[[\,], [\,], \ldots]$. The predicate operators $\exists$, $\doteq$ and *not* can be implemented in a very similar way to their implementation in the depth-first model. So, the only other significant changes from Chapter 3 are:

$$\mathit{true}\ x = [x] : \mathit{repeat}\ [\,], \tag{5.9}$$

$$\mathit{false}\ x = \mathit{repeat}\ [\,]. \tag{5.10}$$

Finally, the book-keeping of the resolution costs for each of the answers to a predicate is implemented by the function *step*, with type $\mathit{Predicate} \to \mathit{Predicate}$. In the breadth-first model, the definition of each predicate needs to be changed to perform a call to *step* on the outermost level; the cost of computation of the predicate should be increased by one each time the definition is unfolded. Therefore, in the breadth-first model, $\mathit{step}\ p(\sigma)$ shifts all the bags of answers to $p(\sigma)$ one position to the right:

$$\mathit{step}\ p\ x = [\,] : (p\ x). \tag{5.11}$$

Further, all the algebraic laws for *true*, *false*, & and $\|$ listed in Chapter 4 hold in this model as well. As before, the proofs of these laws are based on equational reasoning. The proofs here correspond exactly to the ones from Chapter 4. They depend on the definitions of the operators and on the properties (5.3–5.8) of matrix functions for *mmap*, *shuffle* and *zipwith*. For example, in the proof of the associativity of $\|$, we use the associativity property of *zipwith* (5.6), and in the proof of the right-distributivity of & through $\|$ we use the distributivity properties of *mmap* and *shuffle* through *zipwith* (5.7–5.8). A thorough mathematical treatment of all the laws in this model can be found in Spivey [132]. As an example we show the proof of the associativity of & in this model:

$$\begin{aligned}
&(p_1 \mathbin{\&} p_2) \mathbin{\&} p_3 \\
&= \mathit{shuffle} \cdot \mathit{mmap}\ p_3 \cdot \mathit{shuffle} \cdot \mathit{mmap}\ p_2 \cdot p_1 && \text{by (5.2)} \\
&= \mathit{shuffle} \cdot \mathit{shuffle} \cdot (\mathit{mmap}\ \mathit{mmap}\ p_3) \cdot \mathit{mmap}\ p_2 \cdot p_1 && \text{by (5.4)} \\
&= \mathit{shuffle} \cdot \mathit{mmap}\ \mathit{shuffle} \cdot (\mathit{mmap}\ \mathit{mmap}\ p_3) \cdot \mathit{mmap}\ p_2 \cdot p_1 && \text{by (5.5)} \\
&= \mathit{shuffle} \cdot \mathit{mmap}\ (\mathit{shuffle} \cdot \mathit{mmap}\ p_3 \cdot p_2) \cdot p_1 && \text{by (5.3)} \\
&= p_1 \mathbin{\&} (p_2 \mathbin{\&} p_3). && \text{by (5.2)}
\end{aligned}$$

As mentioned earlier in this section, we choose to let all predicates in this model return in infinite stream of answer lists, where in the case of a finite LD-tree, the answer lists eventually become empty. With this choice, we compromise the information about the finiteness of the LD-tree, but in return the algebraic laws remain simple, and the same as in the other models. In the alternative implementation of this model, where the finite LD-tree corresponds to a finite lists of lists of answers, two programs that are identical for all practical purposes could differ in how many empty lists of answers they return after having returned the last "proper" answer. These empty lists may appear due to the "flattening" of the matrix. Since the equality in our algebraic laws means extensional equality of two Haskell programs, we would not be able to equate a program that returns one additional empty list with one that returns none.

The implementation of a breadth first model that incorporates the information about the finiteness of the LD-tree is, however, not difficult. The significant changes are only required for the functions *diag* and *transpose*, where one has to deal with the case where on of the sub-lists terminates before the others:

$$
\begin{aligned}
&\mathit{diag} :: \mathit{Stream}\ (\mathit{Stream}\ a) \rightarrow \mathit{Stream}\ (\mathit{List}\ a)\\
&\mathit{diag}\ [\,] = [\,]\\
&\mathit{diag}\ ([\,] : \mathit{xss}) = [\,]\ \ :\ \ \mathit{diag}\ \mathit{xss}\\
&\mathit{diag}\ ((x : \mathit{xs}) : \mathit{xss}) = [x] : \mathit{zipw}\ (+\!+)\ (\mathit{map}\ \mathit{wrap}\ \mathit{xs})\ (\mathit{diag}\ \mathit{xss}),\\
&\mathit{transpose} :: \mathit{List}\ (\mathit{Stream}\ a) \rightarrow \mathit{Stream}\ (\mathit{List}\ a)\\
&\mathit{transpose}\ [\,] = [\,]\\
&\mathit{transpose}\ (\mathit{xs} : \mathit{xss}) = \mathit{zipw}\ (+\!+)\ (\mathit{map}\ \mathit{wrap}\ \mathit{xs})\ (\mathit{transpose}\ \mathit{xss}).
\end{aligned}
$$

Here the function *zipw* corresponds to the function *zipwith* which has been adapted to deal with input lists of different lengths:

$$
\begin{aligned}
&\mathit{zipw} :: (a \rightarrow a \rightarrow a) \rightarrow \mathit{List}\ a \rightarrow \mathit{List}\ a \rightarrow \mathit{List}\ a\\
&\mathit{zipw}\ f\ \mathit{xs}\ [\,] = \mathit{xs}\\
&\mathit{zipw}\ f\ [\,]\ \mathit{ys} = \mathit{ys}\\
&\mathit{zipw}\ f\ (x : \mathit{xs})\ (y : \mathit{ys}) = f\ x\ y\ \ :\ \ \mathit{zipw}\ f\ \mathit{xs}\ \mathit{ys}.
\end{aligned}
$$

5.2 The general search model

In a model that allows the use of *both* depth-first and breadth-first search, the predicates can be modelled by functions returning lists of trees, or *forests*, of answers. The answers to a predicate call are found in the leaves of the trees in the returned forest, and the cost of each answer corresponds to its depth in the respective tree.

The motivation for choosing forests rather than just trees for the type of answers is that $\parallel$ and & cannot be cost-preserving on trees. If only trees were used, $p_1 \parallel p_2$ would have to combine their trees of answers by inserting them under a new parent node in a new tree, but that would increase the cost of each answer to $p_1 \parallel p_2$ by one. For example, the answers to $p \parallel \mathit{false}$ would in the tree model cost more than the answers to p, which would be wrong – the number of resolution steps performed is the same. Also, in the tree model the $\parallel$ operation would not be associative.

The type *Answer* is same as before. Each inner node in a tree can have an arbitrary number of children; this can be implemented by collecting all the children nodes in a new forest:

$$\begin{aligned}
&\mathbf{type}\ \mathit{Predicate} = \mathit{Answer} \to \mathit{Forest}\ \mathit{Answer},\\
&\mathbf{type}\ \mathit{Forest}\ a = \mathit{List}\ (\mathit{Tree}\ a),\\
&\mathbf{data}\ \mathit{Tree}\ a = \mathit{Leaf}\ a \mid \mathit{Fork}\ (\mathit{Forest}\ a).
\end{aligned}$$

The implementations of $\parallel$ and & operators in this model are similar to the implementations in the stream model, but have to preserve the cost information for answers. The $\parallel$ operator actually stays the same, it just concatenates the two forests of answers, since forests are lists of trees and since costs of answers do not change in the computation of $\parallel$:

$$(p_1 \parallel p_2)\ x = p_1\ x +\!\!+\ p_2\ x. \tag{5.12}$$

During the computation of $(p_1 \,\&\, p_2)(\sigma)$ in the forest model, p_1 computes its answers first and returns them as leaves of the resulting forest. Then p_2 is applied to each leaf σ' in this forest. This application is performed by the function *fmap*, and it results in a new forest of answers at each leaf. These forests are grafted into the corresponding trees by the function *fgraft*:

$$p_1 \mathbin{\&} p_2 = \mathit{fgraft} \cdot \mathit{fmap}\ p_2 \cdot p_1. \tag{5.13}$$

The functions *fgraft* and *fmap* are central for the general search model; using intuition about streams, *fgraft* can be thought of as *concat* for forests, and *fmap* as *map* for forests. They are implemented using the auxiliary functions *tgraft* and *tmap* as follows:

$$\mathit{fgraft} = \mathit{concat} \cdot \mathit{map}\ \mathit{tgraft}, \tag{5.14}$$

$$\mathit{tgraft}\ (\mathit{Leaf}\ x) = x, \tag{5.15}$$

$$\mathit{tgraft}\ (\mathit{Fork}\ x) = [\,\mathit{Fork}\ (\mathit{fgraft}\ x)\,], \tag{5.16}$$

$$\mathit{fmap}\ f = \mathit{map}\ (\mathit{tmap}\ f), \tag{5.17}$$

$$\mathit{tmap}\ f\ (\mathit{Leaf}\ x) = \mathit{Leaf}\ (f\ x), \tag{5.18}$$

$$\mathit{tmap}\ f\ (\mathit{Fork}\ \mathit{xf}) = \mathit{Fork}\ (\mathit{fmap}\ f\ \mathit{xf}). \tag{5.19}$$

From these definitions it can be proved by structural induction that *fmap* and *fgraft* are both homomorphisms with respect to $+\!\!+$, and that they share the standard properties of the functions *map* and *concat*:

$$\mathit{fmap}\ (g \cdot f) = (\mathit{fmap}\ g) \cdot (\mathit{fmap}\ f), \tag{5.20}$$

$$\mathit{fmap}\ f \cdot \mathit{fgraft} = \mathit{fgraft} \cdot \mathit{fmap}\ (\mathit{fmap}\ f), \tag{5.21}$$

$$\mathit{fgraft} \cdot \mathit{fgraft} = \mathit{fgraft} \cdot \mathit{fmap}\ \mathit{fgraft}. \tag{5.22}$$

In terms of category theory, (5.20) expresses the fact that *fmap* is a functor, as are *map* and *mmap*; (5.21) expresses that *fgraft* is a natural transformation, as are *concat* and *shuffle*; and (5.22) expresses the associativity law, and we had the same associativity properties in the previous two models. We shall return to these similarities in the next chapter.

As an example, we give the proof of (5.22). The function *fgraft* is defined through indirect recursion with the function *tgraft*, so a proof of (5.22) requires a simultaneous inductive proof of the equation (5.23):

$$\mathit{fgraft}\ \cdot \mathit{tgraft} = \mathit{tgraft} \cdot \mathit{tmap}\ \mathit{fgraft}. \tag{5.23}$$

Assuming that (5.23) holds, we prove (5.22):

$$
\begin{array}{ll}
\quad \mathit{fgraft} \cdot \mathit{fgraft} & \\
= \mathit{concat} \cdot \mathit{map}\ \mathit{tgraft} \cdot \mathit{concat} \cdot \mathit{map}\ \mathit{tgraft} & \text{by (5.14)} \\
= \mathit{concat} \cdot \mathit{concat} \cdot \mathit{map}\ (\mathit{map}\ \mathit{tgraft}) \cdot \mathit{map}\ \mathit{tgraft} & \text{by (2.3)} \\
= \mathit{concat} \cdot \mathit{map}\ \mathit{concat} \cdot \mathit{map}\ (\mathit{map}\ \mathit{tgraft}) \cdot \mathit{map}\ \mathit{tgraft} & \text{by (2.4)} \\
= \mathit{concat} \cdot \mathit{map}\ (\mathit{concat} \cdot \mathit{map}\ \mathit{tgraft} \cdot \mathit{tgraft}) & \text{by (2.2)} \\
= \mathit{concat} \cdot \mathit{map}\ (\mathit{fgraft} \cdot \mathit{tgraft}) & \text{by (5.14)} \\
= \mathit{concat} \cdot \mathit{map}\ (\mathit{tgraft} \cdot \mathit{tmap}\ \mathit{fgraft}) & \text{by (5.23)} \\
= \mathit{concat} \cdot \mathit{map}\ \mathit{tgraft} \cdot \mathit{map}\ (\mathit{tmap}\ \mathit{fgraft}) & \text{by (2.2)} \\
= \mathit{fgraft} \cdot \mathit{fmap}\ \mathit{fgraft} & \text{by (5.14,5.17)}
\end{array}
$$

To prove (5.23), we need to look at both inductive cases. The proof of the base case, $\mathit{fgraft}\ (\mathit{tgraft}\ (\mathit{Leaf}\ \mathit{xf})) = \mathit{tgraft}\ (\mathit{tmap}\ \mathit{fgraft}\ (\mathit{Leaf}\ \mathit{xf}))$, follows trivialy from the definitions (5.15) and (5.18). In the proof of the step case, if $\mathit{xft} = \mathit{Fork}\ \mathit{xf}$ and the induction hypothesis (5.22) holds of xf, we find:

$$
\begin{array}{ll}
\quad (\mathit{fgraft} \cdot \mathit{tgraft})\ (\mathit{Fork}\ \mathit{xf}) & \\
= \mathit{fgraft}\ [\,\mathit{Fork}\ (\mathit{fgraft}\ \mathit{xf})\,] & \text{by (5.16)} \\
= \mathit{concat}\ (\mathit{map}\ \mathit{tgraft}\ [\,\mathit{Fork}\ (\mathit{fgraft}\ \mathit{xf})\,]) & \text{by (5.14)} \\
= \mathit{concat}\ [\,\mathit{tgraft}\ (\mathit{Fork}\ (\mathit{fgraft}\ \mathit{xf}))\,] & \text{by } (\mathit{map}) \\
= [\,\mathit{tgraft}\ (\mathit{Fork}\ (\mathit{fgraft}\ \mathit{xf}))\,] & \text{by } (\mathit{concat}) \\
= [\,\mathit{Fork}\ (\mathit{fgraft}\ (\mathit{fgraft}\ \mathit{xf}))\,] & \text{by (5.16)} \\
= [\,\mathit{Fork}\ (\mathit{fgraft}\ (\mathit{fmap}\ \mathit{fgraft}\ \mathit{xf}))\,] & \text{by (5.22)} \\
= \mathit{tgraft}\ (\mathit{Fork}\ (\mathit{fmap}\ \mathit{fgraft}\ \mathit{xf})) & \text{by (5.16)} \\
= \mathit{tgraft} \cdot \mathit{tmap}\ \mathit{fgraft}\ (\mathit{Fork}\ \mathit{xf}) & \text{by (5.19)}
\end{array}
$$

Regarding the remainding basic predicate operators of the embedding, not much change is needed. The types are different, but the implementation of $\exists$, $\doteq$ and *not* operators is otherwise analogous to that in the depth-first and breadth-first models.

The same strong analogy holds for the primitive predicates *true* and *false*. In the depth-first model $\mathit{false}(\sigma)$ returned the empty stream and $\mathit{true}(\sigma)$

returned the singleton list containing σ. Now we make $\mathit{false}(\sigma)$ return the empty forest and $\mathit{true}(\sigma)$ returns the one-leaf forest $[\,\mathit{Leaf}\ (\sigma)\,]$:

$$\mathit{true}\ x = [\,\mathit{Leaf}\ x\,], \tag{5.24}$$

$$\mathit{false}\ x = [\ ]. \tag{5.25}$$

In this model, the function *step* pushes all the computed answers one level down the tree by adding a new parent node as a root. Given a forest of answers $p(\sigma)$, the function *step* creates a new a tree with *Fork*, and converts this tree into a singleton forest:

$$\mathit{step}\ p\ x = [\,\mathit{Fork}\ (p\ x)\,].$$

Regarding the algebraic laws for the operators, the laws that were listed in Chapter 4 hold of this model as of the previous two. In the forest model, the proofs for laws regarding *true* and *false* follow directly from the definitions of the operators; the associativity of $\|$ follows from the associativity of $+\!\!+$; the associativity of & has a similar proof as for matrices and uses (5.20–5.22); and the proof of the distributivity of & through $\|$ from the left uses the distributivity of *fgraft* and *fmap* through $+\!\!+$.

5.3 Laws regarding *step*

In Chapter 4, we have listed laws concerning the embedding that are true both in the declarative and the operational sense. However, if we wish to use the laws for program transformation and derivation, we need to step over the "operational" boundary, and be more permissive in our criteria about the usable laws. Indeed, if the transformed predicate is to become computationally superior to the starting predicate, it must have a different search tree, and all our previous laws preserve the shape of the search tree.

For the purposes of program transformation, we wish to view two programs as equal if they *compute* the same collection of answers. This is a stricter notion of equality than the one provided by the declarative reading of logic programs, according to which two programs with the same declarative semantics might produce completely different results (under some search strategies).

On the other hand, the procedural equality of logic programs that arises from the laws (4.1–4.28) is arguably too strict for interesting program transformation techniques.

The laws presented in this section equate predicates with different search trees which still result in the same *set* of computed answers, under any given search strategy. However, now we do not require for the respective streams, matrices or forests of computed answers to be equal.

In order for this criterion to be satisfied under all search strategies, the two predicates must have *similar* search trees, in sense that all the branches containing the finitely reachable leaves must be found in the same left-to-right order. However, the finitely reachable answers may be found at (finitely) different relative depths, and the internal nodes may have different branching factors. Indeed, the search trees of more efficient predicates will typically have shorter paths to the answers leaves and the the internal nodes with the high branching factor will be further away from the root node.

The construction of the search tree for an embedded predicate is captured by the function *step*. As discussed earlier, the rôle of function *step* in fair search models is to account for the cost of an answer; each time a predicate definition is unfolded, *step* records this by either pushing the answers down by one level in the search tree, or down by one row in the matrix. Informally, *step* provides a "timed" interpretation of logic programming, and can be thought of as a *tick* in the timed models of processes, as in CSP [64]. The difference in the shape of the tree, or alternatively in the structure of the matrix, can be captured by manipulating occurences of *step*, and the laws we present in this section capture the manipulations which yield similar trees.

It is worth noting that the laws presented here hold trivially in the unfair search model of depth-first traversal. This is because this model implements *step* as the identity function, since the cost of an answer is inconsequential.

The following properties of *step* preserve the search tree, and therefore the cost and order of answers under any search strategy:

$$(\mathit{step}\ p_1)\ \&\ p_2 = \mathit{step}\ (p_1\ \&\ p_2), \tag{5.26}$$

$$\exists x.\ \mathit{step}\ p = \mathit{step}\ (\exists x.\ p). \tag{5.27}$$

The following laws allow us to equate predicates with similar trees:

$$p_1 \,\&\, (\mathit{step}\ p_2) \simeq \mathit{step}\ (p_1 \,\&\, p_2), \tag{5.28}$$

$$(\mathit{step}\ p_1) \parallel (\mathit{step}\ p_2) \simeq \mathit{step}\ (p_1 \parallel p_2), \tag{5.29}$$

$$\mathit{step}\ p \simeq \mathit{step}\ (\mathit{step}\ p). \tag{5.30}$$

In law (5.28), the *step* on the right-hand side pushes all the answers one level down by inserting a new branch at the bottom, while the *step* on the left-hand side adds a new branch at the top. The law (5.29) lets us move a branching point one step down the tree. Finally, the law (5.30) allows us to shorten a path between the two nodes.

We shall use the laws presented in this section for program transformation in Chapter 9, but for now we return to the list of laws that hold in the procedural sense. We have so far show that the three implementations of the embedding, corresponding to different search models, share the same list of algebraic laws. The laws presented in Chapter 3 hold for the corresponding operators in each model. We now proceed to formalise and explore this strong algebraic relationship between the different models.

Chapter 6

The Relationship Between Search Strategies

In this chapter the depth-first and breadth-first models are shown to be special cases of a general model of tree searching. We show that the laws regarding search are an enrichment of the categorical concept of a monad, and we explore the links between such monads.

6.1 Three search monads

The aim of this section is to present the mathematical framework which will help us explore and express the relationships between our three models.

Moggi introduced in [88], and Wadler later popularised in [144, 145], the idea that many aspects of functional programming, for example laziness or eagerness of evaluation, and even non-functional aspects such as nondeterminism or handling of input and output, can be captured by the monad construction from category theory. Our models of logic programming, and their relationships, relate in a similar manner to concepts from category theory.

It is useful to note that, even though we use the language of category theory to present the results in this chapter, this work could also have been presented with no reference to category theory. Its use here merely shortens the presentation, and offers intuition about some generic properties of the functions involved.

We have so far described implementations of three search strategies for logic programming, and we have seen that the same set of algebraic laws holds for the structuring operators of each model. We now show that the common behaviour of these models can be captured by a certain extension of a *monad*, and that the relationship between the three models can be described as a *monad morphism*. Finally, we show that our forest model is indeed the most general model satisfying our specification of such extended monads; in categorical language, it is the *initial object* of the category of search monads.

A monad is a type constructor T together with a triple $(map_T, unit_T, join_T)$, where map_T, $unit_T$ and $join_T$ are polymorphic functions with types:

$$\begin{aligned}
&map_T :: (a \to b) \to T\ a \to T\ b,\\
&unit_T :: a \to T\ a,\\
&join_T :: T\ (T\ a) \to T\ a.
\end{aligned}$$

Below we denote the identity function as id. For such a triple T to qualify as a monad, the following equalities must be satisfied:

$$map_T\ id = id, \tag{6.1}$$

$$map_T\ (f \cdot g) = map_T\ f \cdot map_T\ g, \tag{6.2}$$

$$map_T\ f \cdot unit_T = unit_T \cdot f, \tag{6.3}$$

$$map_T\ f \cdot join_T = join_T \cdot map_T\ (map_T\ f), \tag{6.4}$$

$$join_T \cdot unit_T = id, \tag{6.5}$$

$$join_T \cdot map_T\ unit_T = id, \tag{6.6}$$

$$join_T \cdot map_T\ join_T = join_T \cdot join_T. \tag{6.7}$$

In terms of category theory, the equalities (6.1) and (6.2) express that T and map_T form a *functor*. The equalities (6.3) and (6.4) express that $unit_T$ and $join_T$ are *natural transformations*, and the equalities (6.5), (6.6) and (6.7) express the *monad laws*; (6.5) is usually referred to as the left unit law, (6.6) as the right unit law, and (6.6) is the associative law for the monad.

Our depth-first model of search can be captured by a monad *Stream*, for which we will use S as a subscript. In this monad, the type constructor is *Stream*, and map_S is the standard stream function *map*; $unit_S$ is the basic list unit constructor $[-]$, and $join_S$ is *concat*. One can easily verify that $(map, [-], concat)$ is a monad. The equations (6.1, 6.2, 6.4) and (6.7) correspond to the standard laws for list operators (2.1–2.4); the rest of the equations follow from the definitions of *map* and *concat*.

For our breadth-first search model, the *Matrix* monad, denoted by M, results from taking *mmap* for map_M, the infinite matrix unit constructor $[[-], [\,], [\,], \ldots]$ for $unit_M$ and *shuffle* for $join_M$. The equations (6.2), (6.4) and (6.7) for this monad correspond to the equations (5.3–5.5), and the remaining equations can be proved from the definitions of the matrix functions *mmap* and *shuffle*.

Finally, the *Forest* monad, denoted by F, results from taking the function *fmap* for map_F, the forest unit constructor $[Leaf\ -]$ for $unit_F$ and *fgraft* for $join_F$. Again, the equations (6.2), (6.4) and (6.7) for this monad are the same as the equations (5.20–5.22) described in Section 5.2, and the remaining ones can be proved from the definitions of the forest functions *fmap* and *fgraft*. For example, the unit laws (6.5) and (6.6) for this monad are:

$$fgraft \cdot [Leaf] = id \qquad\qquad fgraft \cdot fmap\ [Leaf] = id,$$

both of which follow from the definitions of *fgraft*, *concat* and *fmap*.

Given any monad T, we can define a composition operator $\circledast_T$ called the *Kleisli composition* (cf. [10]), such that:

$$(\circledast) :: (a \to T\ b) \to (b \to T\ c) \to (a \to T\ c), \tag{6.8}$$

$$p \circledast_T q = join_T\ \cdot map_T\ q \cdot p. \tag{6.9}$$

The operator $\circledast_T$ is associative with unit element $unit_T$:

$$unit_T \circledast p = p, \tag{6.10}$$

$$p \circledast unit_T = p, \tag{6.11}$$

$$m \circledast (p \circledast q) = (m \circledast p) \circledast q. \tag{6.12}$$

The operator $\circledast_T$ corresponds exactly to the definition of the operator & in each of the models, and unit_T corresponds to our function *true* in each model. We know that in all three models & is associative with unit element *true*, so the laws (6.10–6.12) are satisfied in the *Stream*, *Matrix* and *Forest* monads. In that sense we can say that they capture the algebraic semantics of the operator & and predicate *true* in our three models of logic programming.

The remaining structural parts of each model are the implementations of $\|$, *false*, and *step*. We have seen that $\|$ and *false* have very similar implementations and behaviour in each model. We wish to capture that common behaviour and properties in an extension of the above notion of monad. In this general setting we shall refer to $\|$ in each model as or_T, to *false* as empty_T, and to *step* as wrap_T. We continue to write or_T as an infix operator.

Regarding or_T and empty_T, we know that in all the three models, the operator corresponding to or_T is associative and has the predicate corresponding to empty_T as its unit element, and that $\circledast_T$ distributes through or_T from the left, but not from the right. Further, the predicate empty_T is a left zero for $\circledast_T$:

$$\mathit{empty}_T \ \mathit{or}_T \ p = p, \tag{6.13}$$

$$p \ \mathit{or}_T \ \mathit{empty}_T = p, \tag{6.14}$$

$$p_1 \ \mathit{or}_T \ (p_2 \ \mathit{or}_T \ p_3) = (p_1 \ \mathit{or}_T \ p_2) \ \mathit{or}_T \ p_3, \tag{6.15}$$

$$(p_1 \ \mathit{or}_T \ p_2) \circledast_T p_3 = (p_1 \circledast_T p_3) \ \mathit{or}_T \ (p_2 \circledast_T p_3) \tag{6.16}$$

$$\mathit{empty}_T \circledast_T p = \mathit{empty}_T. \tag{6.17}$$

We can reduce the last two laws in this list using the definition of $\circledast_T$ in terms of join_T and map_T. Then the law (6.16) is equivalent to the two laws (6.18–6.19), and the law (6.17) is equivalent to (6.20–6.21). We have:

$$\mathit{map}_T \ f \ (p_1 \ \mathit{or}_T \ p_2) = (\mathit{map}_T \ f \ p_1) \ \mathit{or}_T \ (\mathit{map}_T \ f \ p_2), \tag{6.18}$$

$$\mathit{join}_T \ (p_1 \ \mathit{or}_T \ p_2) = (\mathit{join}_T \ p_1) \ \mathit{or}_T \ (\mathit{join}_T \ p_2), \tag{6.19}$$

$$\mathit{map}_T \ f \ \mathit{empty}_T = \mathit{empty}_T, \tag{6.20}$$

$$\mathit{empty}_T \ \mathit{join}_T = \mathit{empty}_T. \tag{6.21}$$

Here the law (6.18) simply states that or_T is a natural transformation, and the law (6.20) states that $empty_T$ is a natural transformation.

Regarding $wrap_T$, the implementation of *step* in each model is dictated by the type of predicates in the search model, so, $wrap_T$ captures some essential information about the behaviour of the search models. The central property of $wrap_T$ in each model is that it is a natural transformation, and that it commutes with $join_T$:

$$wrap_T \cdot join_T = join_T \cdot wrap_T. \tag{6.22}$$

In depth-first search model *step* is the identity so this property holds trivially. In the breadth-first model *join* is implemented by *shuffle*, which sorts the answers from the input *Matrix(Matrix)* to a single *Matrix*, while *step* advances all the answers from the input *Matrix* further by one list. Informally, we can either advance all the answers and then *shuffle*, or *shuffle* and then advance. The proof is by structural induction. The argument is the same for the forest model, with *fgraft* and *step*.

Adding or_T, $empty_T$ and $wrap_T$, we define an *extended monad* as a six-tuple:

$$T^+ = (map_T,\ unit_T,\ join_T,\ empty_T,\ or_T,\ wrap_T),$$

such that map_T is a functor, and $unit_T$, $join_T$, $empty_T$, or_T, and $wrap_T$ are natural transformations with the following properties:

1. map_T obeys the functor laws (6.1–6.2),
2. $(map_T, unit_T, join_T)$ form a monad, respecting the laws (6.3–6.7),
3. or_T, $empty_T$ and $wrap_T$ are natural transformations, (6.18, 6.20, 6.22),
4. or_T is associative with unit $empty_T$, as in (6.13–6.15),
5. $join_T$ distributes over or_T, as in law (6.19), and
6. $empty_T$ is a left zero for $join_T$, as in law (6.21).

The first two requirements above are the "inherited" properties of a monad. These six requirements correspond exactly to our algebraic laws. Thus, the extended monads *Stream*, *Matrix* and *Forest* capture the algebraic semantics of the three different scheduling strategies for logic programming.

$$\begin{aligned}
\mathit{Stream} &= (\mathit{map},\ \ \mathit{unit}_S,\ \mathit{concat},\ \mathit{empty}_S,\ \mathit{or}_S,\ \mathit{id}),\\
\mathit{Matrix} &= (\mathit{mmap},\ \mathit{unit}_M,\ \mathit{shuffle},\ \mathit{empty}_M,\ \mathit{or}_M,\ \mathit{wrap}_M),\\
\mathit{Forest} &= (\mathit{fmap},\ \ \mathit{unit}_F,\ \mathit{fgraft},\ \ \mathit{empty}_F,\ \mathit{or}_F,\ \mathit{wrap}_F).
\end{aligned}$$

The function $\circledast_T$ is a Kleisli composition for each basic monad, and is therefore determined by join_T and map_T.

We can easily convince ourselves that all the five requirements above hold for each of these extended monads. The first property holds as a consequence of the way polymorphic recursive data types are defined in Haskell. The second and third property, stating the associativity of $\|$ and &, and their unit elements, can be proved as in Chapter 4 for all other extended monads. The fourth property states that join_T distributes over or_T, and we know from chapters 3 and 5 that *concat*, *shuffle* and *fgraft* distribute over $+\!\!+$, *zipwith* $\uplus$ and $+\!\!+$ respectively. As we have shown in the equational proofs in each model, the right-distributivity of & through $\|$ follows from these two properties. The fifth property states that the zero elements for, *concat*, *shuffle* and *fgraft*, are respectively $[\,]$, $[[\,],[\,],\ldots]$ and $[\,]$; it follows directly from the definitions of these functions. This is the property that is used together with the fact that the map functions in each model leave these terms unchanged, to prove in each model that *empty* is a left zero of &. The last property on the list above requires that wrap_T is well-behaved in the sense discussed before, where the omitted proof can be done by structural induction.

6.2 The relationships between monads

In the most general model, each query to a logic program returns a forest corresponding to the search tree of the query. We now show that the other two search models can be obtained from this most general one.

The forest of answers can be converted to a stream of answers, by traversing the forest in a depth-first manner. The function *dfs* below, with type $\mathit{Forest}\ a \rightarrow \mathit{Stream}\ a$, implement this traversal strategy. Alternatively, the forest can be converted to a matrix of answers, by traversing the tree in a breadth-first manner. This traversal is implemented by the function *bfs* below, with type $\mathit{Forest}\ a \rightarrow \mathit{Stream}\ \mathit{List}\ a$. For the purpose of printing the answers from this breadth-first matrix, it can be flattened to a single stream of answers by simple concatenation.

The *dfs* function applies the auxiliary dfs_1 function to each tree in the input forest and concatenates the resulting streams. The function dfs_1 returns the leaf nodes of each tree in a depth-first manner, by recursively calling *dfs*:

$$\mathit{dfs}\ \mathit{xff} = \mathit{foldr}\ \mathit{concat}\ [\,]\ (\mathit{map}\ \mathit{dfs}_1\ \mathit{xff}), \tag{6.23}$$
$$\mathit{dfs}_1\ (\mathit{Leaf}\ x) = [x], \tag{6.24}$$
$$\mathit{dfs}_1\ (\mathit{Fork}\ \mathit{xf}) = \mathit{dfs}\ \mathit{xf}. \tag{6.25}$$

Then, by (6.23), for a singleton forest $[t]$, we have $\mathit{dfs}\ [t] = \mathit{dfs}_1\ t$, and for a forest $[t_1, \ldots, t_n]$ we get:

$$\mathit{dfs}\ [t_1, \ldots, t_n] = (\mathit{dfs}_1\ t_1) \mathbin{+\!\!+} \ldots \mathbin{+\!\!+} (\mathit{dfs}_1\ t_n).$$

The *bfs* function needs to take account of the cost of the answers. It does this by converting each of the trees in the input forest into matrices, using the auxiliary function bfs_1, and collecting the answers with same cost from each matrix into bags using $\mathit{zipwith}\ \uplus$. We have:

$$\mathit{bfs}\ \mathit{xff} = \mathit{foldr}\ (\mathit{zipwith}\ \uplus)\ \mathit{false}_M\ (\mathit{map}\ \mathit{bfs}_1\ \mathit{xff}), \tag{6.26}$$
$$\mathit{bfs}_1\ (\mathit{Leaf}\ x) = [x] : \mathit{repeat}\ [\,], \tag{6.27}$$
$$\mathit{bfs}_1\ (\mathit{Fork}\ \mathit{xf}) = [\,] : \mathit{bfs}\ \mathit{xf}. \tag{6.28}$$

Again, by (6.26), for a singleton forest $[t]$, we have $\mathit{bfs}\ [t] = \mathit{bfs}_1\ t$, and for a forest $[t_1, \ldots, t_n]$ we get:

$$\mathit{bfs}\ [t_1, \ldots, t_n] = (\mathit{bfs}_1\ t_1)\ (\mathit{zipwith}\ \uplus) \ldots (\mathit{zipwith}\ \uplus)\ (\mathit{bfs}_1\ t_n).$$

Below we argue that any query results in the same stream of depth-first sorted answers regardless whether one computes the answers in the stream model, or one computes the answers by the forest model and then applies *dfs* to this forest. Also, one gets the same matrix of breadth-first sorted answers, either by computing queries directly in the matrix model or by applying *bfs* to the forest resulting from the most general model. Categorically speaking, we show that there exist *morphisms* between the three monads, and that they are exactly the functions *dfs* and *bfs*.

The polymorphic function *dfs* is a morphism between the extended monads *Forest* and *Stream* if it maps the predicates $unit_F$ and $empty_F$ to their counterparts in the stream model and if it preserves the behaviour of the basic predicates operators. Formally, *dfs* is a $Forest \Longrightarrow Stream$ morphism if:

$$dfs \cdot unit_F = unit_S, \tag{6.29}$$

$$dfs \cdot empty_F = empty_S, \tag{6.30}$$

$$dfs \cdot map_F\ f = map_S\ f \cdot dfs, \tag{6.31}$$

$$dfs \cdot (p_1 \circledast_F p_2) = (dfs \cdot p_1) \circledast_S (dfs \cdot p_2), \tag{6.32}$$

$$dfs \cdot (p_1\ or_F\ p_2) = (dfs \cdot p_1)\ or_S\ (dfs \cdot p_2). \tag{6.33}$$

$$dfs \cdot wrap_F = wrap_S \cdot dfs. \tag{6.34}$$

The equations (6.29) and (6.30) follow directly from the definitions of *dfs* and the predicates *true* and *false* in the depth-first and general models. The proof of (6.33) is a simple consequence of the distributivity of *concat* and *map* through $+\!\!+$. The proof of (6.34) follows from the definition of *step* in the forest model and the fact that *step* is identity in the depth-first model.

The proofs of equations (6.31) and (6.32) are slightly more complicated than the others. The proof of (6.31) requires a simultaneous induction for *dfs* and dfs_1, similarly to the proof of (5.22). Equation (6.32) requires in addition a proof of the following lemma:

$$dfs \cdot fgraft = concat \cdot (dfs * dfs). \tag{6.35}$$

The proof of (6.35) can also be done by simultaneous induction on *dfs* and dfs_1. Expressed in a commuting diagram, it means:

$$\begin{tikzcd}
Forest\ (Forest\ X) \arrow[r, "dfs*dfs"] \arrow[d, "fgraft"'] & Stream\ (Stream\ X) \arrow[d, "concat"] \\
Forest\ X \arrow[r, "dfs"'] & Stream\ X
\end{tikzcd}$$

Here $dfs * dfs$ denotes a categorical construction called the *horizontal composition* of natural transformations (cf. [10]). In the case at hand we have:

$$dfs * dfs = dfs \cdot fmap\ dfs = map\ dfs \cdot dfs. \tag{6.36}$$

These equations are instances of the so-called exchange law for natural transformations. The two expressions for $dfs * dfs$ are equal because the diagram below can be proved to commute:

$$\begin{tikzcd}
Forest\ (Forest\ X) \arrow[r, "dfs"] \arrow[d, "fmap\ \ dfs"'] \arrow[dr, "dfs*dfs"] & Forest\ (Stream\ X) \arrow[d, "map\ \ dfs"] \\
Stream\ (Forest\ X) \arrow[r, "dfs"'] & Stream\ (Stream\ X)
\end{tikzcd}$$

This diagram corresponds to the fact that given a forest of forests, and using depth-first search, it does not matter which forest we choose to first flatten to a stream. Using (6.31) and(6.35), the proof of (6.32) is:

$$\begin{aligned}
& dfs \cdot (p_1 \circledast_F p_2) \\
&= dfs \cdot join_F \cdot map_F\ p_2 \cdot p_1 && \text{by (5.13)} \\
&= join_S \cdot map_S\ dfs \cdot dfs \cdot map_F\ p_2 \cdot p_1 && \text{by (6.35, 6.36)} \\
&= join_S \cdot map_S\ dfs \cdot map_S\ p_2 \cdot dfs \cdot p_1 && \text{by (6.31)} \\
&= join_S \cdot map_S\ (dfs \cdot p_2) \cdot dfs \cdot p_1 && \text{by (2.2)} \\
&= (dfs \cdot p_1) \circledast_S (dfs \cdot p_2). && \text{by (3.3)}
\end{aligned}$$

Similarly, to prove that also bfs is a monad morphism, we need to show that it correctly maps predicates $unit_F$ and $empty_F$ and that it preserves the correspondences between the basic operators of the two models:

$$bfs \cdot unit_F = unit_M, \tag{6.37}$$

$$bfs \cdot empty_F = empty_M, \tag{6.38}$$

$$bfs \cdot map_F\ f = map_M\ f \cdot bfs, \tag{6.39}$$

$$bfs \cdot (p_1 \circledast_F p_2) = (bfs \cdot p_1) \circledast_M (bfs \cdot p_2), \tag{6.40}$$

$$bfs \cdot (p_1\ or_F\ p_2) = (bfs \cdot p_1)\ or_M\ (bfs \cdot p_2). \tag{6.41}$$

$$bfs \cdot wrap_F = wrap_M \cdot bfs. \tag{6.42}$$

The proofs of (6.37, 6.38, 6.41) and (6.42) follow the same simple pattern as in for the *dfs* morphism, and are based directly on the definitions of the corresponding functions in the respective models of search. The proof of (6.40) requires the following two lemmas, which correspond to the lemmas (6.35) and (6.36) used in the proof for (6.32) for *dfs*. These lemmas can as well be proved by structural induction on forests:

$$\mathit{bfs} \cdot \mathit{fgraft} = \mathit{shuffle} \cdot (\mathit{bfs} * \mathit{bfs}), \tag{6.43}$$

$$\mathit{bfs} * \mathit{bfs} = \mathit{bfs} \cdot \mathit{fmap}\ \mathit{bfs} = \mathit{mmap}\ \mathit{bfs} \cdot \mathit{bfs}. \tag{6.44}$$

Thus, *bfs* is a morphism between the extended monads *Forest* and *Matrix*.

Another interesting monad for survey of search is the monad *Set* where the answers are returned as sets. This monad would correspond to an algebraic formulation of the least Herbrand models semantics of logic programs. The operators in this monad are less operational; $\|$ would be a union of the least Herbrand models of answers to its two arguments, & would be their intersection, and so on. For some $\mathit{collect}_1$ and $\mathit{collect}_2$ which remove the information about the ordering and the multiplicity of answers, we have:

$$\begin{tikzcd}
 & \mathit{Forest\ monad} \arrow[dl, Rightarrow, "\mathit{dfs}"] \arrow[dd, Rightarrow, "\mathit{collect}_1"] \arrow[dr, Rightarrow, "\mathit{bfs}"] & \\
\mathit{Stream\ monad} \arrow[dr, dotted, "?"'] & & \mathit{Matrix\ monad} \arrow[dl, Rightarrow, "\mathit{collect}_2"] \\
 & \mathit{Set\ monad} &
\end{tikzcd}$$

Concerning this chapter, the interesting question relating to the *Set* monad is whether it plays a special role in the category of search models, that is, whether it is a *final object* of the category. An object C in a category is final if, for any other object D of the category, there is a unique morphism $D \to C$. However, there is no morphism from *Stream* to the *Set* monad, because the depth-first model is not fair, and is consequently not complete. As in Prolog, this search strategy can diverge and does not always return all the answers implied by the program. The lack of this arrow in the diagram above shows that *Set* is not final. In fact, the existence of a morphism from a monad T to the *Set* monad can be taken as the semantic definition of fairness in a search model corresponding to the extended monad T.

6.3 Initiality of the Forest monad

In this section we show the uniqueness of the homomorphism between the *Forest* monad and any extended monad, and therefore conclude that *Forest* is the *initial object* of the category of extended monads. This result follows the properties (1–6) from the definition of the extended monad.

The proof of the uniqueness of homomorphisms between the *Forest* monad and other extended monads is based on a generalisation from functions *dfs* and *bfs* to a function h. Let us make the following recursive definition of a polymorphic function h:

$$h :: \mathit{Forest}\ a \to T\ a$$

$$h\ xs = \mathit{foldr}\ (\mathit{or}_T)\ \mathit{empty}_T\ (\mathit{map}\ h_1\ xs), \tag{a}$$

and a function h_1:

$$h_1 :: \mathit{Tree}\ a \to T\ a$$

$$h_1\ (\mathit{Leaf}\ x) = \mathit{true}_T\ x \tag{b}$$

$$h_1\ (\mathit{Fork}\ \mathit{xf}) = \mathit{wrap}_T\ (h\ \mathit{xf}). \tag{c}$$

Because this is a definition of two functions by mutual structural recursion, we can be sure that *exactly one* pair of functions h and h_1 satisfies (a–c).

According to this definition, the function h recursively applies h_1 to each of the trees in the list that is the input forest; the function h_1 converts each tree into a type T of the respective goal monad – a *Stream*, *Matrix*, or other; finally, h converts the resulting *Stream* T into a T, by folding the stream with the union operator or_T of the extended monad T. Presented with a *Leaf*, the function h_1 uses true_T of the corresponding monad to package an answer in a unit *Stream*, *Matrix*, or other. Presented with a *Forest xf*, h_1 converts it to T by applying the mutually recursive call to h to all the trees in *xf*. In order to preserve the cost information from the input forest, at this stage the function wrap_T needs to be applied.

This definition of h and h_1 is analogous to the definitions for *dfs* and *bfs* and their auxiliary functions. As for those definitions, the relationship between

h and h_1 is captured by the following equation:

$$h\ [t] = h_1\ t. \tag{h}$$

We need to prove that the implementation of h, captured by equations (a–c), implies the same structure preserving properties as respected by *dfs* and *bfs*. Generalising to any extended monad T, these properties are:

$$h \cdot map_F\ f = map_T\ f \cdot h, \tag{I}$$
$$h \cdot join_F = join_T \cdot h * h, \tag{II}$$
$$h \cdot true_F = true_T, \tag{III}$$
$$h \cdot empty_F = empty_T, \tag{IV}$$
$$h \cdot or_F = or_T \cdot\ \ h \times h, \tag{V}$$
$$h \cdot wrap_F = wrap_T \cdot h, \tag{VI}$$

where the pair-product $h \times h$ applies the function h to each element of the pair, and is defined as having the type $Forest\ a \times Forest\ a \to T\ a \times T\ a$; also, as before, the horizontal composition $h * h$ is defined as:

$$h * h :: Forest(Forest\ a) \to T(T\ a)$$
$$h * h = h \cdot map_F\ h = map_T\ h \cdot h. \tag{6.45}$$

The list (I–VI) implicitly specifies the behaviour of h on $\&_F$ through its behaviour on map_F and $join_F$.

We now show that the definition of h, as given by equations (a–c), is equivalent to the set of equations (III–VI). In particular, we show that:

(IV, V) $\Leftrightarrow$ (a),
(III) $\Leftrightarrow$ (b), and
(VI) $\Leftrightarrow$ (c).

Because exactly one h satisfies (a–c), and (a–c) are equivalent to (III–VI), we have that exactly one function, the same h, satisfies (III–VI). Later, we prove that such h also must satisfy (I,II). For simplicity, we prove the equivalences in one direction only, though the same arguments work backwards.

The proof for (IV, V)⇒(a) for the empty list follows from the definitions of *foldr* and $empty_F$:

$$\begin{aligned}
& h\ [\,] \\
& = h\ empty_F && \text{by (5.25)} \\
& = empty_T && \text{by (IV)} \\
& = foldr\ (or_T)\ empty_T\ (map\ h_1\ [\,]). && \text{by defn. } foldr
\end{aligned}$$

For non-empty lists, it follows from (h) and the definitions of *foldr* and or_F:

$$\begin{aligned}
& h\ [t_1, t_2, \ldots, t_n] \\
& = h([t_1]\ or_F\ [t_2]\ or_T\ \ldots\ or_F\ [t_n]) && \text{by (5.12)} \\
& = h[t_1]\ or_T\ h[t_2]\ or_T\ \ldots\ or_T\ h[t_n] && \text{by (V)} \\
& = h_1(t_1)\ or_T\ h_1(t_2)\ or_T\ \ldots\ or_T\ h_1(t_n) && \text{by (h)} \\
& = foldr\ or_T\ empty_T\ (map\ h_1\ [t_1, t_2, \ldots, t_n]). && \text{by defn. } foldr
\end{aligned}$$

The proof for (III) ⇒ (b) uses (h) and the definition of $true_F$:

$$\begin{aligned}
& h_1\ (Leaf\ x) \\
& = h\ [Leaf\ x] && \text{by (h)} \\
& = h\ (true_F\ x) && \text{by defn. } true_F \\
& = true_T\ x. && \text{by (III)}
\end{aligned}$$

Finally, the proof of (VI) ⇒ (c) uses (h) and the definition of $wrap_F$:

$$\begin{aligned}
& h_1\ (Fork\ ts) \\
& = h\ [Fork\ ts] && \text{by (h)} \\
& = h\ (wrap_F\ ts) && \text{by defn. } wrap_F \\
& = wrap_T\ (h\ ts). && \text{by (VI)}
\end{aligned}$$

We can now prove that h satisfies (I) and (II), assuming both (a–c) and (I–VI). We choose to first prove (II). For this we need the following lemma, which can be proved by structural induction on the tree t:

$$h\ (join_F\ [t]) = join_T\ ((h * h)\ [t]). \tag{6.46}$$

The base case uses (a) and (b), with the monad law (6.5) and $*$-definition (6.45):

$$\begin{aligned}
& h\ (join_F\ [Leaf\ xf]) & \\
&= h\ xf & \text{by defn. } \mathit{fgraft} \\
&= join_T\ (true_T\ (h\ xf)) & \text{by (6.5)} \\
&= join_T\ (h_1\ (Leaf\ (h\ xf))) & \text{by (b)} \\
&= join_T\ (h\ [Leaf\ (h\ xf)]) & \text{by (a)} \\
&= join_T\ ((h * h)\ [Leaf\ xf]). & \text{by (6.45)}
\end{aligned}$$

We have already proved that the properties (III–VI) imply (a, b, c), so in these proofs we may use any of those properties as well. The inductive case uses the definitions of $join_F$, $wrap_F$ and (c), and property (VI):

$$\begin{aligned}
& h\ (join_F\ [Fork\ xff]) & \\
&= h\ (Fork\ (join_F\ xff)) & \text{by defn. } \mathit{fgraft} \\
&= wrap_T\ (h\ (join_F\ xff)) & \text{by (c)} \\
&= wrap_T\ (join_T\ ((h * h)\ xff)) & \text{by ind. hyp.} \\
&= join_T\ (wrap_F\ ((h * h)\ xff)) & \text{by (VI)} \\
&= join_T\ ([(h * h)\ xff]). & \text{by defn. } wrap_F
\end{aligned}$$

Then the point-wise proof of (II), which results from applying both sides of the equation to the forest $[t_1, \ldots, t_n]$, is:

$$\begin{aligned}
& h\ (join_F\ [t_1, \ldots, t_n]) & \\
&= h\ ((join_F\ [t_1])\ or_F \ldots or_F\ (join_F\ [t_n])) & \text{by defn. } \mathit{fgraft} \\
&= (h(join_F\ [t_1]))\ or_T \ldots or_T\ (h(join_F\ [t_n])) & \text{by (IV, V)} \\
&= (join_T((h * h)[t_1]))\ or_T \ldots or_T\ (join_T((h * h)[t_n])) & \text{by (6.46)} \\
&= join_T\ (((h * h)[t_1])\ or_T \ldots or_T\ ((h * h)[t_n])) & \text{by (6.19)} \\
&= join_T\ ((h(map_F\ h)[t_1]) or_T \ldots or_T (h(map_F\ h)[t_n])) & \text{by (6.45)} \\
&= join_T\ (h\ ((map_F\ h\ [t_1])\ or_F \ldots or_F\ (map_F\ h\ [t_n]))) & \text{by (V)} \\
&= join_T\ (h\ (map_F\ h)\ [t_1, \ldots, t_n]) & \text{by (6.47)} \\
&= join_T\ ((h * h)\ [t_1, \ldots, t_n]). & \text{by (6.45)}
\end{aligned}$$

In the penultimate step we use the fact that map_F distributes over or_F, which is simply concatenation $+\!\!+$:

$$map_F\ h\ (f_1 +\!\!+ f_2) = map_F\ h\ f_1 +\!\!+ map_F\ h\ f_2. \tag{6.47}$$

The proof of (I) is structurally very similar to the proof of (II) above. Again, we need an additional lemma, proved by structural induction on the tree t:

$$h\ (map_F\ f\ [t]) = map_T\ f\ (h\ [t]). \tag{6.48}$$

The proof of (I) is then:

$$\begin{aligned}
& h\ (map_F f\ [t_1, \ldots, t_n]) \\
&= h\ (map_F f\ [t_1]\ or_F \ldots\ or_F\ map_F f\ [t_n]) && \text{by defn. } fmap \\
&= h(map_F f\ [t_1])\ or_T \ldots or_T\ h(map_F f\ [t_n]) && \text{by (IV, V)} \\
&= map_T f\ (h\ [t_1])\ or_T \ldots or_T\ map_T f\ (h\ [t_n]) && \text{by (6.48)} \\
&= map_T f\ (h\ [t_1]\ or_T \ldots or_T\ h\ [t_n]) && \text{by (6.47)} \\
&= map_T f\ (h\ [t_1, \ldots, t_n]). && \text{by (V)}
\end{aligned}$$

Finally, we can also prove that h preserves the behaviour of &:

$$\begin{aligned}
& h \cdot (p_1 \circledast_F p_2) \\
&= h \cdot join_F \cdot map_F\ p_2 \cdot p_1 && \text{by (5.13)} \\
&= join_F \cdot h * h \cdot map_F\ p_2 \cdot p_1 && \text{by (II)} \\
&= join_F \cdot h \cdot map_F\ h \cdot map_F\ p_2 \cdot p_1 && \text{by (6.45)} \\
&= join_F \cdot h \cdot map_F\ (h \cdot p_2) \cdot p_1 && \text{by (2.2)} \\
&= join_F \cdot map_F\ (h \cdot p_2) \cdot (h \cdot p_1) && \text{by (I)} \\
&= (h \cdot p_1) \circledast_F (h \cdot p_2). && \text{by (5.13)}
\end{aligned}$$

The fact that the monad *Forest* is an initial object in the "category of monads that describe the search part of logic programming" captures the idea that the *dfs* and *bfs* morphisms are unique. Also, it implies that the list of algebraic laws for operators which we provide is *complete*: any additional laws would exclude some of the current members of the category of the search monads. The *Forest* model satisfies *exactly* those laws that are satisfied by every search model.

Chapter 7

Adequacy of the Embedding

In this chapter we describe how a particular selection of the algebraic laws can be used to prove that our translation of logic program into Haskell gives the same effect as LD-resolution. Since the laws we use are provably correct for the implementation of our embedding, we use this connection to argue for the correctness, that is, soundness and completeness, of our embedding.

7.1 Motivation for LD-simulation

We can view LD-resolution as a recipe for growing the LD-tree for a query, starting with just the query itself, and iteratively adding further nodes. Consider the definition of the *append* predicate presented earlier:

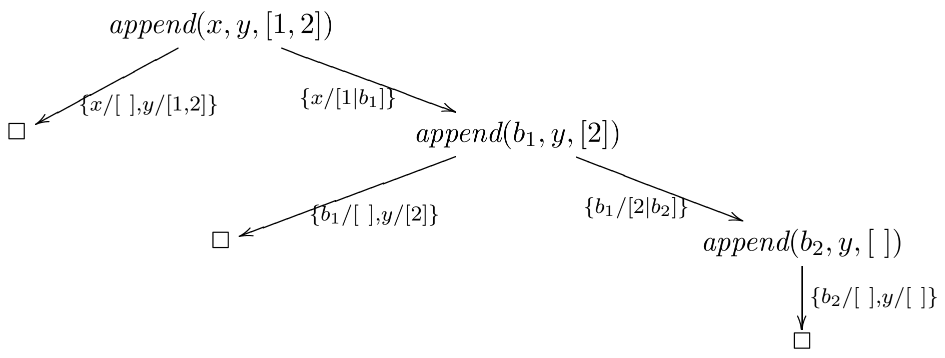
$$
\begin{aligned}
&\mathit{append}(x, y, z) = \\
&\qquad (x \doteq [\,] \mathbin{\&} y \doteq z) \\
&\quad \parallel (\exists a, b, c.\ x \doteq [a|b] \mathbin{\&} z \doteq [a|c] \mathbin{\&} \mathit{append}(b, y, c)).
\end{aligned}
$$

One step of LD-resolution causes the growth of a single branch in the LD-tree, corresponding to the application of a single clause to a non-empty query in the tree. The effect is to move the *frontier* of the examined part of the LD-tree by one node further away from the root.

Our aim is to manipulate predicates such as *append* using the algebraic laws for our embedding, so that the transformations follow the growth of the LD-tree. However, in our setting, we cannot simulate the growth of a single branch, because all the clauses defining the same relation are joined in the completed form of the predicate, and we have no means of applying them separately. Nevertheless, we can simulate several LD-steps at once. The evaluation which proceeds by a *simultaneous growth of all the branches from one node* in the LD-tree is provably equivalent to the computation in the embedding.

Our argument is based on the idea that one such "sprouting" of the LD-tree is equivalent, according to our laws, to the unfolding of a query in the embedding. Therefore the original query q is also inductively equal to $\bar{q}$, the result of unfolding it completely: but that complete unfolding $\bar{q}$ represents the set of answers computed by LD-resolution. The two predicates q and $\bar{q}$ must compute the same stream, matrix or forest of answers, since lazy functional programming computes the same answers from equal expressions.

For example, the terminated LD-tree for the computation of the query $append(x, y, [1, 2])$ is presented below:



In the embedding, successful computations encounter equations, which result immediately in computed answer substitutions; these answers are, in turn, stored in the forest which is the input to the rest of the computation, which corresponds to the frontier of the LD-tree.

For the purposes of simulating LD-resolution in this chapter, we now wish to keep the information about the frontier and all partial answers explicit. The partial answers can be represented explicitly in the predicate, in a form of a conjunction of $\doteq$ equations prefixing each $\|$ branch. For brevity, let us refer to any such conjunction of equations:

$$x_1 \doteq t_1 \,\&\, \ldots \,\&\, x_n \doteq t_n,$$

where no x_i appears in any t_j, as an *environment*, and to each such $x_i \doteq t_i$ as an *assignment*. This environment can be applied to subsequent computation through the appropriate use of the law for the substitution of equals for equals (4.25), the one-point law (4.12), and the distributivity laws for & though $\|$ and $\exists$ (4.6, 4.10). Through a series of applications of these laws, a predicate is transformed from one equivalent form to another. We require each of these forms to be a (possibly empty) disjunction of the form:

$$\|_i \; (\exists \vec{x_i}.\; x_{i1} \doteq t_{i1} \,\&\, \ldots \,\&\, x_{in_i} \doteq t_{in_i} \,\&\, r\; \vec{u_i}),$$

where $x_{i1} \doteq t_{i1} \ldots x_{in_i} \doteq t_{in_i}$ is an environment, and the list of terms $\vec{u_i}$ contains none of $x_{i1} \ldots x_{in_i}$. We shall refer to this form of predicates as the *normal form*.

The combination of laws used to transform one normal form into another is operationally not the same as the executed evaluation of our embedding. However, all the laws used are valid for the implementation of the embedding in all the models. Therefore, we know that the abstract evaluation is equivalent to the normal Haskell one in an *extensional* sense: they compute provably equal collections of answers when provided with the same input.

Because of the left-most selection rule, all the nodes to the left of the current one are fully explored, and in these branches the computational frontier touches the leaves of the LD-tree. In the embedding these branches correspond to the previously evaluated $\|$ disjuncts.

7.2 Example of LD-simulation

We now show how to simulate the growth of the LD-tree for predicate:

$$\mathit{append}(x, y, [1, 2]),$$

using our algebraic laws. For simplicity, in this section we omit the applications of *step*, and address this issue later. After one unfolding of the definition of *append* get the following predicate:

$$\begin{aligned}&(x \doteq [\,] \mathbin{\&} y \doteq [1,2])\\ \| \ &(\exists a, b, c.\ x \doteq [a|b] \mathbin{\&} [1,2] \doteq [a|c] \mathbin{\&} \mathit{append}(b, y, c)).\end{aligned}$$

The first disjunct in this predicate consists only of an environment, and is therefore already in normal form, while the other disjunct needs to be transformed to this form. We can first apply the $\exists$-related law (4.13) to rename the bound variables a, b, c to fresh variable names a_1, b_1 and c_1, followed by an application of the equality law (4.22) to simplify $[1,2] \doteq [a_1|c_1]$. We then use the symmetry property of $\doteq$ (4.26) to get:

$$\begin{aligned}&(x \doteq [\,] \mathbin{\&} y \doteq [1,2])\\ \| \ &(\exists a_1, b_1, c_1.\ x \doteq [a_1|b_1] \mathbin{\&} a_1 \doteq 1 \mathbin{\&} c_1 \doteq [2] \mathbin{\&} \mathit{append}(b_1, y, c_1)).\end{aligned}$$

The second disjunct now has a conjunction of equations as a prefix; however, some of these equalities can be simplified, and some local variables removed, before the normal form is reached. We use the law for substitution of equals for equals (4.28) to propagate the equalities:

$$\begin{aligned}&(x \doteq [\,] \mathbin{\&} y \doteq [1,2])\\ \| \ &(\exists a_1, b_1, c_1.\ x \doteq [1|b_1] \mathbin{\&} a_1 \doteq 1 \mathbin{\&} c_1 \doteq [2] \mathbin{\&} \mathit{append}(b_1, y, [2])),\end{aligned}$$

and subsequently remove the unused local variables a_1 and c_1 with (4.14):

$$\begin{aligned}&(x \doteq [\,] \mathbin{\&} y \doteq [1,2])\\ \| \ &(\exists b_1.\ x \doteq [1|b_1] \mathbin{\&} \mathit{append}(b_1, y, [2])).\end{aligned}$$

This predicate is in normal form. It corresponds to the frontier of the partial LD-tree for $\mathit{append}(x, y, [1,2])$ after the sprouting of all the branches from the root node. The first $\|$ branch corresponds to the first successful branch of the LD-tree, with the answer substitution $\{x/[\,], y/[1,2]\}$. The second $\|$ branch carries the environment with the partial answer $x \doteq [1|b_1]$ at the front, and has the as yet unresolved parts of the predicate, corresponding to the contents of the LD-node $\mathit{append}(b_1, y, [2])$, as the rest.

We now repeat the sprouting process for the node $append(b_1, y, [2])$ in the second branch. After unfolding we get:

$$\begin{aligned}
&\quad (x \doteq [\,] \mathbin{\&} y \doteq [1,2]) \\
&\| \ (\exists b_1.\ x \doteq [1|b_1] \mathbin{\&} \\
&\qquad\qquad (\quad (b_1 \doteq [\,] \mathbin{\&} y \doteq [2]) \\
&\qquad\qquad\ \ \| \ (\exists a, b, c.\ b_1 \doteq [a|b] \mathbin{\&} [2] \doteq [a|c] \mathbin{\&} append(b, y, c)))).
\end{aligned}$$

We now need to distribute $x \doteq [1|b_1]$ through $\|$ in the newly expanded part. At this point there is a slight complication due to the one-sided distributivity of &: we wish to rewrite $x \doteq [1|b_1] \mathbin{\&} (p_1 \parallel p_2)$ to $(x \doteq [1|b_1] \mathbin{\&} p_1) \parallel (x \doteq [1|b_1] \mathbin{\&} p_2)$, but & does not, in general, distribute through $\|$ from the right.

Fortunately, in this case we are able to extend our set of laws with an additional law (7.1), because & distributes through $\|$ from the right when the predicate e in the equation below is restricted to a conjunction of equations. The law holds because such a predicate e can only have zero or one answers. Rather than interrupt the flow of the example, we present the proof of this law at the end of the present section:

$$e \mathbin{\&} (p_1 \parallel p_2) = (e \mathbin{\&} p_1) \parallel (e \mathbin{\&} p_2). \tag{7.1}$$

Returning to the example and the predicate above, using (7.1), we can distribute $x \doteq [1|b_1]$ through $\|$. Then, using law (4.17), we distribute $\exists b_1$ through $\|$. Finally, we apply the law (4.4) about associativity of $\|$ to get:

$$\begin{aligned}
&\quad (x \doteq [\,] \mathbin{\&} y \doteq [1,2]) \\
&\| \ (\exists b_1.\ x \doteq [1|b_1] \mathbin{\&} b_1 \doteq [\,] \ \mathbin{\&} y \doteq [2]) \\
&\| \ (\exists b_1, a, b, c.\ x \doteq [1|b_1] \mathbin{\&} b_1 \doteq [a|b] \mathbin{\&} [2] \doteq [a|c] \mathbin{\&} b_1 \doteq [2|b] \mathbin{\&} \\
&\qquad\qquad append(b, y, c)).
\end{aligned}$$

After simplification of last two $\|$ branches and renaming of b (4.13), we get:

$$\begin{aligned}
&\quad (x \doteq [\,] \mathbin{\&} y \doteq [1,2]) \\
&\| \ (x \doteq [1] \mathbin{\&} y \doteq [2]) \\
&\| \ (\exists b_2.\ x \doteq [1,2|b_2] \mathbin{\&} append(b_2, y, [\,])).
\end{aligned}$$

This normal form predicate corresponds to the LD-tree above after the sprouting of the $\mathit{append}(b_1, y, [2])$ node. The second branch corresponds to a successfully terminated branch in the LD-tree, representing the answer $\{x/[1], y/[2]\}$. In the third branch, the environment represents the combination of substitutions which label the corresponding branch of the LD-tree.

After unfolding $\mathit{append}(b_2, y, [\,])$, and more simplification as before, we get:

$$\begin{aligned}
&\quad (x \doteq [\,] \mathbin{\&} y \doteq [1,2]) \\
&\parallel (x \doteq [1] \mathbin{\&} y \doteq [2]) \\
&\parallel (x \doteq [1,2] \mathbin{\&} y \doteq [\,]) \\
&\parallel (\exists a_3, b_3, c_3.\ x \doteq [1,2,a_3|b_3] \mathbin{\&} [\,] \doteq [a_3|c_3] \mathbin{\&} \mathit{append}(b_3, y, c_3)).
\end{aligned}$$

Finally, in the last $\parallel$ branch the unification $[\,] \doteq [a_3|c_3]$ fails, and is thus equivalent to *false*. This branch thus rewrites to:

$$(\exists a_3, b_3, c_3.\ x \doteq [1,2,a_3|b_3] \mathbin{\&} \mathit{false} \mathbin{\&} \mathit{append}(b_3, y, c_3)).$$

Using the law (4.3), stating that *false* is a left zero of &, we get:

$$(\exists a_3, b_3, c_3.\ x \doteq [1,2,a_3|b_3] \mathbin{\&} \mathit{false})$$

In general case *false* is not a right zero of &, because in the computation of $p \mathbin{\&} \mathit{false}$, the predicate p might diverge. However, if the left argument is a conjunction of equations e, we know that e never diverges, and we can prove:

$$e \mathbin{\&} \mathit{false} = \mathit{false}. \tag{7.2}$$

Thus, using the law (7.2), and subsequently the laws (4.5) and (4.24) we can now remove this branch, and terminate the computation. The resulting predicate is a list of environments, corresponding to the frontier of the completely explored LD-tree:

$$\begin{aligned}
&\quad (x \doteq [\,] \mathbin{\&} y \doteq [1,2]) \\
&\parallel (x \doteq [1] \mathbin{\&} y \doteq [2]) \\
&\parallel (x \doteq [1,2] \mathbin{\&} y \doteq [\,]).
\end{aligned}$$

In this example, for brevity we have omitted *step* from all predicates, even though it occurs at each unfolding of a predicate definition. With *step*, the normal form of a predicate would conform to the grammar:

$$\begin{aligned} nf \;\rightarrow\; & (\exists \vec{x_n}.\; x_1 \doteq t_1 \;\&\; \ldots \;\&\; x_n \doteq t_n \;\&\; r\; \vec{t_m}) \\ & \mid step\; (\|_i\; nf_i). \end{aligned}$$

This form retains the structure of the LD-tree. It slightly complicates the presentation, but not the argument.

Returning to the proof of law (7.1), we need to consider two cases in the proof, since for some input σ, $e(\sigma)$ has either one or zero solutions. If $e(\sigma)$ has exactly one solution, we can write $e(\sigma) = unit_T(\sigma')$, for any search model T. We use the notation from Chapter 6, so & in T is $\circledast_T$. Then:

$$\begin{aligned} & (e \circledast_T p)(\sigma) \\ & = (join_T \cdot map_T\; p \cdot e)(\sigma) && \text{by defn. of } \circledast_T \\ & = (join_T \cdot map_T\; p \cdot unit_T)(\sigma') && \text{by } e(\sigma) = unit_T(\sigma') \\ & = p(\sigma') && \text{by (6.3) and (6.5)} \end{aligned}$$

so, in the forest model, and similarly in the others, for such e and σ, we get:

$$\begin{aligned} & (e \circledast_F (p_1\;\; or_F\;\; p_2))(\sigma) \\ & = (p_1\;\; or_F\;\; p_2)(\sigma') && \text{by eqn. above} \\ & = p_1(\sigma') + p_2(\sigma') && \text{by defn. of } or_F \\ & = (e \circledast_F p_1)(\sigma) + (e \circledast_T p_2)(\sigma) && \text{by eqn. above} \\ & = ((e \circledast_F p_1)\;\; or_F\;\; (e \circledast_F p_2))(\sigma) && \text{by defn. of } or_F \end{aligned}$$

If e has no solutions, we can write $e(\sigma) = empty_T(\sigma)$ for any search model, that is, any extended monad T, and we use the similar argument:

$$\begin{aligned} & (e \circledast_T p)(\sigma) \\ & = (join_T \cdot map_T\; p \cdot e)(\sigma) && \text{by defn. of } \circledast_T \\ & = (join_T \cdot map_T\; p \cdot empty_T)(\sigma) && \text{by } e = empty_T \\ & = empty_T(\sigma). && \text{by (6.21)} \end{aligned}$$

In the last step we use the law for extended monads which states that $empty_T$ is a left zero for $\circledast_T$. For forests we then have the pointwise proof:

$$
\begin{aligned}
& (e \circledast_F (p_1 \; or_F \; p_2))(\sigma) \\
& = empty_F(\sigma) && \text{by eqn. above} \\
& = empty_F(\sigma) +\!\!+ \; empty_F(\sigma) && \text{by defn. of } +\!\!+ \text{ and } empty_F \\
& = (e \circledast_F p_1)(\sigma) +\!\!+ (e \circledast_F p_2)(\sigma) && \text{by eqn. above} \\
& = ((e \circledast_F p_1) \; or_F \; (e \circledast_F p_2))(\sigma) && \text{by defn. of } or_F
\end{aligned}
$$

We now proceed to formalise the approach presented in this example.

7.3 Soundness and completeness of the embedding

The contents of this section are closely related to Clark's [26] treatment of the soundness and completeness (for positive literals) of the evaluation of completed logic programs. This is not surprising, since we both deal with the semantics of completed logic programs and the correspondence of their evaluation to LD-resolution. However, we have different motivations for this approach.

Clark arrives at the completed form of a logic program and draws a connection between LD-resolution in Horn clause logic and deduction in first order logic in order to justify the use of the negation-as-failure inference rule with respect to truth-functional semantics. We arrive at the completed form of a logic program and draw a connection between LD-resolution and lazy evaluation of a Haskell programs in order to justify the use of equational reasoning about logic programs. We shall return to the differences in our approaches at the end of this section.

The example in the previous section makes heavy use of our laws for $\doteq$; they are used for inference about equalities in order to emulate unification. According to Clark [26], such use of an identity theory is correct:

Lemma 7.1
For any relation r, if $r(t'_1, \ldots, t'_n)$ unifies with $r(t''_1, \ldots, t''_n)$, with strong mgu

$\theta = \{x_1/t_1, \dots x_k/t_k\}$, then the following two conjunctions of equations can be proved equivalent according to our algebraic laws:

$$(t'_1 \doteq t''_1 \,\&\, \dots \,\&\, t'_n \doteq t''_n) \;=\; (x_1 \doteq t_1 \,\&\, \dots \,\&\, x_k \doteq t_k) \tag{7.3}$$

On the other hand, if $r(t'_1, \dots, t'_n)$ does not unify with $r(t''_1, \dots, t''_n)$, then we have:

$$(t'_1 \doteq t''_1 \,\&\, \dots \,\&\, t'_k \doteq t''_k) \;=\; \textit{false}. \tag{7.4}$$

In other words, the predicate on the right, representing either the substitution θ or the failed result *false*, can be arrived at either by a simultaneous unification of all the term pairs (t'_i, t''_i), or by rewriting the conjunction of all equations on the left using our $\doteq$-related laws.

According to our laws, these two equations can be proved using induction on the number of steps in the unification algorithm, and relating each of these steps to the $\doteq$-laws. This is not surprising, since the equality-related laws were defined exactly so that they would correspond closely to the unification algorithm described in Chapter 2. In the induction proof, we can use the law (4.22) when the unification algorithm breaks composite terms apart, we use the substitution law (4.28) to apply the computed bindings to the rest of the term, and the laws (4.23–4.24) to give a contradiction when unification would fail.

We proceed to prove that one LD-sprouting step corresponds to a transformation, using our algebraic laws, of a predicate. In order to simplify the proofs that follow, we first establish some notation. The normal form of a predicate, introduced earlier in this chapter, is crucial for the formalisation of the sprouting steps in the embedding. We shall abbreviate the normal form: $\|_i\, (\exists \vec{x_i}.\; x_{i1} \doteq t_{i1} \,\&\, \dots \,\&\, x_{in_i} \doteq t_{in_i} \,\&\, q_i)$ as:

$$\|_i\, (\exists \vec{x}'_i.\; e_{\theta_i} \,\&\, q_i).$$

Here $\vec{x}'_i$ denotes the list of all the local variables of the i^{th} disjunct, and e_{θ_i} is the environment representing the substitution θ_i, which corresponds to the conjunction of assignments $x_{i1} \doteq t_{i1} \,\&\, \dots \,\&\, x_{in_i} \doteq t_{in_i}$. As before, the query

q_i is the conjunction which corresponds to the remaining computation. The empty normal form is equivalent to *false*.

Recall from the definition of SLD-resolution (from Chapter 2) that we use the pair $\langle q; \theta \rangle$ to denote the query q in the environment θ, where, if q is non-empty, that one atom is selected in it. Since we know that a conjunction of equations can be used to represent environments and substitutions, we shall now replace θ with e_θ, where e is an environment representing the substitution θ. According to the definition of an LD-resolvent, where a restriction to a left-most selection rule has been imposed, the LD-resolvent $\langle q', q; e_{\theta\eta} \rangle$ of a pair $\langle l, q; e_\theta \rangle$, with l the selected atom, and the clause c, is:

$$\langle l, q; e_\theta \rangle \underset{\text{SLD}}{\longrightarrow} \langle q', q; e_{\theta\eta} \rangle,$$

where $h \leftarrow q'$ is a variant of c that is variable disjoint with l, q and e_θ; and $l\theta$ and h unify with the mgu η.

This presentation of LD-resolution is convenient for our purposes because it is very close to the normal form of a predicate. In the completed form, where the conjunctions and existential quantifications of local variables are made explicit, we may write this LD-step as:

$$(\exists \vec{x}.\ e_\theta \,\&\, l \,\&\, q) \underset{\text{SLD}}{\longrightarrow} (\exists \vec{x}, \vec{x}'.\ e_{\theta\eta} \,\&\, q' \,\&\, q),$$

Three comments regarding this alternative notation: First, the placement of & between the environment and the rest of the query is justified both logically in the completed form of the predicate, and also by the implementation of our embedding. In the embedding, the interaction of $\doteq$ and & operators is such that, during evaluation, the environments e_θ and $e_{\theta\eta}$ are applied to the rest of the query. This interaction is recorded appropriately with our laws. Second, since we assume a left-most selection rule, the queries to the left of l will already be computed when we select l, and all the information these computed predicates contain is represented by the environment e_θ. Third, the syntactic constraint that $h \leftarrow q'$ must be variable disjoint with l, q and e_θ can be satisfied by appropriately renaming the local variables of q'. These renamed local variables are exactly the variables $\vec{x}'$.

An LD-sprouting step from q results in a disjunction of k queries derivable from q by resolving on the selected literal. In our case, this selected literal must be l, the left-most predicate after the environment. Then, a *sprouting step*, denoted as $\underset{\text{SLD}}{\longmapsto}$, is defined as:

$$\begin{aligned}(\exists \vec{x}.\ e_\theta \,\&\, l \,\&\, q)\ \underset{\text{SLD}}{\longmapsto}\ & (\exists \vec{x}, \vec{x_1}'.\ e_{\theta\eta_1'} \,\&\, q_1' \,\&\, q)\\ \| \ & \ldots\\ \| \ & (\exists \vec{x}, \vec{x_k}'.\ e_{\theta\eta_k'} \,\&\, q_k' \,\&\, q),\end{aligned}$$

where the elements $\vec{x}_i$, $e_{\theta\eta_i'}$ and q_i' of each disjunct are derived as for a single LD-step. If there are no derivable queries from q by resolving on the selected literal, that is, if $k = 0$, we have:

$$(\exists \vec{x}.\ e_\theta \,\&\, l \,\&\, q)\ \underset{\text{SLD}}{\longmapsto}\ \mathit{false}.$$

Both of these derivation steps can be achieved by equational rewriting using the certain sequences of algebraic laws. So we can prove:

Lemma 7.2
A query q is equivalent, according to our algebraic laws, to the disjunction of queries derivable from q by LD-resolving on the left-most literal.

In the proof of this lemma, we need to show that the algebraic laws can simulate LD-sprouting in two cases, corresponding to the definition above: one where the disjunctions of derivable queries is non-empty, and the other one where it is empty. In case of an unsuccessful sprouting, that is, if $k = 0$:

$$\begin{aligned}&\exists \vec{x}.\ (e_\theta \,\&\, l \,\&\, q)\\ &\quad \{\text{literal } l \text{ does not match any clause}\}\\ &= \exists \vec{x}.\ (e_\theta \,\&\, \mathit{false} \,\&\, q)\\ &\quad \{\mathit{false} \text{ is a left zero for } \&\text{, law (4.3)}\}\\ &= \exists \vec{x}.\ (e_\theta \,\&\, \mathit{false})\\ &\quad \{\mathit{false} \text{ is a right zero for } \& \text{ in equations such as } e \,\&\, p\text{, law (7.2)}\}\\ &= \exists \vec{x}.\ (\mathit{false})\\ &\quad \{\vec{x} \text{ does not appear in } \mathit{false}\text{, law (4.15)}\}\\ &= \mathit{false}.\end{aligned}$$

In case of a successful sprouting, that is, if $k \neq 0$, we have:

$$
\begin{aligned}
& \exists \vec{x}.\ (e_\theta \mathbin{\&} l \mathbin{\&} q) \\
& \quad \{\text{selecting } l \text{ and expanding it to general form}\} \\
& = \exists \vec{x}.\ (e_\theta \mathbin{\&} ((\exists \vec{x}_1.\ e_{\eta_1} \mathbin{\&} q_1) \parallel \ldots \parallel (\exists \vec{x}_k.\ e_{\eta_k} \mathbin{\&} q_k)) \mathbin{\&} q) \\
& \quad \{\text{variable renaming, law (4.13)}\} \\
& = \exists \vec{x}.\ (e_\theta \mathbin{\&} (\exists \vec{x}'_1.\ (e_{\eta'_1} \mathbin{\&} q'_1) \parallel \ldots \parallel (\exists \vec{x}'_k.\ e_{\eta'_k} \mathbin{\&} q'_k)) \mathbin{\&} q) \\
& \quad \{\& \text{ distributes through } \parallel \text{ from the left for } e\text{-predicates, law (7.1)}\} \\
& = \exists \vec{x}.\ (((e_\theta \mathbin{\&} (\exists \vec{x}'_1.\ e_{\eta'_1} \mathbin{\&} q'_1)) \parallel \ldots \parallel (e_\theta \mathbin{\&} (\exists \vec{x}'_k.\ e_{\eta'_k} \mathbin{\&} q'_k))) \mathbin{\&} q) \\
& \quad \{\exists \text{ distributes through } \& \text{ , law (4.15)}\} \\
& = \exists \vec{x}.\ (((\exists \vec{x}'_1.\ e_\theta \mathbin{\&} e_{\eta'_1} \mathbin{\&} q'_1) \parallel \ldots \parallel (\exists \vec{x}'_k.\ e_\theta \mathbin{\&} e_{\eta'_k} \mathbin{\&} q'_k)) \mathbin{\&} q) \\
& \quad \{\text{right distributivity of } \& \text{ through } \parallel \text{ , law (4.6)}\} \\
& = \exists \vec{x}.\ ((\exists \vec{x}'_1.\ e_\theta \mathbin{\&} e_{\eta'_1} \mathbin{\&} q'_1 \mathbin{\&} q) \parallel \ldots \parallel (\exists \vec{x}'_k.\ e_\theta \mathbin{\&} e_{\eta'_k} \mathbin{\&} q'_k \mathbin{\&} q)) \\
& \quad \{\exists \text{ distributes through } \parallel \text{ , law (4.17)}\} \\
& = (\exists \vec{x}, \vec{x}'_1.\ e_\theta \mathbin{\&} e_{\eta'_1} \mathbin{\&} q'_1 \mathbin{\&} q) \parallel \ldots \parallel (\exists \vec{x}, \vec{x}'_k.\ e_\theta \mathbin{\&} e_{\eta'_k} \mathbin{\&} q'_k \mathbin{\&} q).
\end{aligned}
$$

Now we can evaluate, separately in each new query, the left-most literal from q'_i in the respective new environments.

Since LD-derivations are obtained by a repeating application of LD-steps, we can simulate LD-derivations by grouping the single LD-steps into sprouting steps, and simulating the sprouting steps by algebraic rewriting according to Lemma 7.2. By induction, we can prove that the construction of a complete LD-evaluation tree is tantamount to a rewriting proof in the embedding. But the algebraic laws which guide this rewriting are provably correct with respect to the execution of the embedding. So we have:

Theorem 7.1
For any computed answer substitution $\theta | Var(q)$ resulting from an LD-derivation $\langle q; \varepsilon \rangle \underset{SLD}{\longrightarrow}^* \langle \Box; \theta \rangle$, there is a corresponding answer substitution θ' returned by the embedding, such that θ and θ' are equal up to renaming of automatically generated variables.

In this proof, let $\bar{q}$ represent the frontier of the completed LD-tree for q.

If the query for q results in k answers θ_i, where $1 \leq i \leq k$, then $\bar{q}$ can be represented as the normal form predicate:

$$\exists \vec{x}_1.\ e_{\theta_1} \ \parallel \ \cdots \ \parallel \ \exists \vec{x}_k.\ e_{\theta_k}.$$

Each of these disjuncts consist only of an existentially quantified environment, corresponding to one computed instance $q\theta_i$, or answer substitution $\theta_i | \mathit{Var}(q)$, of the LD-derivation from the query q.

Given q and $\bar{q}$ as above, we simply use induction on the number of sproutings in an LD-tree, and apply Lemma 7.2 to prove that

$$q \underset{\text{SLD}}{\longrightarrow}^{*} \bar{q} \quad \Leftrightarrow \quad q \ = \ \bar{q}.$$

Consequently, also $q(\varepsilon) \ = \ \bar{q}(\varepsilon)$, so if the embedding computes the answers $\{\theta_1, \dots \theta_k\}$ for $q(\varepsilon)$, they are the same as the ones computed by LD-resolution for $q(\varepsilon)$. Not only do $q(\varepsilon)$ and $\bar{q}(\varepsilon)$ contain the same answers, they also compute the same stream, matrix or forest of answers in the respective implementations of the embedding. So the results returned by the embedding are presented in the same order, under any chosen search strategy, as the answers computed by LD-resolution.

We stress again that even though the computation of $q(\varepsilon)$ in the embedding *does not* follow this process, it *does* give the same answers. This result holds for any search strategy chosen for the traversal of the LD-tree, since the algebraic laws used for the simulation hold in all search models of the embedding.

The Theorem 7.1 is important because it allows us to prove the adequacy of the embedding, under any search strategy and allowing recursive predicates. According to Theorem 2.1 LD-resolution is sound, and according to Theorem 2.2, it is complete. Since the answers computed by the evaluation of our embedding are the same as the ones computed by LD-resolution, we have:

Theorem 7.2
The embedding is sound and complete with respect to the standard declarative semantics of logic programming.

The treatment above deals only with the completeness for evaluations sprouting from positive literals. Even though we have an operator *not* which corresponds to negation of a predicate, here we choose to not elaborate on the semantics of its evaluation. It suffices to say that, since we implement it as a simple negation as failure rule, we adopt the soundness results from Clark and follow the same restrictions. In order to allow recursive predicates we only consider evaluation of ground negated literals, that is, of literals which become ground in the given environment. Non-ground literals could possibly be dealt with by means of some residuation-like technique, but we have not explored this possibility.

As mentioned earlier, our proof of soundness and completeness is reminiscent of Clark's original treatment of the correctness of evaluation of completed logic programs. We now stress our differences. Clark sees the LD-tree as a structural representation of a natural deduction style proof, where alternatives in the proof space become explicit disjunction and and match-failures become false equalities. Consequently his logic operators $\wedge$, $\vee$ and $\exists$ obey all the standard laws of predicate calculus. His predicates are simply first-order logical formulae. On the other hand, our operators &, $\|$ and $\exists$ are, in fact, Haskell functions which are more restricted than their logic counterparts, and our predicates have an operational semantics which corresponds to LD-resolution. Therefore, as we know, & is not commutative, and does not always distribute through $\|$ from the left. Also, we choose to implement & as sequential, rather than parallel, composition – and for this reason we cannot allow a random selection rule. Clark's declarative approach does not have such restrictions.

In this operational respect, our treatment is more related to an approach called "Formulas as Program" taken by Apt and Bezem in [5]. In the next section we relate our treatment to this work, and use this connection as an alternative proof of adequacy of the embedding.

7.4 Relation to Formulas as Programs

In [5] Apt and Bezem provide a computational interpretation of first-order formulae over arbitrary interpretations. This is a generalisation of our approach, since we provide an operational semantics for a subset of first-order logical formulae over a fixed interpretation of all ground terms of the given language. Both our restrictions follow from the standard logic programming tradition. With respect to the approach of Formulas as Programs, our first restriction is primarily of syntactical nature, while the second has significant semantical consequences: it constrains us to Herbrand models, but this restriction allows us to guarantee a decidable and computationally efficient notion of equality, without forcing us to restrict the equality predicates to ground assignments or ground tests.

In this work, as in ours, first-order formulae are viewed as executable programs, searching for a satisfying valuation for the formula in question. The operational semantics is given in terms of evaluation trees for a formula and a given input environment. In contrast, our operational semantics is given by executable Haskell code. The main result in [5] states that this computational mechanism is sound, in the sense that every computed answer valuation of a query validates it. Below we prove that, with appropriate restrictions on the notion of equality, our semantics are equivalent. Therefore we may conclude that also our work is sound.

The generalisation from Herbrand models to arbitrary interpretations is achieved in [5] by lifting the notion of substitutions to *valuations*. Valuations are, like substitutions, defined as single-valued sets of pairs x/d, where x is a variable, but d can be an element of an arbitrary domain, so not necessarily a term. The concepts of empty valuations, compositions of valuations, and of a subsumption ordering on valuations, are parallel to those for substitutions.

In [5] the operational semantics of a formula is defined in terms of a tree $[\phi]_\alpha$, depending on a formula ϕ and the initial valuation α. The root and all internal nodes of the evaluation tree are labelled with a pair consisting of a formula and a valuation, and the leaves of the tree are labelled with either: *error*, corresponding to a diverging computation; *fail*, corresponding to a failed computation; or a valuation α', corresponding to a successful computation and containing a satisfying valuation of the free variables of the root formula.

For example, the evaluation tree for the empty conjunction, denoted as $\square$, is depicted below:

$$\begin{tikzcd}
\Box, \alpha \arrow[d] \\
\alpha
\end{tikzcd}$$

This empty conjunction $\Box$ corresponds to the predicate *true* in our embedding, and as our computation of *true*, this tree just returns the input valuation. However, in our setting the return type is always explicitly a collection, while here that information is implicitly given by the fact that the tree might have several branches, each possibly resulting in a successful valuation.

Such evaluation trees are defined for all formulas, which are inductively defined as follows. All formulas are conjunctions; this corresponds to the form of queries in logic programming. As seen above, the empty conjunction is denoted as $\Box$, and in the non-empty case every conjunct is either an atomic formula or one of the following: disjunction, conjunction, implication, negation, or an existential quantification of another formula. The atomic formulas include equations of terms $s = t$. The operational semantics is defined in terms of lexicographic induction on the pairs (s_1, s_2) where, for any formula $\phi_1 \wedge \psi$, s_1 is the size of the whole formula and s_2 is the size of ϕ_1. We claim that this inductive definition corresponds to our implementation of operators in the embedding, when valuations are restricted to substitutions and the notion of equality is restricted to "is unifiable with":

Lemma 7.3
Given a formula ϕ and substitution α, and a translation p of ϕ to a predicate in our embedding. If the notion of equality is restricted to "is unifiable with", then each state in the evaluation tree $[\phi]_\alpha$ corresponds to a state in the Haskell evaluation of $p(\alpha)$.

The proof is by induction on the structure of the formula ϕ, following the definitions of the evaluation trees.

The base case concerns the leaves of the evaluation tree, and we argue that for each there exists a corresponding result of a Haskell program in our setting. When interpretations are restricted to free term algebras, the notion of valuations coincides with that of substitutions, so a successful leaf in the evaluation tree corresponds a successful answer substitution in the computation of embedding. In our setting an invariably failing computation corresponds to the predicate *false*, which is the zero of each monad and therefore terminates the $\|$ branch where it occurs. Finally, the operational meaning of *error* in our setting is an infinite, or diverging, computation.

The inductive step has six cases, one for each type of conjunctions: where the selected formula is an atom (i), a disjunction (ii), a conjunction (iii), implication (iv), negation (v) or existential quantification (vi).

The (i) is the most interesting case, since this is where the restriction from arbitrary algebras to the free algebra of terms is most significant. When interpretations are not restricted to term algebras, the equality atoms are not guaranteed to be safely evaluated, since equality may involve solving a test for satisfiability of a sequence of constraints, which may be undecidable. Therefore, in [5] computation of $=$ atoms is restricted to a form where this may not happen, as described below.

The concepts of an α-closed term and an α-assignment are, in turn, a generalisation of our ground terms, and a restriction of our $\doteq$-predicates. If α is a valuation, a term t is said to be α-closed if every variable in t gets a value in α. The result of applying the valuation α of some expression E is denoted as E^α and is obtained by replacing each α-closed term t by $t\alpha$. Further, an α-assignment is an equation $s = t$ where one side is a variable that is not α-closed, and the other a term which is α-closed, that is, "ground".

The four trees below show the evaluation of formulas where the left-most literal is an atom. As seen from the trees, also this framework imposes a left-most selection rule. We expand on the four conditions below.

$$\begin{tikzcd}
A \wedge \psi, \alpha \arrow[d, "cond\ 1"] & A \wedge \psi, \alpha \arrow[d, "cond\ 2"] & A \wedge \psi, \alpha \arrow[d, "cond\ 3"] & A \wedge \psi, \alpha \arrow[d, "cond\ 4"] \\
{[\psi]_\alpha} & \mathit{fail} & \mathit{error} & {[\psi]_{\alpha'}}
\end{tikzcd}$$

Conditions 1 and 2 require that the atom A is α-closed and respectively true or false. In our setting, this corresponds to terms that evaluate to ground terms in the input environment; this results either in success, with no new additions to the resulting environment for the rest of the computation, or in failure.

Conditions 3 and 4 correspond to cases where the atom A is not α-closed, and where in the first case it is not an α-assignment while in the second it is. The tree corresponding to condition 3 is one where our restriction the Herbrand algebras allows us a more liberal treatment of atoms; we may have atoms with logical variables which get values later in the computation, and do not have to diverge in this case. The second tree assumes that A is an α-assignment, say, $x = t$. Here the rest of the evaluation consists of evaluating ϕ in the new environment α', which corresponds to α extended with the pair x/t. In our setting, the computation of $(x \doteq t \,\&\, p)(\theta)$ also computes p in the environment $\theta \cup \{x/t\}$, since we know that $x \doteq t$ succeeds whenever x does not appear in t. Again, we do not need to require that $t\theta$ is ground.

The cases (II) and (III) are quite straightforward. The evaluation trees of $\vee$ and $\wedge$ are depicted below. They have the same effect as our implementations of $\|$ and &, where the first uses concatenation to evaluate the two branches independently, while the second behaves as sequential composition:

$$\begin{tikzcd}
& (\phi_1 \vee \phi_2) \wedge \psi, \alpha \arrow[dl] \arrow[dr] & & (\phi_1 \wedge \phi_2) \wedge \psi, \alpha \arrow[d] \\
{[\phi_1 \wedge \psi]_\alpha} & & {[\phi_2 \wedge \psi]_\alpha} & {[\phi_1 \wedge (\phi_2 \wedge \psi)]_\alpha}
\end{tikzcd}$$

In the next three trees we see the evaluation of implication, case (IV). Conditions 5 and 6 require that ϕ_1 is α-closed and respectively failed (containing only failed leaves) or successful (containing at least one success leaf). The last tree, with condition 7, covers all other cases.

$$\begin{tikzcd}
(\phi_1 \to \phi_2) \wedge \psi, \alpha \arrow[d, "cond\ 5"] & (\phi_1 \to \phi_2) \wedge \psi, \alpha \arrow[d, "cond\ 6"] & (\phi_1 \to \phi_2) \wedge \psi, \alpha \arrow[d, "cond\ 7"] \\
{[\psi]_\alpha} & {[\phi_2 \wedge \psi]_\alpha} & error
\end{tikzcd}$$

In the embedding we do not have $\to$, but we have negation and disjunction, and the combination of these operators gives exactly the same operational behaviour as these trees for $\to$. By law (4.6), we know that the evaluation of $(not\ p_1 \parallel p_2)\ \&\ p_3$ is equivalent to evaluating $(not\ p_1\ \&\ p_3) \parallel (p_2\ \&\ p_3)$. These two $\parallel$ branches are evaluated independently, and they correspond to the first and second tree above.

We only compute negation of ground literals, but the same restriction is present in the evaluation trees above; the two successful trees only allow the evaluation of a α-closed negated formula. Such evaluation succeeds or fails without extending the environment, and that is why both ψ and $\phi_2 \wedge \psi$ are evaluated in the old α environment.

The next three trees deal with case (V), the "negation as failure" rule:

$$\begin{tikzcd}
\neg\phi \wedge \psi, \alpha \arrow[d, "\mathit{cond}\ 8"] & \neg\phi \wedge \psi, \alpha \arrow[d, "\mathit{cond}\ 9"] & \neg\phi \wedge \psi, \alpha \arrow[d, "\mathit{cond}\ 10"] \\
{[\psi]_\alpha} & \mathit{fail} & \mathit{error}
\end{tikzcd}$$

Conditions 8 and 9 require that ϕ is α-closed (the same restriction about ground negation as above), and in the first condition $[\phi]_\alpha$ is assumed to contain only failure leaves while in the second it contains at least one success leaf. In the third tree the computation diverges due to non-ground negation. Our implementation behaves the same in all three cases.

For case (VI), the evaluation tree of $\exists$ is presented below:

$$\begin{tikzcd}
\exists x.\ \phi \wedge \psi, \alpha \arrow[d, "\mathit{cond}\ 11"] \\
{[\phi \wedge \psi]_\alpha}
\end{tikzcd}$$

Condition 11 imposes syntactic requirements about the freshness of the variable x: it must not occur neither in the domain of α nor in ϕ. This is assumed fixed via appropriate renaming, which is exactly what we do in our implementation of $\exists$.

So we know that our embedding simulates evaluation trees, under described restrictions on the permitted interpretations and the use of $\doteq$. This gives us an alternative proof of the soundness of the embedding, because the following soundness theorem has been proved for evaluation trees: Let ϕ be a formula and α a valuation. If $[\phi]_\alpha$ contains a success leaf labelled with α', then α' extends α and $\forall(\phi^\alpha)$ is true. If $[\phi]_\alpha$ is failed, then $\exists(\phi^\alpha)$ is false. The proof of this theorem is by lexicographic induction on the structure of the computation.

Regarding program transformation, the notion of equivalence between two formulas in [5] is based on their evaluation trees, similarly to our equality between predicates. Two formulas are equivalent if both computation trees are successful and return the same set of successful leaves, or if both computation trees are failed. In our setting, the equivalence between two formulas is the precise notion of equivalence between two Haskell programs, and as we have shown, it may be used for a complete axiomatisation of the equivalence relation induced by the computation mechanism. This axiomatisation transfers to the evaluation mechanism of Formulas as Programs in the restricted case of logic programs.

Chapter 8

Properties of Predicates

Certain universal properties are satisfied by all predicates, whatever the underlying search model may be. In this chapter we identify such properties and show that they are respected by all predicates defined using the operators of our embedding. We also show that, because of our function *step*, recursive definitions of predicates have a unique fixed point.

8.1 Non-recursive predicates

The implementation of the basic operators of the embedding can be used to recognise the characteristic properties which cause predicates in the embedding, such as *append*, to behave like a corresponding Prolog relation. We call these characteristics the *healthiness* properties of predicates in the embedding. We state and prove three such properties: all predicates are *conservative* (8.1), *monotonic* (8.2) and *oblivious* (8.3), in the following sense:

$$\sigma' \in p(\sigma) \Rightarrow \sigma \sqsubseteq \sigma', \tag{8.1}$$

$$\sigma_1 \sqsubseteq \sigma_2 \Rightarrow p(\sigma_1) \sqsubseteq_* p(\sigma_2), \tag{8.2}$$

$$t_1(\sigma) = t_2(\sigma) \Rightarrow (r\ t_1)(\sigma) = (r\ t_2)(\sigma). \tag{8.3}$$

It is exactly these three properties that we need for our treatment of the denotational semantics of predicates, and the property (8.3) was also used in our argument for the correctness of the Leibniz law (4.28). The relation $\sqsubseteq_*$ used above is the subsumption relation on collections of substitutions, defined in Chapter 2, where for two sets of substitutions S and S', $S \sqsubseteq_* S'$ holds iff for every $\sigma' \in S'$ there exists a more general $\sigma \in S$.

Predicates are conservative because each substitution σ' in the answer returned by a predicate $p(\sigma)$ refines the input substitution σ. Predicates are monotonic in their input, because the more refined the input substitution to a predicate, the more refined will be each of the answers. Finally, relations r definable with our operators do not differentiate between input arguments which have the same value under the input substitution, so they are oblivious to argument names.

Our predicates, in the most general model, return forests. Forests, in essence, are trees with an arbitrary finite branching factor. In this section we consider all such trees that have finite depth. For finite trees, each of these properties can be proved by induction over the structure of predicate expressions. In the next section we show that these properties are such that we may extend this argument to infinitely deep trees as well. In the rest of this chapter we use "predicate" to mean predicate definable in the embedding, that is, a Haskell function built from from $\doteq$-predicates, *true* and *false*, and possibly other healthy primitives, and with &, $\|$, *not* and $\exists$ as the predicate combinators.

Lemma 8.1

All non-recursive predicates definable in terms of the operators are conservative: For any non-recursive predicate p and substitutions σ and σ', if σ' is one of the answers in the collection computed by $p(\sigma)$, then $\sigma'$is a refinement of σ. That is:

$$\sigma' \in p(\sigma) \Rightarrow \sigma \sqsubseteq \sigma'.$$

In the proof of this lemma, the basis of induction concerns atomic predicates, which are either $\doteq$-predicates, or *true* or *false*. In case of equations, the refinement holds by our definition of the $\doteq$ operator. All equations compute the unification of two terms relative to the input substitution. If the unification fails, the result is the empty list of answers, so the property holds trivially. If the unification succeeds, the resulting substitution must be a refinement of the input. In case of *true* the collection of answers returned is the singleton containing the input answer, so the refinement is $\sigma' = \sigma\varepsilon$. In case of *false* the collection of answers returned is empty, so the property holds trivially.

For the inductive step, we assume that the property holds for the predicates p_1 and p_2, so each element in the collections of answers to these predicates must refine the input substitution. The operator $\|$ preserves the conservative property since its collection of answers is the concatenation (or merging, in the *bfs* model) of the collections of answers to the two disjuncts and thus contains the same elements. By the induction hypothesis, each of these elements refines the input substitution.

The operator & preserves the property since in $(p_1 \,\&\, p_2)(\sigma)$ it first computes $p_1(\sigma)$, in which each element σ' refines σ by the induction hypothesis, and then for each σ' it computes $p_2(\sigma')$, where each element σ'' refines σ' again by the hypothesis. But refinement is a transitive relation on substitutions, so each σ'' also refines σ.

We write $(\exists x.\ p_1)$ as a shorthand for *exists* $(\lambda x.\ p_1)$. Here *exists* provides a fresh variable x' to the λ-expression, so in effect it replaces the bound variable x with a fresh variable x' in p_1, let us denote this renamed predicate as p_1'. By the induction hypothesis, each answer σ' to $p_1'(\sigma)$ refines the input σ. Depending on whether we standardise the answers or not, $\exists$ either leaves this substitution unchanged, or removes the variable x' from it. In the first case we already have that $\sigma \sqsubseteq \sigma'$. In the second case, make $\sigma'' = \mathit{drop}_{x'}(\sigma')$. Since $x \notin \mathit{Dom}(\sigma)$, we still have $\sigma \sqsubseteq \sigma''$.

Lemma 8.2
All non-recursive predicates definable in terms of the operators are monotonic: For any non-recursive predicate p and input substitutions σ_1 and σ_2, if σ_2 refines σ_1, then also every answer of $p(\sigma_2)$ refines some answer of $p(\sigma_1)$. That is:

$$\sigma_1 \sqsubseteq \sigma_2 \Rightarrow p(\sigma_1) \sqsubseteq_* p(\sigma_2).$$

The base case of induction again concerns $\doteq$ predicates and *true* and *false*. The property holds trivially for *true* and *false*, because *true* returns the unchanged input substitution in a singleton, and *false* return the empty collection of answers.

In case of $(t_1 \doteq t_2)$, we may use Lemma 7.1 from the previous chapter, stating that $(t_1 \doteq t_2)$ is equivalent to $(x_1 \doteq t'_1 \,\&\, \ldots \,\&\, x_k \doteq t'_k)$, if the unification of t_1 and t_2 succeeds, and to *false*, if it fails. As before, x_i denotes a variable and t'_i denotes a term. If we assume $\sigma_1 \sqsubseteq \sigma_2$, we may write $\sigma_2 = \sigma_1\eta$. The relationship between:

$$(x_1 \doteq t'_1 \,\&\, \ldots \,\&\, x_k \doteq t'_k)(\sigma_1) \quad \text{and} \quad (x_1 \doteq t'_1 \,\&\, \ldots \,\&\, x_k \doteq t'_k)(\sigma_1\eta)$$

must then be one of the following. If the first query returns the empty collection of answers, it is because one or more $x_i \doteq t'_i$ assignments were inconsistent with σ_1. But they must also be inconsistent with $\sigma_1\eta$, so the property holds trivially. If the first query returns a non-empty collection, it will return a singleton with σ'_1 as its only solution, where σ'_1 consists of σ_1 extended with some subset of $\{x_1/t'_1, \ldots, x_k/t'_k\}$, with the duplicated bindings removed from this set. The collection returned by the second query will either be a singleton containing the same answer, or be empty because of some inconsistency between $\{x_1/t'_1, \ldots, x_k/t'_k\}$ and η. In both cases the relation $(t_1 \doteq t_2)(\sigma_1) \sqsubseteq_* (t_1 \doteq t_2)(\sigma_2)$ holds.

For the inductive step, assuming that the predicates p_1 and p_2 are monotonic in the sense above, we show that the remaining operators preserve the property. For $\|$, the collections of answers to $(p_1 \,\|\, p_2)(\sigma_1)$ and $(p_1 \,\|\, p_2)(\sigma_2)$ are, respectively, the concatenation of the collections of answers to the queries $p_1(\sigma_1)$ and $p_2(\sigma_1)$, and $p_1(\sigma_2)$ and $p_2(\sigma_2)$. By induction hypothesis each of the elements of $p_1(\sigma_2)$ and $p_2(\sigma_2)$ refine some element in $p_1(\sigma_1)$ and $p_2(\sigma_1)$, respectively, so this property holds for the union of these collections as well.

Further, for $(p_1 \,\&\, p_2)(\sigma_1)$ and $(p_1 \,\&\, p_2)(\sigma_2)$, by the hypothesis we have that each answer σ'_2 in the collection returned by $p_1(\sigma_2)$ refines some answer σ'_1 from $p_1(\sigma_1)$. So when p_2 is mapped over these σ'_2 and σ'_1, by the induction hypothesis each resulting substitution in $p_2(\sigma'_2)$ must refine some substitution in the collection returned by $p_2(\sigma'_1)$.

Finally, for $\exists x.\ p_1$, by the hypothesis each answer σ_2' to $p_1(\sigma_2)$ refines some answer σ_1' in $p_1(\sigma_1)$. Let $\sigma_2' = \sigma_1'\eta$; due to $\exists x$, renaming of x to fresh x' in $p_1(\sigma_2)$ results in σ_2', where either σ_1' or η (or none) are correspondingly renamed. If the change affects σ_1', then obviously the answer σ_1'' to $\exists x.\ p_1(\sigma_1)$ will change the same way, so σ_2'' refines σ_1''. If it affects η, the shared bindings stay the same so σ_2'' is still a refinement of σ_1', which is identical to σ_1''.

Lemma 8.3
All non-recursive predicates definable in terms of the operators are oblivious: For any non-recursive predicate $r\ t$, terms t_1 and t_2, and input substitution σ, if t_1 and t_2 evaluate to the same term under substitution σ, the query $(r\ t_1)(\sigma)$ is equivalent to $(r\ t_2)(\sigma)$. That is:

$$t_1(\sigma) = t_2(\sigma) \Rightarrow (r\ t_1)(\sigma) = (r\ t_2)(\sigma).$$

In other words, predicates compute the same answer for all arguments that have same values under the input substitutions. The reason for this is that our unification operator computes relative to the input substitution, so the difference is annulled whenever we get to the part of the computation that leads to new components in the answers. The other three combinators merely propagate the relationship down to unification. For example, if $t_1\sigma = t_2\sigma$, then obviously $(r_1\ t_1 \parallel r_2\ t_1)(\sigma)$ will compute the same answers as $(r_1\ t_2 \parallel r_2\ t_2)(\sigma)$.

8.2 Healthiness of recursive predicates

Using the forest model, we now show that the properties considered in the previous section extend to recursive predicates. The argument presented in this section has two parts. First we show that all predicates are such that the healthiness properties are preserved for arbitrary large finite trees. Then we show that the properties we consider here are such that we may generalise their preservation from arbitrary large finite trees to infinite trees.

Some care must be taken here, because the transition to from finite to infinite trees does not hold for many important properties for predicates: for example, in general completeness cannot be proved this way, which is why we base our completeness proof on simulating LD-resolution.

The proof of the first part is based on computation induction, where we show that the computation preserves the healthiness properties in each step, so if a predicate is healthy up to n steps of computation, it will also be healthy for $n + 1$ steps. If we relate this to our general model, the steps of the computation correspond to the depth of the tree for the query. This means that if we can prove that the properties propagate in this way, we can assume that they hold for all queries resulting in an arbitrarily big finite search tree.

Consider a recursively defined predicate $r\ x$. We may assume that such predicates are defined as:

$$r\ x = step\ (\mathcal{F}(r)\ x),$$

where $\mathcal{F}$ is some predicate expression built from the relation r, other healthy predicates previously defined in the program, and atomic predicates $\doteq$, *true* and *false*, using combinators &, $\|$, *not* and $\exists$. We do not wish to restrict the terms on the right to be smaller than on the left, since this may not always be the case. For example, the definition (3.1) of the recursive predicate *append* can alternatively be written as:

$$\begin{aligned}
&append(p,q,r) = step\ (\mathcal{F}(append)(p,q,r))\\
&\qquad where\ \mathcal{F}(app)(p,q,r) =\\
&\qquad\qquad (p \doteq nil\ \&\ q \doteq r)\\
&\qquad\qquad \|\ (\exists x,y,z.\ p \doteq [x|y]\ \&\ r \doteq [x|z]\ \&\ app(y,q,z)).
\end{aligned}$$

For simplicity, we only discuss linear recursion, where $\mathcal{F}$ has at most one recursive call. However, the argument extends to the non-linear case. The first auxiliary result we need is:

Lemma 8.4
$\mathcal{F}$ is non-destructive. That is, the partial answers in the first n levels in the search tree of $(\mathcal{F}(p))(\sigma)$ depend only on the partial answers from the first n levels in the search tree of $p(\sigma)$.

The proof is by induction on the structure of $\mathcal{F}$. For the induction base, $\mathcal{F}(p)$ must either be an equation or *true* or *false*, and therefore does not have any constituent predicates, so the lemma holds trivially in this case.

For the inductive step, we assume that the property holds for p_1 and p_2. Then, $\mathcal{F}(p)$ is either a disjunction $p_1 \parallel p_2$, a conjunction $p_1 \mathbin{\&} p_2$, a negation *not* p_1 or a quantified predicate $\exists x.\ p_1$. The combinators $\parallel$, *not* and $\exists$ preserve the depth of the answers in the search tree, while the answers resulting from & have depth equal to the sum of the depths of answers of the two conjuncts. So all the answers to $(\mathcal{F}(p))(\sigma)$ on the level n come from answers to $p(\sigma)$ on levels n or less. By the induction hypothesis, these answers only depend on levels n or less, so we are done.

Letting *healthy* stand for any the three healthiness properties from the previous section, we are now ready to prove that if a predicate returns healthy answers up to level n in the search tree, then it also returns healthy answers for level $n+1$. This argument is based on the behaviour of *step* function, which pushes each answer one level down the tree.

Lemma 8.5
If $r\ x = step\ (\mathcal{F}(r)\ x)$ and the predicate $r\ x$ is healthy up to n levels, for arbitrary n, then step $(\mathcal{F}(r)\ x)$ must also be healthy up to $n+1$ levels.

The proof is by induction on n. The base case concerns the answers at level 0; but there are no such answers, so the statement holds trivially.

For the induction step, we consider $n \neq 0$. The induction hypothesis states that for any term x and any substitution σ, any answer σ' produced by $(r\ x)(\sigma)$ up to level n is healthy. We denote the restriction of a search tree of $p(\sigma)$ up to level n by $p(\sigma)\!\uparrow\! n$. By definition of *step*, if σ' is an answer in $(step\ (\mathcal{F}(r)\ x)(\sigma))\!\uparrow\!(n+1)$, it must be an answer in $(\mathcal{F}(r)\ x)(\sigma))\!\uparrow\! n$. That is, the answers to $(step\ (\mathcal{F}(r)\ x))(\sigma)$ at level $n+1$ come from the answers to $((\mathcal{F}(r)\ x))(\sigma)$ at level n.

Now, $\mathcal{F}$ builds its resulting predicate from the predicate $r\ x$ and other healthy predicates using $\parallel$, &, *not* and $\exists$ combinators. But we have established that $\mathcal{F}$ is non-destructive, so all the answers to $(\mathcal{F}(r)\ x)(\sigma)$ at level n must come from answers to $r\ x$ and these other predicates from the level

n or less. By the induction hypothesis, they are all healthy.

We have already shown in lemmas 8.1, 8.2 and 8.3 that these operators preserve the three healthiness properties. Therefore, all the answers at level n in $(\mathcal{F}(r)\ x)(\sigma)$ are also healthy, and thus also all the answers at level $n+1$ in $(\mathit{step}\ (\mathcal{F}(r)\ x))(\sigma){\uparrow}(n+1)$ are also healthy.

So, we know that all recursive predicates with finite trees are conservative, monotonic and oblivious. Further, we wish to show that since this is the case for arbitrarily large finite trees, it extends to the infinitely large trees.

Lemma 8.6
The conservative, monotonic and oblivious properties of predicates do not depend on the finiteness of the search tree.

The conservative property of predicates concerns only computed answers. They are always a result of a finite computation, corresponding to a finite branch, and it is irrelevant whether the tree might contain infinite branches. So in this case, the property does not depend on the finiteness of the tree.

The monotonicity property compares two trees for the same predicate p, under different substitutions σ_1 and σ_2 such that $\sigma_1 \sqsubseteq \sigma_2$. The query $p\sigma_1$ involving the more general substitution might result in an infinite tree where the other query $p\sigma_2$ results in a finite tree. For example, in the two queries:

$$\mathit{append}(x_1, x_2, x_3)(\{x_2/[2]\}) \text{ and } \mathit{append}(x_1, x_2, x_3)(\{x_1/[1], x_2/[2]\})$$

the first one results in a tree with infinitely many successful branches, while the second results in a finite tree with a singe successful leaf corresponding to the answer $\{x_1/[1], x_2/[2], x_3/[1,2]\}$. However, the property is stated for *all computed answers* of the more refined query, and each such answer will always be reached at a finite depth in the tree. Relating to our example, the second, more refined, query has only one answer $\{x_1/[1], x_2/[2], x_3/[1,2]\}$, and for this answer there exists a more general answer in the collection returned by the first query: $\{x_1/[x_4], x_2/[2], x_3/[x_4, 2]\}$. But this corresponding less refined answer is always reached at the same depth in the tree, so it does not matter that there are infinitely many other answers.

The oblivious property states that the two trees corresponding to the queries $(r\ t_1)(\sigma)$ and $(r\ t_2)(\sigma)$ are equal under certain conditions. But both finite and infinite trees are equal exactly if they can be proved equal to an arbitrary depth, so this property is conserved by transition to infinite trees.

So, we have shown that the three healthiness properties hold for recursive predicates when their search trees are finite, and that the properties are preserved also when the trees are infinite. Combining the two results, we get:

Theorem 8.1
All predicates are conservative, monotonic and oblivious, in the sense of lemmas 8.1, 8.2 and 8.3.

For the sake of discussion, it is interesting to note why this argument does not apply to some other properties of predicates, such as completeness. The inductive proof of completeness, even in the non-diverging case, depends on the finiteness of the answer tree. Completeness states that the computed tree of the predicate query contains all the correct answers. We can prove by induction that we always compute the subset of correct answers which corresponds to the answers found in the tree up to depth n. However, with recursion, there can be infinitely many answers, and these answers could require increasingly deep branches; even though each finite tree is complete up to a given level, this does not mean that the infinite is as well.

Since we treat completeness through LD-resolution, the recursive predicates do not cause problems.

Another complex area related to recursive logic programs is the issue non-termination. It is a complex issue in program transformation and semantic analysis of Prolog programs. However, non-termination is not a great issue for us for two reasons. First, program transformation in our case is by equivalence, not implication. Since all our steps preserve non-termination, each of transformation step yields two equally non-terminating trees, which is fine. Second, our $\|$ is parallel under fair search strategies; in Prolog the unfair *dfs* is used, and a divergence of one $\|$ branch leads to the divergence of the whole query.

8.3 Denotational semantics of predicates

The semantics of pure logic programming provided by our embedding is denotational, in the sense that the meaning of each predicate arises from the composition of the meanings of its constituents. Indeed, in [3], Apt proposes a denotational semantics for first order logic that is similar to our embedding. The semantics proposed there is the denotational counterpart of the operational semantics from [5], and it is consequently also built in a more general setting of arbitrary interpretations for a given language $\mathcal{L}$, while our embedding is, as pointed out before, set in a Herbrand world of universal term algebras for $\mathcal{L}$.

From a denotational point of view, we need to attribute to each predicate symbol p in the program some well-defined and uniquely minimal relation r, so that the meaning given to p shall be unambiguous. The approach taken in [3] is to define the meaning of a formula relative to input substitutions, as a mapping from an input substitution to a set of answer substitutions. This fits well with our treatment of predicates, since we also view them as mappings from substitutions to collections of substitutions. Recall the type of predicates in the general search model:

$$\mathbf{type}\ \mathit{Predicate} = \mathit{Answer} \to \mathit{Forest}\ \mathit{Answer},$$

where *Answer* is, in essence, a substitution. Each substitution is a finite mapping, so can be viewed as a finite set. A *Forest* is a (possibly empty or infinite) collection of such finite sets. If $\mathcal{P}(A)$ denotes the powerset of the set A, a set-based reading of the type definition above gives us the denotational semantics of predicate p:

$$[\![p]\!] : \mathit{Answer} \to \mathcal{P}(\mathit{Answer}).$$

The basic principle of denotational semantics is that the meaning of each syntactic construct is defined as a semantical function of its direct constituents. This induces a definition driven by the syntactic structure of the language; our primitive syntactic constructs are the atomic predicates $\doteq$, *true* and *false*, and the composite predicates are built with &, $\|$, *not* and $\exists$.

The denotational semantics of the atomic predicates is:

$$[\![\mathit{true}]\!](\sigma) = \{\sigma\}, \tag{8.4}$$

$$[\![\mathit{false}]\!](\sigma) = \emptyset, \tag{8.5}$$

$$[\![t_1 \doteq t_2]\!](\sigma) = \{\sigma\mu \mid \mu \leftarrow \mathit{mgu}(t_1\sigma, t_2\sigma)\}. \tag{8.6}$$

These definitions correspond directly to our implementation of predicates $\doteq$, *true* and *false* in the embedding.

Further, for each of the predicate operators &, $\|$, *not* and $\exists$ we define a corresponding semantic operator, and state equations that relate the two kind of operators. We have:

$$[\![p_1 \parallel p_2]\!](\sigma) = [\![p_1]\!](\sigma) \cup [\![p_2]\!](\sigma), \tag{8.7}$$

$$[\![p_1 \,\&\, p_2]\!](\sigma) = \mathit{union}(\mathit{map}\ [\![p_2]\!]\ ([\![p_1]\!](\sigma))), \tag{8.8}$$

$$[\![\exists x.\ rx]\!](\sigma) = \{\mathit{drop}_v\ (\sigma') \mid \sigma' \in [\![rv]\!](\sigma)\},\ v \text{ is fresh.} \tag{8.9}$$

Here $\cup$ computes a union of two (possibly infinite) sets; however, in the forest model, even infinite sets of answers are represented as finite lists of trees, some of which may be of infinite depth. The union of two such sets of answers can easily be computed as a concatenation of the two lists of trees, which is exactly what our $\|$ operator does. Similarly, & composes the answers in the two sets corresponding to their cost, so $\mathit{union} \cdot \mathit{map}\ [\![p_2]\!]$ computes also on infinite sets. The $\mathit{drop}_v\ (\sigma)$ function removes the variable v if it is in the domain of σ. For simplicity, our implementation of $\exists$ does not perform this elimination, but in principle it is a part of the standardisation of answers described in Chapter 4. The equations (8.7–8.9) above correspond to a set-based reading of the implementation of the operators.

Relating our semantics to that of Apt's [3], the main differences are in the generality of interpretations, and in the consequent treatment of terms and equations. There, the substitutions are generalised to correspond to an arbitrary interpretation $\mathcal{J}$, and a special state *error* is needed to deal with the possibility of a divergence in the computation of $t_1 = t_2$ or elsewhere. So the meaning of a formula ϕ is:

$$[\![\phi]\!] : \mathit{Subs} \rightarrow \mathcal{P}(\mathit{Subs} \cup \{\mathit{error}\}).$$

The reason for our not having *error* in the answer set is that we have fair-search strategies to deal with infinite computation and in our restricted setting equalities never lead to *error*.

The denotational semantics of a term t is in [3] defined relative to the chosen $\mathcal{J}$-substitution and is denoted $[\![t]\!]_{\mathcal{J}}$. It is a "partially-evaluated" term where each maximal ground subterm is replaced by its value in $\mathcal{J}$. An $\mathcal{J}$-substitution can be viewed as a generalisation of our substitutions: if one chooses the Herbrand algebra as the interpretation $\mathcal{J}$, $\mathcal{J}$-substitutions and substitutions coincide and so do the meaning of the term t, $[\![t]\!]_{\mathcal{J}}$, and the term t itself.

The other main difference is in the treatment of equality: our treatment of equality can be more liberal, since unification does not diverge. The denotational semantics of equality in [3] allows only ground equality-tests and arbitrary assignments to variables (with the occurs-check restriction). For other atomic formulae $p(t_1, \dots, t_n)$, the denotational semantics is defined only if they are ground.

With the restrictions of the interpretation to Herbrand interpretations, and the more liberal treatment of equality for our setting, our denotational semantics can be viewed as a restriction of that of [3] to logic programming. Our embedding is provably consistent with this denotational semantics; it is also consistent with the operational semantics of [5], for the restricted case of logic programming. So, the embedding can be used as the link between the two in the restricted setting of logic programming.

8.4 Herbrand semantics of predicates

We now give an alternative denotational semantics for predicates, where the meaning of a predicate is independent of an input substitution. This approach is similar to the traditional declarative semantics of logic programming, where the meaning of a logic program corresponds to the set of atoms that are implied by the program. The relation that we now attribute to each predicate p is the minimal model of p. To differentiate from the semantics $[\![\cdot]\!]$ in the previous section, we denote this semantics by $decl(p)$.

The Least Herbrand Model of p, $\mathit{decl}(p)$, corresponds to the set of all ground substitutions γ which extend some computed answer σ of $p(\varepsilon)$:

$$\mathit{decl}(p) = \{\gamma \mid \exists\sigma.\ \sigma \in \mathit{Set}(p(\varepsilon)) \wedge \sigma \sqsubseteq \gamma\}, \tag{8.10}$$

where $\mathit{Set}(p(\varepsilon))$ corresponds to the set of all answers computed by the predicate call $p(\varepsilon)$. It is a collection of all the leaves in the forest returned by this predicate, and may consequently be an infinite set.

Following the basic compositionality principle of denotational semantics, we again specify how the meaning of the whole is composed from the meaning of its constituent parts. For each of the syntactic operators $\&$, $\|$, $\exists$, and $\doteq$ we define a corresponding semantic operator, and equations that relate the two kind of operators:

$$\begin{aligned}
&\mathit{decl}(p_1 \parallel p_2) = \mathit{decl}(p_1) \cup \mathit{decl}(p_2), && (8.11)\\
&\mathit{decl}(p_1 \mathbin{\&} p_2) = \mathit{decl}(p_1) \cap \mathit{decl}(p_2), && (8.12)\\
&\mathit{decl}(\exists x.\ rx) = \{\mathit{drop}_v(\gamma) \mid \gamma \in \mathit{decl}(rv)\}, \text{where } v \text{ is fresh} && (8.13)\\
&\mathit{decl}(t_1 \doteq t_2) = \{\gamma \mid t_1\gamma = t_2\gamma\}. && (8.14)
\end{aligned}$$

Such equations need to be justified – we need to show that, for example, $\cup$ satisfies (8.11). This claim constitutes the soundness and completeness proof for the implementation of $\|$. We have:

$$\begin{aligned}
&\mathit{decl}(p_1 \parallel p_2)\\
&= \{\gamma \mid \exists\sigma.\ \sigma \in \mathit{Set}((p_1 \parallel p_2)(\varepsilon)) \wedge \sigma \sqsubseteq \gamma\} && \text{by (8.10)}\\
&= \{\gamma \mid \exists\sigma.\ \sigma \in \mathit{Set}((p_1(\varepsilon) \mathbin{+\!\!+} p_2(\varepsilon)) \wedge \sigma \sqsubseteq \gamma\} && \text{by (5.12)}\\
&= \{\gamma \mid \exists\sigma.\ \sigma \in \mathit{Set}((p_1(\varepsilon)) \wedge \sigma \sqsubseteq \gamma\} \cup\\
&\qquad \{\gamma \mid \exists\sigma.\ \sigma \in \mathit{Set}(p_2(\varepsilon)) \wedge \sigma \sqsubseteq \gamma\} && \text{by (8.15)}\\
&= \mathit{decl}(p_1) \cup \mathit{decl}(p_2). && \text{by (8.10)}
\end{aligned}$$

The step marked with (8.15) is justified because Set satisfies the following property with regards to $+\!\!+$:

$$\mathit{Set}(p_1(\sigma) \mathbin{+\!\!+} p_2(\sigma)) = \mathit{Set}(p_1(\sigma)) \cup \mathit{Set}\ (p_2(\sigma)). \tag{8.15}$$

If the answer collection to a predicate call $p_1(\sigma)$ is an infinite stream, the set $Set(p_1(\sigma))$ will also be infinite. The union of two sets, even if they are infinite, should be commutative. The question then, is, whether (8.15) holds in all models of the embedding. In case of the *dfs*-model, where infinite streams are used, the concatenation $p_1(\sigma) +\!\!+\, p_2(\sigma)$ would never reach the second stream if the stream of $p_1(\sigma)$ is infinite. However, in any fair model, including our general-search and *bfs* models, this step holds. This is because the answer collections in each of these models consist a finite list of possibly infinitely deep structures, and the concatenation therefore always computes in a finite time. On the other hand, in case of unfair search, this property does not hold in logic programming either. The same holds for diverging computation, that is, if the search-tree of $p_1(\sigma_1)$ contains infinite branches.

In the proof of (8.12), things are more complicated. We can only prove the soundness part $decl(p_1 \mathbin{\&} p_2) \subseteq decl(p_1) \cap decl(p_2)$:

$$
\begin{aligned}
& decl(p_1 \mathbin{\&} p_2) \\
& = \{\gamma \mid \exists\sigma.\ \sigma \in Set((p_1 \mathbin{\&} p_2)(\varepsilon)) \wedge \sigma \sqsubseteq \gamma\} && \text{by (8.10)} \\
& = \{\gamma \mid \exists\sigma.\ \sigma \in Set(join(map\ p_2(p_1(\varepsilon)) \wedge \sigma \sqsubseteq \gamma\} && \text{by (5.13)} \\
& \subseteq \{\gamma \mid \exists\sigma.\ \sigma \in Set((p_1(\varepsilon)) \wedge \sigma \sqsubseteq \gamma\} \cap \\
& \qquad \{\gamma \mid \exists\sigma.\ \sigma \in Set(p_2(\varepsilon)) \wedge \sigma \sqsubseteq \gamma\} && \text{by } (*) \\
& = decl(p_1) \cap decl(p_2) && \text{by (8.10)}
\end{aligned}
$$

The step marked by $(*)$ is justified by the monotonic (8.2) and conservative (8.1) healthiness properties, as we show below. Let $\sigma \in Set(p_1(\varepsilon))$ and $\sigma' \in Set(p_2(\sigma))$. For any such σ and σ', we have:

$$
\begin{aligned}
& \sigma' \in Set(p_2(\sigma)) \Rightarrow \sigma' \sqsubseteq \sigma. && \text{by (8.1)} \\
& \sigma' \in Set(p_2(\sigma)) \wedge \varepsilon \sqsubseteq \sigma \Rightarrow \exists\theta \in Set(p_2(\varepsilon)).\ \theta \sqsubseteq \sigma', && \text{by (8.2)}
\end{aligned}
$$

So, σ' refines at the same time the answer σ in $Set(p_1(\varepsilon))$ and some answer θ in $Set(p_2(\varepsilon))$. But then, for any such σ', the set of ground refinements of σ' must be in the intersection of the ground refinements of σ and θ.

To prove completeness for our implementation of & in this setting, we need to prove that the set inclusion holds the other way as well. For this we need to use induction, with the assumption that p_1 and p_2 also preserve the set inclusion in the other direction. This induction hypothesis corresponds to Theorem 3 from Clark's paper [26], and to the approach used to prove completeness by Apt in [3], where a predicate is assumed equivalent to the disjunction of its computed answer substitutions written out in an equational form. As their results show, this proof can indeed be carried out, but the argument does not extend to the recursive case. The reasons for this are discussed in Section 8.2: properties such as completeness do not extend trivially from arbitrary large finite trees to infinite trees. They may be another way of extending this argument to the recursive case, but at present we do not know of it. That is one of our reasons for using the simulation of LD-resolution to argue for the soundness and completeness of the embedding, also for recursive predicates.

The argument for the adequacy of our implementation of $\exists$ (8.13), is straight-forward, and follows from the oblivious property of predicates (8.3). The equation (8.14), which implies the soundness and completeness of equations, follows immediately from the implementation of $\doteq$.

8.5 Uniqueness of fixpoints

Let $p = \mathcal{F}(p)$ denote a recursive predicate definition. Below we outline a proof that this equation defines a unique predicate solution. The proof uses metric spaces (cf. Smyth [128]) and a contraction property of predicates to prove its uniqueness. This uniqueness result is best proved for the tree model of logic programs, and can be easily promoted to the other fair models of logic programs, such as the matrix model, but it does not hold for the unfair model of depth-first search. The extension of the argument to mutually recursive predicates is straight-forward, and uses tuples of predicates.

Intuitively, we need to show that there is some well-defined way of measuring the 'distance' between two predicates, that the predicates are well-behaved in a certain sense regarding this distance, and that the functional $\mathcal{F}$ has some contractive effect on it. In what follows below we denote by $\mathbb{T}$ the set of all answer trees, by $\mathbb{P}$ the set of all predicates, by $d_{\mathcal{T}}$ the distance function between two trees and by d_p the distance function between two predicates.

The contractiveness of $\mathcal{F}$ is provided by the *step* function which, as described in section 3.4, determines the level of an answer in the search tree. We define the truncation $(\mathcal{T}{\upharpoonleft}n)$ to be the tree $\mathcal{T}$ restricted to n levels. We will use ${\upharpoonleft}n$ to measure how similar two trees are: the distance between two trees corresponds to their level of agreement, and diminishes the longer it takes to distinguish between them. We first define a difference function $d_{\mathcal{T}}(\mathcal{T}_1, \mathcal{T}_2)$ between two trees $\mathcal{T}_1$ and $\mathcal{T}_2$ to be 2^{-n} (the so-called Baire distance), where $n = max\{n \mid (\mathcal{T}_1{\upharpoonleft}n) = (\mathcal{T}_2{\upharpoonleft}n)\}$, that is, the deepest common level of the two trees. If the trees are equal, we define $d_{\mathcal{T}}(\mathcal{T}_1, \mathcal{T}_2)$ to be 0.

Then, we define $d_p(p, q)$, for two predicates p and q, to be the supremum $sup\{d(p(x), q(x)) \mid x \in \mathit{Subst}\}$ of all the distances between answer trees for p and q for all input substitutions x. This is always defined, because *sup* is 1 in the worst case (when $n = 0$), and 0 in the case when the two predicates are equal.

The set of all trees with the function $d_{\mathcal{T}}$, $(\mathbb{T}, d_{\mathcal{T}})$, forms a *metric space*. This is the case since the function $d_{\mathcal{T}}$ satisfies the following by its definition:

$$d_{\mathcal{T}}(x, y) = 0 \Leftrightarrow x = y, \tag{8.16}$$

$$d_{\mathcal{T}}(x, y) = d_{\mathcal{T}}(y, x), \tag{8.17}$$

$$d_{\mathcal{T}}(x, z) \leq d_{\mathcal{T}}(x, y) + d_{\mathcal{T}}(y, z). \tag{8.18}$$

The space $(\mathbb{T}, d_{\mathcal{T}})$ is actually *ultrametric*, since we can state an even stronger inequality than (8.18): the level of agreement between two trees is less or equal to the *maximum* level of agreement between the two original trees and a third one. This obviously implies that it must be less or equal to their sum. Because all the distances in this space are ≤ 1, we call $(\mathbb{T}, d_{\mathcal{T}})$ 1-bounded. Now, predicates are functions from substitutions to trees of substitutions, and a standard result from the metric space theory guarantees that if $(\mathbb{T}, d_{\mathcal{T}})$ is 1-bounded (ultra)metric space, and the distance between two functions in the space $\mathbb{P}$ is defined as d_p above, we have that the function space $(\mathbb{P}, d_p)$ is ultrametric.

Furthermore, a Cauchy sequence $\mathcal{T}_i$ of trees is defined as:

$$\forall \epsilon > 0.\ \exists i.\ \forall j, k > i.\ \ d(\mathcal{T}_j, \mathcal{T}_k) \leq \epsilon. \tag{8.19}$$

The distance between trees in a Cauchy sequence diminishes as the level of agreement between the trees is getting deeper. For any ϵ, we can guarantee that the trees in the sequence agree on at least i levels. Since this ϵ can get arbitrarily small, the trees in the sequence will have to agree to arbitrary many levels, and the trees of the sequence will resemble more and more the (possibly infinite) tree that is the limit of such a sequence. Thus, every Cauchy sequence $\mathcal{T}_i$ of trees in this space converges to some limit tree $\mathcal{T}$ (8.20):

$$\forall \epsilon > 0.\ \exists i.\ \forall j \geq i.\ \ d(\mathcal{T}_j, \mathcal{T}) \leq \epsilon. \tag{8.20}$$

Since every Cauchy sequence in the space $(\mathbb{T}, d_{\mathcal{T}})$ converges to an element of $\mathbb{T}$, it is a *complete* metric space. This property of the space guarantees that limits are available, whenever necessary, and this is crucial for the fixpoint theory stated further below.

Completeness of the space is preserved on the function space constructed from it. Therefore, we also have:

LEMMA 8.7
$(\mathbb{P}, d_p)$ is a complete metric space.

A function f is said to be *contractive* on some metric space if there exists some constant ρ such that $d(f(x), f(y)) < \rho \times d(x, y)$, for all x and y in that space. In the case of the functional $\mathcal{F}$, which adds at least one level to any particular search tree (because it must contain at least one application of *step*) the distance $d_{\mathcal{T}}$ between the two trees is at least halved. So $\mathcal{F}$ is a contraction on trees, and also on predicates because it will (at least) halve the *sup* as well as the distance for each individual tree, with $\rho = 1/2$. If $\mathcal{F}$ is contractive, we also have that it is *non-destructive*:

$$(\mathcal{T}_1 {\uparrow} n) = (\mathcal{T}_2 {\uparrow} n) \ \Rightarrow\ (\mathcal{F}(\mathcal{T}_1){\uparrow}(n+1)) = (\mathcal{F}(\mathcal{T}_2){\uparrow}(n+1)), \tag{8.21}$$

since the left hand side implies that $d_{\mathcal{T}}(\mathcal{T}_1, \mathcal{T}_2) \leq 2^{-n}$, and the contractiveness of $\mathcal{F}$ results in $d_{\mathcal{T}}(\mathcal{F}(\mathcal{T}_1), \mathcal{F}(\mathcal{T}_2)) \leq 2^{-(n+1)}$ which implies the right hand side. By the definition of d_p, this property also promotes to predicates:

Lemma 8.8
$\mathcal{F} : \mathbb{P} \to \mathbb{P}$ is a contractive function.

Intuitively this must be true since the answers of $(\mathcal{F}(p))(\sigma)$ with depth $n+1$ depend only on the answers of $p(\sigma)$ with depth n or less.

Since we have asserted that $(\mathbb{P}, d_p)$ is a complete metric space and that $\mathcal{F} : \mathbb{P} \to \mathbb{P}$ is a contractive function, Banach's fixed point theorem allows us to assume and compute the unique solutions to our recursively defined predicates in the tree model. So finally, we have:

Theorem 8.2
For all $p \in \mathbb{P}$, the equation $p = \mathcal{F}(p)$ has a unique solution.

As we have seen here, the uniqueness of fixpoints crucially depends on the *step* function, which provides the contraction property for $\mathcal{F}$ in the general model. The same approach works for the breadth-first search model, because it can be made in a metric space in a similar way, and the *step* there will also make $\mathcal{F}$ contractive. However, in the depth-first model, *step* is the identity function, so the fixpoints in this model are not unique.

Chapter 9

Program Transformation

In this chapter we describe how some important program transformation techniques from functional programming can be translated to logic programs, and present three examples where these technique have been successfully applied to derive efficient implementations of logic programs from their specifications. The transformations presented here are based on fold operations, and as such they preserve program termination.

9.1 Algebraic program transformation

The Prolog predicates *rev*1 and *rev*2 are both true exactly if one argument list is the reverse of the other:

$$\begin{array}{ll}
\mathit{rev}1([\,],[\,]). & \mathit{rev}2(A,B) \leftarrow\ \mathit{revapp}(A,[\,],B). \\
\mathit{rev}1([X|A],C) \leftarrow & \mathit{revapp}([\,],B,B). \\
\quad \mathit{rev}1(A,B), \mathit{append}(B,[X],C). & \mathit{revapp}([X|A],B,C) \leftarrow \\
 & \quad \mathit{revapp}(A,[X|B],C).
\end{array}$$

These two predicates have the same Least Herbrand Models, but they have a very different computational behaviour: the time complexity for *rev*1 is quadratic while for *rev*2 it is linear. We now present a general technique for developing the efficient predicate from the clear but inefficient one, in this and similar examples. The approach presented here is primarily inspired by Bird and de Moor's work [18] on similar program synthesis and transformation techniques for functional programming.

Arguably the most general transformational technique in logic programming is the "rules and strategies" approach [99]. In this technique the *rules* perform operations such as an unfolding or folding of clause definitions, introduction of new clause definitions, deletion of irrelevant, failing or subsumed clauses, and certain rearrangements of goals or clauses. Subject to certain conditions, these rules can be proved correct relative to the most common declarative semantics of logic programs.

In this approach, the application of the transformation rules is guided by meta-rules called *strategies*, which prescribe suitable sequences of basic rule applications. The main strategies involve tupling of goals that visit the same data structure in a similar way, generalisation of goals in a clause in order to fold them with some other clause, elimination of unnecessary variables, and fusion of predicates defined by two independent recursive predicates into a single predicate. Such strategies are used as the building blocks of more complex transformation techniques, and for limited classes of predicates these strategies have been well understood and can be seen as the backbone of a compositional method for transforming logic programs.

Our transformational example can indeed be solved by the rules and strategies approach, together with mathematical induction, needed to prove the associativity of *append* on which the transformation depends. The basic strategies involved are tupling and generalisation, and the derivation is simple and semantically correct relative to the least Herbrand model of the two programs. However, there are a few methodological problems in this approach. First, the declarative semantics does not quite capture the behaviour of logic programs when they are evaluated under the standard depth-first search strategy, and we have no clear measure of the reduction of the computation complexity. Second, the application of induction requires a separate form of reasoning. But maybe most importantly, if we did not know of this particular combination of strategies, there would be no systematic method to guide us in the derivation. As far as we know, there are no general results regarding what complex strategies can be applied for families of predicates.

Below we outline a general approach to logic program transformations, and argue that such an approach should be based on higher-order predicates and their properties.

Many problems similar to the one described above have recently been explored and explained for functional programs in [18]. These results build on the ample heritage of program transformation in the functional programming community and are based on laws of algebra and category theory. According to this algebra of functional programming, the program transformation in the example above can be seen as an instance of a more general transformational strategy, valid for an entire family of programs based on functions *foldl* and *foldr* and parametric in the data structure. Algebraic laws regarding such higher-order functions prove to be highly versatile for functional program transformations.

There are two main advantages in using our functional embedding for transformation of logic programs. The first one is that it allows us to reason about logic programs in a simple calculational style, using rewriting and the algebraic laws of combinators. The second, and the more interesting reason, is that many predicates are easily expressible using higher-order functions that accept more basic predicates as arguments.

We can implement the general "prepackaged recursion operators" *foldl*, *foldr*, *map* etc. as functions from predicates to predicates, and thereby get the opportunity to use their algebraic properties for program transformation. This approach avoids the problems related to higher-order unification, while it gives us the power of generic programming and provides the appropriate language and level of abstraction to reason about logic program transformation. Also, even though we use higher-order syntax, we only apply it in a manner which can be translated in a systematic way to the first-order syntax. Consequently, the laws of Chapter 4 that were proved only for the first-order syntax are still applicable.

Even though each particular derivation can be performed in a first-order setting, the *general* strategies guiding the program transformations depend essentially on the higher-order functions. We argue that, as in functional programming, so also in logic programming it is the properties of generic recursion operators that yield generic transformation strategies.

9.2 Emulation of the unfold/fold technique

We begin with the following definition of a predicate $\mathit{adj}1(x, y, xs)$ that holds if x and y are adjacent elements of list xs:

$$\begin{aligned}
&\mathit{adj}1(x, y, l) = (\exists n.\ \mathit{elem}(x, n, l)\ \&\ \mathit{elem}(y, \mathit{succ}(n), l))\\
&\mathit{elem}(x, n, l) =\\
&\qquad (\exists xs.\ n \doteq 0\ \&\ l \doteq \mathit{cons}(x, xs))\\
&\quad \parallel (\exists m, y, ys.\ n \doteq \mathit{succ}(m)\ \&\ l \doteq \mathit{cons}(y, ys)\ \&\ \mathit{elem}(x, n, ys)),
\end{aligned}$$

where $\mathit{succ}(n)$ is the term denoting the successor of numeral n. By transforming this program, we aim to obtain the following, declaratively equivalent, definition of $\mathit{adj}1$ that avoids the auxiliary predicate elem:

$$\begin{aligned}
&\mathit{adj}2(x, y, l) =\\
&\qquad (\exists xs.\ l \doteq \mathit{cons}(x, \mathit{cons}(y, xs)))\\
&\quad \parallel (\exists z, ys.\ l \doteq \mathit{cons}(z, ys)\ \&\ \mathit{adj}2(x, y, ys)).
\end{aligned}$$

The proof of their equivalence can be done by our algebraic laws as follows. We first substitute the definition of elem for its first occurrence in the body of *adj1* and use the distributivity of & and $\exists$ through $\parallel$ (laws 4.6 and 4.8), getting:

$$\begin{aligned}
&\mathit{adj}1(x, y, ps) =\\
&\qquad (\exists n, xs.\ n \doteq 0\ \&\ ps \doteq \mathit{cons}(x, xs)\ \&\ \mathit{elem}(y, \mathit{succ}(n), ps))\\
&\quad \parallel (\exists n, m, z, xs.\ n \doteq \mathit{succ}(m)\ \&\ ps \doteq \mathit{cons}(z, xs)\ \&\\
&\qquad\quad \mathit{elem}(x, m, xs)\ \&\ \mathit{elem}(y, \mathit{succ}(n), ps)).
\end{aligned}$$

Next we apply the one-point rule (4.19) for n and use the substitution of equals for equals (4.28) to replace ps by $\mathit{cons}(x, xs)$, to obtain:

$$\begin{aligned}
&\mathit{adj}1(x, y, ps) =\\
&\qquad (\exists xs.\ ps \doteq \mathit{cons}(x, xs)\ \&\ \mathit{elem}(y, \mathit{succ}(0), \mathit{cons}(x, xs)))\\
&\quad \parallel (\exists m, z, xs.\ ps \doteq \mathit{cons}(z, xs)\ \&\\
&\qquad\quad \mathit{elem}(x, m, xs)\ \&\ \mathit{elem}(y, \mathit{succ}(\mathit{succ}(m)), \mathit{cons}(z, xs))).
\end{aligned}$$

We now reason about the expression $elem(y, succ(0), cons(x, xs))$. The rewrite steps are shown below. We also use the fact that the unification of two non-unifiable terms ($succ(0)$ and 0) equals the predicate *false*, and that $(\exists x.false)$ equals *false* as a consequence of (4.14). We get:

$$\begin{aligned}
&elem(y, succ(0), cons(x, xs))\\
&\qquad = \quad (\exists zs.\ succ(0) \doteq 0 \mathbin{\&} cons(x, xs) \doteq cons(y, zs))\\
&\qquad\quad \parallel (\exists m, z, zs.\ succ(0) \doteq succ(m) \mathbin{\&}\\
&\qquad\qquad\qquad cons(x, xs) \doteq cons(z, zs) \mathbin{\&} elem(y, m, zs))\\
&\qquad = false \parallel (\exists z, zs.\ x \doteq z \mathbin{\&} xs \doteq zs \mathbin{\&} elem(y, 0, zs))\\
&\qquad = elem(y, 0, xs)\\
&\qquad = \ldots\\
&\qquad = (\exists us.\ xs \doteq cons(y, us)).
\end{aligned}$$

We reason similarly about the expression $elem(y, succ(succ(m)), cons(y, xs))$ to get $elem(y, succ(m), xs)$. Then we substitute these back into our previous expression for $adj1(x, y, ps)$, and get:

$$\begin{aligned}
&adj1(x, y, ps) =\\
&\qquad (\exists xs.\ ps \doteq cons(x, xs) \mathbin{\&} \exists us.(xs \doteq cons(y, us))\\
&\quad\ \parallel (\exists m, z, xs.\ ps \doteq cons(z, xs) \mathbin{\&}\\
&\qquad\quad elem(x, m, xs) \mathbin{\&} elem(y, succ(m), xs)).
\end{aligned}$$

Now in the first clause we pull $\exists us$ to the front using (4.8), then eliminate xs using the one-point rule (4.12). In the second clause we move m inwards using (4.8) and then fold last two goals according to the definition of $adj1$:

$$\begin{aligned}
&adj1(x, y, ps) =\\
&\qquad (\exists us.\ ps \doteq cons(x, cons(y, us)))\\
&\quad\ \parallel (\exists z, xs.\ ps \doteq cons(z, xs) \mathbin{\&} adj1(x, y, xs)).
\end{aligned}$$

This equation is equivalent to our definition of $adj2$. Since $adj1$ and $adj2$ satisfy the same guarded recursive equation, we finally appeal to the uniqueness of fixed-points for recursive predicate definitions, and conclude that the predicates $adj1$ and $adj2$ are the same.

In fact, here and in all the following examples we need the function *step* in order to justify this appeal to uniqueness of fixed-points, since it is the *step* function of a fair strategy that guarantees the contractiveness of the predicate space. We have omitted *step* in the example above for simplicity of presentation. We now sketch the changes needed and the applications to the laws of step. The "proper" predicate definition for $adj1$ is:

$$adj1(x, y, l) = step\ (\exists n.\ elem(x, n, l)\ \&\ elem(y, succ(n), l)),$$

and so on for $adj2$ and $elem2$. After the same transformation steps as before, consisting of the substitution of the definition of *elem* for its first occurrence in the body of $adj1$ and the distribution of & and $\exists$ through $\|$, we have two *step* occurrences enclosed by $\exists$ in the new predicate. We then apply the law (5.26, 5.27) relating *step* to $\exists$ and &, which let us extract the two inner *step* occurrences to the front of each $\|$ branch. Next we apply the distributivity of *step* through $\|$ (5.29) to extract these *step*'s to the front:

$$\begin{aligned} adj1(x, y, ps) = {} & step\ (step\ (\\ & \quad (\exists xs.\ ps \doteq cons(x, xs)\ \&\ elem(y, succ(0), cons(x, xs))) \\ & \quad \|\ (\exists m, z, xs.\ ps \doteq cons(z, xs) \\ & \qquad\quad \&\ elem(x, m, xs)\ \&\ elem(y, succ(succ(m)), cons(z, xs))))). \end{aligned}$$

As before, we now reason about the expression $elem(y, succ(0), cons(x, xs))$, but now we get $step\ (step\ (\exists us.\ xs \doteq cons(y, us)))$. From the expression $elem(y, succ(succ(m)), cons(y, xs))$ we get $step\ (step\ (elem(y, succ(m), xs)))$. Substituting these back into above expression for $adj1(x, y, ps)$, we get:

$$\begin{aligned} adj1(x, y, ps) = {} & step\ (step\ (\\ & \quad (\exists xs.\ ps \doteq cons(x, xs)\ \&\ step\ (step\ (\exists us.\ xs \doteq cons(y, us))) \\ & \quad \|\ (\exists m, z, xs.\ ps \doteq cons(z, xs) \\ & \qquad\quad \&\ elem(x, m, xs)\ \&\ step\ (step\ (elem(y, succ(m), xs)))))). \end{aligned}$$

Now we apply in both clauses the weak-equality law (5.28) for moving the *step* from the second conjunct outside &. This is the first point in this example where we are performing rewriting steps that are true in the declarative but not in the procedural sense. We know, however, that the computed answers are still the same. We then eliminate xs from the first clause using the one-point rule (4.19), and contract the definition of $adj1$ in the second clause, to get:

$$\begin{aligned}
&adj1(x,y,ps) = step\ (step\ (step\ (step\ (\\
&\qquad (\exists us.\ ps \doteq cons(x, cons(y, us))) \\
&\quad \parallel (\exists z, xs.\ ps \doteq cons(z, xs)\ \&\ adj1(x,y,xs)))))).
\end{aligned}$$

We apply the other weak-equality law for *step* (5.30) to reduce the four outermost steps to one, and the equation above becomes equivalent to our translation of the second definition of $adj2$.

9.3 Accumulator introduction

We now return to the example introduced at the beginning of this chapter, ignoring *step* for sake of readability. The standard definition of the naive reverse predicate has quadratic time complexity:

$$\begin{aligned}
&rev1(l1, l2) = \\
&\qquad (l1 \doteq nil\ \&\ l2 \doteq nil) \\
&\quad \parallel (\exists x, xs, ys.\ l1 \doteq cons(x, xs)\ \& \\
&\qquad\qquad rev1(xs, ys)\ \&\ append(ys, cons(x, nil), l2)).
\end{aligned}$$

A better definition of reverse uses an accumulating parameter and runs in linear time:

$$\begin{aligned}
&rev2(l1, l2) = revapp(l1, nil, l2) \\
&revapp(l1, acc, l2) = \\
&\qquad (l1 \doteq nil\ \&\ l2 \doteq acc) \\
&\quad \parallel (\exists x, xs.\ l1 \doteq cons(x, xs)\ \&\ revapp(xs, cons(x, acc), l2)).
\end{aligned}$$

We can prove these two definitions equivalent by using the previously mentioned algebraic laws together with structural induction, following the same style of transformation as in the previous section. This approach is similar to the rules and strategies approach for logic program transformation.

However, there is a shorter and more elegant way of proving these predicates equal, by resorting to program derivation techniques based on higher-order *fold* predicates and their properties. The outline of the derivation is:

$$\begin{aligned}
& rev1(xs, ys) && \\
& = \mathit{foldr}\ (snoc, nil)\ (xs, ys) && \text{by defn. of } \mathit{foldr} \text{ and } snoc \\
& = \mathit{foldl}\ (\mathit{flipapp}, nil)\ (xs, ys) && \text{by duality law (9.1), see below} \\
& = revapp(xs, nil, ys) && \text{by defn. of } \mathit{foldl} \\
& = rev2(xs, ys). && \text{by defn. of } rev2
\end{aligned}$$

We denote this derivation by $(*)$ and justify each of the steps below.

The higher-order operators have proved to be fundamental in functional programming, partly because they provide for a disciplined use of recursion, namely a recursive decomposition that follows the structure of the data type. They also satisfy a set of laws that are crucial in the functional program transformations, and we will rely on one of those laws in our derivation $(*)$.

The definitions of some families of higher-order predicates, for example the map and fold predicates over lists or other data structures, can be made without any extensions to the implementation of our embedding. They can be implemented using Haskell's higher-order functions on predicates, so we do not need to resort to the higher-order unification machinery of, say, λ-Prolog. For example, the predicate *foldr*, which holds iff the predicate p applied right-associatively to all the elements of the list l yields the term res, could be defined as:

$$\begin{aligned}
& \mathit{foldr}\ (p, e)\ (l, res) = \\
& \qquad (l \doteq nil\ \&\ e \doteq res) \\
& \qquad \| \ (\exists x, xs, r.\ l \doteq cons(x, xs)\ \& \\
& \qquad\qquad \mathit{foldr}\ (p, e)\ (xs, r)\ \&\ p(x, r, res)),
\end{aligned}$$

where (p, e) are the higher-order parameters to the function *foldr* and (l, res) are the arguments to the resulting predicate. The predicate p corresponds to a binary function to be applied to the consecutive list elements, and e denotes the initial element used to 'start things rolling'. For example, the function *foldr* $(add, 0)$ applied to $([2, 7, 8], res)$ produces the predicate $r_1 \doteq 0$ & $add(8, r_1, r_2)$ & $add(7, r_2, r_3)$ & $add(2, r_3, res)$; when invoked with the appropriate input substitution (say the empty one), this predicate has the effect that *res* unifies with the term denoting the numeral 17.

In the first step of the derivation $(*)$, we use the following predicate *snoc*:

$$snoc\ (x, l, res) = append(l, cons(x, nil), res).$$

The pattern of recursion in the definition of *rev*1 is the same as that captured by *foldr*. Using a result that guarantees that recursive definitions have unique fixed points, we may conclude that *rev*1 is equal to the instance of *foldr* shown in the second line of our derivation $(*)$.

The next step in $(*)$ involves a transition from *foldr* to another higher-order predicate, *foldl*. This left-associative fold over lists could be defined as:

$$\begin{aligned} &foldl\ (p, e)\ (l, res) = \\ &\qquad (l \doteq nil\ \&\ e \doteq res) \\ &\quad \| \ (\exists x, xs, r.\ l \doteq cons(x, xs)\ \& \\ &\qquad\qquad p(e, x, r)\ \&\ foldl\ (p, r)\ (xs, res)). \end{aligned}$$

Roughly speaking, the function call *foldl* $(add, 0)$ $([2, 7, 8], res)$ would return the predicate $add(0, 2, r_1)$ & $add(r_1, 7, r_2)$ & $add(r_2, 8, r_3)$ & $r_3 \doteq res$. Again, this predicate has the effect of setting *res* to 17, but this time the numbers are added from left to right.

The second step in $(*)$ is an instance of the **duality law**:

$$foldr\ (f, e)\ (l, res) = foldl\ (g, e)\ (l, res), \tag{9.1}$$

where f is replaced by *snoc*, g by *flipapp*, and e by *nil*. Intuitively, we replace tail-concatenation of *snoc* by a composition of frontal *append* operations.

The law above holds if f, g and e satisfy the following requirements: f and g must associate with each other, and $f(x, e, res)$ must equal $g(e, x, res)$ for all x and *res*. The predicates f and g associate with each other iff the predicates $(\exists t.\ f(x, t, res)\ \&\ g(y, z, t))$ and $(\exists t.\ g(t, z, res)\ \&\ f(x, y, t))$ are equal. In functional notation this corresponds to $f(x, g(y, z)) = g(f(x, y), z)$.

The proof of (9.1) requires the following auxiliary result:

$$\begin{aligned}&(\exists t.\ f(x, t, \mathit{res})\ \&\ \mathit{foldl}\ (g, y)\ (xs, t))\\ &= (\exists t.\ f(x, y, t)\ \&\ \mathit{foldl}\ (g, t)\ (xs, \mathit{res})).\end{aligned}$$

This is proved by induction, using the associativity assumption about f and g. Then this equality, with y instantiated to e, is used together with the assumption about the equality of $f(x, e, \mathit{res})$ and $g(e, x, \mathit{res})$, in the induction proof for (9.1).

Returning to our derivation $(*)$, we need to check that the duality law really is applicable, so we now prove that the predicates *snoc* and *flipapp* and term *nil* satisfy the requirements for f, g and e. If *flipapp* is defined as:

$$\mathit{flipapp}\ (l, x, \mathit{res}) = \mathit{append}(\mathit{cons}(x, \mathit{nil}), l, \mathit{res}),$$

then we unfold the definition of both functions, and use the associativity of *append* in step marked with $(**)$, to get:

$$\begin{aligned}&(\exists t.\ \mathit{snoc}(x, t, \mathit{res})\ \&\ \mathit{flipapp}(y, z, t))\\ &= (\exists t.\ \mathit{append}(t, \mathit{cons}(x, \mathit{nil}), \mathit{res})\ \&\ \mathit{append}(\mathit{cons}(z, \mathit{nil}), y, t))\\ &= (\exists t.\ \mathit{append}(\mathit{cons}(z, \mathit{nil}), t, \mathit{res})\ \&\ \mathit{append}(y, \mathit{cons}(x, \mathit{nil}), t)) \quad (**)\\ &= (\exists t.\ \mathit{flipapp}(t, z, \mathit{res})\ \&\ \mathit{snoc}(x, y, t)),\end{aligned}$$

and similarly for $\mathit{snoc}(x, \mathit{nil}, \mathit{res})$ and $\mathit{flipapp}(\mathit{nil}, x, \mathit{res})$. The associativity of *append* used in $(**)$ can be shown by induction on the list argument *res*.

For the penultimate step in our derivation $(*)$, we first prove that

$$\mathit{revapp}(l, \mathit{acc}, \mathit{res}) = \mathit{foldl}\ (\mathit{flipapp}, \mathit{acc})\ (l, \mathit{res}),$$

using a simple induction proof. Then, instantiating the arbitrary term *acc* in *foldl* to the term *nil*, we get exactly the $\mathit{foldl}\ (\mathit{flipapp}, \mathit{nil})\ (xs, ys)$ from the third line of $(*)$, so we can rewrite this to a call to $\mathit{revapp}(xs, \mathit{nil}, ys)$ in the fourth line. The final step follows directly from the definition of $\mathit{rev}2$.

9.4 Generate-and-test elimination

We start with the standard implementation of the *naiveSort* predicate that uses the 'generate-and-test' method to sort a list:

$$\begin{aligned}
&naiveSort(l1, l2) = perm(l1, l2) \mathbin{\&} isSorted(l2)\\
&isSorted(l) =\\
&\qquad l \doteq nil\\
&\quad \parallel (\exists x.\ l \doteq cons(x, nil))\\
&\quad \parallel (\exists x, y, l2.\ l \doteq cons(x, cons(y, l2)) \mathbin{\&}\\
&\qquad\qquad le(x, y) \mathbin{\&} isSorted(cons(y, l2))),
\end{aligned}$$

where *perm* has the standard definition, using the auxiliary predicate *delete*:

$$\begin{aligned}
&perm(l1, l2) =\\
&\qquad (l1 \doteq nil \mathbin{\&} l2 \doteq nil)\\
&\quad \parallel (\exists x, xs, zs.\ l2 \doteq cons(x, xs) \mathbin{\&}\\
&\qquad\qquad delete(x, l1, zs) \mathbin{\&} perm(zs, xs))\\
&delete(x, l1, l2) =\\
&\qquad (\exists ys.\ l1 \doteq cons(x, ys) \mathbin{\&} l2 \doteq ys)\\
&\quad \parallel (\exists y, ys, zs.\ l1 \doteq cons(y, ys) \mathbin{\&} l2 \doteq cons(y, zs) \mathbin{\&}\\
&\qquad\qquad delete(x, ys, zs)).
\end{aligned}$$

We now wish to show that *naiveSort* is equivalent to its more efficient variant *iSort*, which performs insertion sort.

Given a predicate $insert(x, zs, l2)$ which is true if the sorted list $l2$ is the result of inserting the element x in the appropriate position in the sorted list zs, the usual implementation of the *iSort* predicate is:

$$\begin{aligned}
&iSort(l1, l2) =\\
&\qquad (l1 \doteq nil \mathbin{\&} l2 \doteq nil)\\
&\quad \parallel (\exists x, ys.\ l1 \doteq cons(x, ys) \mathbin{\&}\\
&\qquad\qquad iSort(ys, zs) \mathbin{\&} insert(x, zs, l2)),
\end{aligned}$$

$$
\begin{aligned}
&\mathit{insert}(x, l1, l2) = \\
&\qquad (l1 \doteq \mathit{nil} \;\&\; l2 \doteq \mathit{cons}(x, \mathit{nil})) \\
&\quad\; \| \; (\exists y, zs.\; l1 \doteq \mathit{cons}(y, zs) \;\&\; l2 \doteq \mathit{cons}(x, \mathit{cons}(y, zs)) \;\&\; \\
&\qquad\qquad \mathit{le}(x, y)) \\
&\quad\; \| \; (\exists y, ys, zs.\; l1 \doteq \mathit{cons}(y, ys) \;\&\; l2 \doteq \mathit{cons}(y, zs) \;\&\; \\
&\qquad\qquad \mathit{gt}(x, y) \;\&\; \mathit{insert}(x, ys, zs)).
\end{aligned}
$$

The outline of this derivation is similar to the previous example, except that the essential step this time uses the fusion law for *fold* instead of the duality law. If a function is expressed as a composition of functions, the fusion law may be applicable; it combines two or more subsidiary computations into a single function. This may eliminate multiple traversals and intermediate data structures, as in the example here. The derivation is:

$$
\begin{aligned}
&\mathit{naiveSort}(l1, l2) \\
&= \mathit{isSorted}(l2) \;\&\; \mathit{perm}(l1, l2) && \text{by defn. of } \mathit{naiveSort} \\
&= \mathit{isSorted}(l2) \;\&\; \mathit{foldr}(\mathit{add}, \mathit{nil})\ (l1, l2) && \text{by defn. of } \mathit{foldr} \\
&= \mathit{foldr}(\mathit{insert}, \mathit{nil})\ (l1, l2) && \text{by fusion (9.2), see below} \\
&= \mathit{iSort}(l1, l2). && \text{by defn. of } \mathit{iSort}
\end{aligned}
$$

In step one we simply unfold the definition of *naiveSort*$(l1, l2)$ and use the commutativity property of &. In the next step we argue that the predicate *perm* is an instance of *foldr*(*add*, *nil*), where the predicate *add* is as defined below. First, we use an auxiliary result stating that the relation *perm* is symmetric, i.e. the defining equation for $\mathit{perm}(zs, xs)$ and $\mathit{perm}(xs, zs)$ can be rewritten to the same recursive equation. Second, we define *add* to be the converse of *delete*, that is, $\mathit{add}(x, zs, l1) = \mathit{delete}(x, l1, zs)$, and we can now rewrite *perm* as:

$$
\begin{aligned}
&\mathit{perm}(l1, l2) = \\
&\qquad (l1 \doteq \mathit{nil} \;\&\; l2 \doteq \mathit{nil}) \\
&\quad\; \| \; (\exists x, xs, zs.\; l2 \doteq \mathit{cons}(x, xs) \;\&\; \\
&\qquad\qquad \mathit{add}(x, zs, l1) \;\&\; \mathit{perm}(xs, zs)).
\end{aligned}
$$

Then, once again using the result about the symmetricity of *perm*, we swap

$l1$ and $l2$ and obtain a recursive equation equivalent to the one defining $\mathit{foldr}\ (\mathit{add}, \mathit{nil})\ (l1, l2)$.

The third step is the major step in this proof, and it is the one where the efficiency gain is achieved, i.e. the one that captures this transformation strategy. It involves the fusion law for *foldr*, which can be proved by induction on the length of the input list. The assumptions for this law[1] are as follows: let predicates f, g and h, and a term e, be such that $f(e)$ holds, and that $f(\mathit{res})\ \&\ g(x, y, \mathit{res})$ rewrites to the same recursive equation as $h(x, y, \mathit{res})\ \&\ f(y)$ for all terms x, y and *res* (in functional notation, $f(g\ x\ y) = h\ x\ (f\ y)$). Then, the **fusion law** states that:

$$f(\mathit{res})\ \&\ \mathit{foldr}\ (g, e)\ (l, \mathit{res}) = \mathit{foldr}\ (h, e)\ (l, \mathit{res}). \tag{9.2}$$

If we now insert our predicate *isSorted* for f, *add* for g, *insert* for h, and *nil* for e, the third step in the main proof is a straight-forward application of the fusion law. We only need to prove that our choices for f, g, h, and e satisfy the two fusion law requirements. The predicate call $\mathit{isSorted}(\mathit{nil})$ holds by definition, and the remaining condition for f, g and h is that:

$$\mathit{isSorted}(\mathit{res})\ \&\ \mathit{add}(x, l, \mathit{res}) = \mathit{insert}(x, l, \mathit{res})\ \&\ \mathit{isSorted}(l).$$

This equality can also be proved by an application of algebraic laws and induction on the lists l and *res*, using the lemma:

$$\begin{aligned}&\mathit{delete}(x, zs, ys)\ \&\ \mathit{isSorted}(\mathit{cons}(y, zs))\\ &\ = gt(x, y)\ \&\ \mathit{delete}(x, zs, ys)\ \&\ \mathit{isSorted}(\mathit{cons}(y, zs))\end{aligned}$$

which can also be proved by induction on the argument lists zs and ys.

In the final step, we simply recognise that $\mathit{iSort}(l1, l2)$ is equivalent to $\mathit{foldr}\ (\mathit{insert}, \mathit{nil})\ (l1, l2)$.

Following a similar approach, we can also derive the equivalence of the naive sort and, for example, *quickSort* or *selectionSort*. Both of these derivations rely on the fusion law, but they are algebraically slightly more advanced than the above derivation of *iSort* because they also involve properties of

[1]the assumptions are slightly simplified for the sake of this example

unfold predicates.

The derivation of *quickSort* uses fold and unfold predicates on trees. The reason for this is that even though *quickSort* is usually represented as a flat recursive predicate, it has a compositional form which is basically a sort on trees where the intermediate tree data type has been eliminated. Essentially, the derivation of *quickSort* involves proving the equality:

$$\begin{aligned}&\mathit{isSorted}(l2)\ \&\ \mathit{perm}(l1,l2)\\&=\mathit{mkTree}(l1,t)\ \&\ \mathit{flatten}(t,l2),\end{aligned}$$

where the predicate $\mathit{mkTree}(l1,t)$ holds if t is an ordered tree:

$$\begin{aligned}&\mathit{mkTree}(l,t) =\\&\qquad (l \doteq \mathit{nil}\ \&\ t \doteq \mathit{null})\\&\qquad \| \ (\exists x, xs, t1, t2, l1, l2, a.\ l \doteq \mathit{cons}(x,xs)\ \&\ t \doteq \mathit{fork}(t1,a,t2)\ \&\\&\qquad\qquad \mathit{split}(l,l1,a,l2)\ \&\ \mathit{mkTree}(l1,t1)\ \&\ \mathit{mkTree}(l2,t2))\\&\mathit{split}(l,l1,a,l2) =\\&\qquad (\exists y, ys.\ l \doteq \mathit{cons}(y,ys)\ \&\ a \doteq y\ \&\\&\qquad\qquad \mathit{filter}\ (\lambda x.\mathit{le}(x,y))\ (l,l2)\ \&\ \mathit{filter}\ (\lambda x.\mathit{gt}(x,y))\ (l,l1)),\end{aligned}$$

where $\mathit{filter}\ (g)\ (l,l1)$ is a higher order predicate that holds if $l1$ contains all the elements of l that satisfy g, and $\mathit{flatten}(t,l)$ holds if the list l corresponds to the flattened tree t. The terms representing trees and fold functions on trees are defined similarly to the corresponding definitions for lists.

Chapter 10

Advanced Program Transformation

Optimisation problems are well suited for logic programming, because they often require a simple specification and a subsequent search for the best solution in a combinatorial space. In this chapter we show how our algebraic calculus of relations can be applied to an analysis and the derivation of the appropriate efficient algorithm for certain families of optimisation problems. The main purpose of this chapter is to make a tentative connection, and to provide a fresh perspective on the work of Bird and de Moor in [18]; we do not attempt a thorough semantical analysis of their theorems in the setting of our embedding.

10.1 Optimisation problems

Dynamic programming[1] is the name for a general strategy used in algorithms that organises the computation so that subproblems are evaluated once instead of many times; traditionally this is done by combining a recurrence equation with tabling or memoing. As applied to combinatorial optimisation problems, dynamic programming was first popularised by Bellman in

[1]The nomenclature in the dynamic programming literature is somewhat inconsistent, so in what follows we have chosen to follow a neutral source [32].

[107], where he introduced the *Principle of Optimality* which states that an optimal solution is composed of optimal solutions to subproblems. This is the essential (though only sufficient and not necessary) condition for the dynamic programming technique to be applicable. Greedy algorithms also suppose that the principle of optimality holds, but in addition they exploit a greedy condition which guarantees that on basis of some local information only the best subproblem needs to be computed. Some problems fall in between these two extremes of pursuing all or only one of the recursive decompositions.

The problems which satisfy the principle of optimality can be divided into two categories. Some are naturally solved in a top-down way, through a decomposition of a recursive data-type and a consequent combination of the partial answers. Others are better solved in a bottom-up way, where a recursive data-type which contains the answers is being composed from some seed. Provided that the conditions mentioned above hold, greedy algorithms can be derived for both of these classes of problems, but dynamic programming, as treated here, only applies to problems with bottom-up, compositional specifications. For the problems which do not satisfy the greedy condition yet require a decompositional solution, we derive a third alternative: thinning algorithms.

From a programmers point of view, two main questions arise from this description of algorithms for optimisation problems. One is how to *formalise* the aforementioned conditions for each of the three strategies, in such a way that programmers can easily analyse the problem and identify the appropriate algorithm design strategy. The second is how can the programmers use this information to *derive* the correct and efficient algorithm within the given strategy. In the traditional, tabulating, approach to dynamic programming, there is no general answer to these two questions.

These two questions have recently been addressed by the relational algebraic approach to optimisation problems, and applied in the framework of functional programming. Helman [61] was the first to separate the ideas of problem structure and computation, and his ideas have been generalised by de Moor and Bird [18, 39] and later by Curtis [36].

Here, we apply these results to the setting of logic programs by showing how this approach can be used to answer these questions for three standard examples in dynamic programming: the string-edit, the minimum lateness and the 1/0 knapsack problem. In section 10.2 we present the general framework, and in section 10.3 we formulate in terms of higher-order logic programs the three central theorems for the classification of optimisation problems and for the derivation of the respective algorithms. In sections 10.4, 10.5 and 10.6 we show how the theorems can be applied to the problems mentioned above. The full code for the three examples, written in HiLog, can be found in the appendix B.

10.2 Algebraic approach to optimisation problems

The goal behind the algebraic approach to optimisation problems is to provide a general tool for both the analysis and the derivation of the appropriate efficient algorithm. The starting point for both these tasks is a uniform specification of optimisation problems. In this section, we first present this specification, and then we prepare the ground for the analysis and the derivation that will be presented in the following section.

In order to make the theorems as general as possible, we use higher-order predicates. In many higher-order logic programming languages, these predicates would have to be defined in terms of `call/n`, which applies a given predicate to the rest of the argument list. In our embedding, however, we have direct access to higher-order functions such as *map* and *fold*. Also, for the sake of simplicity, when we need to return a collection of answers to a given predicate rather than the individual answers, we define a function *bagof*, corresponding to the standard logic programming predicate `bagof/3`. Also, in the examples in this chapter we will use interchangeably the notation $cons(x, xs)$ and $[x|xs]$, depending on the context.

Optimisation problems ask for the best solution among all the solutions to a particular problem. This specification is most naturally formulated as a composition of two relations: first generate all the possible solutions to the problem, and then test this collection of answers to find the best answers among them.

Which answers are "best" depends on the particular ordering of solutions which will inevitably vary from problem to problem, such as maximising the value, minimising the delay, etc. To abstract from such particulars, we write r to denote the ordering of the solutions, and use a higher-order predicate $\mathit{best}(r, \mathit{bag}, \mathit{out})$ that is true if out contains the r-optimal solutions in the collection bag. Given a predicate $\mathit{bagof}(x, p(x), \mathit{bag})$ that collects in bag all the x's satisfying $p(x)$, the specification can be implemented as:

$$
\begin{aligned}
&\mathit{optimal}\; r\; (\mathit{in},\ \mathit{out}) = \\
&\qquad (\exists x, \mathit{bag}.\ \mathit{bagof}\,(x,\ \mathit{solution}(\mathit{in}, x),\ \mathit{bag})\ \& \\
&\qquad\qquad\qquad \mathit{best}\ (r,\ \mathit{bag},\ \mathit{out})).
\end{aligned}
$$

The relation r must be a preorder, i.e. a reflexive and transitive relation. However, some elements of bag may not be related by it, so it need not be a connected preorder. We say that a solution a is better then b with respect to r if and only if the predicate $r(a, b)$ holds, and the predicate $\mathit{best}(r, x, c)$ holds when r happens to be connected and c matches *all* the optimal elements in x with respect to r:

$$
\begin{aligned}
&\mathit{best}\ (r, x, c) = \\
&\qquad x \doteq [c] \\
&\quad \parallel (\exists a, b, y.\ x \doteq [a|[b|y]]\ \&\ r(a, b)\ \&\ \mathit{best}\ (r,\ [a|y],\ c)) \\
&\quad \parallel (\exists a, b, y.\ x \doteq [a|[b|y]]\ \&\ r(b, a)\ \&\ \mathit{best}\ (r,\ [b|y],\ c)).
\end{aligned}
$$

Further, we observe that most of the common optimisation problems can be formulated in terms of some initial data-types, such as lists or trees, and that the computation of individual solutions performed by $\mathit{solution}(\mathit{in}, x)$ can be expressed in terms of higher-order relations over these initial data-types, such as *fold* or *unfold*. We simplify the presentation by expressing all the results in terms of lists, which suffices for the purposes of our three examples, although the more general setting of [18] includes any initial data-types.

The *fold* relation for lists collapses an input list to a value according to a given relation p, while *unfold* constructs from an input value a list according to the relation p. They are converses of each other; declaratively, we do not need two separate definitions as we do in functional programming languages, as we could simply reverse the roles of two arguments in to produce a list rather than consume a list. Operationally, however, we need to reverse the order of the two premises so that Prolog and other systems with an unfair selection rule can solve the subgoals in the correct order. Still, in the relational setting of logic programming and of our embedding their definitions are almost identical, the only difference being ordering of the two literals in the second clause:

$$\begin{aligned}
&\mathit{fold}\ (p, e)\ (l, \mathit{res}) = \\
&\qquad (l \doteq \mathit{nil}\ \&\ e \doteq \mathit{res}) \\
&\quad\ \| \ (\exists x, \mathit{xs}, r.\ l \doteq [x|\mathit{xs}]\ \&\ \mathit{fold}\ (p, e)\ (\mathit{xs}, r)\ \&\ p(x, r, \mathit{res})),
\end{aligned}$$

$$\begin{aligned}
&\mathit{unfold}\ (p, e)\ (\mathit{res}, l) = \\
&\qquad (l \doteq \mathit{nil}\ \&\ e \doteq \mathit{res}) \\
&\quad\ \| \ (\exists x, \mathit{xs}, r.\ l \doteq [x|\mathit{xs}]\ \&\ p(x, r, \mathit{res})\ \&\ \mathit{unfold}\ (p, e)\ (\mathit{xs}, r)).
\end{aligned}$$

In case of *fold* it is expected that the input parameter will be the third and the output will be the fourth argument, while for *unfold* their positions will be swapped. For example, $\mathit{fold}\ (\mathit{add},\ 0)\ ([2, 7, 8], x)$ will be true for $\{x/17\}$, while $\mathit{unfold}\ (\mathit{add},\ 0)\ (y, 17)$ will return $\{y/[2, 7, 8]\}$ as one of its many answers, for an appropriate definition of the predicate *add*.

As for an application of *fold* in a computation of a particular *solution* in *bagof*, it is the natural predicate to use whenever we are given a list and need to decompose it, say, in order to find an optimal way to select some elements from it. For example, in the knapsack problem, if *in* is a list of items and the task is to select the most valuable sublist of it within a given knapsack capacity, a solution *out* can be computed by a *fold* which uses the term *emptysack* as the initial input to the computation and the relation *consumeone* to consume the consequent elements of the *in* list:

$$\begin{aligned}
&\mathit{solution}\ (\mathit{in}, \mathit{out}) = \\
&\qquad \mathit{fold}\ (\mathit{consumeone},\ \mathit{emptysack})\ (\mathit{in},\ \mathit{out}).
\end{aligned}$$

Alternatively, whenever we need to compose the solutions from some seeds, as for example in the string edit problem where we are looking at sequences of editing operations between two strings, we will use *unfold* to implement the predicate *solution*. The solutions can be computed by an *unfold* which uses the term *emptyedit* as the initial edit empty sequence and the relation *addone* to produce the consequent edit instructions in the list *out*. Predicate *addone* builds the list *out* from its previous result and the seed *in*, which simply contains the two strings to be edited:

$$\begin{aligned} &\mathit{solution}\ (\mathit{in}, \mathit{out}) = \\ &\qquad \mathit{unfold}\ (\mathit{addone},\ \mathit{emptyedit})\ (\mathit{out},\ \mathit{in}). \end{aligned}$$

Finally, the notion of refinement in the next section is as follows: the derived predicate *fast* refines the specification *spec* if for all inputs *in*, we have:

$$\mathit{fast}\ (\mathit{in}, \mathit{out}) \Rightarrow \mathit{spec}\ (\mathit{in}, \mathit{out}).$$

With these preliminaries out of the way, we are now ready to express the three central optimisational theorems regarding the applicability and the derivation of dynamic, greedy and thinning algorithms. The proofs of these theorems are omitted here, but can be found in a more general form in [18]. These proofs, as the ones in the previous chapter, are based on equational reasoning about the algebraic properties of the relations involved, which is why we refer to this as an "algebraic" approach. As already seen in the previous chapter, in this style of program calculation, theorems about higher-order functions like *fold* play a central role.

10.3 The three theorems

Before we start introducing the main theorems of this chapter, we need the notion of *monotonicity* of predicates. The predicate $p((a, x), \mathit{newx})$ constructs a solution *newx* by incrementing the partial solution x by a. We say that p is monotonic, or order preserving, on a preorder r if, for any arguments x_i and x_j, if x_i is better than x_j with respect to r, no matter how p extends the inferior solution to some newx_j, we can always find at least one way to extend by p the superior solution so that the resulting newx_i is better than newx_j. If this is the case, we know that it is safe to throw the inferior solutions away and only extend the best ones. Formally:

$$\begin{aligned} & r\ (x_i, x_j) \wedge p\ ((a, x_j), newx_j)\ \Rightarrow \\ & \qquad (\exists newx_i.\ p\ ((a, x_i), newx_i) \wedge r\ (newx_i, newx_j)). \end{aligned} \tag{10.1}$$

The predicate p can be non-deterministic. When p happens to be deterministic, in particular, when it is the list constructor *cons* defined by $cons((a, x), [a|x])$, the condition (10.1) simplifies to:

$$r(x_i, x_j)\ \Rightarrow\ r([a|x_i], [a|x_j]). \tag{10.2}$$

The **Dynamic Theorem** is applicable to problems in which it is natural for the partial results to be computed by an *unfold*, i.e. by constructing candidate answers from a seed. The theorem also guides the first part of the derivation of the dynamic program, but the programmer is still left to his own ingenuity to derive the appropriate tabling scheme. If (10.2) holds for some given r, the specification:

$$\begin{aligned} & d\ (in, out) = \\ & \qquad (\exists x, bag.\ bagof\ (x,\ unfold\ (step, base)\ (x, in),\ bag)\ \& \\ & \qquad\qquad best\ (r, bag, out)), \end{aligned}$$

can, for any given *step* and *base*, be refined to:

$$\begin{aligned} & d\ (in, out) = \\ & \qquad out \doteq nil \\ & \quad \parallel (\exists x, bag, bag1.\ bagof\ ((a, x),\ step((a, x), in),\ bag)\ \& \\ & \qquad\qquad consmap\ (d,\ bag,\ bag1)\ \& \\ & \qquad\qquad best\ (r, bag1, out)), \end{aligned}$$

where the auxiliary predicate *consmap* is defined as:

$$\begin{aligned} & consmap\ (p, l1, l2) = \\ & \qquad (l1 \doteq nil\ \&\ l2 \doteq nil) \\ & \quad \parallel (\exists a, x, y, newx, newy.\ l1 \doteq [(a, x)|y]\ \&\ l2 \doteq [[a|newx]\ |\ newy]\ \& \\ & \qquad\qquad p\ (x, newx)\ \&\ consmap\ (p, y, newy)). \end{aligned}$$

The derived program is better since it filters the input through *best* at each level of recursion, rather than maintain all the unprofitable solutions and choosing the optimal ones at the very end, as the specification does. The second program for d describes a recursive scheme in which first *step((a,x),in)* is applied to *in* in all possible ways. These results $(a_1, x_1), \ldots, (a_n, x_n)$ are collected by *bagof* in the bag *bag*. Then the recursive calls to d are applied by *consmap* to each of the new seeds $x_1, \ldots, x_n$. These calls to d generate the lists $newx_1, \ldots, newx_n$, which are consequently "consed" with $a_1, \ldots, a_n$, resulting in the list *bag1* of solution lists.

The monotonic condition (10.2) is actually the *Principle of Optimality* stated formally for lists. Here is the reason why it is needed in this derivation. Each seed x_i may generate many alternatives for $newx_i$, and all of these will be contained in $bag1$. However, since each of these $newx_i$ is consed with the same a_i, according to (10.2), the best x_i's will always lead to the best solutions. That is why we only need to consider the result of *best* for each decomposition of subproblems.

If the set of decompositions associated with each subproblem is overlapping, a naive evaluation of the derived program will involve much repeating work. However, several declarative languages provide a built-in tabulation facility where the solutions to subproblems can be implicitly recorded and retrieved for subsequent use, and with such evaluation the derived program would have polynomial rather than exponential time complexity.

Notice that such implicit tabulation would not reduce the time complexity of the original specification in the same way; this is because the specification produces a bag with all the solutions and if there are exponentially many of them, the *best* predicate must take exponential time.

Similarly, the **Greedy Theorem** is used to check whether one can solve an optimisational problem by a greedy algorithm, and for the positive instances to derive this algorithm from the specification. While the dynamic theorem allows us to improve the efficiency of the specification by only considering the best partial solutions for *each* decomposition, the greedy theorem goes much further: the derived greedy program arrives to the optimal solution by only computing the best partial solutions of *one*, best, decomposition at each recursion level. Obviously, the conditions required by this theorem must be rather strong, since we need an additional ordering which will provide us with a hint exactly which decomposition to use, based on local information.

There are actually two greedy theorems, one for *fold* and one for *unfold*, and here we choose to present only the one relevant for our examples, based on *unfold*. If the monotonicity condition (10.2) is satisfied, and if we can find a preorder q defined on pairs which represent problem decompositions, such that:

$$\begin{aligned} & q((a_i, in_i), (a_j, in_j)) \wedge \mathit{unfold}\ (\mathit{step}, \mathit{base})\ (\mathit{out}_j, in_j) \;\Rightarrow \\ & \quad (\exists \mathit{out}_i.\ \mathit{unfold}\ (\mathit{step}, \mathit{base})\ (\mathit{out}_i, in_i) \wedge r\ ([a_i|\mathit{out}_i], [a_j|\mathit{out}_j])). \end{aligned} \tag{10.3}$$

Then the following program segment:

```
$g\ (in, out) =$
    $(\exists x, bag.\ bagof\ (x, unfold(step, base)\ (x, in), bag)\ \&$
            $best\ (r, bag, out)),$
```

can be refined to:

```
$g\ (in, out) =$
    $out \doteq nil$
  $\|\ (\exists x, a, a1, bag1.\ out \doteq [a1|newx1]\ \&$
            $bagof\ ((a, x), step((a, x), in), bag)\ \&$
            $best\ (q, bag, (a1, x1))\ \&\ g\ (x1, newx1)).$
```

As in the dynamic theorem, the condition (10.2) enables us to consider only the best partial solutions for each decomposition. Even better, only the best decomposition is needed, and the greedy condition (10.3) uses q to chose this decomposition, say, (a_i, in_i). In cases where the input is such that *best* matches more than one element in *bag*, we can refine this program further by replacing *best* with a predicate *onebest* which is matched only once.

Note that q is an ordering on decompositions, while r is an ordering on results. If we have an ordering q such that for any pair of decompositions, say (a_i, in_i) and (a_j, in_j), if (a_i, in_i) is preferred to (a_j, in_j) by q, then we know the following: for any possible outcome out_j of the inferior decomposition, we can find at least one outcome out_i of the superior decomposition, such that the total result $[a_i|out_i]$ will be better than $[a_j|out_j]$, according to the ordering r. This is why, in the derived program, we can after each *step* safely chose the best decomposition $(a1, x1)$ according to q, and recursively apply g only to this optimal decomposition without considering others.

On the continuum of the optimisation problems that satisfy the principle of optimality, on one extreme we find the problems requiring dynamic programming, and on the other problems that can be solved by greedy algorithms.

Some problems fall between these two extremes of all and one: in [18] there is a refined dynamic theorem with an additional condition which allows us to move away from the "all" extreme to "some", where an additional preorder is used discard some of the decompositions, and pursue only those which incorporate all the potentially profitable solutions. This approach is called "thinning".

We have previously mentioned that two distinct forms of greedy theorem exists, one for problems defined in terms of *fold* and one for those defined in terms of *unfold*. The dynamic theorem, however, only works for problems specified with *unfold*. Problems that satisfy the principle of optimality and require a specification in terms of *fold* can be solved by the **Thinning Theorem**. This theorem specifies a monotonicity condition that can be used to discard some of the unprofitable decompositions in the derived program.

Assume that we have preorders r and q respectively on solutions and on problem decompositions, with the following specifications. The preorder q is a sub-relation of preorder r, meaning $q(x, y) \Rightarrow r(x, y)$, and it is not necessarily connected. The predicate *step* is monotonic (as in (10.1)) on the converse of q, which we denote by q°.

Furthermore, assume that we have predicates *thin* and *pow*, with the following specifications. The predicate $thin(r, xs, ys)$ holds if ys is a subset of xs and $\forall x \in xs.\ \exists y \in ys.\ r(y, x)$, that is, for each solution in xs there is a better solution in ys. Finally, $pow(p, (a, xs), y)$ holds if $\exists x \in xs.\ p((a, x), y)$, that is, if p applied to some arbitrary element of x of xs yields y. Then the program:

$$\begin{aligned}
&t\ (\mathit{in}, \mathit{out}) = \\
&\qquad (\exists x, \mathit{bag}.\ \mathit{bagof}\,(x,\ \mathit{fold}\ (\mathit{step},\ \mathit{base})\ (\mathit{in}, x), \mathit{bag})\ \& \\
&\qquad\qquad\qquad \mathit{best}\ (r, \mathit{bag}, \mathit{out})),
\end{aligned}$$

can be refined to:

$$\begin{aligned}
&t\ (\mathit{in}, \mathit{out}) = \\
&\qquad (\exists \mathit{bag}.\ \mathit{fold}\ (\mathit{tstep},\ [\mathit{base}])\ (\mathit{in}, \mathit{bag})\ \& \\
&\qquad\qquad\qquad \mathit{best}\ (r, \mathit{bag}, \mathit{out})),
\end{aligned}$$

where the auxiliary predicate *tstep* is defined as:

$$\begin{aligned}
&\mathit{tstep}\ ((a, \mathit{as}), \mathit{ys}) = \\
&\qquad (\exists x, \mathit{xs}.\ \mathit{bagof}\,(x, \mathit{pow}(\mathit{step}, (a, \mathit{as}), x),\ \mathit{xs})\ \& \\
&\qquad\qquad\qquad \mathit{thin}\ (q, \mathit{xs}, \mathit{ys})).
\end{aligned}$$

The motivation here is that we promote *bagof* in *fold*, so that we can thin it at each recursion stage. At each stage, we use a *pow* operator to apply *step* in all possible ways to the bag of partial solutions, then collect the results. The role of *thin* is to use the preorder q to shrink the size of the bag of solutions.

The specification of $\mathit{thin}(q, x, y)$ in effect means that if an element in x is worse than some other element with respect to q, then by monotonicity we need not keep it in y. Obviously, this specification allows *thin* to return its input unshrank, and in that case the derived program is as inefficient as the specification. To gain from this refinement without relying on tabling, we need to find a q that removes a considerable portion of the bag of the partial solutions. On the other hand, if we manage to find a q so strong that it is a connected preorder, we can simply keep the best partial solution at each stage, which would correspond to a greedy algorithm.

Finally, the premises for the three theorems are actually stronger than they need to be, so both (10.2) and (10.3) can be additionally restricted. We shall prefer to use these weaker conditions in section 10.6, so we define them as well. A weaker form of (10.2) requires that x_i and x_j are composed from the same seed *in*:

$$\begin{aligned}&\exists in.\ (unfold\ (step, base)\ (x_i, in) \wedge unfold\ (step, base)\ (x_j, in)) \wedge r\ (x_i, x_j)\\ &\qquad \Rightarrow\ r\ ([a|x_i], [a|x_j]) \qquad (10.4)\end{aligned}$$

and, similarly, a weaker form of (10.3) requires that both (a_i, in_i) and (a_j, in_j) are decompositions of the same input *in*:

$$\begin{aligned}&\exists in.\ (step\ ((a_i, in_i), in) \wedge step\ ((a_j, in_j), in))\\ &\wedge\ q\ ((a_i, in_i), (a_j, in_j)) \wedge unfold\ (step, base)\ (out_j, in_j) \qquad (10.5)\\ &\qquad \Rightarrow\ (\exists out_i.\ unfold\ (step, base)\ (out_i, in_i) \wedge r\ ([a_i|out_i], [a_j|out_j]))\end{aligned}$$

Now we go on to apply these three theorems to three classic optimisation problems: dynamic theorem to the string edit problem, thinning theorem to 1/0 knapsack problem, and greedy theorem to the minimal lateness problem.

10.4 Dynamic programming example: string edit

Given two strings x and y, the string edit problem asks for the minimal sequence of editing operations required to transform x into y. The choice of the editing operations varies in different formulations of this problem, and we choose the simplest possible set: *insert* a character into x, *delete* a character from x, and *copy* a character in x, that is, retain it. These three operations contain enough information to construct both strings from scratch, if one interprets *copy* a as "append a to both strings", *insert* a as "append a to the right string" and *delete* a as "append a to the left string".

We choose to represent the strings as lists of characters and the edit sequence as a list containing pairs of (*op*, *char*), where *op* is one of *ins del* or *cpy*. Each operation costs one unit, so the optimal edit sequence is one with a minimal length. Since there might be more than one such solution, we choose the first.

The string edit problem constructs the lists of editing instructions *out* by means of an *unfold* from the seed $(s1, s2)$, containing the two input strings:

$$\begin{aligned}
&\mathit{edit}\ ((s1, s2), \mathit{out}) = \\
&\qquad (\exists x, \mathit{bag}.\ \mathit{bagof}\ (x,\ \mathit{unfold}\ (\mathit{step}, (\mathit{nil}, \mathit{nil}))\ (x, (s1, s2)),\ \mathit{bag})\ \& \\
&\qquad\qquad\qquad \mathit{best}\ (\mathit{lleq},\ \mathit{bag},\ \mathit{out})),
\end{aligned}$$

where the predicate *step* applies one editing instruction to the pair of input strings, and the predicate *lleq* compares the lengths of two edit sequences:

$$\begin{aligned}
&\mathit{step}\ (\mathit{in}, \mathit{out}) = \\
&\qquad (\exists x, y, a, b.\ \mathit{in} \doteq ((\mathit{cpy}, a), (x, y))\ \&\ \mathit{out} \doteq ([a|x], [a|y]) \\
&\qquad\qquad\qquad \|\ \mathit{in} \doteq ((\mathit{del}, a), (x, \mathit{nil}))\ \&\ \mathit{out} \doteq ([a|x], \mathit{nil}) \\
&\qquad\qquad\qquad \|\ \mathit{in} \doteq ((\mathit{del}, a), (x, [b|y]))\ \&\ \mathit{out} \doteq ([a|x], [b|y]) \\
&\qquad\qquad\qquad \|\ \mathit{in} \doteq ((\mathit{ins}, b), (\mathit{nil}, y))\ \&\ \mathit{out} \doteq (\mathit{nil}, [b|y]) \\
&\qquad\qquad\qquad \|\ \mathit{in} \doteq ((\mathit{ins}, b), ([a|x], y))\ \&\ \mathit{out} \doteq ([a|x], [b|y])), \\
&\mathit{lleq}(x, y) = \\
&\qquad (\exists m, n.\ \mathit{length}(x, m)\ \&\ \mathit{length}(y, n)\ \&\ m =< n).
\end{aligned}$$

Obviously *lleq* is monotonic on *cons*, since consing an element to two sequences preserves the length comparison between them. So (10.2) holds and we can apply the dynamic theorem, which results in the following program:

$$\begin{aligned}
&\mathit{edit2}\ ((s1, s2),\ \mathit{out}) = \\
&\qquad (s1 \doteq \mathit{nil}\ \&\ s2 \doteq \mathit{nil}\ \&\ \mathit{out} \doteq \mathit{nil}) \\
&\qquad \|\ (\exists x, \mathit{bag}, \mathit{bag1}.\ \mathit{bagof}\ (x,\ \mathit{step}(x, (s1, s2)),\ \mathit{bag})\ \& \\
&\qquad\qquad\qquad\qquad \mathit{consmap}\ (\mathit{edit2},\ \mathit{bag},\ \mathit{bag1})\ \& \\
&\qquad\qquad\qquad\qquad \mathit{best}\ (\mathit{lleq},\ \mathit{bag1},\ \mathit{out})).
\end{aligned}$$

Using XSB [119], or some other tabling Prolog system, *edit*2 can be automatically tabled and the execution complexity becomes polynomial[2] because there are only $m * n$ different subproblems, where m and n are the lengths of the two sequences. The complexity of the specification *edit*, on the other side, would remain exponential even after tabling.

[2]Since we are tabling strings, lookups require string comparisons and are potentially expensive, but these strings are prefixes of a fixed string, so we can use integers as keys.

10.5 Thinning example: 1/0 knapsack

Given n items, each of weight w_i and of value v_i, and a knapsack of capacity K, the goal is to find the subset of items with maximal total value whose total weight does not exceed K. The naive implementation of this problem constructs all subsets of items within weight limit and returns the subset with the greatest value; the size of the powerset of a set of n elements is 2^n, so the complexity of this algorithm is exponential in the number of items:

$$\begin{aligned}
&\mathit{knapsack}\ (w, \mathit{in}, \mathit{out}) = \\
&\qquad (\exists x, \mathit{bag}.\ \mathit{bagof}\ (x,\ \mathit{fold}\ (\mathit{step}(w), (\mathit{nil}, 0, 0))\ (\mathit{in}, x),\ \mathit{bag})\ \& \\
&\qquad\qquad\qquad \mathit{best}\ (\mathit{vgeq},\ \mathit{bag},\ \mathit{out})).
\end{aligned}$$

The predicate $\mathit{knapsack}(w, x, y)$ holds if y is the optimal way to select items from the list x within weight w. We adopt a *fold* to capture the process of examining all the items one by one. In each *step* we have two choices: to ignore this item, or to add it to the bag if the total weight does not exceed our limit:

$$\begin{aligned}
&\mathit{step}\ (w, (a, x), y) = \\
&\qquad x \doteq y \\
&\quad\ \| \ (\mathit{addone}(a, x, y)\ \&\ \mathit{within}(w, y)).
\end{aligned}$$

The explicit use of nondeterminism here is suggestive and we consider that the relations of a logic programming language give us in this case a clear notational advantage over conventional functional languages. The predicate $\mathit{addone}(a, x, y)$ holds if y is a partial solution gotten by adding the item a to the partial solution x. The predicate $\mathit{within}(w, y)$ holds if the total weight of all the items in list y is within the weight limit w.

The auxiliary *vgeq* simply compares the values of two collections:

$$\mathit{vgeq}\ (a, b) = (\exists \mathit{va}, \mathit{vb}.\ \mathit{value}(a, \mathit{va})\ \&\ \mathit{value}(b, \mathit{vb})\ \&\ \mathit{va} >= \mathit{vb}).$$

Since the specification of *knapsack* is expressed in terms of *fold*, we can try to apply either the thinning or the greedy theorem. Optimally, we would like to try to use the greedy theorem, since it results in the simplest and the most efficient algorithm. Unfortunately, *step* is not monotonic on *vgeq*. We cannot give up a selection of items just because it is less valuable than another selection, because the *step* might not be able to add it to the partially filled knapsack due to overflow.

However, we can identify a sub-relation q of *value* such that *step* is monotonic on a converse of q. If a selection of items is not only less valuable, but also heavier than another selection, then this choice of items is definitely not leading to the optimal solution. Given an auxiliary predicate *wleq* which is true if the weight of its first argument is less than or equal to that of its second argument, we define q simply as the conjunction:

$$q\ (a, b) = vgeq\ (a, b)\ \&\ wleq\ (a, b).$$

The proof that *step* is monotonic on q° follows directly from the definition of *step*. If $q^\circ(a, b)$ holds, then a has smaller value and greater weight than b. Given a partial solution x, we can always find a way to use *step* to extend it with these two items so that the resulting solutions, say xa and xb, are related by $q^\circ(xa, xb)$, i.e. so that the solution resulting from the less valuable and heavier item will also be less valuable and heavier.

Thus, we know that we can apply the thinning theorem. Actually, a special form of this theorem, known as the *Binary Thinning Theorem*, can be applied in this case, because the definition of *step* has only two alternatives. This theorem states that under the conditions described above, we can derive the following program from our knapsack specification:

$$\begin{aligned}
&knapsack2\ (w, in, out) = \\
&\qquad (\exists list.\ fold\ (step3(w), [(nil, 0, 0)])\ (in, list)\ \& \\
&\qquad\qquad head\ (list,\ out)), \\
&step3\ (w, (a, x), ys) = \\
&\qquad (\exists y, bag1, bag2.\ bagof\ (y, pow(step1(w), (a, x), y), bag1)\ \& \\
&\qquad\qquad bagof\ (y, pow(step2(w), (a, x), y), bag2)\ \& \\
&\qquad\qquad merge\ (vgeq, bag1, bag2, bag)\ \& \\
&\qquad\qquad thin\ (q, bag, ys)).
\end{aligned}$$

Instead of referring to a theorem not presented here, we could have used simple algebraic calculations to derive this program from the code resulting from the original thinning theorem in a few steps. The sketch of the main steps in this proof is as follows. First, *best* r is refined by a composition of predicates *sort* r° and *head*, since taking the first element of the list sorted in reverse order of r gives the optimal element. Then, the fusion theorem is used to push *sort* into the *fold*, and the conditions of the fusion theorem are used to calculate the composition of predicates *bagof*, *merge* and *thin* used in the code above.

Let n be the number of items, and w their total weight. As discussed earlier, the time complexity of the specification is $O(2^n)$. The derived program computes *step*3 n times, once for each item in the input list. The size of the bag in *bagof* is in this case bounded by the the total weight w because for each weight the bag contains at most one element, and *thin* and *merge* are easily implemented such that they are linear in the length of the input lists, so the time complexity of the derived program is $O(n * w)$.

This problem is traditionally solved by tabling, where one builds a table containing a row for each item and a column for each weight, up to the total sum of the weights of all items. In each entry the best possible value within the given subset of items and the given weight is recorded. Since the number of entries is the product of the number of items and the total weight, the running time becomes polynomial, so the complexities of the tabled program and our program are comparable. However, unlike the tabling program, our program also works for non-integer weights and values, but in that case the running time becomes exponential.

10.6 Greedy example: minimal tardiness

The minimum tardiness problem is a scheduling problem from Operations Research. Given a bag of jobs, it is required to find some scheduling of it, that is, some permutation of the bag, that minimises the worst penalty incurred if the scheduled jobs are not completed in their due time. Each job j is associated with three quantities: the *completion time* $ct(j, c)$, determining how long it takes to complete the job; the *due time* $dt(j, d)$, determining the latest time before which the job must be completed; and a *weighting* $wt(j, w)$, measuring the importance of the job.

Given predicates $\mathit{bagify}(x, in)$, which holds if x is some permutation of the bag in, and $\mathit{costleq}(a, b)$, which holds if the schedule a has a cost less than or equal to the schedule b, the scheduling problem *sche* can be specified as:

$$\begin{aligned}
&\mathit{sche}\ (in, out) = \\
&\qquad (\exists x, bag.\ \mathit{bagof}\ (x, \mathit{bagify}(x, in), bag)\ \& \\
&\qquad\qquad\qquad \mathit{best}\ (\mathit{costleq}, bag, out)).
\end{aligned}$$

We can implement *bagify* using *unfold* with the auxiliary predicate *bcons*. In one direction, *bcons* adds an item a to a bag x; used in the reverse way, it nondeterministically picks an arbitrary item from the bag and pairs it with the rest of the bag. With *unfold*, it can be used to generate all the permutations for a bag. The predicate *costleq* is as expected:

$$\mathit{bagify}\ (y, x) = \mathit{unfold}\ (\mathit{bcons}, \mathit{nil})\ (y, x),$$

$$\begin{aligned}
&\mathit{bcons}\ (x, ys) = \\
&\qquad (\exists a, z.\ x \doteq (a, z)\ \&\ ys \doteq [a|z]) \\
&\quad \parallel (\exists a, b, y, z.\ x \doteq (b, [a|z])\ \&\ ys \doteq [a|y]\ \& \\
&\qquad\qquad\qquad \mathit{bcons}\ ((b, z), y)),
\end{aligned}$$

$$\begin{aligned}
&\mathit{costleq}\ (x, y) = \\
&\qquad (\exists cx, cy.\ \mathit{cost}(x, cx)\ \&\ \mathit{cost}(y, cy)\ \&\ cx =< cy).
\end{aligned}$$

The cost of a scheduling is the maximum penalty of any scheduled job. The relation $\mathit{penalty}((j, x), p)$ below denotes that the penalty p is incurred when the job j is performed after some jobs x. Notice that in this example schedules should be read backwards, i.e. the last job is written at the head of the list. If a job is completed before its due time, according to the definition below its assigned penalty is negative, but since we only need to be concerned with the maximum penalty, we choose to ignore negative penalties in the definition of *cost*. To this end we use the predicate *bmax* which simply relates the maximum of its first and second arguments to the third. Given a predicate *totaltime* which calculates the time taken to complete all the jobs in x by summing up their completion time, the definitions of *penalty* and *cost* are:

$$\begin{aligned}
&\mathit{penalty}\ ((j,x),p) = \\
&\qquad (\exists c, w, d, tt.\ ct(j,c)\ \&\ wt(j,w)\ \&\ dt(j,d)\ \& \\
&\qquad\qquad\qquad \mathit{totaltime}(x,tt)\ \&\ p\ \mathit{is}\ (tt+c-d)*w),
\end{aligned}$$

$$\begin{aligned}
&\mathit{cost}\ (x,c) = \\
&\qquad (x \doteq \mathit{nil}\ \&\ c \doteq 0) \\
&\quad \parallel (\exists j, z, c1, c2.\ x \doteq [j|z]\ \&\ \mathit{penalty}((j,z),c1)\ \& \\
&\qquad\qquad\qquad \mathit{cost}(z,c2)\ \&\ \mathit{bmax}(c1,c2,c)).
\end{aligned}$$

As the number of permutations of a list is exponential in its length, the above specification of *sche* takes exponential time to run. Fortunately, the two conditions of the greedy theorem hold for this specification, so we can derive a greedy algorithm for this problem. We give an informal argument for their validity below, and a formal, calculational proof can be found in [18].

The monotonicity condition states that if the cost of a scheduling x is less than or equal to the cost of scheduling y, attaching job j to both of them does not change the ordering. In this example we find the weaker version (10.4) easier to prove. Given a job j, schedules $s1$, $s2$, and a bag b, we claim that:

$$\begin{aligned}
&(\exists b.\ \mathit{bagify}\ (s1,b) \wedge \mathit{bagify}\ (s2,b)) \wedge \mathit{costleq}\ (s1,s2) \\
&\qquad \Rightarrow\ \ \mathit{costleq}\ ([j|s1],[j|s2]).
\end{aligned}$$

The premise $(\exists b.\ \mathit{bagify}(s1,b) \wedge \mathit{bagify}(s2,b))$, also called the context, says that both $s1$ and $s2$ are schedulings of the same bag of jobs b. For any permutation of a bag of jobs b, i.e. any schedule resulting form b, the total completion time must be the same. Therefore also adding the same job to both schedules results in a same completion time. But the penalty of $[j|s1]$ only depends on the weight of j and the total completion time of $s1$, so the penalties for doing j after $s1$ and $s2$ are the same. Since the cost of a scheduling is defined to be the maximum penalty, the monotonicity condition trivially holds.

Proving the greedy condition is trickier, and again we chose to prove the weaker version (10.5). We need to invent an ordering which will help us to choose the best job to pick in each stage. Formally, we need to find an ordering q such that, for jobs $j1$, $j2$, bags $b1$, $b2$, b and schedules $s1$, $s2$, we can prove:

$$\begin{aligned}&(\exists b.\ \mathit{bcons}((j1,b1),b) \wedge \mathit{bcons}((j2,b2),b))\\ &\wedge\ q((j1,b1),(j2,b2)) \wedge \mathit{bagify}(s2,b2)\\ &\qquad\Rightarrow\ (\exists s1.\ \mathit{bagify}(s1,b1) \wedge \mathit{costleq}([j1|s1],[j2|s2])). \qquad (*)\end{aligned}$$

We claim that the ordering *penaltyleq* is the right choice for q, that is, that in each step we only need to follow the decomposition with the least penalty job:

$$\begin{aligned}&\mathit{penaltyleq}\ (x,y) =\\ &\qquad(\exists px,py.\ \mathit{penalty}(x,px)\ \&\ \mathit{penalty}(y,py)\ \&\ px =< py).\end{aligned}$$

We refer to the two parts in the antecedent of the greedy condition $(*)$ as the context and the premise, respectively. The context of the greedy condition states that $(j1, b1)$ and $(j2, b2)$ are decompositions of the same bag b. Further, the premise $\mathit{penaltyleq}((j1, b1),(j2, b2))$ means that the job $j1$ after any scheduling of $b1$ incurs less penalty than the job $j2$ after any scheduling of $b2$.

If the context and the premise hold, then, we argue, for any scheduling $s2$ of the bag $b2$, there must exist some way to construct a schedule $s1$ out of the bag $b1$, such that the total schedule $[j1|s1]$ has lower cost than the schedule $[j2|s2]$.

The argument is as follows. Remember that the cost of $[j1|s1]$ is the maximum of the cost of $s1$ and the penalty of $j1$. Regarding the cost of $s1$, notice first that inserting a job into a schedule either keeps the total schedule cost the same or increases it. Because both $[j1|s1]$ and $[j2|s2]$ are schedulings of the same bag b, we can chose the schedule $s1$ to be the same as schedule

$[j2|s2]$ without the job $j1$; then the cost of $s1$ must be less than or equal to the cost of $[j2|s2]$. Regarding the penalty of $j1$ after $s1$, by the premise we know that it is less than or equal to the penalty of $j2$ after $s2$.

And then, through a direct application of the greedy theorem, we derive:

$$\begin{aligned}
&\mathit{sche2}\ (b, y) = \\
&\qquad b \doteq \mathit{nil}\,\&\ y \doteq \mathit{nil} \\
&\qquad \|\ (\exists x, \mathit{bag}, j1, s1, b1.\ y \doteq [j1|s1]\ \&\ \mathit{bagof}\ (x, \mathit{bcons}(x, b), \mathit{bag})\ \& \\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathit{best}\ (\mathit{penaltyleq}, \mathit{bag}, (j1, b1))\ \& \\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathit{sche2}(b1, s1).
\end{aligned}$$

The complexity of this is cubic in n, since there are totally n recursive calls to *sche*2, in each call we need to examine among a linear number of decompositions to pick the one with least penalty, and the calculation of penalty also takes linear time (to sum up the completion time). We could have refined the data structure such that computing the penalty takes constant time, but we keep the code in this form to emphasise the structure of the program described in the theorem.

For completeness, we finish this chapter with a few words about dynamic programming in logic programming: traditional tabulation methods have been successfully used to solve dynamic programming problems in logic programming: examples of tabular logic programming systems include XSB [119], DyALog [140], and B-Prolog [148]. There is an alternative approach, advocated by Clocksin in [31], where recomputation is avoided by using data-flow analysis at compile time; this approach is implemented for logic programming in [22, 41]. However, they do not focus on algorithms; we do.

Finally, the three theorems in this chapter are presented in specialised forms for lists. The more general form can be found in [18], where an optimisation problem is represented as *fold* and *unfold* for any initial data-type. However, not all dynamic programming or greedy algorithms can be expressed in terms of *fold* and *unfold*. Curtis [36] generalises the model further to cover most dynamic programming or greedy problems.

Chapter 11

Evaluation and Future Work

In the final chapter we summarise the work presented in this thesis, present and discuss some areas of related work and outline possible directions for further research.

11.1 Summary of the thesis and conclusions

The work presented in this thesis is a part of a research program which explores the application of algebra to *program transformation and calculation* for declarative languages. In this case, the general conceptual advantage of using an algebra, instead of using specific language semantics, is that many different programming styles can be specified in the same formalism, so that the approach elucidates similarities between the different declarative styles. Also, the techniques presented hold for whole families of programs which are found in each of the declarative styles.

Since the algebraic approach is closely related to the equational style of functional programming, functional programs can be treated directly in this setting, using a flexible range of formal analysis techniques and symbolic simulation. Therefore, most of the work in the area of algebraic program transformation has been focused on functional programming, where the results are plentiful. However, the other main family of declarative languages, logic programming, has not yet been studied in this setting.

The objective of this thesis was to investigate whether the algebraic approach to program transformation and derivation is similarly well-suited to model and analyse logic programs.

In this thesis we have proposed and studied an embedding of pure logic programs into lazy functional ones. This work was not aimed towards an implementation of a new programming language, although a language implementation based on the embedding is conceivable. Rather, this work was directed towards producing and using a theoretical tool (with a simple implementation) for the analysis of different aspects of logic programs. Our three top-level goals were:

- the core implementation of the operators of the embedding,
- the operational analysis of the combinators of the embedding, and
- the application of the embedding to program transformation.

Each of these tasks turned out to be quite tractable in the embedding, in the sense that they could be carried out in a reasonably straightforward and self-explanatory manner. Indeed, this simplicity was the key idea and the main strength of the embedding. The procedural semantics of the embedded program proved to be both transparent and very close to the declarative semantics of the original logic program. The embedding is abstract enough to be flexible: it permits experimenting with different search strategies and higher-order logic programs. It also facilitates reasoning about logic programs and can used to generalise the analysis of programs and program transformation.

Based on this work, our two major conclusions are that:

- algebraic semantics is a suitable semantic framework for logic programming; it facilitates a compositional approach to reasoning about logic programs, and yields several interesting theoretical insights; and
- program transformation based on algebraic laws and higher-order functions is more generic than the traditional methods employed for logic program transformation.

On a pragmatic level, we found algebraic semantics an appealing tool for reasoning about logic programs because it is based on a fairly natural and well-known formalism, and because of its abstract modelling of the computation process and support for equational reasoning.

We also found that the two major practical advantages of our functional embedding were, first, that it made expressible language principles that are normally "outside" the language, such as properties of the predicate operators & and || which are implicit in logic program, and second, that it was simple. We now expand on these results and conclusions, relating them to the relevant chapters.

There have been several earlier research contributions exploring the algebraic model of logic programming. The first one has been the work of Clark on *completion of logic programs* [26], a transformation of a definite logic program into an equational form, where the implicit logic operators that form the logic program are made explicit. The original goal of this work had been to allow negation in clause bodies. Although the completed form of a logic program has been shown to be adequate for reasoning about logic programs, since it preserves the semantics, this equational form had not been, to the best of our knowledge, used for transfer of methodology from functional programming. A variety of executable implementations of logic programming in a functional setting are based on this result, notably [11, 63, 114, 117], but they did not attempt to exploit the full potential of the algebraic framework. Nevertheless, this work also indicated that algebra is indeed well-suited to modelling and analysing logic programs.

For building the embedding, we chose Haskell as a meta-language, in which other languages – in this case several variants of logic programming, including Prolog – were represented and executed. Specifically, the implementation of the embedding extends the standard library of Haskell, and makes extensive use of Haskell's *lazy* and *higher-order* features. Besides giving a great saving in implementation effort, the design of the embedding as an extension of Haskell also meant that the embedding inherited and extended the rich algebra of Haskell, including laws for higher-order functions and many important theorems, including those implied by the theory of [18]. Our experience was that it was rewarding to have a concrete, working prototype to experiment with; and that the proofs of the proposed algebraic properties were greatly simplified by only requiring the standard algebra of functions.

As an alternative to Haskell any polymorphically-typed lazy language and λ-abstractions would do. Our implementation showed that any such functional language contains much of the expressive power of functional logic languages, but to achieve the full power of these languages further extensions (e.g. typed unification) would be needed.

The implementation of the embedding was presented in Chapter 3. It was implemented by translating each logic programming primitive to a function or an operator. The implementation was centred on six functions, *true*, *false* and $\doteq$ for defining primitive predicates, and &, $\|$ and $\exists$ for constructing composite predicates from other predicates. In this manner we could implement the substitutional operators and the search-control operators separately; so this could be seen as a realisation of Kowalski's slogan that programs are logic plus control, bearing in mind that we deal only with the search strategy of the "control part", but not the selection rule. Our implementation is arguably the simplest possible formalisation of different denotational semantics of logic programming and can be thought of as an executable operational semantics for logic programming. The embedding shows that any pure logic program can be quite naturally and directly expressed as an executable function.

The use of lazy lists in the implementation gives rise to a natural implementation of the possibly infinite search-space and the depth-first search strategy of Prolog. We called this the *depth-first model* of logic programming. In Chapter 5 we described how the embedding can be changed to implement breadth-first search by lifting the operations to streams of lists, where each list has one higher cost that the predecessor; we called this the *breadth-first model*. Further, we offered a third, more flexible model based on lists of trees which accommodates *both* search strategies; we called this the *general model* of logic programming. We found that the the changes required to implement each of the different search models were small and simple, and that the polymorphism and higher-order functions of Haskell simplified this task further; this was a substantially more pleasant situation than the one faced by implementors of different search strategies in deep embeddings, such as interpreters based on LD-resolution.

Our deliberately minimalist approach laid bare the algebraic laws of logic programming. In Chapter 4 we have derived a set of algebraic laws serving as a specification of the basic operators of the embedding. These laws were chosen such that they respect the operational behaviour of logic programs as well. For example, the predicate *false* is a left zero for &, but the & operator is strict in its left argument, so *false* is not a right zero. The & operator distributes through $\|$ from the right, but not from the left. Other identities that are satisfied by the connectives of propositional logic are not shared by our operators because in our stream-based implementation, answers are produced in a definite order and with definite multiplicity. This behaviour mirrors the operational behaviour of Prolog, and as it turns out, more generally SLD-resolution with a left-most selection rule under any search strategy.

However, the embedding and the laws also exposed this operational behaviour in a compositional manner, and made it explicit in an equational setting. Therefore we could make it subject to algebraic methods for reasoning about operational properties of programs, search strategies, and program transformation. Many properties of logic programs, for example, the associativity of *append* in fair search models, were easily expressed in the embedding, because of explicit scoping; on the other hand, in a Prolog setting, expressing this property would require some awkward coding using additional parameters.

Our approach is in a contrast to the standard methods used for reasoning about Prolog, or other logic programming languages. They are traditionally explained in terms of their declarative and operational semantics, where for a given logic program, these are respectively considered its specification and its model of execution. One of the main attractions of logic programming is its abstract and simple declarative semantics, but this semantics does not allow for reasoning about the results of a concrete query to a concrete logic program. The operational semantics does permit such reasoning, but it is generally too low-level for equational reasoning. We have shown that a semantics based on algebra can, to some extent, fill the gap between these other two semantics.

We have thus developed three different implementations of a logic programming language. In comparing their logical and computational properties we have concentrated on those algebraic laws which are shared in all three execution models, and in this sense the three models proved to be strongly consistent with each other and with the declarative reading. In Chapter 6 we used concepts from category theory to formalise the relationship between the three models, by defining three *semi-distributive monads* – extensions of the stream, matrix and forest monads – that satisfy certain laws and that capture each of the computation models. Then we proved the existence of unique mappings between the monad corresponding to the most general model and the other two monads, where the two mappings correspond exactly to the depth-first and breadth-first traversal of the search tree of a logic program.

Finally, we have also proved the forest monad to be the *initial object* of this category of search monads; this corresponds to the fact that the general search model sums up all the information that is common to all the elements in the "category of search-strategies" in logic programming. This means that any search strategy that is compositional, meaning that it obeys our list of laws for the explicit logic operators, can be obtained by searching the tree.

In order to use the algebraic framework of the embedding for purposes of modelling and reasoning about logic programs, we also needed to show that our implementation of the embedding and the resulting algebraic specification preserve the standard understanding of pure logic programs. We showed that our algebraic semantics complements more abstract semantics, such as Least Herbrand Models, as well as more concrete, execution-based ones, such as LD-resolution, by providing:

1. the possibility of executing, and deriving, correct implementations from the equational form by simulating LD-resolution;
2. a precise mathematical model of the system against which more abstract specifications, such as the set-based Least Herbrand Model semantics, can be proved correct by means of inductive reasoning; and
3. support for formal reasoning and analysis at the equational logic level, such as symbolic simulation, and transformation, derivation, and a

wide range of formal methods for the analysis of programs.

We addressed the first point in Chapter 7. Our argument for the correctness of the embedding with respect to LD-resolution was *indirect*: first, in Chapter 6 we showed that our algebraic laws are provably correct for all the models of the embedding, and second, in Chapter 7 we showed that the algebraic laws can be used to simulate LD-resolution. This also entails the soundness and completeness of the embedding with respect to the Least Herbrand Model semantics for logic programs.

To summarise our argument for the *operational validity of our transformations* of logic programs: in Chapter 3 we have shown how to translate into Haskell, in a compositional way, the fragment of the first-order logic which corresponds to pure logic programs. Next, the laws concerning these translations of the formulas were stated and proved in Chapter 4. Finally, in Chapter 7 we prove in Theorem 7.1 that the successful outcomes of the LD-resolution can be simulated by means of the algebraic laws of Chapter 4. These laws allow us to transform a given query in presence of the completed definition of the program in question to the equational representation of the computed substitution. Theorem 7.2 provides this result. Theorem 7.1 also guarantees that for each transformation of a query to the equational representation of the computed substitution there corresponds a successful LD-derivation.

We addressed the second point in Chapter 8. There we showed that predicates have a set based reading which corresponds to the Least Herbrand Model for the predicate, and that the operators of the embedding preserve this reading. We also related the embedding to the denotational semantics of first-order logical formulas given by Apt in [3], and to the operational semantics of first-order logical formulas given by Apt and Bezem in [5]. We argued that our embedding conforms to both of these semantics for the restricted case of logic programs and Herbrand interpretations, and can therefore used to help in bridging the gap between the more abstract semantics and the operational one, showing their equivalence in the case of logic programs.

After showing that the algebraic semantics is a suitable semantic framework for logic programs, we investigated how well the analysis and transformation methods of algebraic semantics, already known to work well for functional programming (see for example [15, 16]), specialise to logic programs. For this purpose, we have chosen two groups of examples: one consists of well-known problems of program transformation, where standard techniques are used, and the other of optimisation problems, for which the equational transformations are less widely known.

In Chapter 9 we described the algebraic transformation methods for the first group, covering the unfold/fold technique, accumulating parameters, and the transformation of generate-and-test programs. These examples have already been studied for logic programming, but we believe that our treatment is simpler and more *generic* than the previously used rules-and-strategies approach. In addition, our treatment highlights the algebraic similarities between functional and logic programming. Finally, the algebraic transformations can now be related to the underlying operational semantics, allowing one to argue about the preservation of computational properties such as termination. This relationship has not been explored before in the case of logic program transformation.

We found that there are two main *practical* advantages in using our functional embedding for transformation of logic programs. The first one is that it allows us to reason about logic programs in a simple calculational style, using rewriting and the algebraic laws of combinators; equations work better in this setting than implications. The second, and the more important advantage, is that many predicates are easily expressible using higher-order functions that can take predicates as arguments. The appropriate transformation strategy became obvious once we presented the predicates in a higher-order form using folds, and we could readily apply the ever important fold laws, such as the fusion law.

The examples we studied in Chapter 10 proved that the algebraic approach to program transformation is truly general, in the sense that the methods apply to whole families of examples, and to a large variety of such groups of examples. The case studies we have undertaken have been the specification and analysis of the 1/0 knapsack, the string edit, and the minimal tardiness problems. These algorithms arise in a variety of real-world problems, such as DNA-sequencing, natural language processing, and operations research, and therefore present challenging specification and analysis problems of a kind typically not included in formal methods for logic program transformation.

To perform these transformations, we have used the embedding to translate all examples from a logic to a functional setting. We have then used the algebraic representation to analyse the examples, and to select and apply the appropriate program transformation strategy. We found that in each of the three examples, the particular transformations arise from a straight-forward application of respective theorems from the Algebra of Programming in [18].

We found that, while the traditional approach views dynamic programming and greedy algorithms as separate and unrelated, the approach we take from [18] relates them to the same specification and thereby also makes clear the differences between the two strategies. Further, this approach made it easier to derive the efficient algorithms from the common specification, thus achieving two things: it helped us to write clear and efficient code and made us aware of the algorithmical issues.

These examples have also taught us a lesson about the expressiveness of the two programming styles, and of our embedding. We would argue that the problems studied here make a good case for functional logic programming languages, as we found that we needed the best of both worlds for a simple and clear presentation. Originally we wrote the examples in HiLog [24], a higher-order logic programming language which supports tabling (in XSB). This language provided relations and non-determinism, which were required by the theorems and the examples. In [18] the examples are written in Haskell, and non-determinism had to be simulated. Also, the predicate *bagof* and the use of logical variables were a big convenience. On the other hand, higher-order functions are essential for the derivations, and we find that the *call* notation is somewhat cumbersome. In HiLog we also missed currying, and types were needed to make the theorems more intuitive, because many of our predicates use sets with non-trivial structure.

In summary, a language that incorporates these features would be the perfect tool for declarative algorithm derivations. There exist several good candidates, such as Mercury, Curry, Oz or Escher, but one does not even need to go that far: as we have shown, our embedding can add all the desirable relational features to, say, Haskell, for a cost of a few dozen lines of code.

Even *bagof* came for free in our setting. The present simple implementation of the embedding posed problems for the treatments of arithmetics and we missed built-in operators; however, this is not a conceptual problem, just a programming one.

11.2 Related work

Translating logic programs into functional ones, or to the related area of term rewriting systems, has long been a subject of investigation. The literature presenting a translation from logic to functional programs, or rewriting systems, is divided into two main groups. One group focuses on translations where nondeterminism or failed computations are not permitted. They do not concern themselves with the search-related issues, and they are mainly interested in using the translated form of the program for proving properties, such as termination, of the original logic program. The other group study the extent to which the full computational mechanism of a logic program is preserved, and most often provide implementations of a Prolog like LD-resolution with depth-first search, in which failure and multiple answers are allowed.

The earliest translations were motivated by a better insight on the relation between functional and logic languages, and belong to the second group mentioned above. Reddy [108, 110] and Robinson [116, 114] described in the mid-eighties translations of logic programming languages to functional programming languages.

The next generation of translations belongs to the first group above, focusing on using the translation to rewriting systems for proving logic program properties, most often termination. However, all these translations impose the restriction that the original logic program must be *well-moded* or *simply moded*, a property which can be considered 'the functional core of logic programs'. Such logic programs have a straightforward left-to-right dataflow model and prohibit the use of logic variables in complex data structures such as difference lists. As stated by Etalle and Mountjoy in [46], 'It is now widely accepted that "complex" logical variables, the possibility of a dynamic selection rule, and general properties of non-well-moded programs are exclusive features of logic programs. This is not quite right.' We return to [46] later.

For example, Krishna Rao, Kapur and Shyamasundar present in [71] a termination preserving, but complex, translation of well-moded logic programs into unconditional rewriting systems. Ganzinger and Waldmann [51] describe a simple translation of well-moded logic programs into conditional rewriting systems and a criterion of *quasi-reductivity* as a method to derive whether a logic program terminates in a unique result. Marchiori describes in [82] a termination preserving, but complex, translation from well-moded and simply-moded logic programs to term rewriting systems, and Arts and Zantema describe in [7] a translation of logic programs into constructor systems, and a method for proving the termination of constructor systems which implies the termination of the original program. Finally, also in this group, van Raamsdonk presents in [139] a translation similar to that of [51], where a well-moded logic program is translated into a termination preserving conditional rewriting system with explicit substitutions. The paper contains also a correctness result relating a successful resolution sequence to a rewriting sequence resulting in a normal form expression containing the *single* computed answer substitution.

Our work belongs to the other group, since we explicitly choose to deal with nondeterminism and logic variables. As already mentioned, Reddy was the first to show [108, 110] that many logic programs can be translated in a straightforward way into functional programs using list comprehensions. He used mode annotations, specifying which arguments are expected to be ground, to turn relations into functions; he thereby restricted the class of logic programs used in translations. The languages LogLisp and Super [30, 114, 116] of Robinson use a compositional approach similar to our embedding, in that an existing functional language (Lisp) is used to model the behaviour of logic programs, and a set of logic programming primitives is implemented as functions. Wand presents in [147] another compositional embedding of Prolog into Scheme. On a more abstract level, Baudinet presents in [11] a model-theoretic, denotational semantics for logic programs that is equivalent to the denotational semantics which arises from our embedding, as described in Chapter 8; she also expresses *cut* in this semantics, and uses the semantics to analyses termination of queries. Hinze presents in [63] a translation of nondeterministic logic programs in Haskell, using a monadic approach. His translation appeared approximately at the same time as ours, and the two transformations are quite similar, but we use them for different purposes. We focus on applications of the embedding for program transformations; he focuses on its application to program termination analysis, and uses it as an example of the expressiveness of monads in Haskell. In [85, 86] McPhee and de Moor propose a system for compositional logic programming; however, their focus is more on implementation whereas ours is on applications to program transformation. Billaud [14] describes the list of algebraic laws that capture depth-first search, the same list as our laws for & and ‖, but he does not generalise to the other basic operators in the completed form of a logic program, and he does not generalise to the other search models.

In [117] Ross proposes a compositional algebraic semantics for Prolog, based on the process algebra CCS, but he is not concerned with other search strategies or program transformation. Also, in [58], Hamfelt, Nilsson and Vitoria present a compositional embedding of pure logic programs into equational programs, using a point-free style of Bird and de Moor [18] and quasi higher-order predicates to implement the logic operators, and prove this semantics equivalent to the usual Kowalski-van Emden fixpoint semantics of logic programs. We believe that our Haskell embedding and the operational semantics arising from our embedding are simpler. The similarities continue on a denotational level: [58] identifies a canonical form of predicates involving only combinators for conjunction, disjunction and generalised projection, parallel to our operators &, ‖ and ∃, with the corresponding denotational semantics, respectively intersection, union, and an operation corresponding to a projection on the Herbrand universe.

We discuss in some length the relationship of our work to [46], since it is the most recent translation-oriented work we have found to date. In [46] a *literal*, syntactic, translation of consistent logic programs to Haskell is proposed; no restrictions to well- or simply-modedness of logic programs are made, but they assume that arguments are moded. Our paths diverge from the very start, since they do not consider logic programs which may return more than one answer, stating that 'such programs are intrinistically logic programs and therefore do not belong to our target'. They subsequently discuss the relationship between lazy evaluation and logical variables, dynamic scheduling, and nondeterminism, and our standpoints differ on all of these. They view logical variables as an exclusive feature of logic programs; our work does not elaborate upon this point, but we use "complex" logic variables such as lists throughout our examples. Concerning dynamic scheduling, [46] state that 'the lazy computing mechanism compensates for the lack of control over dynamic scheduling'; we discuss this issue in some length in Chapter 3, where we argue why it is more difficult to vary the selection rule than the search strategy, and we point to the work of McPhee [85] for a deeper analysis of dynamic selection rules in a lazy functional setting. About non-determinism [46] state that only *input discriminative* nondeterministic logic programs may be "safely" translated to Haskell, where this property guarantees that for any consistent query corresponds to an SLD-tree with only one successful branch. They do mention that multiple answers can be returned can be returned by a list of successes, such as advocated by Wadler [141] and used by us, or Reddy [108, 110], Robinson [114, 116] and others. However, the position of [46] is that nondeterminism should be considered a peculiarity exclusive to logic programming. They imply that this restriction to determinate programs is necessary to obtain an '*literal translation* in which the computational mechanism of the resulting functional program is as similar as possible to the one of the original logic program.' Our view is that this restriction is not necessary, since our embedding has *provably* the same computational model, LD-resolution, as the original logic program, and in addition allows a variation in the search strategy, while allowing nondeterminism.

Another large area of related work is the transformation of logic programs. As discussed in Chapter 9, the work in logic program transformation can be divided in two main categories: approaches based on "rules and strategies", and approaches based on "schemata". For an excellent survey of these techniques, see [98] by Pettorossi and Proietti. In short, the rules and strategies approach has a set of fine grained, elementary rules, and the application of these rules is guided by strategies, metarules which prescribe suitable sequences of applications of rules. The central rule in this technique is "unfold-fold", originally introduced by Burstall and Darlington already in 1977 in [23]. In the alternative approach, based on schemas, the program derivation is guided by recognition of higher-order clichés, that is, recursion schemes of universal nature. Traditionally, schemas are expressed in a first-order setting, so a program schema is an abstraction of a program, where some terms, goals and clauses are replaced by metavariables. In the extensive existing literature, the schema transformations deal with recursion removal and the reduction of nondeterminism in generate-and-test programs. Our style of program transformation is somewhere between these two groups. It is similar to the schema approach, where the schemas in our case correspond to applications of theorems regarding fold operators. However, we do not transform whole programs at once, just parts of predicates. On the other hand, our algebraic laws can also be viewed as the rules from the rules and strategies approach; in that case, applications of theorems about fold operators correspond to strategies. Therefore we choose to not adhere to this division in this summary of related work, but to rather address some of the more recent results in program transformation.

In [100], Pettorossi and Proietti present the syntax and operational semantics for a higher-order logic programming language, which is an extension of a definite logic program with goals as predicate arguments. This allows them to apply some of the ideas from higher-order based transformations [18] or continuation based transformations [146] in functional programming. They provide a set of transformational rules together with a correctness theorem, which ensures that if a goal succeeds or fail in the given program, then also the derived program succeeds or fails; however, they do not preserve computed answers. They do not seem to deal with termination in the general case, but rely on the operational semantics based on *universal termination*. In such semantics, the meaning of a goal is defined iff all LD-derivations starting from that goal are finite. Our transformation technique preserves termination and the computed answers.

Transformation of definite logic programs has also been extensively studied by Bossi and Cocco. In [20], they study the correctness conditions for fold-unfold transformations in *left terminating* logic programs. A logic program P is left terminating iff all ground goals universally terminate in P. They present a transformational system which preserves the computed answers, but the folding rule they present there requires exactly one clause whose body is a conjunction of atoms. Since we deal with folding and unfolding in an equational setting, we do not have to place such restrictions. Also, [20] does not guarantee termination preservation. In [21], Bossi, Cocco and Etalle present the sufficient conditions for the goal rearrangements to preserve left termination. Our rules preserve general termination.

Hamfelt and Nilsson take a schema-oriented approach to program transformation, based on a higher-order form of logic programs and on theorems relating the higher-order predicates. This approach is very close to ours, also in the sense that they rely on these theorems not only for declarative equality of the original and derived programs, but also on their procedural equality, in the sense that they compute the same answers and have the same termination behaviour. In [56] they describe a set of higher-order relational recursion operators, which is intended to cover all forms of recursive predicate formulations met in logic programming practice. In [93] they identify a class of primitive recursive list processing logic programs which can be formulated using the relational counterparts of *foldl* and *foldr* from functional programming. They prove the duality theorem connecting these relational fold operators and their termination behaviour. Since they only use higher-order relations which can be replaced by explicit recursions in a pure logic program, they refer to them as quasi higher-order predicates. A similar system of recursion operators for quasi higher-order predicates was proposed simultaneously and independently in [52] for λPROLOG. In [58] Hamfelt, Nilsson and Vitoria propose a compositional form of definite logic programs, called Combilog, mentioned earlier. In this work they view predicates as being formed by composition of basic predicates or program defined predicates by means of combining forms functioning as higher-order predicates. In [57] they apply a recursion schema based transformation approach to such compositional logic programs, and also come to the conclusion that the higher-order representation may help support the declarative understanding of logic programs.

Finally, Pettorossi and Proietti present in [105] a transformation strategy based on the introduction of lists and higher-order predicates on lists; this strategy can be viewed as a rules-and-strategies counterpart of our example for derivation of efficient reverse.

As stated before, the transformational examples we present in this thesis are taken from Bird and de Moor's [18]. For most of these examples there exists a substantial body of research, and we choose not to elaborate on all of them. We focus on the work related to the derivation of the sorting example.

Sorting algorithms are traditionally classified according to their main operational semantics, for example, whether they do sorting by insertion, partitioning and so on. More recently top-down program synthesis has been used as a basis for classification. Darlington presents in [37] a family of six sorting algorithms, where the classification is based on program transformation on recursion equations, using fold-unfold as the key transformation rule. He derives and classifies *quick sort*, *selection sort*, *merge sort*, *insertion sort*, *exchange sort* and *bubble sort*. Clark and Darlington derive in [28] the first four algorithms listed above, using similar transformational rules in a first-order predicate logic notation. Green and Barstow synthesise in [54] the same six algorithms using the divide-and-conquer paradigm; they show how their system can automatically construct the programs, and analyse the facts and rules used in the process.

Smith describes in [127, 126] yet another method for derivation and classification of the first four sorting algorithms by top-down decomposition of specifications into subproblem specifications, followed by a synthesis of concrete program for the subproblems, and then a bottom-up composition of these. The classification criterion is the design strategy chosen in the decomposition and composition phases. Dromey uses in [44] Dijkstra's constructive weakest pre-condition technique to derive sorting algorithm from a specification which is in the form of a pair of pre- and post-condition, and manipulating the post-condition. Lau and Prestwich derive in [73] the sorting algorithms listed above and also *bsort*, *radix exchange sort*, *distribution sort*, *block bubble sort* and *external merge sort*. Their approach is also based on the fold-unfold rule, and describes the computation of a semi-automatics logic programming system which produces a recursive logic procedure by top-down syntheses from its given specification.

In Chapter 6 we refer to category theory in order to give a rigorous analysis of the relationship between our three implementations. We do not aim to give a categorical semantics to logic programming; however, this has been done in several ways before, and we describe this area of work as the last subtopic for our related work.

Category theory has been successfully used to give a mathematical treatment of several aspects of the syntax and the semantics of programming languages [101], such as types [35, 72], state [88, 89, 94, 130, 143], non-determinism [96] or polymorphism [102]. The reason for the popularity of this approach is that it captures the desired features in a generic and implementation independent, yet rigorous, manner. For the same reasons, the categorical approach has more recently been applied to logic programming features as well.

The first logic programming feature analysed in a categorical setting was unification, in a paper by Burstall and Rydeheard in [118]. Further, logic programs were described in a topos theory by Aspetti and Martini [8] and the syntax of Horn clause programming over the Herbrand Universe was was formalised in first-order categorical logic. Logic program transitions and structure were analysed and described using indexed monoidal categories by Corradini, Asperti and Montanari in [33, 34]. Concurrent constraint logic programming was treated in categorical setting by Panangaden, Scott, Seely and Saraswat in [97]. Horn clause resolution was described via indexed categories by Power and Kinoshita in [103]. Constraint Horn clause programming was formulated using categories and institutions by Diaconescu in [43]. The evaluation of a logic program was studied in a categorical framework of types and realisability by Pym in [106]. A categorical treatment of non-declarative extensions of logic programming, notably constraints and uniform proof systems, based on a resolution system over finite product τ-categories, has been undertaken by Finkelstein, Freyd and Lipton in [75]. Non-deterministic SLD-resolution proofs are represented as arrows in an extension of a base syntactic category, subject to certain categorical constraints on data, by Lipton and McGrail in [69, 84]. However, all this research is only marginally related to the work presented in this thesis. We only apply category theory briefly, in order to make a formal argument that our list of algebraic laws is "reasonable" and complete; our main goal is to use the laws for program transformation, and not a thorough categorical analysis of the logic programming paradigm.

Finally, the work presented here has been cited in the following two papers: in [25] Claessen and Ljunglöf extend our embedding to incorporate functional logic programming. In [137] Todoran and Papaspyrou give denotational models to parallel logic programming languages, where the semantics arises from a functional embedding akin to ours.

11.3 Functional logic programming

Our work is in some ways related to the area of functional logic programming (FLP), a style of programming which subsumes purely functional languages as well as pure logic programming. The integrated languages of FLP have more expressive power than functional languages due to features like inversion and logical variables, and they have more efficient operational behaviour than logic languages due to deterministic evaluation. At present, the research focus concerning such integrated languages is mainly on the improvement of execution principles and efficient implementation for integrated languages.

However, the major motivating factor behind this programming style lies beyond the purely operational aspects of programming: it is to enable the sharing of developments on many facets of declarative programming language research between the two programming styles. Our work is closer in intention to this aspect of unification of the two styles, and we believe that our examples in the area of program transformation could help assure both functional and logic programmers of the advantages of this unified style.

In order to easier contrast the approach taken by different FLP languages and our embedding, we now present a summary of the execution principles of FLP. This summary is based on Hanus [59], and on [13, 38, 77, 79, 112].

The integration of the operational principles of functional and logic programming can be approached from both styles, but in both cases it yields similar results. One way of allowing function definitions inside logic programs is to treat functions as atoms with a special predicate symbol =, taking this to mean "evaluates to" rather that the usual syntactic unifiability on the two arguments. Such atoms are called *equations*. Then clauses, queries and programs can be defined as in pure logic programming. Given a clause:

$$p_0 \leftarrow p_1, \dots, p_n,$$

if p_0 is an equation, this clause is also called an equation; if $n > 0$, it is called *conditional*, otherwise it is called an unconditional equation. Equations are always used from left to right, so they are also called rewrite rules. For example, the function *append* can be defined with unconditional equations:

$$\mathit{append}([\,], x) = x.$$
$$\mathit{append}([x|a], b) = [x|\mathit{append}(a, b)].$$

Terms such as $\mathit{append}([1], [2])$ may be used as any other terms in programs containing these definitions. When such functional term only contains ground terms, it is simply evaluated as a rewrite step. However, it may also be non-ground, so we might have queries such as:

$$\mathit{append}(l, [2]) = [1, 2].$$

To compute such results in general one has two alternatives: either to search for the right instantiation of the new variable l occuring in the condition, or to delay the computation of such atoms until they are sufficiently instantiated. The first approach is called *narrowing* and the second *residuation*.

The fundamental idea behind narrowing, as introduced by Slagle [125], is to use unification rather than matching in the rewrite step, if the function call contains free variables. The unification required to do this is computed with respect to a set E of axioms, and is called E-unification. Unfortunately, E-unification is undecidable even for simple equational axioms like distributivity and associativity of functions. The practical solution to problems involving E-unification is to impose restrictions on the definition of the equality predicate, that is, on the rewrite rules in the system. The aim, then, is to find restrictions which are acceptable from a programming point of view, and which ensure the existence of a usable algorithm. Such restrictions often include confluence and termination of the rewrite rules.

In one step of narrowing, a non-variable subterm of the query is unified with the left-hand side of some rule and the instantiated subterm is replaced by the right-hand side of that rule. We refer to the subterm chosen for unification as the narrowing position. A narrowing step from $\langle q; \sigma \rangle$ to $\langle q'; \sigma\eta \rangle$, with the unifier η, is denoted as:

$$\langle q; \sigma \rangle \leadsto_{\eta} \langle q'; \sigma\eta \rangle.$$

For example, in order to solve $\mathit{append}(l, [2]) = [1, 2]$, the second rule for *append* is applied, followed by the first rule:

$$\begin{array}{ll} & \langle\ \mathit{append}(l,[2]) = [1,2]\ ;\ \varepsilon\ \rangle \\ \leadsto_{\{l/[x|a]\}} & \langle\ [x|\mathit{append}(a,[2])] = [1,2]\ ;\ \{l/[x|a]\}\ \rangle \\ \leadsto_{\{a/[\,]\}} & \langle\ [x,2] = [1,2]\ ;\ \{l/[x], a/[\,]\}\ \rangle \end{array}$$

The final equation instantiates x to 1, so the computed solution is $\{l/[1]\}$.

Given a confluent and terminating set of rewrite rules defined by an unconditional set of equations E, narrowing is sound and complete in the sense that each computed substitution is a unifier w.r.t. E, and for each unifier w.r.t. E exists a more general computed substitution.

However, there is an obvious practical difficulty in the definition of narrowing: if more than one left-hand side is unifiable with the selected subterm, the evaluation must choose a suitable rule nondeterministically. To guarantee completeness, each rule must be applied at each non-variable subterm of the given query, and this yields a huge search space with many infinite paths even for a small program. In order to use narrowing as a practical operational semantics, further restrictions are necessary.

One possible restriction is called *basic narrowing* [67]. Here the narrowing step may only be performed at a subterm which is not part of a computed substitution, but belongs to an original program clause or query. Basic narrowing is also sound and complete in the sense above, because searching for narrowing positions inside substitutions is superfluous, but is still highly nondeterministic.

For a restricted set of functional logic programs, called constructor-based functional logic programs, the deterministic strategies of functional programming can be simulated with *innermost* [49] and *outermost* [91, 109] narrowing strategies, corresponding to eager and lazy evaluation in functional programming. Another improvement is *selection* narrowing [19], where a selection rule, similar to the selection rule of SLD-resolution, is employed to select exactly one innermost position for each narrowing step. Leftmost innermost basic narrowing, a combination of these three strategies, can be shown to be equivalent to LD-resolution after appropriate translation of the functional logic program into a logic one. There are further improvements on narrowing which result in an execution mechanism that is superior to LD-resolution.

Outermost narrowing is based upon lazy evaluation, where the next narrowing position must be an outermost one, and is complete if the rule set is confluent, terminating and if the selection strategy is well-behaved in a certain sense. Lazy evaluation allows infinite data structures; but if the rewrite relation is not terminating, which is the case when infinite data structures are involved, confluence also becomes undecidable, so a series of strong restrictions on the rules is required to ensure completeness. Finally, in the case of conditional equations, *conditional* narrowing [68] is used.

With all forms of narrowing, an uninstantiated value of an argument must be guessed in a nondeterministic way. The alternative is a reduction strategy based on delaying evaluation of functions until it is possible in a deterministic way, when all the arguments are sufficiently instantiated, so that they are reducible to unique ground value. This mechanism is often called *residuation* [136]. The basic operational semantics is SLD-resolution, with an extended unification procedure in which any function call in a term is evaluated before it is unified with another term.

This evaluation strategy seems preferable to narrowing since it preserves the deterministic nature of functions. For example, the query

$$append([1], l) = [x], l = [2].$$

solves the first literal by producing the residual $append([1], l) = [x]$, which will be proved or disproved as soon as the variable l becomes ground. After solving the second literal $l = [2]$, the residual $append([1], [2]) = [1, 2]$ can be proved true, binding x to $[1, 2]$ in the process.

This delay principle is satisfactory in many cases, but is incomplete in general. The evaluation may result in an infinite derivation path, generating infinitely many residuals, and even if these residuals become together unsolvable at some point in the derivation, this is not detected since they are simply delayed. On the other hand, a functional logic language based on narrowing can solve such a query in a finite search space.

11.4 Further work

As with any research, more work can always be done. There are also some weaknesses of the approach taken here. Maybe the most important deficiency of this approach is the expressiveness and the implementation of the embedding. We have chosen to focus on methodological issues so simplicity of the embedding was therefore a priority. The result of this choice is that the current implementation is too slow for any practical purpose, and that it does not have the full power of functional logic programming languages. Further, we have automated neither the translation of logic programs nor the program transformation techniques. We hope to address the last issue in the future, as pointed out below. Another possible criticism is that we have chosen to list a small (but complete) set of basic algebraic laws that can be used for program transformation, rather than explore the "schemas" or "strategies" beyond the standard fold-theorems from the functional setting of Haskell.

We devote the rest of this section to outlining the most promising directions such further work could go in. Some are extensions of the embedding that will require some conceptual ground-work but should otherwise be just a matter of implementation, and some are open problems.

An embedding of a functional logic programming language. This embedding could be achieved by extending the embedding. We can implement something similar to the residuation approach in our embedding by letting the predicates in the embedding be functions from sets of constraints to sets of constraints. We would like to see what algebraic properties such a language has, and see whether we can find some interesting examples for program transformation, similar to the ones presented here.

An embedding of a constraint programming language. This is also achievable without major changes to our embedding. Based on our embedding and our discussions at ICLP'99, Stuckey [135] has implemented an embedding of a simple constraint programming language in Haskell. After this implementation, Stuckey remarked that this implementation was superior in efficiency to a previous one which he implemented with Wadler, based on a monads in Haskell. Apparently, even though simpler, the approach based on the embedding avoids the unnecessary copying of the constraint store. There are several promising research avenues based on this implementation: our favourite ones are an algebraic specification of a constraint language, and subsequent applications the program transformation from functional programming. Both the implementation and the examples may help functional programmers realise how close this constraint-based style is to functional programming, and might lead to a further cross-fertilisation of the methodologies for these declarative styles.

An embedding that incorporates Haskell's type system. Ideally, it would be nice to be able to mix Haskell's term types within one Prolog-like piece of code, thus supporting a "typed logic programming" paradigm in Haskell. This topic connects to the rich literature on modular semantics through monadic interpreters in Haskell. Indeed, this idea is explored and implemented in [25], which is based on our embedding presented in this monadic manner. In this work all the types are introduced individually, by introducing a new monad for each "by hand". We are interested in exploring alternative approaches, possibly based on dependent types.

An efficient implementation of the embedding. At present, the embedding is intentionally simple but slow, and should be thought of as a prototype for a more efficient implementation. This implementation would be achieved by making the operators of the embedding *primitives* in the functional language, instead of them being library functions. The challenging problem here is that the built-in substitutions and unification must interact correctly with laziness.

A better timing-analysis of the logic programs. At the moment we have an informal approach to the efficiency of the original and the derived logic program. We need better formal techniques for analysing the time requirements of non-trivial logic programs, but this is not easy because laziness is involved.

Automation of some examples. All the derivations of the more efficient programs presented in this thesis are done "by hand". The recent work of Sittampalam and de Moor [40, 124] deals with a tool for an automatic derivation of the more efficient program for some of the same examples, in a functional setting. It is possible that this tool could be extended to work on logic programming examples, or functional logic or constraint programming as well, thorough the use of the appropriate embedding.

Using the embedding to transfer other ideas. So far, we have focused on program transformation. However, there are other areas of research in functional programming that seem amenable to migration to other declarative styles. For example, in [95], Okasaki presents a suite of efficient, purely declarative data-structures supporting features such as random access or efficient catenation. Since the embedding is placed in the same lazy setting of Haskell, it would be interesting to find out how well such data-structures extend to logic programming.

Categorical analysis of search in recursive programs. Our analysis in Chapter 6 does not consider recursive programs, since we do not include a fixpoint operator in the definition of an extended monad, that is, the definition of a search strategy. Haskell does not have constructs for defining recursive equations over a monadic type, but Launchbury, Lewis and Cook propose in [74] as an extension of Haskell a special fixpoint operator `mfix` for monadic types. In [45], Erkök and Launchbury pursue an axiomatic analysis of `mfix`, by postulating three axioms that characterise the behaviour of `mfix` and show that these are satisfied in several individual monads, called recursive monads. It would be interesting to see whether our monads of search can be turned into recursive search monads, following this approach.

Analysis of the computational complexity of predicates. In Chapter

7 we show that a finite number of computational steps in LD-resolution can be simulated by a finite number of computational steps in our embedding. The steps in the embedding are simulated by the algebraic laws, which preserve the operational behaviour of the embedding. However, it is not clear whether these numbers of reduction steps are always proportional, and it would be interesting to see whether we can use th embedding to deduce a result about the computational complexity of the original logic program.

The original purpose of the embedding was algebraic program transformation, but it proved to have many other interesting properties and theoretical applications. As argued here, there remain many topics for further research related to the theoretical, and practical, applications of the embedding; however, we believe that, to a large extent, the plethora of questions it opens is one of the main attractions of this algebraic approach.

What we call the beginning is often the end
And to make an end is to make a beginning.
The end is where we start from.
T. S. Elliot, Four Quartets

Appendix A

Code for the Embedding

A.1 General types

```
> data Term = Func Func [Term] | Var Var
> type Func = String
> data Var = Name String | Gen Int

> type Subst = [(Var, Term)]

> type Predicate = Answer -> [Answer]
> type Answer = (Subst, Int)
```

A.2 Basic predicates and combinators

```
> no :: Predicate
> no x = []

> yes :: Predicate
> yes x = [x]

> infixl 6 |||
> (|||) :: Predicate -> Predicate -> Predicate
> (p ||| q) x = (p x) ++ (q x)

> infixl 7 &&&
> (&&&) :: Predicate -> Predicate -> Predicate
> (p &&& q) x = concat (map q (p x))

> eqn :: (Term,Term) -> Predicate
> eqn (t1,t2) (s,n) =
>    case (unify s (t1,t2)) of
>       Just u -> [(u,n)]
>       Nothing -> []

> exists :: Int -> ([Term] -> Predicate) -> Predicate
> exists k f (s,n) = f vs (s, n+k)
>                    where vs = map makevar [n..n+k-1]
> neg :: Predicate -> Predicate
> neg p (s,n) = yes (s,n), if res == []
>     = no (s,n), otherwise
>    where res = p (s,n)
```

A.3 Integer and list modelling

```
> zero :: Term
> zero = atom "0"

> succ :: Term -> Term
> succ t = Func "succ" [t]

> cons :: Term -> Term -> Term
> cons x a = Func "cons" [x,a]

> nil :: Term
> nil = atom "nil"

> atom :: String -> Term
> atom a = Func a []

-- plus(X,0,X).
-- plus(X,S(Y),S(Z)) :- plus(X,Y,Z).

> plus :: (Term,Term,Term) -> Predicate
> plus (p,q,r) =
>    (exists 1 (\ [x] -> eqn(p,x) &&& eqn(q,(Func "0" [])) &&& eqn(r,x)))
>    |||
>    (exists 2 (\ [x,y,z] -> eqn(p,x) &&& eqn(q,(Func "succ" [y])) &&&
>       eqn(r,(Func "succ" [z])) &&& plus(x,y,z)))

-- minus(X,0,X).
-- minus(0,X,0).
-- minus(S(X),S(Y),Z) :- minus(X,Y,Z).

> minus :: (Term,Term,Term) -> Predicate
> minus (p,q,r) =
>    (exists 1 (\ [x] -> eqn(p,x) &&& eqn(q,zero) &&& eqn(r,x)))
>    |||
>    (exists 1 (\ [x] -> eqn(p,zero) &&& eqn(q,x) &&& eqn(r,zero)))
>    |||
>    (exists 3 (\ [x,y,z] -> eqn(p,(Func "succ" [x])) &&&
>       eqn(q,(Func "succ" [y])) &&& eqn(r,z) &&& minus(x,y,z)))
```

A.4 Auxiliary functions

```
> makevar :: Int -> Term
> makevar i = Var (Name ("x"++(show i)))

> prolog :: Predicate -> [String] -> String
> prolog p vars = print vars result
>   where result = p startAnswer

> startAnswer::Answer
> startAnswer = ([],0)

> print :: [String] -> [Answer] -> String
> print vars ((s,n):others) = "no", if s==[]
>   = "yes", if (vars==[] && not (s==[]))
>   = instsubst s (filter (onlyinput vars) s) ++
>        "\n" ++ (print vars others), otherwise
> print vars [] = ""

> onlyinput :: [String] -> (Var,Term) -> Bool
> onlyinput vars (v,t) = True, if member vars (varname v)
>   = False, otherwise

> all :: [String] -> (Var,Term) -> Bool
> all vars (v,t) = True

> member :: [String] -> String -> Bool
> member [] x = False
> member (y:ys) x = True, if x==y
>   = member ys x, otherwise

> varname :: Var -> String
> varname (Name v) = v

> instsubst :: Subst -> [(Var,Term)] -> String
> instsubst s [] = ""
> instsubst s ((v,t):others) =
>   varname v ++ " = " ++ show (inst s t) ++ "\n"
>      ++ instsubst s others
```

A.5 Substitutions and unification

```
> data Maybe a = Just a | Nothing

> tryfold :: (a -> b -> Maybe a) -> a -> [b] -> Maybe a
> tryfold f x [] = Just x
> tryfold f x (y:ys) =
>   case f x y of
>     Just z -> tryfold f z ys
>     Nothing -> Nothing

> tryassoc :: [(a, b)] -> a -> Maybe b
> tryassoc [] x = Nothing
> tryassoc ((u,v):ps) x =
>     if u == x then Just v else tryassoc ps x

-- subst s t is one step of applying a substitution

> subst s (Func f xs) = Func f xs
> subst s (Var v) =
>   case tryassoc s v of
>     Just t -> subst s t
>     Nothing -> Var v

-- inst s t applies substitution s to term t: it
-- iterates to a fixpoint

> inst s t =
>   case subst s t of
>     Func f xs -> Func f (map (inst s) xs)
>     Var v -> Var v

-- unify does unification relative to an exiting substitution

> unify :: Subst -> (Term, Term) -> Maybe Subst
> unify s (Var v, t2) = univar s v t2
> unify s (t1, Var v) = univar s v t1
> unify s (Func f a, Func g b) =
>   if f == g then tryfold unify s (zip (a,b)) else Nothing

> univar s v t =
>   case tryassoc s v of
>     Just u -> unify s (u, t)
>     Nothing -> if t' == Var v then Just s else Just ((v,t'):s)
>                where t' = subst s t
```

Appendix B

Code for problems from Chapter 9

B.1 General definitions

```
fold(_,E,[],E).
fold(P,E,[A|X],B) :-
        fold(P,E,X,B1), call(P,(A,B1),B).

unfold(_,E,[],E).
unfold(P,E,[A|X],B) :-
        call(P,(A,B1),B), unfold(P,E,X,B1).

best(_,[A],A).
best(R,[A,B|X],C) :- call(R,A,B), best(R,[A|X],C).
best(R,[_,B|X],C) :- call(R,A,B), best(R,[B|X],C).

onebest(_,[A],A).
onebest(R,[A,B|X],C) :- call(R,A,B), !, onebest(R,[A|X],C).
onebest(R,[_,B|X],C) :- onebest(R,[B|X],C).

merge(_,[],X,X).
merge(_,X,[],X).
merge(R,[A|X],[B|Y],[A|Z]) :- call(R,A,B),!,merge(R,X,[B|Y],Z).
merge(R,[A|X],[B|Y],[B|Z]) :- merge(R,[A|X],Y,Z).

thin(_,[],[]).
thin(R,[A|X],Y) :-
        thin(R,X,Y1), bump(R,A,Y1,Y).

bump(_,A,[],[A]).
bump(R,A,[B|X],Y) :-
        call(R,A,B) -> Y = [A|X] ;
        call(R,B,A) -> Y = [B|X] ;
                       Y = [A,B|X].

consmap(_,[],[]).
consmap(P,[(A,X)|Y],[[A|NewX]|NewY]) :-
        call(P,X,NewX), consmap(P,Y,NewY).

pow(P,(A,XS),Y) :-
        member(X,XS),call(P,(A,X),Y).

head([A|_],A).

bmax(X,Y,X) :- X >= Y.
bmax(X,Y,Y) :- X < Y.
```

B.2 Definitions for string edit

```
%% the input strings (S1,S2) are given as character lists

%% Problem specification

edit((S1,S2),Out) :-
        bagof(X,unfold(step,([],[]),X,(S1,S2)),Bag),
        best(lleq, Bag, Out).

step(((cpy,A),(X,Y)),     ([A|X],[A|Y])).
step(((del,A),(X,[])),    ([A|X],[])).
step(((del,A),(X,[B|Y])), ([A|X],[B|Y])).
step(((ins,B),([],Y)),    ([],[B|Y])).
step(((ins,B),([A|X],Y)), ([A|X],[B|Y])).

lleq(X,Y) :- length(X,M), length(Y,N), M =< N.

%% Refined dynamic programming algorithm

edit2(([],[]),[]).
edit2((S1,S2),Out) :-
        bagof((A,X), step((A,X),(S1,S2)), Bag),
        consmap(edit2,Bag,Bag1),
        best(lleq,Bag1,Out).
```

B.3 Definitions for 1/0 knapsack

```
%% items should be declared in the form item(Name, Value, Weight).

%% Problem specification

knapsack(W,In,Out) :-
        bagof(X,fold(step(W),([],0,0),In,X),Bag),
        best(vgeq,Bag,Out).

step(W,(A,X),Y) :- step1(W,(A,X),Y).
step(W,(A,X),Y) :- step2(W,(A,X),Y).
step1(_,(_,X),X).
step2(W,(A,X),Y) :- addone(A,X,Y), within(W,Y).
```

```
addone(A,(NS,VS1,WS1),([A|NS],VS2,WS2)) :-
         item(A,V,W), VS2 is V + VS1, WS2 is W + WS1.

%% The refined thinning algorithm

knapsack2(W,In,Out) :-
         fold(step3(W),[([],0,0)],In,List),
         head(List,Out).

step3(W,(A,X),YS) :-
         bagof(Y,pow(step1(W),(A,X),Y),Bag1),
         bagof(Y,pow(step2(W),(A,X),Y),Bag2),
         merge(vgeq,Bag1,Bag2,Bag),
         thin(q,Bag,YS).

%% Auxilary functions for problem specification

within(W,X) :- weight(X,WX), W >= WX.

value((_,VS,_),VS).
weight((_,_,WS),WS).

vgeq(A,B) :-
         value(A,VA), value(B,VB), VA >= VB.
wleq(A,B) :-
         weight(A,WA), weight(B,WB), WA =< WB.
q(A,B) :-
         vgeq(A,B), wleq(A,B).
```

B.4 Definitions for minimal tardiness

```
%% jobs should be declared in the form job(Name, CT, DT, WT).

%% Problem Specification

sche(In,Out) :-
         bagof(X,bagify(X,In),Bag),
         best(costleq,Bag,Out).

bagify(Y,X) :- unfold(bcons,[],Y,X).

bcons((A,X), [A|X]).
bcons((B,[A|X]), [A|Y]) :- bcons((B,X),Y).

%% The refined greedy algorithm.

sche2([],[]).
sche2(B,[J1|S1]) :-
         bagof((A,X), bcons((A,X),B), Bag),
         onebest(penaltyleq,Bag,(J1,B1)),
         sche2(B1,S1).

%% Auxilary functions for problem specification
```

```
ct(J,C) :- job(J,C,_,_).
dt(J,D) :- job(J,_,D,_).
wt(J,W) :- job(J,_,_,W).

penalty((J,X),P) :-
        ct(J,C), wt(J,W),dt(J,D),
        totaltime(X,TT),
        P is (TT+C-D)*W.

totaltime([],0).
totaltime([J|X],T) :-
        totaltime(X,T1), ct(J,T2), T is T1+T2.

cost([],0).
cost([J|X],C) :-
        penalty((J,X),C1), cost(X,C2), bmax(C1,C2,C).

costleq(X,Y) :-
        cost(X,CX), cost(Y,CY), CX =< CY.
penaltyleq(X,Y) :-
        penalty(X,PX), penalty(Y,PY), PX =< PY.
```

Bibliography

[1] Antoy, S. Lazy evaluation in logic. In *PLILP 91, Passau, Germany* (1991), vol. 528 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 371–382.

[2] Apt, K. R. *From Logic Programming to Prolog.* Prentice Hall, 1997.

[3] Apt, K. R. A denotational semantics for first-order logic. In *Proc. of the Computational Logic Conference* (2000).

[4] Apt, K. R. The logic programming paradigm: A tutorial. lecture notes, May 2000.

[5] Apt, K. R., and Bezem, M. Formulas as programs. In *The Logic Programming Paradigm: a 25 Years Perspective*, K. R. Apt, V. Marek, M. Truszczynski, and D. S. Warren, Eds. Springer-Verlag, 1999.

[6] Apt, K. R., and Emden, M. H. Contributions to the theory of logic programming. *Journal of the ACM 29*, 3 (July 1982), 841–862.

[7] Arts, T., and Zantema, H. Termination of logic programs using semantic unificition. In *Proc. of the 5th Int. Workshop on Logic Program Synthesis and Transformation* (1995), vol. 1048 of *LNCS*, Springer Verlag.

[8] Asperti, A., and Martini, S. Projections instead of variables, a category theoretic interpretation of logic programs. In *Proc. of 6th ICLP* (1989), MIT Press, pp. 337–352.

[9] Backus, J. From function level semantics to program transformations and optimization. In *Mathematical foundations of software development, Vol. 1* (1985), Springer LNCS 185.

[10] Barr, M., and Wells, C. *Category Theory for Computing Science.* Prentice Hall, 1995.

[11] Baudinet, M. Proving termination properties of Prolog programs: A semantic approach. In *Third Annual Symposium on Logic in Computer Science* (Edinburgh, 1988), IEEE Computer Society, pp. 336–347.

[12] Bekkers, Y., and Tarau, P. Monadic constructs for logic programming. In *Proceedings of ILPS'95* (Portland, USA, 1995), J. Lloyd, Ed., MIT Press, pp. 51–65.

[13] Bellia, M., and Levi, G. The relation between logic and functional languages: a survey. *Journal of Logic Programming 3*, 3 (1986), 317–236.

[14] Billaud, M. Simple operational and denotational semantics for prolog with cut. *Theoretical Computer Science 71(2)* (1988), 193–208.

[15] Bird, R. Algebraic identities for program calculation. *Computer Journal* (1989).

[16] Bird, R. Functional algorithm design. *Science of Computer Programming 26* (1996), 15–31.

[17] Bird, R. *Introduction to Functional Programming using Haskell.* Prentice Hall, 1998.

[18] Bird, R. S., and de Moor, O. *Algebra of Programming.* Prentice Hall, 1997.

[19] Bosco, P. G., Giovanetti, E., and Moiso, C. Narrowing vs. sld-resolution. *Theoretical Computer Science*, 59 (1988), 3–23.

[20] Bossi, A., and Cocco, N. Preserving universal termination through unfold/fold. In *Proc. of ALP'94* (Berlin, 1994), vol. 850, LNCS, pp. 269–286.

[21] Bossi, A., Cocco, N., and Etalle, S. Transformation of left terminating programs. In *Proc. of LOPSTR'99* (Venezia, Italy, 1999), LNCS 1817, pp. 156 – 175.

[22] Bruynooghe, M., De Raedt, L., and De Schreye, D. Explanation based program transformation. In *Proc. of the 11th Intl. Conference on Artificial Intelligence* (1989), pp. 407–412.

[23] Burstall, R. M., and Darlington, J. A transformation system for developing recursive programs. *Journal of the ACM 24*, 1 (1977).

[24] Chen, W., Kifer, M., and Warren, D. S. HiLog: A foundation for higher-order logic programming. *Journal of Logic Programming 15*, 3 (1993), 187–230.

[25] Claessen, K., and Ljunglöf, P. Typed logical variables in Haskell. In *Haskell Workshop* (Montreal, Canada, 2000), Univeristy of Nottingham Technical Report.

[26] Clark, K. L. Negation as failure. In *Logic and Data Bases*, H. Gallaire and J. Minker, Eds. Plenum Press, 1978, pp. 293–322.

[27] Clark, K. L. Predicate logic as a computational formalism. Research Monograph, Imperial College, Univ. of London, 1980.

[28] Clark, K. L., and Darlington, J. Algorithm classification through synthesis. *The Computer Journal 23*, 1 (1980), 61–65.

[29] Clark, K. L., and Sickel, S. Predicate logic: a calculus for deriving programs. In *Proc. of 5th Intl. Joint Conference on Artificial Inteligence* (Cambridge, Massachusetts, 1977).

[30] Clark, K. L., and Tärnlund, S.-A., Eds. *Logic Programming.* No. 16 in A.P.I.C Studies in Data Processing. Academic Press, 1982.

[31] Clocksin, W. F. Logic-programming specification and execution of dynamic-programming problems. *Journal of Logic Programming 12*, 4 (1990).

[32] Cormen, T. H., Leiserson, C. E., and Rivest, R. L. *Introduction to algorithms.* McGraw-Hill, 1990.

[33] Corradini, A., and Asperti, A. A categorical model for logic programs: Indexed monoidal categories. In *Proc. of REX Workshop* (1992), Lecture Notes in Conputer Science, Springer.

[34] Corradini, A., and Montanari, U. An algebraic semantics for structured transition systems and its application to logic programs. *Theoretical Computer Science 103* (1992), 51–106.

[35] Crole, R. L. *Categories for Types.* Cambridge University Press, 1993.

[36] Curtis, S. *A Relational Approach to Optimization Problems.* PhD thesis, University of Oxford, 1995.

[37] Darlington, J. A synthesis of several sorting algorithms. *Acta Informatica 11*, 1 (1978).

[38] Darlington, J., Field, A. J., and Pull, H. The unification of functional and logic languages. In *Logic Programming: Functions, Relations and Equations*, D. DeGroot and G. Lindstrom, Eds. Prentice Hall, 1986, pp. 37–70.

[39] de Moor, O. *Categories, relations and dynamic programming.* PhD thesis, Computing Laboratory, Oxford, 1992.

[40] de Moor, O., and Sittampalam, G. Generic program transformation. In *Procs. 3rd International Summer School on Advanced Functional Programming* (1998).

[41] De Schreye, D., Martens, B., Sablon, G., and Bruynooghe, M. Compiling bottom-up and mixed derivations into top-down executable logic programs. Tech. rep., Katholieke Univ. Leuven, 1989.

[42] Debray, S. K., and Mishra, P. Denotational and operational semantics for Prolog. *Journal of Logic Programming 5* (1988).

[43] Diaconescu, R. *Category Semantics for Equational and Constraint Logic Programming.* PhD thesis, Oxford University, 1994.

[44] Dromey, R. G. Derivation of sorting algorithms from a specification. *The Computer Journal 30* (1987), 512–518.

[45] Erkök, L., and Launchbury, J. Recursive monad bindings. *ACM Sigplan Notices 35*, 9 (2000), 174–185.

[46] Etalle, S., and Mountjoy, J. The lazy functional side of logic programming. In *Proc. of the Int. Workshop on Logic Program Synthesis and Transformation* (2000).

[47] Felleisen, M. Transliterating Prolog into Scheme. Tech. Rep. 182, Indiana University CS Department, 1985.

[48] Freyd, P. J., and A., S. Some semantic aspects of polymorphic lambda calculus. In *Proc. of Logic in Computer Science* (1987), pp. 315–319.

[49] Fribourg, L. SLOG: A logic programming language interpreter based on on clausal superposition and rewriting. In *Proc. IEEE Internat. Symposium on Logic Programming* (1985).

[50] Friedman, D. P., Wand, M., and Haynes, C. T., Eds. *Essentials of Programming Languages.* MIT Press, 1992.

[51] Ganzinger, H., and Waldmann, U. Termination proofs of well-moded logic programs via conditional rewrite systems. In *Proc. of the 3rd International Workshop on Conditional Term Rewriting Systems* (1993), vol. 656 of *LNCS*, pp. 430–437.

[52] Gegg-Harrison, T. S. Representing logic program schemata in λ Prolog. In *Proceedings of the 12th International Conference on Logic Programming* (June 1995).

[53] Goguen, J. A., and Meseguer, J. Models and equality for logical programming. In *Proceedings, TAPSOFT87* (1987), Springer LNCS 250.

[54] Green, C., and Barstow, D. On program synthesis knowledge. *Artificial Intelligence 10* (1987), 241–279.

[55] Hamfelt, A., and Nilsson, J. F. Declarative logic programming with primitive recursive relations on lists. In *Proc. of the Joint International Conference and Symposium on Logic Programming* (1996), M. Maher, Ed., MIT Press, pp. 230–243.

[56] Hamfelt, A., and Nilsson, J. F. Towards a logic programming methodology based on higher-order predicates. *New Generation Computing 15*, 4 (1997), 421 – 448.

[57] Hamfelt, A., and Nilsson, J. F. Inductive synthesis of logic programs by composition of combinatory program scehmes. In *Proc. 8th International Workshop on Logic-based Program Synthesis and Transformation* (Manchester, United Kingdom, 1998), P. Flener, Ed.

[58] Hamfelt, A., Nilsson, J. F., and Vitoria, A. A combinatory form of pure logic programs and its compositional semantics. submitted for publication, 1998.

[59] Hanus, M. The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming 19*, 20 (1994), 583–628.

[60] Hanus, M., Kuchen, H., and Moreno-Navarro, J. J. Curry: A truly functional logic language. In *Proc. ILPS'95 Workshop on Visions for the Future of Logic Programming* (1995), pp. 95–107.

[61] Helman, P. The principle of optimality in the design of efficient algorithms. *Journal of Mathematical Analysis and Applications 119* (1986), 97–127.

[62] Henkin, L., Monk, J. D., and Tarski, A. *Cylindric Algebras, Part I.* North-Holland, 1971.

[63] Hinze, R. Prological features in a functional setting - axioms and implementations. In *Proc. of FLOPS'98* (Fuji, 1998).

[64] Hoare, C. A. R. *Communicating Sequential Processes.* Prentice Hall, 1985.

[65] Hoare, C. A. R., and He, J. *Unifying Theories of Programming.* Prentice Hall, 1998.

[66] Hogger, C. J. Derivation of logic programs. *Journal of the ACM 28*, 2 (April 1981), 372–392.

[67] Hullot, J.-M. Canonical forms and unification. In *Conference on Automated Deduction* (1980), pp. 318–334.

[68] Hussmann, H. Unification in conditional-equational theories. In *Proc. EURO-CAL'85* (1985), no. 204 in LNCS, Springer, pp. 543–553.

[69] J., L., and McGrail, R. Encapsulating data in logic programming via categorical constraints. In *Proc. of ALP'98, Principles of Declarative Programming* (1998), vol. 1490 of *LNCS*, Springer Verlag.

[70] Jones, N. D., and Mycroft, A. Stepwise development of operational and denotational semantics for Prolog. *IEEE* (1984).

[71] Krishna Rao, M. R. K., Kapur, D., and Shyamasundar, R. K. A transformation methodology for proving termination of logic programs. In *Proc. of the 5th Workshop on Computer Science Logic* (1991), Springer Verlag, pp. 213–226.

[72] Lambek, J., and Scott, P. J. *Introduction to Higher Order Categorical Logic.* Cambridge, 1986.

[73] Lau, K. K., and Prestwich, S. D. Synthesis of logic programs for recursive sorting algorithm. Tech. rep., Dept. of Computer Science, Univ. of Manchester, 1988.

[74] Launchbury, J., Lewis, J. R., and Cook, B. On embedding a michroarchitectural design language within haskell. In *ACM Sigplan Int. Conf. on Functional Programming* (1999), ACM Press, pp. 60–69.

[75] Lipton, J., Finkelstein, S. E., and Freyd, P. J. A new framework for declarative programming: Categorial perspectives. In *Proc. of ELP* (1996), pp. 209–211.

[76] Lloyd, J. W. *Foundations of Logic Programming.* Springer Verlag, 1993.

[77] Lloyd, J. W. Combining functional and logic programing languages. In *Proc. Eleventh International Logic Programming Symposium* (1994), M. Bruynooghe, Ed.

[78] Lloyd, J. W. Declarative programming in Escher. Tech. Rep. CSTR-95-013, Department of Computer Science, University of Bristol, June 1995.

[79] Lloyd, J. W. Programming in an integrated functional and logic language. *The Journal of Functional and Logic Programming* (1998). to appear.

[80] Luttringhaus, S. An interpreter with lazy evaluation for PROLOG with functions. In *Proceedings of the 2nd Workshop on Computer Science Logic* (1989), Springer LNCS 385.

[81] Manes, E. G., and Arbib, M. A. *Algebraic Approaches to Program Semantics*. Springer-Verlag, 1986.

[82] Marchiori, M. Logic programs as term rewriting systems. In *Proceedings of the 4th International Conference on Algebraic and Logic Programming* (1994), vol. 850 of *LNCS*, pp. 223–241.

[83] Martelli, A., and Montanari, U. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems 4* (1982), 258–282.

[84] McGrail, R. *Monads and Control in Logic Programming*. PhD thesis, Wesleyan UNiversity, 1997.

[85] McPhee, R. *Compositional Logic Programming*. PhD thesis, Oxford University Computing Laboratory, 2001.

[86] McPhee, R., and de Moor, O. Compositional logic programming. In *Proceedings of the JICSLP'96 post-conference workshop: Multi-paradigm logic programming* (1996), Report 96-28, Technische Universität Berlin.

[87] Meertens, L. Algorithmics — towards programming as a mathematical activity. In *Mathematics and Computer Science* (1987), J. W. De Bakker, M. Hazewinkel, and J. K. Lenstra, Eds., vol. 1 of *CWI Monographs*, North-Holland, pp. 3–42.

[88] Moggi, E. Computational lambda-calculus and monads. In *Symposium on Logic in Computer Science* (June 1989), IEEE.

[89] Moggi, E. Notions of computations and monads. *Infomation and Computation 93* (1991).

[90] Moreno-Navarro, J., and Roderiguez-Artalejo, M. Logic programming with functions and predicates: The language Babel. *Journal of Logic Programming 12*, 3 (1992), 191–223.

[91] Moreno-Navarro, J. J., Kuchen, H., Loogen, R., and Rodríguez-Artalejo, M. Lazy narrowing in a graph machine. In *Proc. Second International Conference on Algebraic and Logic Programming* (1990), LNCS 463, Springer, pp. 298–317.

[92] Narain, S. A technique for doing lazy evaluation in Prolog. *Journal of Logic Programming 3*, 3 (1986).

[93] Nilsson, J. F., and Hamfelt, A. Constructing logic programs with higher order predicates. In *Proc. Joint Conference on Declarative Programming* (1995), pp. 307–312.

[94] O'Hearn, P. W., and Tennent, R. D. Parametricity and local variables. Tech. rep., Syracuse University, 1993.

[95] Okasaki, C. *Purely Functional Data Structures.* Cambridge University Press, 1998.

[96] Ong, L. Non-determinism in functional programming. In *Proc. of the 8th IEEE Symposium on Logic and Computer Science* (1993).

[97] Panangaden, P., Sarawat, V., Scott, P. J., and Seely, R. A. G. *Lecture Notes in Computer Science*, vol. 666. Springer, 1993, ch. A Hyperdoctrinal View of Constraint Systems.

[98] Pettorossi, A., and Proietti, M. Transformation of logic programs: Foundations and techniques. *ACM Computing surveys 19* (1994).

[99] Pettorossi, A., and Proietti, M. *Handbook of Logic in Artificial Intelligence and Logic Programming*, vol. 5. Oxford University Press, 1998, ch. Transformation of Logic Programs, pp. 697–787.

[100] Pettorossi, A., and Proietti, M. Transformation rules for a higher order logic programming language. Tech. Rep. R.525, IASI - CNR, Roma, Italy, 2000.

[101] Pitt, D., Abramsky, S., Poigne, A., and Rydeheard, D., Eds. *Category Theory in Computer Science*, vol. 240 of *Lecture Notes in Computer Science.* Springer, 1985.

[102] Plotkin, G., and Reynolds, J. *Logical Foundations of Functional Programming.* Addison-Wesley, 1990, ch. On Functors Expressible in the Polymorphic Lambda Calculus.

[103] Power, J., and Kinoshita, Y. A new foundation for logic programming. In *Extensions of Logic Programming* (1996), Springer, Ed.

[104] Proietti, M., and Pettorossi, A. Semantics preserving transformation rules for Prolog. In *Proceedings of PEPM91* (1991), vol. 26 of *Sigplan Notices.*

[105] Proietti, M., and Pettorossi, A. Program derivation via list introduction. In *Proceedings of IFIP TC2 Working Conference on Algorithmic Languages and Calculi* (Le bischenberg, France, 1997).

[106] Pym, D. Functorial kripke models of the λ-calculus. In *Workshop on Category Theory and Logic Programming* (Cambridge, 1995). Lecture at Newton Institute Semantics Programme.

[107] R., B. *Dynamic Programming.* Princeton University Press, 1957.

[108] Reddy, U. S. Transformation of logic programs into functional programs. In *Proc. of ALP'90* (1984), pp. 187–196.

[109] Reddy, U. S. Narrowing as the operational semantics of functional languages. In *Symposium on Logic Programming* (Boston, 1985), IEEE.

[110] Reddy, U. S. *Logic Languages based on Functions: Semantics and Implementation.* PhD thesis, University of Utah, 1986. Technical Report UIUCDCS-R-86-1305, University of Illinois at Urbana-Champaign.

[111] Reddy, U. S. On the relationship between logic and functional languages. In *Logic Programming: Functions, Relations and Equations.* Prentice Hall, 1986, pp. 3–36.

[112] Reddy, U. S. Functional logic languages part I. In *Proceedings of a Workshop on Graph Reduction* (1987), Springer LNCS 279.

[113] Robinson, J. A. A machine-oriented logic based on the resolution principle. *Journal of the ACM 12*, 1 (January 1965), 23–41.

[114] Robinson, J. A. Beyond LogLisp: combining functional and relational programming in a reduction setting. *Machine intelligence 11* (1988), 57–68.

[115] Robinson, J. A. Memories of the past and challenges for the future. In *Computational Logic* (2000), no. 1861 in LNCS, Springer, pp. 1–24.

[116] Robinson, J. A., and Sibert, E. E. LogLisp: An alternative to Prolog. *Machine Intelligence 10* (1982), 399–419.

[117] Ross, B. J. Using algebraic semantics for proving Prolog termination and transformation. *Proceedings of the UKALP 1991* (1991).

[118] Rydeherad, D. E., and Burstall, R. M. *Category Theory and Computer Programming.* Springer, 1985, ch. A Categorical Unification Algorithm.

[119] SAGONAS, K., SWIFT, T., AND WARREN, D. S. XSB as an efficient deductive database engine. In *Proc. of ACM SIGMOD Intl Conference on the Management of Data* (Minneapolis, Minnesota, May 1994), pp. 442–453.

[120] SERES, S., AND MU, S. C. Optimisation problems in logic programming: an algebraic approach. In *Proceedings of LPSE'00* (London, UK, 2000).

[121] SERES, S., AND SPIVEY, J. M. Functional reading of logic programs. *Journal of Universal Computer Science* (2000).

[122] SERES, S., AND SPIVEY, J. M. Higher-order transformation of logic programs. In *Proceedings of LOPSTR'00* (London, UK, 2000).

[123] SERES, S., SPIVEY, J. M., AND HOARE, C. A. R. Algrebra of logic programming. In *Proceedings of ICLP'99* (Las Cruces, USA, 1999).

[124] SITTAMPALAM, G. *Higher-order matching for program transformation.* PhD thesis, University of Oxford, 2001.

[125] SLAGLE, J. R. Automated theorem-proving for theories with simplifiers, commutativity and associativity. *Journal of the ACM 21*, 4 (1974), 622–642.

[126] SMITH, D. R. The design of divide and conquer algorithms. *Science of Computer Programming 5* (1985), 37–58.

[127] SMITH, D. R. Top-down synthesis of divide-and-conquer algorithms. *Artificial Intelligence 27*, 1 (1985), 43–96.

[128] SMYTH, M. B. *Handbook of Logic in Computer Science.* Oxford University Press, 1990, ch. Topology.

[129] SOMOGYI, Z., HENDERSON, F. J., AND CONWAY, T. The implementation of Mercury: an efficient purely declarative logic programming language. In *Proceedings of the Australian Computer Science Conference* (Glenelg, Australia, February 1995), pp. 127–140. available at http://www.cs.mu.oz.au/mercury/papers.html.

[130] SPIVEY, J. M. A functional theory of exceptions. *Science of Computer Programming 14*, 1 (1990), 25–42.

[131] SPIVEY, J. M. *An Introduction to Logic Programming through Prolog.* Prentice Hall, 1996.

[132] SPIVEY, J. M. Combinators for breadth-first search. *Journal of Functional Programming* (2000). Accepted for publication.

[133] Spivey, J. M., and Seres, S. The algebra of searching. In *Festschritf in hounour of C.A.R. Hoare* (1999).

[134] Spivey, J. M., and Seres, S. Embedding Prolog in Haskell. In *Proceedings of Haskell'99* (Paris, France, 1999).

[135] Stuckey, P. Embedding constraint programming in Haskell. Private communication., 2000.

[136] Subrahmanyam, P. A., and You, J.-H. *Logic Programming, Functions, Relations and Equations.* Prentice Hall, 1986, ch. FUNLOG: a Computational Model Integrating Logic Programming and Functional Programming, pp. 157–198.

[137] Todoran, E., and Papaspyrou, N. Continuations for parallel logic programming. In *Principles and Practice of Declarative Programming* (Montreal, Canada, 2000).

[138] van Emden, M. H., and Kowalski, R. A. The semantics of predicate logic as a programming language. *Journal of the ACM 23*, 4 (October 1976), 722–742.

[139] van Raamsdonk, F. Translating logic programs into conditional rewriting systems. In *Proc. of the 14th International Conference on Logic Programming* (1997), MIT Press, pp. 168–182.

[140] Villemonte de la Clergerie, E. A tool for abstract interpretation: Dynamic programming. In *Proc. of JTASPEFL* (1991).

[141] Wadler, P. How to replace failure by a list of successes. In *2'nd International Conference on Functional Programming Languages and Computer Architecture* (Nancy, France, September 1985), Springer-Verlag.

[142] Wadler, P. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science* (1990).

[143] Wadler, P. Comprehending monads. *Mathematical Structures in Computer Science 2* (1992), 461–493.

[144] Wadler, P. The essence of functional programming. In *19'th Annual Symposium on Principles of Programming Languages* (January 1992).

[145] Wadler, P. Monads for functional programming. In *Advanced Functional Programming.* Springer LNCS 925, 1995.

[146] Wand, M. Continuation-based program transformation strategies. *Journal of the ACM 27*, 1 (1980).

[147] Wand, M. A semantic algebra for logic programming. Tech. Rep. 148, Indiana University, 1983.

[148] Zhou, N.-F. Beta-prolog: An exted prolog with boolean tables for combinatorial search. In *Proc. 5th IEEE Intl Conference on Tools with Artificial Intelligence* (November 1993), pp. 312–319.