LOGIC, PROGRAMMING AND PROLOG (2ED)

Ulf Nilsson and Jan Małuszyński

This book was previously published by John Wiley & Sons Ltd. The book was originally published in 1990 with the second edition in 1995. The copyright was reverted back to the authors in November 2000.

For further information about updates and supplementary material please check out the book web-site at

`http://www.ida.liu.se/~ulfni/lpp`

or contact the authors at `ulfni@ida.liu.se` and `janma@ida.liu.se`.

Contents

Preface

Since the first edition of this book the field of logic programming has developed and matured in many respects. This has been reflected by the large number of textbooks that appeared in that period. These books usually fall into one of the following three categories:

- books which provide a *theoretical basis* for logic programming;
- books which describe how to write programs in *Prolog* (sometimes even in particular Prolog systems);
- books which describe alternative logic programming languages like *constraint logic programming*, *deductive databases* or *concurrent logic programming*.

Objectives

The main objective of both editions of this textbook is to provide a uniform account of *both* the foundations of logic programming and simple programming techniques in the programming language Prolog. The discussion of the foundations also facilitates a systematic survey of variants of the logic programming scheme, like constraint logic programming, deductive databases or concurrent logic programming. This book is *not* primarily intended to be a theoretical handbook on logic programming. Nor is it intended to be a book on advanced Prolog programming or on constraint logic programming. For each of these topics there are more suitable books around. Because of the diversity of the field there is of course a risk that nothing substantial is said about anything. We have tried to compensate for this risk by limiting our attention to (what we think are) the most important areas of logic programming and by providing the interested reader with pointers containing suggestions for further reading. As a consequence of this:

- the theoretical presentation is limited to well-established results and many of the most elaborate theorems are stated only with hints or pointers to their proofs;
- most of the program examples are small programs whose prime aim is to illustrate the principal use of logic programming and to inspire the reader to apply similar techniques when writing "real" logic programs.

The objectives of the book have not changed since the first edition, but its content has been revised and updated to reflect the development of the field.

Prerequisites

Like many other textbooks, this book emerged out of lecture notes which finally stabilized after several years of teaching. It has been used as introductory reading in the logic programming course for third year undergraduate students mainly from the computer science curriculum at Linköping University. To take full benefit from the book, introductory courses in logic and discrete mathematics are recommended. Some basic knowledge in automata theory may be helpful but is not strictly necessary.

Organization

The book is divided into three parts:

- Foundations;
- Programming in Logic;
- Alternative Logic Programming Schemes.

The first part deals with the logical aspects of logic programming and tries to provide a logical understanding of the programming language Prolog. Logic programs consist of logical formulas and computation is the process of deduction or proof construction. This makes logic programming fundamentally different from most other programming languages, largely a consequence of the fact that logic is considerably much older than electronic computers and not restricted to the view of computation associated with the Von Neumann machine. The main difference between logic programming and conventional programming languages is the *declarative* nature of logic. A program written in, for instance, Fortran can, in general, not be understood without taking *operational* considerations into account. That is, a Fortran program cannot be understood without knowing *how* it is going to be executed. In contrast to that, logic has no inherent concept of execution and logic formulas can be understood without any notion of evaluation or execution in mind. One of the most important aims of this book is to emphasize this distinction between logic programs and programs written in traditional programming languages.

Chapter 1 contains a recapitulation of notions basic to logic in general. Readers who are already well acquainted with predicate logic can without problem omit this chapter. The chapter discusses concepts related both to model- and proof-theory of predicate logic including notions like *language*, *interpretation*, *model*, *logical consequence*, *logical inference*, *soundness* and *completeness*. The final section introduces the concept of *substitution* which is needed in subsequent chapters.

Chapter 2 introduces the restricted language of *definite programs* and discusses the model-theoretic consequences of restricting the language. By considering only definite programs it suffices to limit attention to so-called *Herbrand interpretations* making the model-theoretic treatment of the language much simpler than for the case of full predicate logic.

The operational semantics of definite programs is described in Chapter 3. The starting point is the notion of *unification*. A unification algorithm is provided and proved correct. Some of its properties are discussed. The unification algorithm is the basis for *SLD-resolution* which is the only inference rule needed for definite programs. Soundness and completeness of this rule are discussed.

The use of *negation* in logic programming is discussed in Chapter 4. It introduces the *negation-as-finite-failure* rule used to implement negation in most Prolog systems and also provides a logical justification of the rule by extending the user's program with additional axioms. Thereafter definite programs are generalized to *general* programs. The resulting proof-technique of this language is called *SLDNF-resolution* and is a result of combining SLD-resolution with the negation-as-finite-failure rule. Results concerning soundness of both the negation-as-finite-failure rule and SLDNF-resolution are discussed. Finally some alternative approaches based on three-valued logics are described to explain alternative views of negation in logic programming.

The final chapter of Part I introduces two notions available in existing Prolog systems. *Cut* is introduced as a mechanism for reducing the overhead of Prolog computations. The main objective of this section is to illustrate the effect of cut and to point out cases when its use is motivated, and cases of misuse of cut. The conclusion is that cut should be used with great care and can often be avoided. For example, cut is not used in subsequent chapters, where many example programs are presented. The second section of Chapter 5 discusses the use of predefined *arithmetic* predicates in Prolog and provides a logical explanation for them.

The second part of the book is devoted to some simple, but yet powerful, programming techniques in Prolog. The goal is not to study implementation-specific details of different Prolog systems nor is it our aim to develop real-size or highly optimized programs. The intention is rather to emphasize two basic principles which are important to appreciate before one starts considering writing "real" programs:

- logic programs are used to describe *relations*, and
- logic programs have both a declarative and an operational meaning. In order to write good programs it is important to keep both aspects in mind.

Part II of the book is divided into several chapters which relate logic programming to different fields of computer science while trying to emphasize these two points.

Chapter 6 describes logic programming from a *database* point of view. It is shown how logic programs can be used, in a coherent way, as a framework for representing relational databases and for retrieving information out of them. The chapter also contains some extensions to traditional databases. For instance, the ability to define infinite relations and the use of structured data.

Chapter 7 demonstrates techniques for defining relations on *recursive data-structures*, in particular on *lists*. The objective is to study how recursive data-structures give rise to recursive programs which can be defined in a uniform way by means of inductive definitions. The second part of the chapter presents an alternative representation of lists and discusses advantages and disadvantages of this new representation.

Chapter 8 introduces the notion of *meta*- and *object*-language and illustrates how to use logic programs for describing SLD-resolution. The ability to do this in a simple way facilitates some very powerful programming techniques. The chapter also introduces some (controversial) built-in predicates available in most Prolog implementations.

Chapter 9 is a continuation of Chapter 8. It demonstrates how to extend an interpreter from Chapter 8 into a simple *expert-system* shell. The resulting program can be used as a starting point for developing a full-scale expert system.

Historically one of the main objectives for implementing Prolog was its application for natural language processing. Chapter 10 shows how to describe grammars in Prolog, starting from context-free grammars. Thereafter larger classes of languages are considered. The last two sections introduce the notion of *Definite Clause Grammars* (DCGs) commonly used for describing both natural and artificial languages in Prolog.

The last chapter of Part II elaborates on results from Chapter 6. The chapter demonstrates simple techniques for solving search-problems in state-transition graphs and raises some of the difficulties which are inherently associated with such problems.

The final part of the book gives a brief introduction to some extensions of the logic programming paradigm, which are still subject of active research.

Chapter 12 describes a class of languages commonly called *concurrent logic programming languages*. The underlying execution model of these languages is based on concurrent execution. It allows therefore for applications of logic programming for description of concurrent processes. The presentation concentrates on the characteristic principles of this class of languages, in particular on the mechanisms used to enforce *synchronization* between parallel processes and the notion of *don't care nondeterminism*.

Chapter 13 discusses an approach to integration of logic programming with *functional* programming based on the use of *equations*. The notion of *E-unification* (unification modulo a set E of equations) is introduced and properties of E-unification algorithms are discussed. Finally it is shown how to generalize the notion of SLD-resolution to incorporate E-unification instead of "ordinary" unification.

Chapter 14 concerns the use of *constraints* in logic programming. The constraint logic programming scheme has attracted a great many people because of its generality, elegance and expressive power. A rigorous semantical framework is briefly described. The main ideas are illustrated using examples from several constraint domains.

The final chapter of Part III concerns the optimization of queries to deductive databases. The chapter provides an alternative to SLD-resolution as the inference mechanism in a query-answering system and discusses the principal idea of several optimizations described in the literature.

In addition the book contains three appendices. The first of them provides bibliographical remarks to most of the chapters of the book including suggestions for further reading. The second appendix contains a brief account of set theoretic notions used throughout the book and the final appendix contains solutions and hints for some of the exercises which are available in the main text.

What is new in the second edition?

The second edition of the book contains one new chapter on query optimization in deductive databases (Chapter 15). Three chapters have also been substantially revised: The presentation of unification in Chapter 3 has been modified to facilitate better integration with Chapters 13 (equational logic programming) and 14 (constraint logic programming). To simplify the presentation of constraint logic programming, Chapter 3 also introduces the notion of derivation trees. Secondly, chapter 4 on negation has been completely revised. In particular, the definition of SLDNF-resolution has been improved and two new sections have been added covering alternative approaches to negation — three-valued completion and well-founded semantics. Finally, Chapter 14 has been substantially extended providing the theoretical foundation of the constraint logic programming scheme and several examples of constraint logic programming languages. Most of the remaining chapters have undergone minor modifications; new examples and exercises have been included, the bibliographical remarks have been updated and an appendix on basic set theory has been added.

Acknowledgements

The authors would like to thank a number of persons for their involvement in the course of writing the first and second edition of this book. In particular, Roland Bol, Staffan Bonnier, Lars Degerstedt, Włodzimierz Drabent and all other members of the Logic Programming Laboratory. We are also indebted to students, who lived through draft versions of the book and provided invaluable feedback. Thanks are also due to Gu Xinli, Jalal Maleki, Mirka Miłkowska, Simin Nadjm-Tehrani, Torbjörn Näslund and Linda Smith who devoted much of their time reading parts of the manuscript. Needless to say, the remaining flaws are to be attributed to the authors.

Our deepest gratitude also to Roslyn Meredith and Rosemary Altoft at John Wiley, and the anonymous referees whose comments influenced the final structure and contents of both editions of the book.

Finally we should mention that the material presented in this book is closely related to our research interests. We gratefully acknowledge the financial support of our research projects by the Swedish Research Council for Engineering Sciences (TFR) and by Linköping University.

Linköping, Sweden
June 1995

Ulf Nilsson
Jan Małuszyński

PART I
FOUNDATIONS

Chapter 1

Preliminaries

1.1 Logic Formulas

When describing some state of affairs in the real world we often use *declarative*[1] sentences like:

(i) "Every mother loves her children"

(ii) "Mary is a mother and Tom is Mary's child"

By applying some general rules of reasoning such descriptions can be used to draw new conclusions. For example, knowing (i) and (ii) it is possible to conclude that:

(iii) "Mary loves Tom"

A closer inspection reveals that (i) and (ii) describe some *universe* of persons and some *relations* between these individuals — like "... is a mother", "... is a child of ..." or the relation "... loves ..." — which may or may not hold between the persons.[2] This example reflects the principal idea of *logic programming* — to describe possibly infinite relations on objects and to apply the programming system in order to draw conclusions like (iii).

For a computer to deal with sentences like (i)–(iii) the *syntax* of the sentences must be precisely defined. What is even more important, the rules of reasoning — like the one

[1]The notion of declarative sentence has its roots in linguistics. A declarative sentence is a complete expression of natural language which is either true or false, as opposed to e.g. imperative or interrogative sentences (commands and questions). Only declarative sentences can be expressed in predicate logic.

[2]Some people would probably argue that "being a mother" is not a relation but rather a property. However, for the sake of uniformity properties will be called relations and so will statements which relate more than two objects (like "... is the sum of ... and ...").

which permits inferring (*iii*) from (*i*) and (*ii*) — must be carefully formalized. Such problems have been studied in the field of mathematical logic. This chapter surveys basic logical concepts that are used later on in the book to relate logic programming and logic. (For basic set theoretic notions see Appendix B.)

The first concept considered is that of *logic formulas* which provide a formalized syntax for writing sentences like (*i*)–(*iii*). Such sentences refer to *individuals* in some *world* and to *relations* between those individuals. Therefore the starting point is an assumption about the alphabet of the language. It must include:

- symbols for denoting individuals (e.g. the symbol *tom* may be used to denote the person Tom of our example). Such symbols will be called *constants*;

- symbols for denoting relations (*loves*, *mother*, *child_of*). Such symbols are called *predicate symbols.*

Every predicate symbol has an associated natural number, called its arity. The relation named by an n-ary predicate symbol is a set of n-tuples of individuals; in the example above the predicate symbol *loves* denotes a set of pairs of persons, including the pair Mary and Tom, denoted by the constants *mary* and *tom*.

With the alphabet of constants, predicate symbols and some auxiliary characters, sentences of natural language like "Mary loves Tom" can be formalized as formulas like $loves(mary, tom)$.

The formal language should also provide the possibility of expressing sentences like (*i*) which refers to *all* elements of the described "world". This sentence says that "for all individuals X and Y, if X is a mother and Y is a child of X then X loves Y". For this purpose, the language of logic introduces the symbol of *universal quantifier* "$\forall$" (to be read "for every" or "for all") and the alphabet of *variables.* A variable is a symbol that refers to an unspecified individual, like X and Y above. Now the sentences (*i*)–(*iii*) can be formalized accordingly:

$$\forall X\ (\forall Y\ ((mother(X) \land child\_of(Y, X)) \supset loves(X, Y))) \tag{1}$$

$$mother(mary) \land child\_of(tom, mary) \tag{2}$$

$$loves(mary, tom) \tag{3}$$

The symbols "$\land$" and "$\supset$" are examples of *logical connectives* which are used to combine logic formulas — "$\land$" reads "and" and is called *conjunction* whereas "$\supset$" is called *implication* and corresponds to the "if-then" construction above. Parentheses are used to disambiguate the language.

Another connective which will be used frequently is that for expressing negation. It is denoted by "$\neg$" (with reading "not"). For example the sentence "Tom does not love Mary" can be formalized as the formula:

$$\neg loves(tom, mary)$$

In what follows the symbol "$\exists$" is also sometimes used. It is called the *existential quantifier* and reads "there exists". The existential quantifier makes it possible to express the fact that, in the world under consideration, there exists at least one individual which is in a certain relation with some other individuals. For example the sentence "Mary has a child" can be formalized as the formula:

$$\exists X \; \mathit{child\_of}(X, \mathit{mary})$$

On occasion the logical connectives "$\vee$" and "$\leftrightarrow$" are used. They formalize the connectives "or" and "if and only if" ("iff").

So far individuals have been represented only by constants. However it is often the case that in the world under consideration, some "individuals" are "composed objects". For instance, in some world it may be necessary to discuss relations between families as well as relations between persons. In this case it would be desirable to refer to a given family by a construction composed of the constants identifying the members of the family (actually what is needed is a *function* that constructs a family from its members). The language of logic offers means of solving this problem. It is assumed that its alphabet contains symbols called *functors* that represent functions over object domains. Every functor has assigned a natural number called its arity, which determines the number of arguments of the function. The constants can be seen as 0-ary functors. Assume now that there is a ternary[3] functor *family*, a binary functor *child* and a constant *none*. The family consisting of the parents Bill and Mary and children Tom and Alice can now be represented by the construction:

$$\mathit{family}(\mathit{bill}, \mathit{mary}, \mathit{child}(\mathit{tom}, \mathit{child}(\mathit{alice}, \mathit{none})))$$

Such a construction is called a *compound term*.

The above informal discussion based on examples of simple declarative sentences gives motivation for introducing basic constructs of the language of symbolic logic. The kind of logic used here is called *predicate logic*. Next a formal definition of this language is given. For the moment we specify only the form of allowed sentences, while the meaning of the language will be discussed separately. Thus the definition covers only the *syntax* of the language separated from its *semantics*.

From the syntactic point of view logic formulas are finite sequences of symbols such as variables, functors and predicate symbols. There are infinitely many of them and therefore the symbols are usually represented by finite strings of primitive characters. The representation employed in this book usually conforms to that specified in the ISO standard of the programming language Prolog (1995). Thus, the *alphabet* of the language of predicate logic consists of the following classes of symbols:

- *variables* which will be written as alphanumeric identifiers beginning with capital letters (sometimes subscriped). Examples of variables are $X, Xs, Y, X_7, \ldots$;
- *constants* which are numerals or alphanumeric identifiers beginning with lower-case letters. Examples of constants are x, $\mathit{alf}, \mathit{none}, 17, \ldots$;
- *functors* which are alphanumeric identifiers beginning with lower-case letters and with an associated arity > 0. To emphasize the arity n of a functor f it is sometimes written in the form f/n;

[3]Usually the terms *nullary*, *unary*, *binary* and *ternary* are used instead of 0-ary, 1-ary, 2-ary and 3-ary.

- *predicate symbols* which are usually alphanumeric identifiers starting with lower-case letters and with an associated arity ≥ 0. The notation p/n is used also for predicate symbols;
- *logical connectives* which are $\wedge$ (conjunction), $\neg$ (negation), $\leftrightarrow$ (logical equivalence), $\supset$ (implication) and $\vee$ (disjunction);
- *quantifiers* — $\forall$ (universal) and $\exists$ (existential);
- *auxiliary* symbols like parentheses and commas.

No syntactic distinction will be imposed between constants, functors and predicate symbols. However, as a notational convention we use $a, b, c, \ldots$ (with or without adornments) to denote constants and $X, Y, Z, \ldots$ to denote variables. Functors are denoted $f, g, h, \ldots$ and $p, q, r, \ldots$ are used to denote predicate symbols. Constants are sometimes viewed as nullary functors. Notice also that the sets of functors and predicate symbols may contain identical identifiers with different arities.

Sentences of natural language consist of words where objects of the described world are represented by nouns. In the formalized language of predicate logic objects will be represented by strings called *terms* whose syntax is defined as follows:

Definition 1.1 (Terms) The set $\mathcal{T}$ of *terms* over a given alphabet $\mathcal{A}$ is the smallest set such that:

- any constant in $\mathcal{A}$ is in $\mathcal{T}$;
- any variable in $\mathcal{A}$ is in $\mathcal{T}$;
- if f/n is a functor in $\mathcal{A}$ and $t_1, \ldots, t_n \in \mathcal{T}$ then $f(t_1, \ldots, t_n) \in \mathcal{T}$.

∎

In this book terms are typically denoted by s and t.

In natural language only certain combinations of words are meaningful sentences. The counterpart of sentences in predicate logic are special constructs built from terms. These are called *formulas* or well-formed formulas (*wff*) and their syntax is defined as follows:

Definition 1.2 (Formulas) Let $\mathcal{T}$ be the set of terms over the alphabet $\mathcal{A}$. The set $\mathcal{F}$ of *wff* (with respect to $\mathcal{A}$) is the smallest set such that:

- if p/n is a predicate symbol in $\mathcal{A}$ and $t_1, \ldots, t_n \in \mathcal{T}$ then $p(t_1, \ldots, t_n) \in \mathcal{F}$;
- if F and $G \in \mathcal{F}$ then so are $(\neg F)$, $(F \wedge G)$, $(F \vee G)$, $(F \supset G)$ and $(F \leftrightarrow G)$;
- if $F \in \mathcal{F}$ and X is a variable in $\mathcal{A}$ then $(\forall X F)$ and $(\exists X F) \in \mathcal{F}$.

∎

Formulas of the form $p(t_1, \ldots, t_n)$ are called *atomic formulas* (or simply *atoms*).

In order to adopt a syntax similar to that of Prolog, formulas in the form $(F \supset G)$ are instead written in the form $(G \leftarrow F)$. To simplify the notation parentheses will be removed whenever possible. To avoid ambiguity it will be assumed that the connectives have a binding-order where $\neg$, $\forall$ and $\exists$ bind stronger than $\vee$, which in turn binds stronger than $\wedge$ followed by $\supset$ (i.e. $\leftarrow$) and finally $\leftrightarrow$. Thus $(a \leftarrow ((\neg b) \wedge c))$ will be simplified into $a \leftarrow \neg b \wedge c$. Sometimes binary functors and predicate symbols are written in infix notation (e.g. $2 \leq 3$).

Let F be a formula. An occurrence of the variable X in F is said to be *bound* either if the occurrence follows directly after a quantifier or if it appears inside the subformula which follows directly after "$\forall X$" or "$\exists X$". Otherwise the occurrence is said to be *free*. A formula with no free occurrences of variables is said to be *closed*. A formula/term which contains no variables is called *ground*.

Let $X_1, \ldots, X_n$ be all variables that occur free in a formula F. The closed formula of the form $\forall X_1(\ldots(\forall X_n\, F)\ldots)$ is called the *universal closure* of F and is denoted $\forall\, F$. Similarly, $\exists\, F$ is called the *existential closure* of F and denotes the formula F closed under existential quantification.

1.2 Semantics of Formulas

The previous section introduced the language of formulas as a formalization of a class of declarative statements of natural language. Such sentences refer to some "world" and may be true or false in this world. The meaning of a logic formula is also defined relative to an "abstract world" called an (algebraic) *structure* and is also either true or false. In other words, to define the meaning of formulas, a formal connection between the language and a structure must be established. This section discusses the notions underlying this idea.

As stated above declarative statements refer to individuals, and concern relations and functions on individuals. Thus the mathematical abstraction of the "world", called a structure, is a nonempty set of individuals (called the *domain*) with a number of relations and functions defined on this domain. For example the structure referred to by the sentences (*i*)–(*iii*) may be an abstraction of the world shown in Figure 1.1. Its domain consists of three individuals — Mary, John and Tom. Moreover, three relations will be considered on this set: a unary relation, "... is a mother", and two binary relations, "... is a child of ..." and "... loves ...". For the sake of simplicity it is assumed that there are no functions in the structure.

The building blocks of the language of formulas are constants, functors and predicate symbols. The link between the language and the structure is established as follows:

Definition 1.3 (Interpretation) An interpretation $\Im$ of an alphabet $\mathcal{A}$ is a non-empty domain $\mathcal{D}$ (sometimes denoted $|\Im|$) and a mapping that associates:

- each constant $c \in \mathcal{A}$ with an element $c_\Im \in \mathcal{D}$;
- each n-ary functor $f \in \mathcal{A}$ with a function $f_\Im : \mathcal{D}^n \to \mathcal{D}$;
- each n-ary predicate symbol $p \in \mathcal{A}$ with a relation $p_\Im \subseteq \underbrace{\mathcal{D} \times \cdots \times \mathcal{D}}_{n}$.

∎

The interpretation of constants, functors and predicate symbols provides a basis for assigning truth values to formulas of the language. The meaning of a formula will be

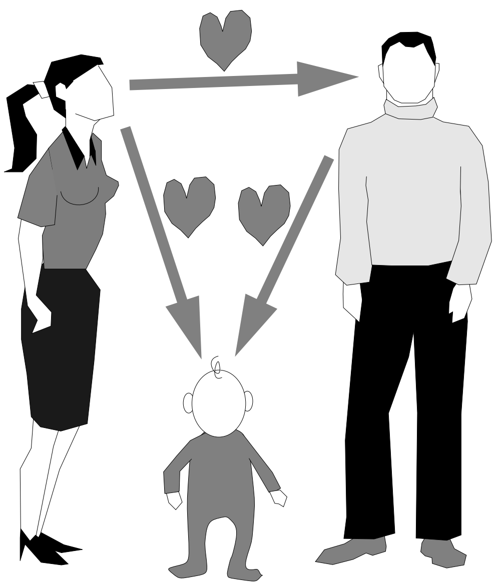


Figure 1.1: A family structure

defined as a function on meanings of its components. First the meaning of terms will be defined since they are components of formulas. Since terms may contain variables the auxiliary notion of *valuation* is needed. A valuation φ is a mapping from variables of the alphabet to the domain of an interpretation. Thus, it is a function which assigns objects of an interpretation to variables of the language. By the notation $\varphi[X \mapsto t]$ we denote the valuation which is identical to φ except that $\varphi[X \mapsto t]$ maps X to t.

Definition 1.4 (Semantics of terms) Let $\Im$ be an interpretation, φ a valuation and t a term. Then the *meaning* $\varphi_\Im(t)$ of t is an element in $|\Im|$ defined as follows:

- if t is a constant c then $\varphi_\Im(t) := c_\Im$;
- if t is a variable X then $\varphi_\Im(t) := \varphi(X)$;
- if t is of the form $f(t_1, \ldots, t_n)$, then $\varphi_\Im(t) := f_\Im(\varphi_\Im(t_1), \ldots, \varphi_\Im(t_n))$.

∎

Notice that the meaning of a compound term is obtained by applying the function denoted by its main functor to the meanings of its principal subterms, which are obtained by recursive application of this definition.

Example 1.5 Consider a language which includes the constant *zero*, the unary functor s and the binary functor *plus*. Assume that the domain of $\Im$ is the set of the natural numbers ($\mathbb{N}$) and that:

$$zero_\Im \quad := \quad 0$$

$$\begin{array}{rcl} s_{\Im}(x) & := & 1+x \\ plus_{\Im}(x,y) & := & x+y \end{array}$$

That is, *zero* denotes the *natural number* 0, s denotes the successor function and *plus* denotes the addition function. For the interpretation $\Im$ and a valuation φ such that $\varphi(X) := 0$ the meaning of the term $plus(s(zero), X)$ is obtained as follows:

$$\begin{array}{rcl} \varphi_{\Im}(plus(s(zero),X)) & = & \varphi_{\Im}(s(zero)) + \varphi_{\Im}(X) \\ & = & (1+\varphi_{\Im}(zero)) + \varphi(X) \\ & = & (1+0)+0 \\ & = & 1 \end{array}$$

∎

The meaning of a formula is a truth value. The meaning depends on the components of the formula which are either (sub-) formulas or terms. As a consequence the meanings of formulas also rely on valuations. In the following definition the notation $\Im \models_{\varphi} Q$ is used as a shorthand for the statement "Q is true with respect to $\Im$ and φ" and $\Im \not\models_{\varphi} Q$ is to be read "Q is false w.r.t. $\Im$ and φ".

Definition 1.6 (Semantics of wff's) Let $\Im$ be an interpretation, φ a valuation and Q a formula. The meaning of Q w.r.t. $\Im$ and φ is defined as follows:

- $\Im \models_{\varphi} p(t_1, \ldots, t_n)$ iff $\langle \varphi_{\Im}(t_1), \ldots, \varphi_{\Im}(t_n) \rangle \in p_{\Im}$;
- $\Im \models_{\varphi} (\neg F)$ iff $\Im \not\models_{\varphi} F$;
- $\Im \models_{\varphi} (F \wedge G)$ iff $\Im \models_{\varphi} F$ and $\Im \models_{\varphi} G$;
- $\Im \models_{\varphi} (F \vee G)$ iff $\Im \models_{\varphi} F$ or $\Im \models_{\varphi} G$ (or both);
- $\Im \models_{\varphi} (F \supset G)$ iff $\Im \models_{\varphi} G$ whenever $\Im \models_{\varphi} F$;
- $\Im \models_{\varphi} (F \leftrightarrow G)$ iff $\Im \models_{\varphi} (F \supset G)$ and $\Im \models_{\varphi} (G \supset F)$;
- $\Im \models_{\varphi} (\forall X F)$ iff $\Im \models_{\varphi[X \mapsto t]} F$ for every $t \in |\Im|$;
- $\Im \models_{\varphi} (\exists X F)$ iff $\Im \models_{\varphi[X \mapsto t]} F$ for some $t \in |\Im|$.

∎

The semantics of formulas as defined above relies on the auxiliary concept of valuation that associates variables of the formula with elements of the domain of the interpretation. It is easy to see that the truth value of a closed formula depends only on the interpretation. It is therefore common practice in logic programming to consider all formulas as being implicitly universally quantified. That is, whenever there are free occurrences of variables in a formula its universal closure is considered instead. Since the valuation is of no importance for closed formulas it will be omitted when considering the meaning of such formulas.

Example 1.7 Consider Example 1.5 again. Assume that the language contains also a unary predicate symbol p and that:

$$p_{\Im} := \{\langle 1\rangle, \langle 3\rangle, \langle 5\rangle, \langle 7\rangle, \ldots\}$$

Then the meaning of the formula $p(zero) \wedge p(s(zero))$ in the interpretation $\Im$ is determined as follows:

$$\begin{array}{rll} \Im \models p(zero) \wedge p(s(zero)) & \text{iff} & \Im \models p(zero) \text{ and } \Im \models p(s(zero)) \\ & \text{iff} & \langle \varphi_{\Im}(zero)\rangle \in p_{\Im} \text{ and } \langle \varphi_{\Im}(s(zero))\rangle \in p_{\Im} \\ & \text{iff} & \langle \varphi_{\Im}(zero)\rangle \in p_{\Im} \text{ and } \langle 1 + \varphi_{\Im}(zero)\rangle \in p_{\Im} \\ & \text{iff} & \langle 0\rangle \in p_{\Im} \text{ and } \langle 1\rangle \in p_{\Im} \end{array}$$

Now $\langle 1\rangle \in p_{\Im}$ but $\langle 0\rangle \notin p_{\Im}$ so the whole formula is false in $\Im$. ▮

Example 1.8 Consider the interpretation $\Im$ that assigns:

- the persons Tom, John and Mary of the structure in Figure 1.1 to the constants *tom*, *john* and *mary*;
- the relations "... is a mother", "... is a child of ..." and "... loves ..." of the structure in Figure 1.1 to the predicate symbols *mother*/1, *child_of*/2 and *loves*/2.

Using the definition above it is easy to show that the meaning of the formula:

$$\forall X\, \exists Y\, loves(X, Y)$$

is false in $\Im$ (since Tom does not love anyone), while the meaning of formula:

$$\exists X\, \forall Y\, \neg loves(Y, X)$$

is true in $\Im$ (since Mary is not loved by anyone). ▮

1.3 Models and Logical Consequence

The motivation for introducing the language of formulas was to give a tool for describing "worlds" — that is, algebraic structures. Given a set of closed formulas P and an interpretation $\Im$ it is natural to ask whether the formulas of P give a proper account of this world. This is the case if all formulas of P are true in $\Im$.

Definition 1.9 (Model) An interpretation $\Im$ is said to be a *model* of P iff every formula of P is true in $\Im$. ▮

Clearly P has infinitely many interpretations. However, it may happen that none of them is a model of P. A trivial example is any P that includes the formula $(F \wedge \neg F)$ where F is an arbitrary (closed) formula. Such sets of formulas are called *unsatisfiable*. When using formulas for describing "worlds" it is necessary to make sure that every description produced is *satisfiable* (that is, has at least one model), and in particular that the world being described is a model of P.

Generally, a satisfiable set of formulas has (infinitely) many models. This means that the formulas which properly describe a particular "world" of interest at the same time describe many other worlds.

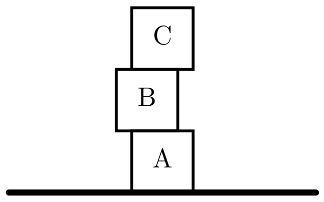


Figure 1.2: An alternative structure

Example 1.10 Figure 1.2 shows another structure which can be used as a model of the formulas (1) and (2) of Section 1.1 which were originally used to describe the world of Figure 1.1. In order for the structure to be a model the constants *tom*, *john* and *mary* are interpreted as the boxes 'A', 'B' and 'C' respectively — the predicate symbols *loves*, *child_of* and *mother* are interpreted as the relations "... is above ...", "... is below ..." and "... is on top". ▮

Our intention is to use the description of the world of interest to obtain more information about this world. This new information is to be represented by new formulas not explicitly included in the original description. An example is the formula (3) of Section 1.1 which is obtained from (1) and (2). In other words, for a given set P of formulas other formulas (say F) which are also true in the world described by P are searched for. Unfortunately, P itself has many models and does not uniquely identify the "intended model" which was described by P. Therefore it must be required that F is true in every model of P to guarantee that it is also true in the particular world of interest. This leads to the fundamental concept of *logical consequence.*

Definition 1.11 (Logical consequence) Let P be a set of closed formulas. A closed formula F is called a logical consequence of P (denoted $P \models F$) iff F is true in every model of P. ▮

Example 1.12 To illustrate this notion by an example it is shown that (3) is a logical consequence of (1) and (2). Let $\Im$ be an arbitrary interpretation. If $\Im$ is a model of (1) and (2) then:

$$\Im \models \forall X(\forall Y((\mathit{mother}(X) \land \mathit{child\_of}(Y,X)) \supset \mathit{loves}(X,Y))) \quad (4)$$

$$\Im \models \mathit{mother}(\mathit{mary}) \land \mathit{child\_of}(\mathit{tom},\mathit{mary}) \quad (5)$$

For (4) to be true it is necessary that:

$$\Im \models_\varphi \mathit{mother}(X) \land \mathit{child\_of}(Y,X) \supset \mathit{loves}(X,Y) \quad (6)$$

for any valuation φ — specifically for $\varphi(X) = \mathit{mary}_\Im$ and $\varphi(Y) = \mathit{tom}_\Im$. However, since these individuals are denoted by the constants *mary* and *tom* it must also hold that:

$$\Im \models \mathit{mother}(\mathit{mary}) \land \mathit{child\_of}(\mathit{tom},\mathit{mary}) \supset \mathit{loves}(\mathit{mary},\mathit{tom}) \quad (7)$$

Finally, for this to hold it follows that $\mathit{loves}(\mathit{mary},\mathit{tom})$ must be true in $\Im$ (by Definition 1.6 and since (5) holds by assumption). Hence, any model of (1) and (2) is also a model of (3). ▮

This example shows that it may be rather difficult to prove that a formula is a logical consequence of a set of formulas. The reason is that one has to use the semantics of the language of formulas and to deal with all models of the formulas.

One possible way to prove $P \models F$ is to show that $\neg F$ is false in every model of P, or put alternatively, that the set of formulas $P \cup \{\neg F\}$ is unsatisfiable (has no model). The proof of the following proposition is left as an exercise.

Proposition 1.13 (Unsatisfiability) Let P be a set of closed formulas and F a closed formula. Then $P \models F$ iff $P \cup \{\neg F\}$ is unsatisfiable. ∎

It is often straightforward to show that a formula F is not a logical consequence of the set P of formulas. For this, it suffices to give a model of P which is not a model of F.

Example 1.14 Let P be the formulas:

$$\forall X(r(X) \supset (p(X) \vee q(X))) \tag{8}$$

$$r(a) \wedge r(b) \tag{9}$$

To prove that $p(a)$ is not a logical consequence of P it suffices to consider an interpretation $\Im$ where $|\Im|$ is the set consisting of the two persons "Adam" and "Eve" and where:

$$\begin{array}{ll}
a_\Im := \text{Adam} & \\
b_\Im := \text{Eve} & \\
p_\Im := \{\langle\text{Eve}\rangle\} & \text{\% the property of being female} \\
q_\Im := \{\langle\text{Adam}\rangle\} & \text{\% the property of being male} \\
r_\Im := \{\langle\text{Adam}\rangle, \langle\text{Eve}\rangle\} & \text{\% the property of being a person}
\end{array}$$

Clearly, (8) is true in $\Im$ since "any person is either female or male". Similarly (9) is true since "both Adam and Eve are persons". However, $p(a)$ is false in $\Im$ since Adam is not a female. ∎

Another important concept based on the semantics of formulas is the notion of *logical equivalence.*

Definition 1.15 (Logical equivalence) Two formulas F and G are said to be logically equivalent (denoted $F \equiv G$) iff F and G have the same truth value for all interpretations $\Im$ and valuations φ. ∎

Next a number of well-known facts concerning equivalences of formulas are given. Let F and G be arbitrary formulas and $H(X)$ a formula with zero or more free occurrences of X. Then:

$$\begin{array}{rcll}
\neg\neg F & \equiv & F & \\
F \supset G & \equiv & \neg F \vee G & \\
F \supset G & \equiv & \neg G \supset \neg F & \\
F \leftrightarrow G & \equiv & (F \supset G) \wedge (G \supset F) & \\
\neg(F \vee G) & \equiv & \neg F \wedge \neg G & \text{DeMorgan's law} \\
\neg(F \wedge G) & \equiv & \neg F \vee \neg G & \text{DeMorgan's law} \\
\neg\forall X H(X) & \equiv & \exists X \neg H(X) & \text{DeMorgan's law} \\
\neg\exists X H(X) & \equiv & \forall X \neg H(X) & \text{DeMorgan's law}
\end{array}$$

and if there are no free occurrences of X in F then:

$$\forall X(F \vee H(X)) \equiv F \vee \forall X H(X)$$

Proofs of these equivalences are left as an exercise to the reader.

1.4 Logical Inference

In Section 1.1 the sentence (*iii*) was obtained by reasoning about the sentences (*i*) and (*ii*). The language was then formalized and the sentences were expressed as the logical formulas (1), (2) and (3). With this formalization, reasoning can be seen as a process of manipulation of formulas, which from a given set of formulas, like (1) and (2), called the *premises*, produces a new formula called the *conclusion*, for instance (3). One of the objectives of the symbolic logic is to formalize "reasoning principles" as formal re-write rules that can be used to generate new formulas from given ones. These rules are called *inference rules*. It is required that the inference rules correspond to correct ways of reasoning — whenever the premises are true in any world under consideration, any conclusion obtained by application of an inference rule should also be true in this world. In other words it is required that the inference rules produce only logical consequences of the premises to which they can be applied. An inference rule satisfying this requirement is said to be *sound*.

Among well-known inference rules of predicate logic the following are frequently used:

- *Modus ponens* or elimination rule for implication: This rule says that whenever formulas of the form F and $(F \supset G)$ belong to or are concluded from a set of premises, G can be inferred. This rule is often presented as follows:

$$\frac{F \quad F \supset G}{G} \quad (\supset \mathrm{E})$$

- Elimination rule for universal quantifier: This rule says that whenever a formula of the form $(\forall X F)$ belongs to or is concluded from the premises a new formula can be concluded by replacing all free occurrences of X in F by some term t which is *free for* X (that is, all variables in t remain free when X is replaced by t: for details see e.g. van Dalen (1983) page 68). This rule is often presented as follows:

$$\frac{\forall X F(X)}{F(t)} \quad (\forall \mathrm{E})$$

- Introduction rule for conjunction: This rule states that if formulas F and G belong to or are concluded from the premises then the conclusion $F \wedge G$ can be inferred. This is often stated as follows:

$$\frac{F \quad G}{F \wedge G} \quad (\wedge \mathrm{I})$$

Soundness of these rules can be proved directly from the definition of the semantics of the language of formulas.

Their use can be illustrated by considering the example above. The premises are:

$$\forall X\ (\forall Y\ (mother(X) \land child\_of(Y, X) \supset loves(X, Y))) \tag{10}$$
$$mother(mary) \land child\_of(tom, mary) \tag{11}$$

Elimination of the universal quantifier in (10) yields:

$$\forall Y\ (mother(mary) \land child\_of(Y, mary) \supset loves(mary, Y)) \tag{12}$$

Elimination of the universal quantifier in (12) yields:

$$mother(mary) \land child\_of(tom, mary) \supset loves(mary, tom) \tag{13}$$

Finally *modus ponens* applied to (11) and (13) yields:

$$loves(mary, tom) \tag{14}$$

Thus the conclusion (14) has been produced in a formal way by application of the inference rules. The example illustrates the concept of *derivability*. As observed, (14) is obtained from (10) and (11) not directly, but in a number of inference steps, each of them adding a new formula to the initial set of premises. Any formula F that can be obtained in that way from a given set P of premises is said to be *derivable* from P. This is denoted by $P \vdash F$. If the inference rules are sound it follows that whenever $P \vdash F$, then $P \models F$. That is, whatever can be derived from P is also a logical consequence of P. An important question related to the use of inference rules is the problem of whether all logical consequences of an arbitrary set of premises P can also be derived from P. In this case the set of inference rules is said to be *complete*.

Definition 1.16 (Soundness and Completeness) A set of inference rules are said to be *sound* if, for every set of closed formulas P and every closed formula F, whenever $P \vdash F$ it holds that $P \models F$. The inference rules are *complete* if $P \vdash F$ whenever $P \models F$. ▮

A set of premises is said to be *inconsistent* if any formula can be derived from the set. Inconsistency is the proof-theoretic counterpart of unsatisfiability, and when the inference system is both sound and complete the two are frequently used as synonyms.

1.5 Substitutions

The chapter is concluded with a brief discussion on *substitutions* — a concept fundamental to forthcoming chapters. Formally a substitution is a mapping from variables of a given alphabet to terms in this alphabet. The following syntactic definition is often used instead:

Definition 1.17 (Substitutions) A *substitution* is a finite set of pairs of terms $\{X_1/t_1, \dots, X_n/t_n\}$ where each t_i is a term and each X_i a variable such that $X_i \neq t_i$ and $X_i \neq X_j$ if $i \neq j$. The *empty substitution* is denoted ϵ. ▮

The *application* $X\theta$ of a substitution θ to a variable X is defined as follows:

$$X\theta \quad := \quad \begin{cases} t \text{ if } X/t \in \theta. \\ X \text{ otherwise} \end{cases}$$

In what follows let $Dom(\{X_1/t_1, \ldots, X_n/t_n\})$ denote the set $\{X_1, \ldots, X_n\}$. Also let $Range(\{X_1/t_1, \ldots, X_n/t_n\})$ be the set of all variables in $t_1, \ldots, t_n$. Thus, for variables not included in $Dom(\theta)$, θ behaves as the identity mapping. It is natural to extend the domain of substitutions to include also terms and formulas. In other words, it is possible to *apply* a substitution to an arbitrary term or formula in the following way:

Definition 1.18 (Application) Let θ be a substitution $\{X_1/t_1, \ldots, X_n/t_n\}$ and E a term or a formula. The application $E\theta$ of θ to E is the term/formula obtained by simultaneously replacing t_i for every free occurrence of X_i in E $(1 \leq i \leq n)$. $E\theta$ is called an *instance* of E. ▮

Example 1.19

$$\begin{aligned} p(f(X,Z), f(Y,a))\{X/a, Y/Z, W/b\} &= p(f(a,Z), f(Z,a)) \\ p(X,Y)\{X/f(Y), Y/b\} &= p(f(Y), b) \end{aligned}$$

▮

It is also possible to compose substitutions:

Definition 1.20 (Composition) Let θ and σ be two substitutions:

$$\begin{aligned} \theta &:= \{X_1/s_1, \ldots, X_m/s_m\} \\ \sigma &:= \{Y_1/t_1, \ldots, Y_n/t_n\} \end{aligned}$$

The *composition* $\theta\sigma$ of θ and σ is obtained from the set:

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

by removing all $X_i/s_i\sigma$ for which $X_i = s_i\sigma$ $(1 \leq i \leq m)$ and by removing those Y_j/t_j for which $Y_j \in \{X_1, \ldots, X_m\}$ $(1 \leq j \leq n)$. ▮

It is left as an exercise to prove that the above syntactic definition of composition actually coincides with function composition (see exercise 1.13).

Example 1.21

$$\{X/f(Z), Y/W\}\{X/a, Z/a, W/Y\} = \{X/f(a), Z/a, W/Y\}$$

▮

A kind of substitution that will be of special interest are the so-called idempotent substitutions:

Definition 1.22 (Idempotent substitution) A substitution θ is said to be *idempotent* iff $\theta = \theta\theta$. ▮

It can be shown that a substitution θ is *idempotent* iff $Dom(\theta) \cap Range(\theta) = \varnothing$. The proof of this is left as an exercise and so are the proofs of the following properties:

Proposition 1.23 (Properties of substitutions) Let θ, σ and γ be substitutions and let E be a term or a formula. Then:

- $E(\theta\sigma) = (E\theta)\sigma$
- $(\theta\sigma)\gamma = \theta(\sigma\gamma)$
- $\epsilon\theta = \theta\epsilon = \theta$

∎

Notice that composition of substitutions is not commutative as illustrated by the following example:

$$\{X/f(Y)\}\{Y/a\} = \{X/f(a), Y/a\} \neq \{Y/a\}\{X/f(Y)\} = \{Y/a, X/f(Y)\}$$

Exercises

1.1 Formalize the following sentences of natural language as formulas of predicate logic:

a) Every natural number has a successor.
b) Nothing is better than taking a nap.
c) There is no such thing as negative integers.
d) The names have been changed to protect the innocent.
e) Logic plays an important role in all areas of computer science.
f) The renter of a car pays the deductible in case of an accident.

1.2 Formalize the following sentences of natural language into predicate logic:

a) A bronze medal is better than nothing.
b) Nothing is better than a gold medal.
c) A bronze medal is better than a gold medal.

1.3 Prove Proposition 1.13.

1.4 Prove the equivalences in connection with Definition 1.15.

1.5 Let $F := \forall X\, \exists Y p(X,Y)$ and $G := \exists Y\, \forall X p(X,Y)$. State for each of the following four formulas whether it is satisfiable or not. If it is, give a model with the natural numbers as domain, if it is not, explain why.

$$(F \wedge G) \quad (F \wedge \neg G) \quad (\neg F \wedge \neg G) \quad (\neg F \wedge G)$$

1.6 Let F and G be closed formulas. Show that $F \equiv G$ iff $\{F\} \models G$ and $\{G\} \models F$.

1.7 Show that P is unsatisfiable iff there is some closed formula F such that $P \models F$ and $P \models \neg F$.

1.8 Show that the following three formulas are satisfiable only if the interpretation has an infinite domain

$$\forall X \neg p(X, X)$$
$$\forall X \forall Y \forall Z (p(X, Y) \wedge p(Y, Z) \supset p(X, Z))$$
$$\forall X \exists Y p(X, Y)$$

1.9 Let F be a formula and θ a substitution. Show that $\forall F \models \forall(F\theta)$.

1.10 Let P_1, P_2 and P_3 be sets of closed formulas. Redefine $\models$ in such a way that $P_1 \models P_2$ iff every formula in P_2 is a logical consequence of P_1. Then show that $\models$ is transitive — that is, if $P_1 \models P_2$ and $P_2 \models P_3$ then $P_1 \models P_3$.

1.11 Let P_1 and P_2 be sets of closed formulas. Show that if $P_1 \subseteq P_2$ and $P_1 \models F$ then $P_2 \models F$.

1.12 Prove Proposition 1.23.

1.13 Let θ and σ be substitutions. Show that the composition $\theta\sigma$ is equivalent to function composition of the mappings denoted by θ and σ.

1.14 Show that a substitution θ is idempotent iff $\mathit{Dom}(\theta) \cap \mathit{Range}(\theta) = \varnothing$.

1.15 Which of the following statements are true?

- if $\sigma\theta = \delta\theta$ then $\sigma = \delta$
- if $\theta\sigma = \theta\delta$ then $\sigma = \delta$
- if $\sigma = \delta$ then $\sigma\theta = \delta\theta$

Chapter 2

Definite Logic Programs

2.1 Definite Clauses

The idea of logic programming is to use a computer for drawing conclusions from declarative descriptions. Such descriptions — called logic programs — consist of finite sets of logic formulas. Thus, the idea has its roots in the research on *automatic theorem proving*. However, the transition from experimental theorem proving to applied logic programming requires improved efficiency of the system. This is achieved by introducing restrictions on the language of formulas — restrictions that make it possible to use the relatively simple and powerful inference rule called the *SLD-resolution principle*. This chapter introduces a restricted language of *definite logic programs* and in the next chapter their computational principles are discussed. In subsequent chapters a more unrestrictive language of so-called *general* programs is introduced. In this way the foundations of the programming language Prolog are presented.

To start with, attention will be restricted to a special type of *declarative* sentences of natural language that describe positive *facts* and *rules*. A sentence of this type either states that a relation holds between individuals (in case of a fact), or that a relation holds between individuals *provided* that some other relations hold (in case of a rule). For example, consider the sentences:

(*i*) "Tom is John's child"

(*ii*) "Ann is Tom's child"

(*iii*) "John is Mark's child"

(*iv*) "Alice is John's child"

(*v*) "The grandchild of a person is a child of a child of this person"

These sentences may be formalized in two steps. First atomic formulas describing facts are introduced:

$$child(tom, john) \quad (1)$$
$$child(ann, tom) \quad (2)$$
$$child(john, mark) \quad (3)$$
$$child(alice, john) \quad (4)$$

Applying this notation to the final sentence yields:

"For all X and Y, $grandchild(X, Y)$ if
there exists a Z such that $child(X, Z)$ and $child(Z, Y)$" (5)

This can be further formalized using quantifiers and the logical connectives "$\supset$" and "$\wedge$", but to preserve the natural order of expression the implication is reversed and written "$\leftarrow$":

$$\forall X\, \forall Y\, (grandchild(X, Y) \leftarrow \exists\, Z\, (child(X, Z) \wedge child(Z, Y))) \quad (6)$$

This formula can be transformed into the following equivalent forms using the equivalences given in connection with Definition 1.15:

$$\forall X\, \forall Y\, (grandchild(X, Y) \vee \neg\, \exists\, Z\, (child(X, Z) \wedge child(Z, Y)))$$
$$\forall X\, \forall Y\, (grandchild(X, Y) \vee \forall\, Z\, \neg\, (child(X, Z) \wedge child(Z, Y)))$$
$$\forall X\, \forall Y\, \forall\, Z\, (grandchild(X, Y) \vee \neg\, (child(X, Z) \wedge child(Z, Y)))$$
$$\forall X\, \forall Y\, \forall\, Z\, (grandchild(X, Y) \leftarrow (child(X, Z) \wedge child(Z, Y)))$$

We now focus attention on the language of formulas exemplified by the example above. It consists of formulas of the form:

$$A_0 \leftarrow A_1 \wedge \cdots \wedge A_n \quad \text{(where } n \geq 0)$$

or equivalently:

$$A_0 \vee \neg A_1 \vee \cdots \vee \neg A_n$$

where $A_0, \ldots, A_n$ are atomic formulas and all variables occurring in a formula are (implicitly) universally quantified over the whole formula. The formulas of this form are called *definite clauses.* Facts are definite clauses where $n = 0$. (Facts are sometimes called unit-clauses.) The atomic formula A_0 is called the *head* of the clause whereas $A_1 \wedge \cdots \wedge A_n$ is called its *body.*

The initial example shows that definite clauses use a restricted form of existential quantification — the variables that occur only in body literals are existentially quantified over the body (though formally this is equivalent to universal quantification on the level of clauses).

2.2 Definite Programs and Goals

The logic formulas derived above are special cases of a more general form, called *clausal form.*

Definition 2.1 (Clause) A *clause* is a formula $\forall(L_1 \vee \cdots \vee L_n)$ where each L_i is an atomic formula (a positive literal) or the negation of an atomic formula (a negative literal). ▮

As seen above, a *definite clause* is a clause that contains exactly one positive literal. That is, a formula of the form:

$$\forall(A_0 \vee \neg A_1 \vee \cdots \vee \neg A_n)$$

The notational convention is to write such a definite clause thus:

$$A_0 \leftarrow A_1, \ldots, A_n \quad (n \geq 0)$$

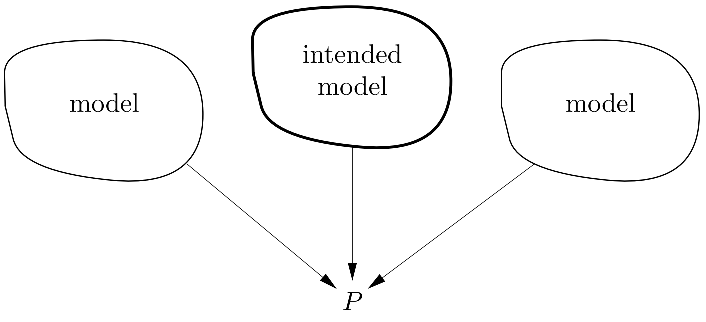
If the body is empty (i.e. if $n = 0$) the implication arrow is usually omitted. Alternatively the empty body can be seen as a nullary connective $\blacksquare$ which is true in every interpretation. (Symmetrically there is also a nullary connective $\square$ which is false in every interpretation.) The first kind of logic program to be discussed are programs consisting of a finite number of definite clauses:

Definition 2.2 (Definite programs) A *definite program* is a finite set of definite clauses. ▮

To explain the use of logic formulas as programs, a general view of logic programming is presented in Figure 2.1. The programmer attempts to describe the *intended model* by means of declarative sentences (i.e. when writing a program he has in mind an algebraic structure, usually infinite, whose relations are to interpret the predicate symbols of the program). These sentences are definite clauses — facts and rules. The program is a set of logic formulas and it may have many models, including the intended model (Figure 2.1(a)). The concept of intended model makes it possible to discuss correctness of logic programs — a program P is incorrect iff the intended model is not a model of P. (Notice that in order to prove programs to be correct or to test programs it is necessary to have an alternative description of the intended model, independent of P.)

The program will be used by the computer to draw conclusions about the intended model (Figure 2.1(b)). However, the only information available to the computer about the intended model is the program itself. So the conclusions drawn must be true in *any* model of the program to guarantee that they are true in the intended model (Figure 2.1(c)). In other words — the soundness of the system is a necessary condition. This will be discussed in Chapter 3. Before that, attention will be focused on the practical question of how a logic program is to be used.

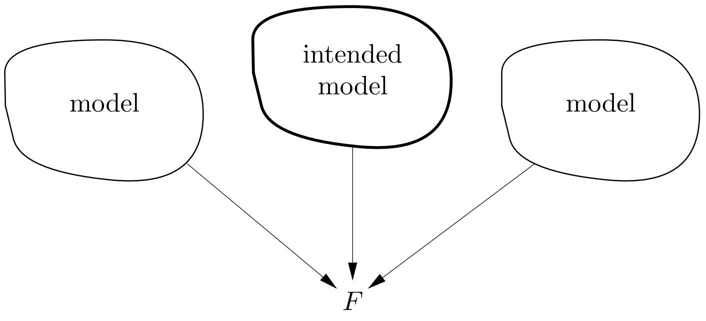
The set of logical consequences of a program is infinite. Therefore the user is expected to *query* the program selectively for various aspects of the intended model. There is an analogy with relational databases — facts explicitly describe elements of the relations while rules give intensional characterization of some other elements.



(a)

$$P \vdash F$$

(b)



(c)

Figure 2.1: General view of logic programming

Since the rules may be recursive, the relation described may be infinite in contrast to the traditional relational databases. Another difference is the use of variables and compound terms. This chapter considers only "queries" of the form:

$$\forall(\neg(A_1 \wedge \cdots \wedge A_m))$$

Such formulas are called *definite goals* and are usually written as:

$$\leftarrow A_1, \ldots, A_m$$

where A_i's are atomic formulas called *subgoals*. The goal where $m = 0$ is denoted $\square$[1] and called the *empty* goal. The logical meaning of a goal can be explained by referring to the equivalent universally quantified formula:

$$\forall X_1 \cdots \forall X_n \, \neg(A_1 \wedge \cdots \wedge A_m)$$

where $X_1, \ldots, X_n$ are all variables that occur in the goal. This is equivalent to:

$$\neg \, \exists X_1 \cdots \exists X_n \, (A_1 \wedge \cdots \wedge A_m)$$

This, in turn, can be seen as an existential question and the system attempts to deny it by constructing a counter-example. That is, it attempts to find terms $t_1, \ldots, t_n$ such that the formula obtained from $A_1 \wedge \cdots \wedge A_m$ when replacing the variable X_i by t_i $(1 \leq i \leq n)$, is true in any model of the program, i.e. to construct a logical consequence of the program which is an instance of a conjunction of all subgoals in the goal.

By giving a definite goal the user selects the set of conclusions to be constructed. This set may be finite or infinite. The problem of how the machine constructs it will be discussed in Chapter 3. The section is concluded with some examples of queries and the answers obtained to the corresponding goals in a typical Prolog system.

Example 2.3 Referring to the family-example in Section 2.1 the user may ask the following queries (with the corresponding goal):

Query	Goal
"Is Ann a child of Tom?"	$\leftarrow child(ann, tom)$
"Who is a grandchild of Ann?"	$\leftarrow grandchild(X, ann)$
"Whose grandchild is Tom?"	$\leftarrow grandchild(tom, X)$
"Who is a grandchild of whom?"	$\leftarrow grandchild(X, Y)$

The following answers are obtained:

- Since there are no variables in the first goal the answer is simply "yes";
- Since the program contains no information about grandchildren of Ann the answer to the second goal is "no one" (although most Prolog implementations would answer simply "no";

[1]Of course, formally it is not correct to write $\leftarrow A_1, \ldots, A_m$ since "$\leftarrow$" should have a formula also on the left-hand side. The problem becomes even more evident when $m = 0$ because then the right-hand side disappears as well. However, formally the problem can be viewed as follows — a definite goal has the form $\forall(\neg(A_1 \wedge \cdots \wedge A_m))$ which is equivalent to $\forall(\square \vee \neg(A_1 \wedge \cdots \wedge A_m \wedge \blacksquare))$. A nonempty goal can thus be viewed as the formula $\forall(\square \leftarrow (A_1 \wedge \cdots \wedge A_m))$. The empty goal can be viewed as the formula $\square \leftarrow \blacksquare$ which is equivalent to $\square$.

- Since Tom is the grandchild of Mark the answer is $X = mark$ in reply to the third goal;
- The final goal yields three answers:

$$\begin{array}{ll} X = tom & Y = mark \\ X = alice & Y = mark \\ X = ann & Y = john \end{array}$$

It is also possible to ask more complicated queries, for example "Is there a person whose grandchildren are Tom and Alice?", expressed formally as:

$$\leftarrow grandchild(tom, X), grandchild(alice, X)$$

whose (expected) answer is $X = mark$. ▮

2.3 The Least Herbrand Model

Definite programs can only express positive knowledge — both facts and rules say which elements of a structure are in a relation, but they do not say when the relations do not hold. Therefore, using the language of definite programs, it is not possible to construct contradictory descriptions, i.e. unsatisfiable sets of formulas. In other words, every definite program has a model. This section discusses this matter in more detail. It shows also that every definite program has a well defined *least* model. Intuitively this model reflects all information expressed by the program and nothing more.

We first focus attention on models of a special kind, called *Herbrand models*. The idea is to abstract from the actual meanings of the functors (here, constants are treated as 0-ary functors) of the language. More precisely, attention is restricted to the interpretations where the domain is the set of variable-free terms and the meaning of every ground term is the term itself. After all, it is a common practice in databases — the constants *tom* and *ann* may represent persons but the database describes relations between the persons by handling relations between the terms (symbols) no matter whom they represent.

The formal definition of such domains follows and is illustrated by two simple examples.

Definition 2.4 (Herbrand universe, Herbrand base) Let $\mathcal{A}$ be an alphabet containing at least one constant symbol. The set $U_{\mathcal{A}}$ of all ground terms constructed from functors and constants in $\mathcal{A}$ is called the *Herbrand universe* of $\mathcal{A}$. The set $B_{\mathcal{A}}$ of all ground, atomic formulas over $\mathcal{A}$ is called the *Herbrand base* of $\mathcal{A}$. ▮

The Herbrand universe and Herbrand base are often defined for a given *program*. In this case it is assumed that the alphabet of the program consists of exactly those symbols which appear in the program. It is also assumed that the program contains at least one constant (since otherwise, the domain would be empty).

Example 2.5 Consider the following definite program P:

$$\begin{array}{l} odd(s(0)). \\ odd(s(s(X))) \leftarrow odd(X). \end{array}$$

The program contains one constant (0) and one unary functor (s). Consequently the Herbrand universe looks as follows:

$$U_P = \{0, s(0), s(s(0)), s(s(s(0))), \ldots\}$$

Since the program contains only one (unary) predicate symbol (*odd*) it has the following Herbrand base:

$$B_P = \{odd(0), odd(s(0)), odd(s(s(0))), \ldots\}$$

∎

Example 2.6 Consider the following definite program P:

$$\begin{array}{l} owns(owner(corvette), corvette). \\ happy(X) \leftarrow owns(X, corvette). \end{array}$$

In this case the Herbrand universe U_P consists of the set:

$$\{corvette, owner(corvette), owner(owner(corvette)), \ldots\}$$

and the Herbrand base B_P of the set:

$$\{owns(s,t) \mid s,t \in U_P\} \cup \{happy(s) \mid s \in U_P\}$$

∎

Definition 2.7 (Herbrand interpretations) A Herbrand interpretation of P is an interpretation $\Im$ such that:

- the domain of $\Im$ is U_P;
- for every constant c, $c_\Im$ is defined to be c itself;
- for every n-ary functor f the function $f_\Im$ is defined as follows

  $$f_\Im(x_1, \ldots, x_n) := f(x_1, \ldots, x_n)$$

  That is, the function $f_\Im$ applied to n ground terms composes them into the ground term with the principal functor f;
- for every n-ary predicate symbol p the relation $p_\Im$ is a subset of U_P^n (the set of all n-tuples of ground terms).

∎

Thus Herbrand interpretations have predefined meanings of functors and constants and in order to specify a Herbrand interpretation it suffices to list the relations associated with the predicate symbol. Hence, for an n-ary predicate symbol p and a Herbrand interpretation $\Im$ the meaning $p_\Im$ of p consists of the following set of n-tuples: $\{\langle t_1, \ldots, t_n\rangle \in U_P^n \mid \Im \models p(t_1, \ldots, t_n)\}$.

Example 2.8 One possible interpretation of the program P in Example 2.5 is $odd_{\Im} = \{\langle s(0)\rangle, \langle s(s(s(0)))\rangle\}$. A Herbrand interpretation can be specified by giving a family of such relations (one for every predicate symbol). ▮

Since the domain of a Herbrand interpretation is the Herbrand universe the relations are sets of tuples of ground terms. One can define all of them at once by specifying a set of *labelled* tuples, where the labels are predicate symbols. In other words: A Herbrand interpretation $\Im$ can be seen as a subset of the Herbrand base (or a possibly infinite relational database), namely $\{A \in B_P \mid \Im \models A\}$.

Example 2.9 Consider some alternative Herbrand interpretations for P of Example 2.5.

$$\begin{array}{rcl}
\Im_1 & := & \varnothing \\
\Im_2 & := & \{odd(s(0))\} \\
\Im_3 & := & \{odd(s(0)), odd(s(s(0)))\} \\
\Im_4 & := & \{odd(s^n(0)) \mid n \in \{1,3,5,7,\ldots\}\} \\
 & = & \{odd(s(0)), odd(s(s(s(0)))), \ldots\} \\
\Im_5 & := & B_P
\end{array}$$

▮

Definition 2.10 (Herbrand model) A Herbrand model of a set of (closed) formulas is a Herbrand interpretation which is a model of every formula in the set. ▮

It turns out that Herbrand interpretations and Herbrand models have two attractive properties. The first is pragmatic: In order to determine if a Herbrand interpretation $\Im$ is a model of a universally quantified formula $\forall F$ it suffices to check if all ground instances of F are true in $\Im$. For instance, to check if $A_0 \leftarrow A_1, \ldots, A_n$ is true in $\Im$ it suffices to show that if $(A_0 \leftarrow A_1, \ldots, A_n)\theta$ is a ground instance of $A_0 \leftarrow A_1, \ldots, A_n$ and $A_1\theta, \ldots, A_n\theta \in \Im$ then $A_0\theta \in \Im$.

Example 2.11 Clearly $\Im_1$ cannot be a model of P in Example 2.5 as it is not a Herbrand model of $odd(s(0))$. However, $\Im_2, \Im_3, \Im_4, \Im_5$ are all models of $odd(s(0))$ since $odd(s(0)) \in \Im_i$, $(2 \leq i \leq 5)$.

Now, $\Im_2$ is not a model of $odd(s(s(X))) \leftarrow odd(X)$ since there is a ground instance of the rule — namely $odd(s(s(s(0)))) \leftarrow odd(s(0))$ — such that all premises are true: $odd(s(0)) \in \Im_2$, but the conclusion is false: $odd(s(s(s(0)))) \notin \Im_2$. By a similar reasoning it follows that $\Im_3$ is not a model of the rule.

However, $\Im_4$ is a model also of the rule; let $odd(s(s(t))) \leftarrow odd(t)$ be any ground instance of the rule where $t \in U_P$. Clearly, $odd(s(s(t))) \leftarrow odd(t)$ is true if $odd(t) \notin \Im_4$ (check with Definition 1.6). Furthermore, if $odd(t) \in \Im_4$ then it must also hold that $odd(s(s(t))) \in \Im_4$ (cf. the the definition of $\Im_4$ above) and hence $odd(s(s(t))) \leftarrow odd(t)$ is true in $\Im_4$. Similar reasoning proves that $\Im_5$ is also a model of the program. ▮

The second reason for focusing on Herbrand interpretations is more theoretical. For the restricted language of definite programs, it turns out that in order to determine whether an atomic formula A is a logical consequence of a definite program P it suffices to check that every Herbrand model of P is also a Herbrand model of A.

Theorem 2.12 Let P be a definite program and G a definite goal. If $\Im'$ is a model of $P \cup \{G\}$ then $\Im := \{A \in B_P \mid \Im' \models A\}$ is a Herbrand model of $P \cup \{G\}$. ∎

Proof: Clearly, $\Im$ is a Herbrand interpretation. Now assume that $\Im'$ is a model and that $\Im$ is not a model of $P \cup \{G\}$. In other words, there exists a ground instance of a clause or a goal in $P \cup \{G\}$:

$$A_0 \leftarrow A_1, \ldots, A_m \quad (m \geq 0)$$

which is not true in $\Im$ ($A_0 = \Box$ in case of a goal).

Since this clause is false in $\Im$ then $A_1, \ldots, A_m$ are all true and A_0 is false in $\Im$. Hence, by the definition of $\Im$ we conclude that $A_1, \ldots, A_m$ are true and A_0 is false in $\Im'$. This contradicts the assumption that $\Im'$ is a model. Hence $\Im$ is a model of $P \cup \{G\}$. ∎

Notice that the form of P in Theorem 2.12 is restricted to definite programs. In the general case, nonexistence of a Herbrand model of a set of formulas P does not mean that P is unsatisfiable. That is, there are sets of formulas P which do not have a Herbrand model but which have other models.[2]

Example 2.13 Consider the formulas $\{\neg p(a), \exists X\, p(X)\}$ where $U_P := \{a\}$ and $B_P := \{p(a)\}$. Clearly, there are only two Herbrand interpretations — the empty set and B_P itself. The former is not a model of the second formula. The latter is a model of the second formula but not of the first.

However, it is not very hard to find a model of the formulas — let the domain be the natural numbers, assign 0 to the constant a and the relation $\{\langle 1\rangle, \langle 3\rangle, \langle 5\rangle, \ldots\}$ to the predicate symbol p (i.e. let p denote the "odd"-relation). Clearly this is a model since "0 is not odd" and "there exists a natural number which is odd, e.g. 1". ∎

Notice that the Herbrand base of a definite program P always *is* a Herbrand model of the program. To check that this is so, simply take an arbitrary ground instance of any clause $A_0 \leftarrow A_1, \ldots, A_m$ in P. Clearly, all $A_0, \ldots, A_m$ are in the Herbrand base. Hence the formula is true. However, this model is rather uninteresting — every n-ary predicate of the program is interpreted as the full n-ary relation over the domain of ground terms. More important is of course the question — what are the *interesting* models of the program? Intuitively there is no reason to expect that the model includes more ground atoms than those which follow from the program. By the analogy to databases — if John is not in the telephone directory he probably has no telephone. However, the directory gives only positive facts and if John has a telephone it is not a contradiction to what is said in the directory.

The rest of this section is organized as follows. First it is shown that there exists a *unique* minimal model called the *least Herbrand model* of a definite program. Then it is shown that this model really contains all positive information present in the program.

The Herbrand models of a definite program are subsets of its Herbrand base. Thus the set-inclusion is a natural ordering of such models. In order to show the existence of least models with respect to set-inclusion it suffices to show that the intersection of all Herbrand models is also a (Herbrand) model.

[2] More generally the result of Theorem 2.12 would hold for any set of *clauses*.

Theorem 2.14 (Model intersection property) Let M be a non-empty family of Herbrand models of a definite program P. Then the intersection $\Im := \bigcap M$ is a Herbrand model of P. ▮

Proof: Assume that $\Im$ is not a model of P. Then there exists a ground instance of a clause of P:

$$A_0 \leftarrow A_1, \ldots, A_m \quad (m \geq 0)$$

which is not true in $\Im$. This implies that $\Im$ contains $A_1, \ldots, A_m$ but not A_0. Then $A_1, \ldots, A_m$ are elements of every interpretation of the family M. Moreover there must be at least one model $\Im_i \in M$ such that $A_0 \notin \Im_i$. Thus $A_0 \leftarrow A_1, \ldots, A_m$ is not true in this $\Im_i$. Hence $\Im_i$ is not a model of the program, which contradicts the assumption. This concludes the proof that the intersection of any set of Herbrand models of a program is also a Herbrand model. ▮

Thus by taking the intersection of all Herbrand models (it is known that every definite program P has at least one Herbrand model — namely B_P) the least Herbrand model of the definite program is obtained.

Example 2.15 Let P be the definite program $\{male(adam), female(eve)\}$ with obvious intended interpretation. P has the following four Herbrand models:

$$\{male(adam), female(eve)\}$$
$$\{male(adam), male(eve), female(eve)\}$$
$$\{male(adam), female(eve), female(adam)\}$$
$$\{male(adam), male(eve), female(eve), female(adam)\}$$

It is not very hard to see that any intersection of these yields a Herbrand model. However, all but the first model contain atoms incompatible with the intended one. Notice also that the intersection of all four models yields a model which corresponds to the intended model. ▮

This example indicates a connection between the least Herbrand model and the intended model of a definite program. The intended model is an abstraction of the world to be described by the program. The world may be richer than the least Herbrand model. For instance, there may be more female individuals than just Eve. However, the information not included explicitly (via facts) or implicitly (via rules) in the program cannot be obtained as an answer to a goal. The answers correspond to *logical consequences* of the program. Ideally, a ground atomic formula $p(t_1, \ldots, t_n)$ is a logical consequence of the program iff, in the intended interpretation $\Im$, t_i denotes the individual x_i and $\langle x_1, \ldots, x_n \rangle \in p_\Im$. The set of all such ground atoms can be seen as a "coded" version of the intended model. The following theorem relates this set to the least Herbrand model.

Theorem 2.16 The least Herbrand model M_P of a definite program P is the set of all ground atomic logical consequences of the program. That is, $M_P = \{A \in B_P \mid P \models A\}$. ▮

Proof: Show first $M_P \supseteq \{A \in B_P \mid P \models A\}$: It is easy to see that every ground atom A which is a logical consequence of P is an element of M_P. Indeed, by the definition of logical consequence A must be true in M_P. On the other hand, the definition of Herbrand interpretation states that A is true in M_P iff A is an element of M_P.

Then show that $M_P \subseteq \{A \in B_P \mid P \models A\}$: Assume that A is in M_P. Hence it is true in every Herbrand model of P. Assume that it is not true in some non-Herbrand model $\Im'$ of P. But we know (see Theorem 2.12) that the set $\Im$ of all ground atomic formulas which are true in $\Im'$ is a Herbrand model of P. Hence A cannot be an element of $\Im$. This contradicts the assumption that there exists a model of P where A is false. Hence A is true in every model of P, that is $P \models A$, which concludes the proof. ∎

The model intersection property expressed by Theorem 2.14 does not hold for arbitrary formulas as illustrated by the following example.

Example 2.17 Consider the formula $p(a) \vee q(b)$. Clearly, both $\{p(a)\}$ and $\{q(b)\}$ are Herbrand models of the formula. However, the intersection $\{p(a)\} \cap \{q(b)\} = \varnothing$ is not a model. The two models are examples of *minimal* models — that is, one cannot remove any element from the model and still have a model. However, there is no *least* model — that is, a unique minimal model. ∎

2.4 Construction of Least Herbrand Models

The question arises how the least Herbrand model can be constructed, or approximated by successive enumeration of its elements. The answer to this question is given by a *fixed point* approach to the semantics of definite programs. (A fixpoint of a function $f : \mathcal{D} \to \mathcal{D}$ is an element $x \in \mathcal{D}$ such that $f(x) = x$.) This section gives only a sketch of the construction. The discussion of the relevant theory is outside of the scope of this book. However, the intuition behind the construction is the following:

A definite program consists of facts and rules. Clearly, all ground instances of the facts must be included in every Herbrand model. If a Herbrand interpretation $\Im$ does not include a ground instance of a fact A of the program then A is not true in $\Im$ and $\Im$ is not a model.

Next, consider a rule $A_0 \leftarrow A_1, \ldots, A_m$ where $(m > 0)$. This rule states that whenever $A_1, \ldots, A_m$ are true then so is A_0. In other words, take any ground instance $(A_0 \leftarrow A_1, \ldots, A_m)\theta$ of the rule. If $\Im$ includes $A_1\theta, \ldots, A_m\theta$ it must also include $A_0\theta$ in order to be a model.

Consider the set $\Im_1$ of all ground instances of facts in the program. It is now possible to use every instance of each rule to augment $\Im_1$ with new elements which necessarily must belong to every model. In that way a new set $\Im_2$ is obtained which can be used again to generate more elements which must belong to the model. This process is repeated as long as new elements are generated. The new elements added to $\Im_{i+1}$ are those which *must follow immediately* from $\Im_i$.

The construction outlined above can be formally defined as an iteration of a transformation T_P on Herbrand interpretations of the program P. The operation is called the *immediate consequence operator* and is defined as follows:

Definition 2.18 (Immediate consequence operator) Let *ground*(P) be the set of all ground instances of clauses in P. T_P is a function on Herbrand interpretations of P defined as follows:

$$T_P(I) := \{A_0 \mid A_0 \leftarrow A_1, \ldots, A_m \in \mathit{ground}(P) \wedge \{A_1, \ldots, A_m\} \subseteq I\}$$

∎

For definite programs it can be shown that there exists a least interpretation $\Im$ such that $T_P(\Im) = \Im$ and that $\Im$ is identical with the least Herbrand model M_P. Moreover, M_P is the limit of the increasing, possibly infinite sequence of iterations:

$$\varnothing, \quad T_P(\varnothing), \quad T_P(T_P(\varnothing)), \quad T_P(T_P(T_P(\varnothing))), \quad \ldots$$

There is a standard notation used to denote elements of the sequence of interpretations constructed for P. Namely:

$$\begin{array}{rcl} T_P \uparrow 0 & := & \varnothing \\ T_P \uparrow (i+1) & := & T_P(T_P \uparrow i) \\ T_P \uparrow \omega & := & \displaystyle\bigcup_{i=0}^{\infty} T_P \uparrow i \end{array}$$

The following example illustrates the construction:

Example 2.19 Consider again the program of Example 2.5.

$$\begin{array}{rcl} T_P \uparrow 0 & = & \varnothing \\ T_P \uparrow 1 & = & \{\mathit{odd}(s(0))\} \\ T_P \uparrow 2 & = & \{\mathit{odd}(s(s(s(0)))), \mathit{odd}(s(0))\} \\ & \vdots & \\ T_P \uparrow \omega & = & \{\mathit{odd}(s^n(0)) \mid n \in \{1, 3, 5, \ldots\}\} \end{array}$$

∎

As already mentioned above it has been established that the set constructed in this way is identical to the least Herbrand model.

Theorem 2.20 Let P be a definite program and M_P its least Herbrand model. Then:

- M_P is the least Herbrand interpretation such that $T_P(M_P) = M_P$ (i.e. it is the least fixpoint of T_P).
- $M_P = T_P \uparrow \omega$.

∎

For additional details and proofs see for example Apt (1990), Lloyd (1987) or van Emden and Kowalski (1976).

Exercises

2.1 Rewrite the following formulas in the form $A_0 \leftarrow A_1, \ldots, A_m$:

$$\begin{gathered}\forall X(p(X) \vee \neg q(X)) \\ \forall X(p(X) \vee \neg \exists\, Y\,(q(X,Y) \wedge r(X))) \\ \forall X(\neg p(X) \vee (q(X) \supset r(X))) \\ \forall X(r(X) \supset (q(X) \supset p(X)))\end{gathered}$$

2.2 Formalize the following scenario as a definite program:

> Basil owns Fawlty Towers. Basil and Sybil are married. Polly and Manuel are employees at Fawlty Towers. Smith and Jones are guests at Fawlty Towers. All hotel-owners and their spouses serve all guests at the hotel. All employees at a hotel serve all guests at the hotel. All employees dislike the owner of the workplace. Basil dislikes Manuel.

Then ask the queries "Who serves who?" and "Who dislikes who?".

2.3 Give the Herbrand universe and Herbrand base of the following definite program:

$$\begin{array}{l} p(f(X)) \leftarrow q(X, g(X)). \\ q(a, g(b)). \\ q(b, g(b)). \end{array}$$

2.4 Give the Herbrand universe and Herbrand base of the following definite program:

$$\begin{array}{l} p(s(X), Y, s(Z)) \leftarrow p(X, Y, Z). \\ p(0, X, X). \end{array}$$

2.5 Consider the Herbrand universe consisting of the constants a, b, c and d. Let $\Im$ be the Herbrand interpretation:

$$\{p(a), p(b), q(a), q(b), q(c), q(d)\}$$

Which of the following formulas are true in $\Im$?

$$\begin{array}{ll} (1) & \forall X p(X) \\ (2) & \forall X q(X) \\ (3) & \exists X(q(X) \wedge p(X)) \\ (4) & \forall X(q(X) \supset p(X)) \\ (5) & \forall X(p(X) \supset q(X)) \end{array}$$

2.6 Give the least Herbrand model of the program in exercise 2.3.

2.7 Give the least Herbrand model of the program in exercise 2.4. *Hint*: the model is infinite, but a certain pattern can be spotted when using the T_P-operator.

2.8 Consider the following program:

$$\begin{array}{l} p(0). \\ p(s(X)) \leftarrow p(X). \end{array}$$

Show that $p(s^n(0)) \in T_P \uparrow m$ iff $n < m$.

2.9 Let P be a definite program and $\Im$ a Herbrand interpretation. Show that $\Im$ is a model of P iff $T_P(\Im) \subseteq \Im$.

Chapter 3

SLD-Resolution

This chapter introduces the inference mechanism which is the basis of most logic programming systems. The idea is a special case of the inference rule called the *resolution principle* — an idea that was first introduced by J. A. Robinson in the mid-sixties for a richer language than definite programs. As a consequence, only a specialization of this rule, that applies to definite programs, is presented here. For reasons to be explained later, it will be called the *SLD-resolution principle.*

In the previous chapter the model-theoretic semantics of definite programs was discussed. The SLD-resolution principle makes it possible to draw correct conclusions from the program, thus providing a foundation for a logically sound *operational semantics* of definite programs. This chapter first defines the notion of SLD-resolution and then shows its correctness with respect to the model-theoretic semantics. Finally SLD-resolution is shown to be an instance of a more general notion involving the construction of proof trees.

3.1 Informal Introduction

Every inference rule of a logical system formalizes some natural way of reasoning. The presentation of the SLD-resolution principle is therefore preceded by an informal discussion about the underlying reasoning techniques.

The sentences of logic programs have a general structure of logical implication:

$$A_0 \leftarrow A_1, \ldots, A_n \qquad (n \geq 0)$$

where $A_0, \ldots, A_n$ are atomic formulas and where A_0 may be absent (in which case it is a goal clause). Consider the following definite program that describes a world where "parents of newborn children are proud", "Adam is the father of Mary" and "Mary is newborn":

$$\begin{array}{l} proud(X) \leftarrow parent(X,Y), newborn(Y). \\ parent(X,Y) \leftarrow father(X,Y). \\ parent(X,Y) \leftarrow mother(X,Y). \\ father(adam, mary). \\ newborn(mary). \end{array}$$

Notice that this program describes only “positive knowledge” — it does not state who is *not* proud. Nor does it convey what it means for someone not to be a parent. The problem of expressing negative knowledge will be investigated in detail in Chapter 4 when extending definite programs with negation.

Say now that we want to ask the question “Who is proud?”. The question concerns the world described by the program P, that is, the intended model of P. We would of course like to see the answer “Adam” to this question. However, as discussed in the previous chapters predicate logic does not provide the means for expressing this type of *interrogative* sentences; only *declarative* ones. Therefore the question may be formalized as the goal clause:

$$\leftarrow proud(Z) \tag{G_0}$$

which is an abbreviation for $\forall Z\, \neg proud(Z)$ which in turn is equivalent to:

$$\neg \exists Z\, proud(Z)$$

whose reading is “Nobody is proud”. That is, a negative answer to the query above. The aim now is to show that this answer is a false statement in every model of P (and in particular in the intended model). Then by Proposition 1.13 it can be concluded that $P \models \exists Z\, proud(Z)$. Alas this would result only in a “yes”-answer to the original question, while the expected answer is “Adam”. Thus, the objective is rather to find a substitution θ such that the set $P \cup \{\neg\, proud(Z)\theta\}$ is unsatisfiable, or equivalently such that $P \models proud(Z)\theta$.

The starting point of reasoning is the assumption G_0 — “For any Z, Z is not proud”. Inspection of the program reveals a rule describing one condition for someone to be proud:

$$proud(X) \leftarrow parent(X,Y), newborn(Y). \tag{C_0}$$

Its equivalent logical reading is:

$$\forall(\neg proud(X) \supset \neg(parent(X,Y) \wedge newborn(Y)))$$

Renaming X into Z, elimination of universal quantification and the use of *modus ponens* with respect to G_0 yields:

$$\neg\, (parent(Z,Y) \wedge newborn(Y))$$

or equivalently:

$$\leftarrow parent(Z,Y), newborn(Y). \tag{G_1}$$

Thus, one step of reasoning amounts to replacing a goal G_0 by another goal G_1 which is true in any model of $P \cup \{G_0\}$. It now remains to be shown that $P \cup \{G_1\}$ is unsatisfiable. Note that G_1 is equivalent to:

$$\forall Z \, \forall Y \, (\neg \mathit{parent}(Z, Y) \lor \neg \mathit{newborn}(Y))$$

Thus G_1 can be shown to be unsatisfiable with P if in every model of P there is some individual who is a parent of a newborn child. Thus, check first whether there are any parents at all. The program contains a clause:

$$\mathit{parent}(X, Y) \leftarrow \mathit{father}(X, Y). \qquad (C_1)$$

which is equivalent to:

$$\forall(\neg \mathit{parent}(X, Y) \supset \neg \mathit{father}(X, Y))$$

Thus, G_1 reduces to:

$$\leftarrow \mathit{father}(Z, Y), \mathit{newborn}(Y). \qquad (G_2)$$

The new goal G_2 can be shown to be unsatisfiable with P if in every model of P there is some individual who is a father of a newborn child. The program states that "Adam is the father of Mary":

$$\mathit{father}(\mathit{adam}, \mathit{mary}). \qquad (C_2)$$

Thus it remains to be shown that "Mary is not newborn" is unsatisfiable together with P:

$$\leftarrow \mathit{newborn}(\mathit{mary}). \qquad (G_3)$$

But the program also contains a fact:

$$\mathit{newborn}(\mathit{mary}). \qquad (C_3)$$

equivalent to $\neg \mathit{newborn}(\mathit{mary}) \supset \Box$ leading to a refutation:

$$\Box \qquad (G_4)$$

The way of reasoning used in this example is as follows: to show existence of something, assume the contrary and use *modus ponens* and elimination of the universal quantifier to find a counter-example for the assumption. This is a general idea to be used in computations of logic programs. As illustrated above, a single computation (reasoning) step transforms a set of atomic formulas — that is, a *definite goal* — into a new set of atoms. (See Figure 3.1.) It uses a selected atomic formula $p(s_1, \ldots, s_n)$ of the goal and a selected program clause of the form $p(t_1, \ldots, t_n) \leftarrow A_1, \ldots, A_m$ (where $m \geq 0$ and $A_1, \ldots, A_m$ are atoms) to find a common instance of $p(s_1, \ldots, s_n)$ and $p(t_1, \ldots, t_n)$. In other words a substitution θ is constructed such that $p(s_1, \ldots, s_n)\theta$ and $p(t_1, \ldots, t_n)\theta$ are identical. Such a substitution is called a *unifier* and the problem of finding unifiers will be discussed in the next section. The new goal is constructed from the old one by replacing the selected atom by the set of body atoms of the clause and applying θ to all

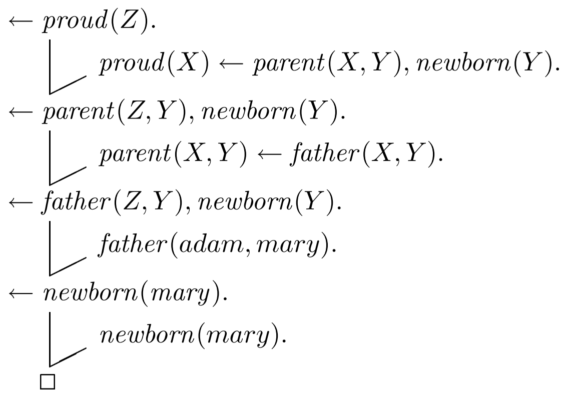

$\leftarrow proud(Z).$
$proud(X) \leftarrow parent(X,Y), newborn(Y).$
$\leftarrow parent(Z,Y), newborn(Y).$
$parent(X,Y) \leftarrow father(X,Y).$
$\leftarrow father(Z,Y), newborn(Y).$
$father(adam, mary).$
$\leftarrow newborn(mary).$
$newborn(mary).$
$\Box$

Figure 3.1: Refutation of $\leftarrow proud(Z)$**.**

atoms obtained in that way. This basic computation step can be seen as an inference rule since it transforms logic formulas. It will be called the resolution principle for definite programs or *SLD-resolution* principle. As illustrated above it combines in a special way *modus ponens* with the elimination rule for the universal quantifier.

At the last step of reasoning the empty goal, corresponding to falsity, is obtained. The final conclusion then is the negation of the initial goal. Since this goal is of the form $\forall \neg(A_1 \wedge \cdots \wedge A_m)$, the conclusion is equivalent (by DeMorgan's laws) to the formula $\exists(A_1 \wedge \cdots \wedge A_m)$. The final conclusion can be obtained by the inference rule known as *reductio ad absurdum*. Every step of reasoning produces a substitution. Unsatisfiability of the original goal $\leftarrow A_1, \ldots, A_m$ with P is demonstrated in k steps by showing that its instance:

$$\leftarrow (A_1, \ldots, A_m)\theta_1 \cdots \theta_k$$

is unsatisfiable, or equivalently that:

$$P \models (A_1 \wedge \cdots \wedge A_m)\theta_1 \cdots \theta_k$$

In the example discussed, the goal "Nobody is proud" is unsatisfiable with P since its instance "Adam is not proud" is unsatisfiable with P. In other words — in every model of P the sentence "Adam is proud" is true.

It is worth noticing that the unifiers may leave some variables unbound. In this case the universal closure of $(A_1 \wedge \cdots \wedge A_m)\theta_1 \cdots \theta_k$ is a logical consequence of P. Examples of such answers will appear below.

Notice also that generally the computation steps are not *deterministic* — any atom of a goal may be selected and there may be several clauses matching the selected atom. Another potential source of non-determinism concerns the existence of alternative unifiers for two atoms. These remarks suggest that it may be possible to construct (sometimes infinitely) many solutions, i.e. counter-examples for the initial goal. On the other hand it may also happen that the selected atom has no matching clause.

If so, it means that, using this method, it is not possible to construct any counter-example for the initial goal. The computation may also loop without producing any solution.

3.2 Unification

As demonstrated in the previous section, one of the main ingredients in the inference mechanism is the process of making two atomic formulas syntactically equivalent. Before defining the notion of SLD-resolution we focus on this process, called *unification*, and give an algorithmic solution — a procedure that takes two atomic formulas as input, and either shows how they can be instantiated to identical atoms or, reports a failure.

Before considering the problem of unifying atoms (and terms), consider an ordinary equation over the natural numbers ($\mathbb{N}$) such as:

$$2x + 3 \doteq 4y + 7 \tag{5}$$

The equation has a set of *solutions*; that is, valuations $\varphi\colon \{x, y\} \to \mathbb{N}$ such that $\varphi_{\Im}(2x+3) = \varphi_{\Im}(4y + 7)$ where $\Im$ is the standard interpretation of the arithmetic symbols. In this particular example there are infinitely many solutions ($\{x \mapsto 2, y \mapsto 0\}$ and $\{x \mapsto 4, y \mapsto 1\}$ etc.) but by a sequence of syntactic transformations that preserve the set of all solutions the equation may be transformed into an new equation that compactly represents *all* solutions to the original equation:

$$x \doteq 2(y + 1) \tag{6}$$

The transformations exploit domain knowledge (such as commutativity, associativity etc.) specific to the particular interpretation. In a logic program there is generally no such knowledge available and the question arises how to compute the solutions of an equation without *any* knowledge about the interpretation of the symbols. For example:

$$f(X, g(Y)) \doteq f(a, g(X)) \tag{7}$$

Clearly it is no longer possible to apply all the transformations that were applied above since the interpretation of $f/2$, $g/1$ is no longer fixed. However, any solution of the equations:

$$\{X \doteq a, g(Y) \doteq g(X)\} \tag{8}$$

must clearly be a solution of equation (7). Similarly, any solution of:

$$\{X \doteq a, Y \doteq X\} \tag{9}$$

must be a solution of equations (8). Finally any solution of:

$$\{X \doteq a, Y \doteq a\} \tag{10}$$

is a solution of (9). By analogy to (6) this is a compact representation of *some* solutions to equation (7). However, whether it represents *all* solution depends on how the symbols $f/2$, $g/1$ and a are interpreted. For example, if $f/2$ denotes integer addition, $g/1$ the successor function and a the integer zero, then (10) represents only one solution to equation (7). However, equation (7) has infinitely many integer solutions — any φ such that $\varphi(Y) = 0$ is a solution.

On the other hand, consider a Herbrand interpretation $\Im$; Solving of an equation $s \doteq t$ amounts to finding a valuation φ such that $\varphi_\Im(s) = \varphi_\Im(t)$. Now a valuation in the Herbrand domain is a mapping from variables of the equations to ground terms (that is, a substitution) and the interpretation of a ground term is the term itself. Thus, a solution in the Herbrand domain is a grounding substitution φ such that $s\varphi$ and $t\varphi$ are identical ground terms. This brings us to the fundamental concept of unification and unifiers:

Definition 3.1 (Unifier) Let s and t be terms. A substitution θ such that $s\theta$ and $t\theta$ are identical (denoted $s\theta = t\theta$) is called a *unifier* of s and t. ▮

The search for a unifier of two terms, s and t, will be viewed as the process of solving the equation $s \doteq t$. Therefore, more generally, if $\{s_1 \doteq t_1, \ldots, s_n \doteq t_n\}$ is a set of equations, then θ is called a unifier of the set if $s_i\theta = t_i\theta$ for all $1 \leq i \leq n$. For instance, the substitution $\{X/a, Y/a\}$ is a unifier of equation (7). It is also a unifier of (8)–(10). In fact, it is the only unifier as long as "irrelevant" variables are not introduced. (For instance, $\{X/a, Y/a, Z/a\}$ is also a unifier.) The transformations informally used in steps (7)–(10) preserve the set of all solutions in the Herbrand domain. (The full set of transformations will soon be presented.) Note that a solution to a set of equations is a (grounding) unifier. Thus, if a set of equations has a unifier then the set also has a solution.

However, not all sets of equations have a solution/unifier. For instance, the set $\{sum(1, 1) \doteq 2\}$ is not unifiable. Intuitively *sum* may be thought of as integer addition, but bear in mind that the symbols have no predefined interpretation in a logic program. (In Chapters 13–14 more powerful notions of unification are discussed.)

It is often the case that a set of equations have more than one unifier. For instance, both $\{X/g(Z), Y/Z\}$ and $\{X/g(a), Y/a, Z/a\}$ are unifiers of the set $\{f(X, Y) \doteq f(g(Z), Z)\}$. Under the first unifier the terms instantiate to $f(g(Z), Z)$ and under the second unifier the terms instantiate to $f(g(a), a)$. The second unifier is in a sense more restrictive than the first, as it makes the two terms ground whereas the first still provides room for some alternatives in that is does not specify how Z should be bound. We say that $\{X/g(Z), Y/Z\}$ is *more general* than $\{X/g(a), Y/a, Z/a\}$. More formally this can be expressed as follows:

Definition 3.2 (Generality of substitutions) A substitution θ is said to be *more general* than a substitution σ (denoted $\sigma \preceq \theta$) iff there exists a substitution ω such that $\sigma = \theta\omega$. ▮

Definition 3.3 (Most general unifier) A unifier θ is said to be a most general unifier (mgu) of two terms iff θ is more general than any other unifier of the terms. ▮

Definition 3.4 (Solved form) A set of equations $\{X_1 \doteq t_1, \ldots, X_n \doteq t_n\}$ is said to be in *solved form* iff $X_1, \ldots, X_n$ are distinct variables none of which appear in $t_1, \ldots, t_n$. ▮

There is a close correspondence between a set of equations in solved form and the most general unifier(s) of that set as shown by the following theorem:

Proposition 3.5 Let $\{X_1 \doteq t_1, \dots, X_n \doteq t_n\}$ be a set of equations in solved form. Then $\{X_1/t_1, \dots, X_n/t_n\}$ is an (idempotent) mgu of the solved form. ▮

Proof: First define:

$$\mathcal{E} := \{X_1 \doteq t_1, \dots, X_n \doteq t_n\}$$
$$\theta := \{X_1/t_1, \dots, X_n/t_n\}$$

Clearly θ is an idempotent unifier of $\mathcal{E}$. It remains to be shown that θ is more general than any other unifier of $\mathcal{E}$.

Thus, assume that σ is a unifier of $\mathcal{E}$. Then $X_i\sigma = t_i\sigma$ for $1 \leq i \leq n$. It must follow that $X_i/t_i\sigma \in \sigma$ for $1 \leq i \leq n$. In addition σ may contain some additonal pairs $Y_1/s_1, \dots, Y_m/s_m$ such that $\{X_1, \dots, X_n\} \cap \{Y_1, \dots, Y_m\} = \varnothing$. Thus, σ is of the form:

$$\{X_1/t_1\sigma, \dots, X_n/t_n\sigma, Y_1/s_1, \dots, Y_m/s_m\}$$

Now $\theta\sigma = \sigma$. Thus, there exists a substitution ω (viz. σ) such that $\sigma = \theta\omega$. Therefore, θ is an idempotent mgu. ▮

Definition 3.6 (Equivalence of sets of equations) Two sets of equations $\mathcal{E}_1$ and $\mathcal{E}_2$ are said to be *equivalent* if they have the same set of unifiers. ▮

Note that two equivalent sets of equations must have the same set of solutions in any Herbrand interpretation.

The definition can be used as follows: to compute a most general unifier $\text{MGU}(s, t)$ of two terms s and t, first try to transform the equation $\{s \doteq t\}$ into an equivalent solved form. If this fails then $\text{MGU}(s, t) =$ **failure**. However, if there is a solved form $\{X_1 \doteq t_1, \dots, X_n \doteq t_n\}$ then $\text{MGU}(s, t) = \{X_1/t_1, \dots, X_n/t_n\}$.

Figure 3.2 presents a (non-deterministic) algorithm which takes as input a set of equations $\mathcal{E}$ and terminates returning either a solved form equivalent to $\mathcal{E}$ or **failure** if no such solved form exists. Note that constants are viewed as function symbols of arity 0. Thus, if an equation $c \doteq c$ gets selected, the equation is simply removed by case 1. Before proving the correctness of the algorithm some examples are used to illustrate the idea:

Example 3.7 The set $\{f(X, g(Y)) \doteq f(g(Z), Z)\}$ has a solved form since:

$$\begin{aligned}\{f(X, g(Y)) \doteq f(g(Z), Z)\} \quad &\Rightarrow \quad \{X \doteq g(Z), g(Y) \doteq Z\}\\ &\Rightarrow \quad \{X \doteq g(Z), Z \doteq g(Y)\}\\ &\Rightarrow \quad \{X \doteq g(g(Y)), Z \doteq g(Y)\}\end{aligned}$$

The set $\{f(X, g(X), b) \doteq f(a, g(Z), Z)\}$, on the other hand, does not have a solved form since:

$$\begin{aligned}\{f(X, g(X), b) \doteq f(a, g(Z), Z)\} \quad &\Rightarrow \quad \{X \doteq a, g(X) \doteq g(Z), b \doteq Z\}\\ &\Rightarrow \quad \{X \doteq a, g(a) \doteq g(Z), b \doteq Z\}\end{aligned}$$

```
Input: A set $\mathcal{E}$ of equations.
Output: An equivalent set of equations in solved form or failure.

repeat
  select an arbitrary $s \doteq t \in \mathcal{E}$;
  case $s \doteq t$ of
    $f(s_1, \dots, s_n) \doteq f(t_1, \dots, t_n)$ where $n \geq 0 \Rightarrow$
        replace equation by $s_1 \doteq t_1, \dots, s_n \doteq t_n$;        % case 1
    $f(s_1, \dots, s_m) \doteq g(t_1, \dots, t_n)$ where $f/m \neq g/n \Rightarrow$
        halt with failure;                                  % case 2
    $X \doteq X \Rightarrow$
        remove the equation;                                % case 3
    $t \doteq X$ where $t$ is not a variable $\Rightarrow$
        replace equation by $X \doteq t$;                   % case 4
    $X \doteq t$ where $X \neq t$ and $X$ has more than one occurrence in $\mathcal{E} \Rightarrow$
        if $X$ is a proper subterm of $t$ then
          halt with failure                                 % case 5a
        else
          replace all other occurrences of $X$ by $t$;      % case 5b
  esac
until no action is possible on any equation in $\mathcal{E}$;
halt with $\mathcal{E}$;
```

Figure 3.2: Solved form algorithm

$$\begin{aligned}
&\Rightarrow \{X \doteq a, a \doteq Z, b \doteq Z\}\\
&\Rightarrow \{X \doteq a, Z \doteq a, b \doteq Z\}\\
&\Rightarrow \{X \doteq a, Z \doteq a, b \doteq a\}\\
&\Rightarrow \textbf{failure}
\end{aligned}$$

The algorithm fails since case 2 applies to $b \doteq a$. Finally consider:

$$\begin{aligned}
\{f(X, g(X)) \doteq f(Z, Z)\} &\Rightarrow \{X \doteq Z, g(X) \doteq Z\}\\
&\Rightarrow \{X \doteq Z, g(Z) \doteq Z\}\\
&\Rightarrow \{X \doteq Z, Z \doteq g(Z)\}\\
&\Rightarrow \textbf{failure}
\end{aligned}$$

The set does not have a solved form since Z is a proper subterm of $g(Z)$. ▮

Theorem 3.8 The solved form algorithm in Figure 3.2 terminates and returns an equivalent solved form or **failure** if no such solved form exists. ▮

Proof: First consider termination: Note that case 5b is the only case that may increase the number of symbol occurrences in the set of equations. However, case 5b can be applied at most once for each variable X. Thus, case 5b can be applied only a finite number of times and may introduce only a finite number of new symbol occurrences. Case 2 and case 5a terminate immediately and case 1 and 3 strictly decrease the number of symbol occurrences in the set. Since case 4 cannot be applied indefinitely, but has to be intertwined with the other cases it follows that the algorithm always terminates.

It should be evident that the algorithm either returns **failure** or a set of equations in solved form. Thus, it remains to be shown that each iteration of the algorithm preserves equivalence between successive sets of equations. It is easy to see that if case 2 or 5a apply to some equation in:

$$\{s_1 \doteq t_1, \ldots, s_n \doteq t_n\} \tag{$\mathcal{E}_1$}$$

then the set cannot possibly have a unifier. It is also easy to see that if any of case 1, 3 or 4 apply, then the new set of equations has the same set of unifiers. Finally assume that case 5b applies to some equation $s_i \doteq t_i$. Then the new set is of the form:

$$\{s_1\theta \doteq t_1\theta, \ldots, s_{i-1}\theta \doteq t_{i-1}\theta, s_i \doteq t_i, s_{i+1}\theta \doteq t_{i+1}\theta, \ldots s_n\theta \doteq t_n\theta\} \tag{$\mathcal{E}_2$}$$

where $\theta := \{s_i/t_i\}$. First assume that σ is a unifier of $\mathcal{E}_1$ — that is, $s_j\sigma = t_j\sigma$ for every $1 \leq j \leq n$. In particular, it must hold that $s_i\sigma = t_i\sigma$. Since s_i is a variable which is not a subterm of t_i it must follow that $s_i/t_i\sigma \in \sigma$. Moreover, $\theta\sigma = \sigma$ and it therefore follows that σ is a unifier also of $\mathcal{E}_2$.

Next, assume that σ is a unifier of $\mathcal{E}_2$. Thus, $s_i/t_i\sigma \in \sigma$ and $\theta\sigma = \sigma$ which must then be a unifier also of $\mathcal{E}_1$. ∎

The algorithm presented in Figure 3.2 may be very inefficient. One of the reasons is case 5a; That is, checking if a variable X occurs inside another term t. This is often referred to as the *occur-check*. Assume that the time of occur-check is linear with respect to the size $|t|$ of t.[1] Consider application of the solved form algorithm to the equation:

$$g(X_1, \ldots, X_n) \doteq g(f(X_0, X_0), f(X_1, X_1), \ldots, f(X_{n-1}, X_{n-1}))$$

where $X_0, \ldots, X_n$ are distinct. By case 1 this reduces to:

$$\{X_1 \doteq f(X_0, X_0), X_2 \doteq f(X_1, X_1), \ldots, X_n \doteq f(X_{n-1}, X_{n-1})\}$$

Assume that the equation selected in step i is of the form $X_i = f(\ldots, \ldots)$. Then in the k-th iteration the selected equation is of the form $X_k \doteq \mathcal{T}_k$ where $\mathcal{T}_{i+1} := f(\mathcal{T}_i, \mathcal{T}_i)$ and $\mathcal{T}_0 := X_0$. Hence, $|\mathcal{T}_{i+1}| = 2|\mathcal{T}_i| + 1$. That is, $|\mathcal{T}_n| > 2^n$. This shows the exponential dependency of the unification time on the length of the structures. In this example the growth of the argument lengths is caused by duplication of subterms. As a matter of fact, the same check is repeated many times. Something that could be avoided by sharing various instances of the same structure. In the literature one can find linear algorithms but they are sometimes quite elaborate. On the other hand, Prolog systems usually "solve" the problem simply by omitting the occur-check during unification. Roughly speaking such an approach corresponds to a solved form algorithm where case 5a–b is replaced by:

[1]The size of a term is the total number of constant, variable and functor occurrences in t.

```
$X \doteq t$ where $X \neq t$ and $X$ has more than one occurrence in $\mathcal{E} \Rightarrow$
        replace all other occurrences of $X$ by $t$;            % case 5
```

A pragmatic justification for this solution is the fact that rule 5a (occur check) never is used during the computation of many Prolog programs. There are sufficient conditions which guarantee this, but in general this property is undecidable. The ISO Prolog standard (1995) states that the result of unification is undefined if case 5b can be applied to the set of equations. Strictly speaking, removing case 5a causes looping of the algorithm on equations where case 5a would otherwise apply. For example, an attempt to solve $X \doteq f(X)$ by the modified algorithm will produce a new equation $X \doteq f(f(X))$. However, case 5 is once again applicable yielding $X \doteq f(f(f(f(X))))$ and so forth. In practice many Prolog systems do not loop, but simply bind X to the infinite structure $f(f(f(\ldots)))$. (The notation $X/f(\infty)$ will be used to denote this binding.) Clearly, $\{X/f(\infty)\}$ is an infinite "unifier" of X and $f(X)$. It can easily be represented in the computer by a finite cyclic data structure. But this amounts to generalization of the concepts of term, substitution and unifier for the infinite case not treated in classical logic. Implementation of unification without occur-check may result in unsoundness as will be illustrated in Example 3.21.

Before concluding the discussion about unification we study the notion of most general unifier in more detail. It turns out that the notion of mgu is a subtle one; For instance, there is generally not a unique most general unifier of two terms s and t. A trivial example is the equation $f(X) \doteq f(Y)$ which has at least two mgu's; namely $\{X/Y\}$ and $\{Y/X\}$. Part of the confusion stems from the fact that $\preceq$ ("being more general than") is not an ordering relation. It is reflexive: That is, any substitution θ is "more general" than itself since $\theta = \theta\epsilon$. As might be expected it is also transitive: If $\theta_1 = \theta_2\omega_1$ and $\theta_2 = \theta_3\omega_2$ then obviously $\theta_1 = \theta_3\omega_2\omega_1$. However, $\preceq$ is not anti-symmetric. For instance, consider the substitution $\theta := \{X/Y, Y/X\}$ and the identity substitution ϵ. The latter is obviously more general than θ since $\theta = \epsilon\theta$. But θ is also more general than ϵ, since $\epsilon = \theta\theta$. It may seem odd that two distinct substitutions are more general than one another. Still there is a rational explanation. First consider the following definition:

Definition 3.9 (Renaming) A substitution $\{X_1/Y_1, \ldots, X_n/Y_n\}$ is called a *renaming substitution* iff $Y_1, \ldots, Y_n$ is a permutation of $X_1, \ldots, X_n$. ▮

A renaming substitution represents a *bijective* mapping between variables (or more generally terms). Such a substitution always preserves the structure of a term; if θ is a renaming and t a term, then $t\theta$ and t are equivalent but for the names of the variables. Now, the fact that a renaming represents a bijection implies that there must be an inverse mapping. Indeed, if $\{X_1/Y_1, \ldots, X_n/Y_n\}$ is a renaming then $\{Y_1/X_1, \ldots, Y_n/X_n\}$ is its inverse. We denote the inverse of θ by θ^{-1} and observe that $\theta\theta^{-1} = \theta^{-1}\theta = \epsilon$.

Proposition 3.10 Let θ be an mgu of s and t and assume that ω is a renaming. Then $\theta\omega$ is an mgu of s and t. ▮

The proof of the proposition is left as an exercise. So is the proof of the following proposition:

Proposition 3.11 Let θ and σ be substitutions. If $\theta \preceq \sigma$ and $\sigma \preceq \theta$ then there exists a renaming substitution ω such that $\sigma = \theta\omega$ (and $\theta = \sigma\omega^{-1}$). ▮

Thus, according to the above propositions, the set of all mgu's of two terms is closed under renaming.

3.3 SLD-Resolution

The method of reasoning discussed informally in Section 3.1 can be summarized as the following inference rule:

$$\frac{\forall\,\neg\,(A_1 \wedge \cdots \wedge A_{i-1} \wedge A_i \wedge A_{i+1} \wedge \cdots \wedge A_m) \qquad \forall\,(B_0 \leftarrow B_1 \wedge \cdots \wedge B_n)}{\forall\,\neg\,(A_1 \wedge \cdots \wedge A_{i-1} \wedge B_1 \wedge \cdots \wedge B_n \wedge A_{i+1} \wedge \cdots \wedge A_m)\theta}$$

or (using logic programming notation):

$$\frac{\leftarrow A_1, \ldots, A_{i-1}, A_i, A_{i+1}, \ldots, A_m \qquad B_0 \leftarrow B_1, \ldots, B_n}{\leftarrow (A_1, \ldots, A_{i-1}, B_1, \ldots, B_n, A_{i+1}, \ldots, A_m)\theta}$$

where

(*i*) $A_1, \ldots, A_m$ are atomic formulas;

(*ii*) $B_0 \leftarrow B_1, \ldots, B_n$ is a (renamed) definite clause in P ($n \geq 0$);

(*iii*) $\textsc{mgu}(A_i, B_0) = \theta$.

The rule has two premises — a goal clause and a definite clause. Notice that each of them is separately universally quantified. Thus the scopes of the quantifiers are disjoint. On the other hand, there is only one universal quantifier in the conclusion of the rule. Therefore it is required that the sets of variables in the premises are disjoint. Since all variables of the premises are bound it is always possible to *rename* the variables of the definite clause to satisfy this requirement (that is, to apply some renaming substitution to it).

The goal clause may include several atomic formulas which unify with the head of some clause in the program. In this case it may be desirable to introduce some deterministic choice of the selected atom A_i for unification. In what follows it is assumed that this is given by some function which for a given goal selects the subgoal for unification. The function is called the *selection function* or the *computation rule.* It is sometimes desirable to generalize this concept so that, in one situation, the computation rule selects one subgoal from a goal G but, in another situation, selects another subgoal from G. In that case the computation rule is not a function on goals but something more complicated. However, for the purpose of this book this extra generality is not needed.

The inference rule presented above is the only one needed for definite programs. It is a version of the inference rule called the *resolution principle*, which was introduced by J. A. Robinson in 1965. The resolution principle applies to clauses. Since definite clauses are restricted clauses the corresponding restricted form of resolution presented below is called *SLD-resolution* (Linear resolution for Definite clauses with Selection function).

Next the use of the SLD-resolution principle is discussed for a given definite program P. The starting point, as exemplified in Section 3.1, is a definite goal clause G_0 of the form:

$$\leftarrow A_1, \ldots, A_m \qquad (m \geq 0)$$

From this goal a subgoal A_i is selected (if possible) by the computation rule. A new goal clause G_1 is constructed by selecting (if possible) some renamed program clause $B_0 \leftarrow B_1, \ldots, B_n$ ($n \geq 0$) whose head unifies with A_i (resulting in an mgu θ_1). If so, G_1 will be of the form:

$$\leftarrow (A_1, \ldots, A_{i-1}, B_1, \ldots, B_n, A_{i+1}, \ldots, A_m)\theta_1$$

(According to the requirement above, the variables of the program clause are being renamed so that they are different from those of G_0.) Now it is possible to apply the resolution principle to G_1 thus obtaining G_2, etc. This process may or may not terminate. There are two cases when it is not possible to obtain G_{i+1} from G_i:

- the first is when the selected subgoal cannot be resolved (i.e. is not unifiable) with the head of any program clause;
- the other case appears when $G_i = \Box$ (i.e. the empty goal).

The process described above results in a finite or infinite sequence of goals starting with the initial goal. At every step a program clause (with renamed variables) is used to resolve the subgoal selected by the computation rule $\Re$ and an mgu is created. Thus, the full record of a reasoning step would be a pair $\langle G_i, C_i \rangle$, $i \geq 0$, where G_i is a goal and C_i a program clause with renamed variables. Clearly, the computation rule $\Re$ together with G_i and C_i determines (up to renaming of variables) the mgu (to be denoted θ_{i+1}) produced at the $(i+1)$-th step of the process. A goal G_{i+1} is said to be *derived* (*directly*) from G_i and C_i via $\Re$ (or alternatively, G_i and C_i *resolve* into G_{i+1}).

Definition 3.12 (SLD-derivation) Let G_0 be a definite goal, P a definite program and $\Re$ a computation rule. An *SLD-derivation* of G_0 (using P and $\Re$) is a finite or infinite sequence of goals:

$$G_0 \overset{C_0}{\rightsquigarrow} G_1 \cdots G_{n-1} \overset{C_{n-1}}{\rightsquigarrow} G_n \ldots$$

where each G_{i+1} is derived directly from G_i and a renamed program clause C_i via $\Re$. ∎

Note that since there are usually infinitely many ways of renaming a clause there are formally infinitely many derivations. However, some of the derivations differ only in the names of the variables used. To avoid some technical problems and to make the renaming of variables in a derivation consistent, the variables in the clause C_i of a derivation are renamed by adding the subscript i to *every* variable in the clause. In what follows we consider only derivations where this renaming strategy is used.

Each finite SLD-derivation of the form:

$$G_0 \overset{C_0}{\rightsquigarrow} G_1 \cdots G_{n-1} \overset{C_{n-1}}{\rightsquigarrow} G_n$$

yields a sequence $\theta_1, \ldots, \theta_n$ of mgu's. The composition

$$\theta := \begin{cases} \theta_1\theta_2 \cdots \theta_n & \text{if } n > 0 \\ \epsilon & \text{if } n = 0 \end{cases}$$

of mgu's is called the *computed substitution* of the derivation.

Example 3.13 Consider the initial goal $\leftarrow proud(Z)$ and the program discussed in Section 3.1.

$$\begin{aligned} G_0 &: \leftarrow proud(Z). \\ C_0 &: proud(X_0) \leftarrow parent(X_0, Y_0), newborn(Y_0). \end{aligned}$$

Unification of $proud(Z)$ and $proud(X_0)$ yields e.g. the mgu $\theta_1 = \{X_0/Z\}$. Assume that a computation rule which always selects the leftmost subgoal is used (if nothing else is said, this computation rule is used also in what follows). Such a computation rule will occasionally be referred to as *Prolog's* computation rule since this is the computation rule used by most Prolog systems. The first derivation step yields:

$$\begin{aligned} G_1 &: \leftarrow parent(Z, Y_0), newborn(Y_0). \\ C_1 &: parent(X_1, Y_1) \leftarrow father(X_1, Y_1). \end{aligned}$$

In the second resolution step the mgu $\theta_2 = \{X_1/Z, Y_1/Y_0\}$ is obtained. The derivation then proceeds as follows:

$$\begin{aligned} G_2 &: \leftarrow father(Z, Y_0), newborn(Y_0). \\ C_2 &: father(adam, mary). \end{aligned}$$

$$\begin{aligned} G_3 &: \leftarrow newborn(mary). \\ C_3 &: newborn(mary). \end{aligned}$$

$$G_4 : \square$$

The computed substitution of this derivation is:

$$\begin{aligned} \theta_1\theta_2\theta_3\theta_4 &= \{X_0/Z\}\{X_1/Z, Y_1/Y_0\}\{Z/adam, Y_0/mary\}\epsilon \\ &= \{X_0/adam, X_1/adam, Y_1/mary, Z/adam, Y_0/mary\} \end{aligned}$$

A derivation like the one above is often represented graphically as in Figure 3.1. ▮

Example 3.14 Consider the following definite program:

$$\begin{aligned} 1 &: grandfather(X, Z) \leftarrow father(X, Y), parent(Y, Z). \\ 2 &: parent(X, Y) \leftarrow father(X, Y). \\ 3 &: parent(X, Y) \leftarrow mother(X, Y). \\ 4 &: father(a, b). \\ 5 &: mother(b, c). \end{aligned}$$

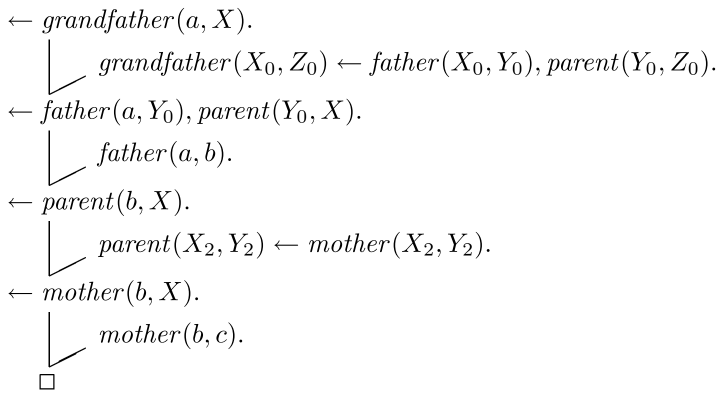

$\leftarrow grandfather(a, X).$
$grandfather(X_0, Z_0) \leftarrow father(X_0, Y_0), parent(Y_0, Z_0).$
$\leftarrow father(a, Y_0), parent(Y_0, X).$
$father(a, b).$
$\leftarrow parent(b, X).$
$parent(X_2, Y_2) \leftarrow mother(X_2, Y_2).$
$\leftarrow mother(b, X).$
$mother(b, c).$
$\Box$

Figure 3.3: SLD-derivation

Figure 3.3 depicts a finite SLD-derivation of the goal $\leftarrow grandfather(a, X)$ (again using Prolog's computation rule). ▮

SLD-derivations that end in the empty goal (and the bindings of variables in the initial goal of such derivations) are of special importance since they correspond to refutations of (and provide answers to) the initial goal:

Definition 3.15 (SLD-refutation) A (finite) SLD-derivation:

$$G_0 \overset{C_0}{\rightsquigarrow} G_1 \cdots G_n \overset{C_n}{\rightsquigarrow} G_{n+1}$$

where $G_{n+1} = \Box$ is called an *SLD-refutation* of G_0. ▮

Definition 3.16 (Computed answer substitution) The computed substitution of an SLD-refutation of G_0 restricted to the variables in G_0 is called a *computed answer substitution* for G_0. ▮

In Examples 3.13 and 3.14 the computed answer substitutions are $\{Z/adam\}$ and $\{X/c\}$ respectively.

For a given initial goal G_0 and computation rule, the sequence $G_1, \ldots, G_{n+1}$ of goals in a finite derivation $G_0 \rightsquigarrow G_1 \cdots G_n \rightsquigarrow G_{n+1}$ is determined (up to renaming of variables) by the sequence $C_0, \ldots, C_n$ of (renamed) program clauses used. This is particularly interesting in the case of refutations. Let:

$$G_0 \overset{C_0}{\rightsquigarrow} G_1 \cdots G_n \overset{C_n}{\rightsquigarrow} \Box$$

be a refutation. It turns out that if the computation rule is changed there still exists another refutation:

$$G_0 \overset{C_0'}{\rightsquigarrow} G_1' \cdots G_n' \overset{C_n'}{\rightsquigarrow} \Box$$

$$\begin{tikzcd}
\leftarrow \mathit{grandfather}(a,X). \arrow[d, "\mathit{grandfather}(X_0,Z_0) \leftarrow \mathit{father}(X_0,Y_0), \mathit{parent}(Y_0,Z_0)."] \\
\leftarrow \mathit{father}(a,Y_0), \mathit{parent}(Y_0,X). \arrow[d, "\mathit{father}(a,b)."] \\
\leftarrow \mathit{parent}(b,X). \arrow[d, "\mathit{parent}(X_2,Y_2) \leftarrow \mathit{father}(X_2,Y_2)."] \\
\leftarrow \mathit{father}(b,X).
\end{tikzcd}$$

Figure 3.4: Failed SLD-derivation

of G_0 which has the same computed answer substitution (up to renaming of variables) and where the sequence $C'_0, \ldots, C'_n$ of clauses used is a permutation of the sequence $C_0, \ldots, C_n$. This property will be called *independence of the computation rule* and it will be discussed further in Section 3.6.

Not all SLD-derivations lead to refutations. As already pointed out, if the selected subgoal cannot be unified with any clause, it is not possible to extend the derivation any further:

Definition 3.17 (Failed derivation) A derivation of a goal clause G_0 whose last element is not empty and cannot be resolved with any clause of the program is called a *failed* derivation. ▮

Figure 3.4 depicts a failed derivation of the program and goal in Example 3.14. Since the selected literal (the leftmost one) does not unify with the head of any clause in the program, the derivation is failed. Note that a derivation is failed even if there is some other subgoal but the selected one which unifies with a clause head.

By a *complete derivation* we mean a *refutation*, a *failed derivation* or an *infinite derivation*. As shown above, a given initial goal clause G_0 may have many complete derivations via a given computation rule $\Re$. This happens if the selected subgoal of some goal can be resolved with more than one program clause. All such derivations may be represented by a possibly infinite tree called the SLD-tree of G_0 (using P and $\Re$).

Definition 3.18 (SLD-tree) Let P be a definite program, G_0 a definite goal and $\Re$ a computation rule. The SLD-tree of G_0 (using P and $\Re$) is a (possibly infinite) labelled tree satisfying the following conditions:

- the root of the tree is labelled by G_0;
- if the tree contains a node labelled by G_i and there is a renamed clause $C_i \in P$ such that G_{i+1} is derived from G_i and C_i via $\Re$ then the node labelled by G_i has a child labelled by G_{i+1}. The edge connecting them is labelled by C_i.

▮

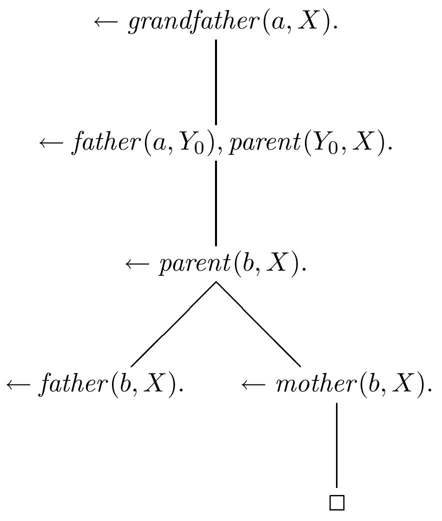


Figure 3.5: SLD-tree of $\leftarrow$ *grandfather*(a, X)

The nodes of an SLD-tree are thus labelled by goals of a derivation. The edges are labelled by the clauses of the program. There is in fact a one-to-one correspondence between the paths of the SLD-tree and the complete derivations of G_0 under a fixed computation rule $\Re$. The sequence:

$$G_0 \overset{C_0}{\rightsquigarrow} G_1 \cdots G_k \overset{C_k}{\rightsquigarrow} \cdots$$

is a complete derivation of G_0 via $\Re$ iff there exists a path of the SLD-tree of the form $G_0, G_1, \ldots, G_k, \ldots$ such that for every i, the edge $\langle G_i, G_{i+1} \rangle$ is labelled by C_i. Usually this label is abbreviated (e.g. by numbering the clauses of the program) or omitted when drawing the tree. Additional labelling with the mgu θ_{i+1} or some part of it may also be included.

Example 3.19 Consider again the program of Example 3.14. The SLD-tree of the goal $\leftarrow$ *grandfather*(a, X) is depicted in Figure 3.5. ▮

The SLD-trees of a goal clause G_0 are often distinct for different computation rules. It may even happen that the SLD-tree for G_0 under one computation rule is finite whereas the SLD-tree of the same goal under another computation rule is infinite. However, the independence of computation rules means that for every refutation path in one SLD-tree there exists a refutation path in the other SLD-tree with the same length and with the same computed answer substitution (up to renaming). The sequences of clauses labelling both paths are permutations of one another.

3.4 Soundness of SLD-resolution

The method of reasoning presented informally in Section 3.1 was formalized as the SLD-resolution principle in the previous section. As a matter of fact one more inference rule is used after construction of a refutation. It applies the computed substitution of the refutation to the body of the initial goal to get the final conclusion. This is the most interesting part of the process since if the initial goal is seen as a query, the computed substitution of the refutation restricted to its variables is an answer to this query. It is therefore called a computed answer substitution. In this context it is also worth noticing the case when no answer substitution exists for a given query. Prolog systems may sometimes discover this and deliver a "no" answer. The logical meaning of "no" will be discussed in the next chapter.

As discussed in Chapter 1, the introduction of formal inference rules raises the questions of their *soundness* and *completeness*. Soundness is an essential property which guarantees that the conclusions produced by the system are correct. Correctness in this context means that they are logical consequences of the program. That is, that they are true in every model of the program. Recall the discussion of Chapter 2 — a definite program describes many "worlds" (i.e. models), including the one which is meant by the user, the intended model. Soundness is necessary to be sure that the conclusions produced by any refutation are true in every world described by the program, in particular in the intended one.

This raises the question concerning the soundness of the SLD-resolution principle. The discussion in Section 3.1 gives some arguments which may by used in a formal proof. However, the intermediate conclusions produced at every step of refutation are of little interest for the user of a definite program. Therefore the soundness of SLD-resolution is usually understood as correctness of computed answer substitutions. This can be stated as the following theorem (due to Clark (1979)).

Theorem 3.20 (Soundness of SLD-resolution) Let P be a definite program, $\Re$ a computation rule and θ an $\Re$-computed answer substitution for a goal $\leftarrow A_1, \dots, A_m$. Then $\forall\,((A_1 \wedge \cdots \wedge A_m)\theta)$ is a logical consequence of the program. ▮

Proof: Any computed answer substitution is obtained by a refutation of the goal via $\Re$. The proof is based on induction over the number of resolution steps of the refutation.

First consider refutations of length one. This is possible only if $m = 1$ and A_1 resolves with some fact A with the mgu θ_1. Hence $A_1\theta_1$ is an instance of A. Now let θ be θ_1 restricted to the variables in A_1. Then $A_1\theta = A_1\theta_1$. It is a well-known fact that the universal closure of an instance of a formula F is a logical consequence of the universal closure of F (cf. exercise 1.9, Chapter 1). Hence the universal closure of $A_1\theta$ is a logical consequence of the clause A and consequently of the program P.

Next, assume that the theorem holds for refutations with $n-1$ steps. Take a refutation with n steps of the form:

$$G_0 \overset{C_0}{\leadsto} G_1 \cdots G_{n-1} \overset{C_{n-1}}{\leadsto} \square$$

where G_0 is the original goal clause $\leftarrow A_1, \dots, A_m$.

Now, assume that A_j is the selected atom in the first derivation step and that C_0 is a (renamed) clause $B_0 \leftarrow B_1, \dots, B_k$ $(k \geq 0)$ in P. Then $A_j\theta_1 = B_0\theta_1$ and G_1 has to be of the form:

$$\leftarrow (A_1, \dots, A_{j-1}, B_1, \dots, B_k, A_{j+1}, \dots, A_m)\theta_1$$

By the induction hypothesis the formula:

$$\forall\,(A_1 \wedge \ldots \wedge A_{j-1} \wedge B_1 \wedge \ldots \wedge B_k \wedge A_{j+1} \wedge \ldots \wedge A_m)\theta_1 \cdots \theta_n \tag{11}$$

is a logical consequence of the program. It follows by definition of logical consequence that also the universal closure of:

$$(B_1 \wedge \ldots \wedge B_k)\theta_1 \cdots \theta_n \tag{12}$$

is a logical consequence of the program. By (11):

$$\forall\,(A_1 \wedge \ldots \wedge A_{j-1} \wedge A_{j+1} \wedge \ldots \wedge A_m)\theta_1 \cdots \theta_n \tag{13}$$

is a logical consequence of P. Now because of (12) and since:

$$\forall\,(B_0 \leftarrow B_1 \wedge \ldots \wedge B_k)\theta_1 \cdots \theta_n$$

is a logical consequence of the program (being an instance of a clause in P) it follows that:

$$\forall\, B_0\theta_1 \cdots \theta_n \tag{14}$$

is a logical consequence of P. Hence by (13) and (14):

$$\forall\,(A_1 \wedge \ldots \wedge A_{j-1} \wedge B_0 \wedge A_{j+1} \wedge \ldots \wedge A_m)\theta_1 \cdots \theta_n \tag{15}$$

is also a logical consequence of the program. But since θ_1 is a most general unifier of B_0 and A_j, B_0 can be replaced by A_j in (15). Now let θ be $\theta_1 \cdots \theta_n$ restricted to the variables in $A_1, \ldots, A_m$ then:

$$\forall\,(A_1 \wedge \ldots \wedge A_m)\theta$$

is a logical consequence of P, which concludes the proof. ▮

It should be noticed that the theorem does not hold if the unifier is computed by a "unification" algorithm without occur-check. For illustration consider the following example.

Example 3.21 A term is said to be *f-constructed* with a term T if it is of the form $f(T, Y)$ for any term Y. A term X is said to be *bizarre* if it is f-constructed with itself. (As discussed in Section 3.2 there are no "bizarre" terms since no term can include itself as a proper subterm.) Finally a term X is said to be *crazy* if it is the second direct substructure of a bizarre term. These statements can be formalized as the following definite program:

$$\begin{array}{l} \mathit{f\_constructed}(f(T,Y),T). \\ \mathit{bizarre}(X) \leftarrow \mathit{f\_constructed}(X,X). \\ \mathit{crazy}(X) \leftarrow \mathit{bizarre}(f(Y,X)). \end{array}$$

Now consider the goal $\leftarrow crazy(X)$ — representing the query "Are there any crazy terms?". There is only one complete SLD-derivation (up to renaming). Namely:

$$
\begin{array}{lcl}
G_0 & : & \leftarrow crazy(X) \\
C_0 & : & crazy(X_0) \leftarrow bizarre(f(Y_0, X_0)) \\
 & & \\
G_1 & : & \leftarrow bizarre(f(Y_0, X)) \\
C_1 & : & bizarre(X_1) \leftarrow f\_constructed(X_1, X_1) \\
 & & \\
G_2 & : & \leftarrow f\_constructed(f(Y_0, X), f(Y_0, X))
\end{array}
$$

The only subgoal in G_2 does not unify with the first program clause because of the occur-check. This corresponds to our expectations: Since, in the intended model, there are no bizarre terms, there cannot be any crazy terms. Since SLD-resolution is sound, if there were any answers to G_0 they would be correct also in the intended model.

Assume now that a "unification" algorithm without occur-check is used. Then the derivation can be extended as follows:

$$
\begin{array}{lcl}
G_2 & : & \leftarrow f\_constructed(f(Y_0, X), f(Y_0, X)) \\
C_2 & : & f\_constructed(f(T_2, Y_2), T_2) \\
 & & \\
G_3 & : & \square
\end{array}
$$

The "substitution" obtained in the last step is $\{X/Y_2, Y_0/f(\infty, Y_2), T_2/f(\infty, Y_2)\}$ (see Section 3.2). The resulting answer substitution is $\{X/Y_2\}$. In other words the conclusion is that every term is crazy, which is not true in the intended model. Thus it is not a logical consequence of the program which shows that the inference is no longer sound. ∎

3.5 Completeness of SLD-resolution

Another important problem is whether all correct answers for a given goal (i.e. all logical consequences) can be obtained by SLD-resolution. The answer is given by the following theorem, called the completeness theorem for SLD-resolution (due to Clark (1979)).

Theorem 3.22 (Completeness of SLD-resolution) Let P be a definite program, $\leftarrow A_1, \ldots, A_n$ a definite goal and $\Re$ a computation rule. If $P \models \forall(A_1 \wedge \cdots \wedge A_n)\sigma$, there exists a refutation of $\leftarrow A_1, \ldots, A_n$ via $\Re$ with the computed answer substitution θ such that $(A_1 \wedge \cdots \wedge A_n)\sigma$ is an instance of $(A_1 \wedge \cdots \wedge A_n)\theta$. ∎

The proof of the theorem is not very difficult but is rather long and requires some auxiliary notions and lemmas. It is therefore omitted. The interested reader is referred to e.g. Apt (1990), Lloyd (1987), Stärk (1990) or Doets (1994).

Theorem 3.22 shows that even if all correct answers cannot be computed using SLD-resolution, every correct answer is an instance of some computed answer. This is

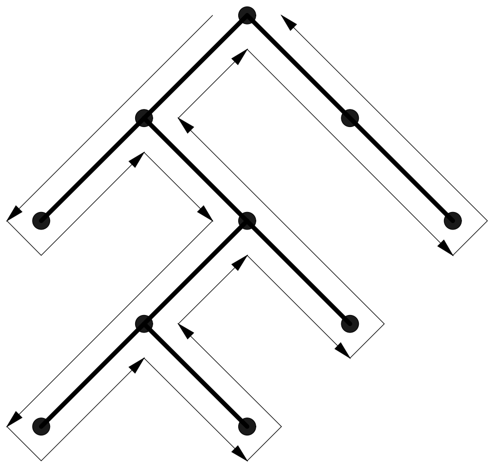


Figure 3.6: Depth-first search with backtracking

due to the fact that only most general unifiers — not arbitrary unifiers — are computed in derivations. However every particular correct answer is a special instance of some computed answer since all unifiers can always be obtained by further instantiation of a most general unifier.

Example 3.23 Consider the goal clause $\leftarrow p(X)$ and the following program:

$$p(f(Y)).$$
$$q(a).$$

Clearly, $\{X/f(a)\}$ is a correct answer to the goal — that is:

$$\{p(f(Y)), q(a)\} \models p(f(a))$$

However, the only computed answer substitution (up to renaming) is $\{X/f(Y_0)\}$. Clearly, this is a more general answer than $\{X/f(a)\}$. ▮

The completeness theorem confirms *existence* of a refutation which produces a more general answer than any given correct answer. However the problem of how to *find* this refutation is still open. The refutation corresponds to a complete path in the SLD-tree of the given goal and computation rule. Thus the problem reduces to a systematic search of the SLD-tree. Existing Prolog systems often exploit some ordering on the program clauses, e.g. the textual ordering in the source program. This imposes the ordering on the edges descending from a node of the SLD-tree. The tree is then traversed in a *depth-first* manner following this ordering. For a finite SLD-tree this strategy is complete. Whenever a leaf node of the SLD-tree is reached the traversal continues by *backtracking* to the last preceding node of the path with unexplored branches (see Figure 3.6). If it is the empty goal the answer substitution of the completed refutation is reported before backtracking. However, as discussed in Section 3.3 the SLD-tree may be infinite. In this case the traversal of the tree will never

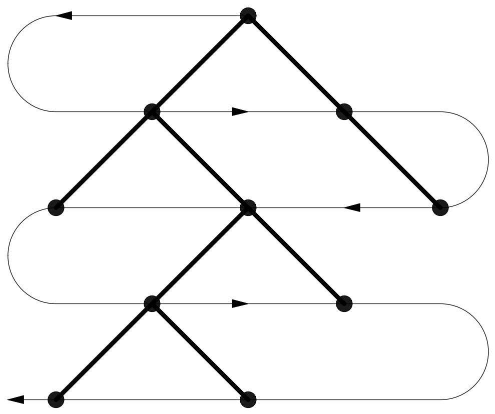


Figure 3.7: Breadth-first search

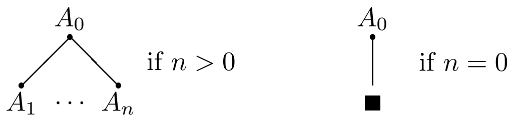
terminate and some existing answers may never be computed. This can be avoided by a different strategy of tree traversal, like for example the *breadth-first* strategy illustrated in Figure 3.7. However this creates technical difficulties in implementation due to very complicated memory management being needed in the general case. Because of this, the majority of Prolog systems use the depth-first strategy for traversal of the SLD-tree.

3.6 Proof Trees

The notion of SLD-derivation resembles the notion of derivation used in formal grammars (see Chapter 10). By analogy to grammars a derivation can be mapped into a graph called a *derivation tree*. Such a tree is constructed by combining together elementary trees representing renamed program clauses. A definite clause of the form:

$$A_0 \leftarrow A_1, \ldots, A_n \qquad (n \geq 0)$$

is said to have an *elementary tree* of one of the forms:



Elementary trees from a definite program P may be combined into *derivation trees* by combining the root of a (renamed) elementary tree labelled by $p(s_1, \ldots, s_n)$ with the leaf of another (renamed) elementary tree labelled by $p(t_1, \ldots, t_n)$. The joint node is labelled by an equation $p(t_1, \ldots, t_n) \doteq p(s_1, \ldots, s_n)$.[2] A derivation tree is said to be *complete* if it is a tree and all of its leaves are labelled by ■. Complete derivation trees are also called *proof trees*. Figure 3.8 depicts a proof tree built out of the following elementary trees from the program in Example 3.14:

[2]Strictly speaking equations may involve terms only. Thus, the notation $p(t_1, \ldots, t_n) \doteq p(s_1, \ldots, s_n)$ should be viewed as a shorthand for $t_1 \doteq s_1, \ldots, t_n \doteq s_n$.

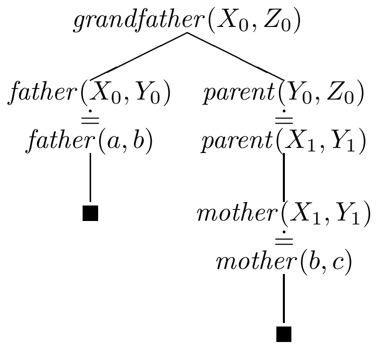
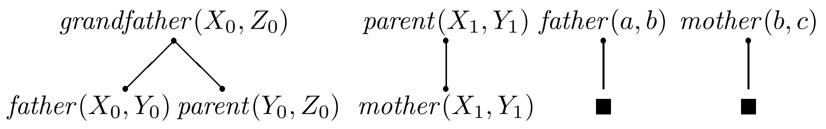


Figure 3.8: Consistent proof tree



A derivation tree or a proof tree can actually be viewed as a collection of equations. In the particular example above:

$$\{X_0 \doteq a, Y_0 \doteq b, Y_0 \doteq X_1, Z_0 \doteq Y_1, X_1 \doteq b, Y_1 \doteq c\}$$

In this example the equations can be transformed into solved form:

$$\{X_0 \doteq a, Y_0 \doteq b, X_1 \doteq b, Z_0 \doteq c, Y_1 \doteq c\}$$

A derivation tree or proof tree whose set of equations has a solution (i.e. can be transformed into a solved form) is said to be *consistent*. Note that the solved form may be obtained in many different ways. The solved form algorithm is not specific as to what equation to select from a set — any selection order yields an equivalent solved form.

Not all derivation trees are consistent. For instance, the proof tree in Figure 3.9 does not contain a consistent collection of equations since the set:

$$\{X_0 \doteq a, Y_0 \doteq b, Y_0 \doteq X_1, Z_0 \doteq c, X_1 \doteq a, Y_1 \doteq b\}$$

does not have a solved form.

The idea of derivation trees may easily be extended to incorporate also atomic *goals*. An atomic goal $\leftarrow A$ may be seen as an elementary tree with a single node, labelled by A, which can only be combined with the root of other elementary trees. For instance, proof tree (a) in Figure 3.10 is a proof tree involving the goal $\leftarrow grandfather(X, Y)$. Note that the solved form of the associated set of equations provides an answer to the initial goal — for instance, the solved form:

$$\{X \doteq a, Y \doteq c, X_0 \doteq a, Y_0 \doteq b, Y_0 \doteq X_1, Z_0 \doteq Y_1, X_1 \doteq b, Y_1 \doteq c\}$$

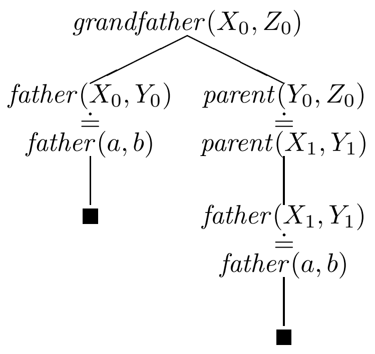


Figure 3.9: Inconsistent proof tree

of the equations associated with proof tree (a) in Figure 3.10 provides an answer substitution $\{X/a, Y/c\}$ to the initial goal.

The solved form of the equations in a consistent derivation tree can be used to simplify the derivation tree by instantiating the labels of the tree. For instance, applying the substitution $\{X/a, Y/c, X_0/a, Y_0/b, Z_0/c, X_1/b, Y_1/c\}$ (corresponding to the solved form above) to the nodes in the proof tree yields a new proof tree (depicted in Figure 3.11). However, nodes labelled by equations of the form $A \doteq A$ will usually be abbreviated A so that the tree in Figure 3.11 is instead written as the tree (d) in Figure 3.10. The equations of the simplified tree are clearly consistent.

Thus the search for a consistent proof tree can be seen as two interleaving processes: The process of combining elementary trees and the simplification process working on the equations of the already constructed part of the derivation tree. Note in particular that it is not necessary to simplify the whole tree at once — the tree (a) has the following associated equations:

$$\{X \doteq X_0, Y \doteq Z_0, X_0 \doteq a, Y_0 \doteq b, \underline{Y_0 \doteq X_1}, \underline{Z_0 \doteq Y_1}, X_1 \doteq b, Y_1 \doteq c\}$$

Instead of solving all equations only the underlined equations may be solved, resulting in an mgu $\theta_1 = \{Y_0/X_1, Z_0/Y_1\}$. This may be applied to the tree (a) yielding the tree (b). The associated equations of the new tree can be obtained by applying θ_1 to the previous set of equations after having removed the previously solved equations:

$$\{X \doteq X_0, Y \doteq Y_1, X_0 \doteq a, X_1 \doteq b, \underline{X_1 \doteq b}, \underline{Y_1 \doteq c}\}$$

Solving of the new underlined equations yields a mgu $\theta_2 = \{X_1/b, Y_1/c\}$ resulting in the tree (c) and a new set of equations:

$$\{\underline{X \doteq X_0}, \underline{Y \doteq c}, \underline{X_0 \doteq a}, \underline{b \doteq b}\}$$

Solving all of the remaining equations yields $\theta_3 = \{X/a, Y/c, X_0/a\}$ and the final tree (d) which is trivially consistent.

Notice that we have not mentioned *how* proof trees are to be constructed or in which order the equations are to be solved or checked for consistency. In fact, a whole spectrum of strategies is possibile. One extreme is to first build a complete proof

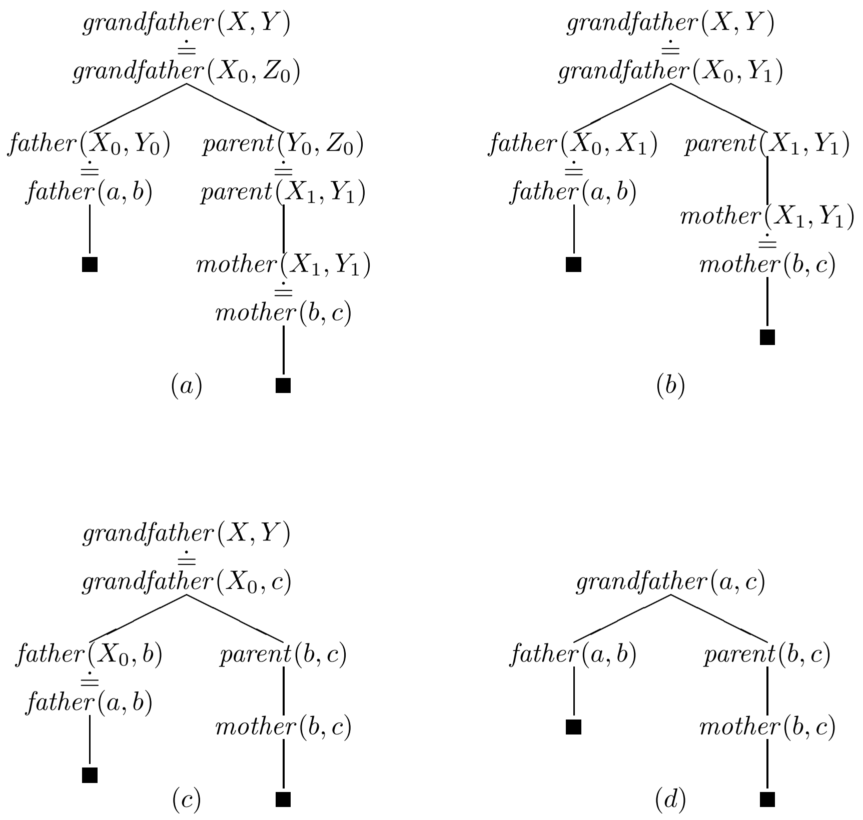


Figure 3.10: Simplification of proof tree

tree and then check if the equations are consistent. At the other end of the spectrum equations may be checked for consistency while building the tree. In this case there are two possibilities — either the whole set of equations is checked every time a new equation is added or the tree is simplified by trying to solve equations as soon as they are generated. The latter is the approach used in Prolog — the tree is built in a depth-first manner from left to right and each time a new equation is generated the tree is simplified.

From the discussion above it should be clear that many derivations may map into the same proof tree. This is in fact closely related to the intuition behind the independence of the computation rule — take "copies" of the clauses to be combined together. Rename each copy so that it shares no variables with the other copies. The clauses are then combined into a proof tree. A computation rule determines the order in which the equations are to be solved but the solution obtained is independent of this order (up to renaming of variables).

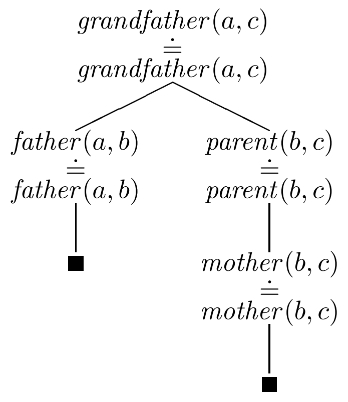


Figure 3.11: Resolved proof tree

Exercises

3.1 What are the mgu's of the following pairs of atoms:

$p(X, f(X))$	$p(Y, f(a))$
$p(f(X), Y, g(Y))$	$p(Y, f(a), g(a))$
$p(X, Y, X)$	$p(f(Y), a, f(Z))$
$p(a, X)$	$p(X, f(X))$

3.2 Let θ be an mgu of s and t and ω a renaming substitution. Show that $\theta\omega$ is an mgu of s and t.

3.3 Let θ and σ be substitutions. Show that if $\theta \preceq \sigma$ and $\sigma \preceq \theta$ then there exists a renaming substitution ω such that $\sigma = \theta\omega$.

3.4 Let θ be an idempotent mgu of s and t. Prove that σ is a unifier of s and t iff $\sigma = \theta\sigma$.

3.5 Consider the following definite program:

$$\begin{array}{l} p(Y) \leftarrow q(X, Y), r(Y). \\ p(X) \leftarrow q(X, X). \\ q(X, X) \leftarrow s(X). \\ r(b). \\ s(a). \\ s(b). \end{array}$$

Draw the SLD-tree of the goal $\leftarrow p(X)$ if Prolog's computation rule is used. What are the computed answer substitutions?

3.6 Give an example of a definite program, a goal clause and two computation rules where one computation rule leads to a finite SLD-tree and where the other computation rule leads to an infinite tree.

3.7 How many consistent proof trees does the goal $\leftarrow p(a, X)$ have given the program:

$$\begin{array}{l} p(X, Y) \leftarrow q(X, Y). \\ p(X, Y) \leftarrow q(X, Z), p(Z, Y). \\ q(a, b). \\ q(b, a). \end{array}$$

3.8 Let θ be a renaming substitution. Show that there is only one substitution σ such that $\sigma\theta = \theta\sigma = \epsilon$.

3.9 Show that if $A \in B_P$ and $\leftarrow A$ has a refutation of length n then $A \in T_P \uparrow n$.

Chapter 4

Negation in Logic Programming

4.1 Negative Knowledge

Definite programs express positive knowledge; the facts and the rules describe that certain objects are in certain relations with one another. The relations are made explicit in the least Herbrand model — the set of all ground atomic consequences of the program. For instance, consider the following program:

$$
\begin{array}{l}
above(X, Y) \leftarrow on(X, Y).\\
above(X, Y) \leftarrow on(X, Z), above(Z, Y).\\
on(c, b).\\
on(b, a).
\end{array}
$$

The program describes the situation depicted in Figure 1.2: The object 'C' is on top of 'B' which is on top of 'A' and an object is above a second object if it is either on top of it or on top of a third object which is above the second object. The least Herbrand model of the program looks as follows:

$$\{on(b, a), on(c, b), above(b, a), above(c, b), above(c, a)\}$$

Note that neither the program nor its least Herbrand model include negative information, such as 'A' is not on top of any box or 'B' is not above 'C'. Also in real life the negative information is seldom stated explicitly. Swedish Rail in its timetable explicitly states that there is a daily train from Linköping to Stockholm scheduled to depart at 9:22, but it does not explicitly state that there is no train departing at 9:56 or 10:24.

Thus, in many real-life situations the lack of information is taken as evidence to the contrary — since the timetable does not indicate a departure from Linköping to Stockholm at 10:24 one does not plan to take such a train. This is because we

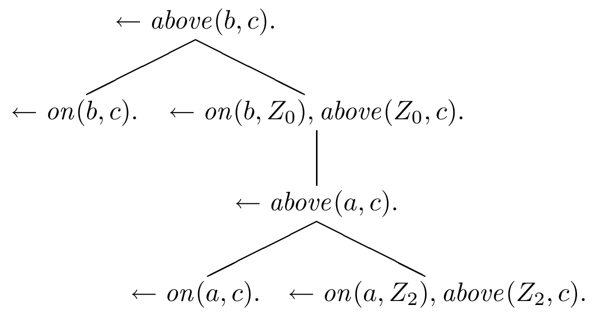

$\leftarrow above(b, c).$
$\leftarrow on(b, c).$
$\leftarrow on(b, Z_0), above(Z_0, c).$
$\leftarrow above(a, c).$
$\leftarrow on(a, c).$
$\leftarrow on(a, Z_2), above(Z_2, c).$

Figure 4.1: Finitely failed SLD-tree

assume that the timetable lists *all* trains from Linköping to Stockholm. This idea is the intuition behind the so-called *closed world assumption* (*cwa*).

The closed world assumption is a mechanism that allows us to draw negative conclusions based on the lack of positive information. The *cwa* is a rule which is used to derive the statement $\neg A$ provided that A is a ground atomic formula which cannot be derived by the inference rules of the used system, e.g. by SLD-resolution. This can be expressed as the following "inference rule"[1]:

$$\frac{P \nvdash A}{\neg A} \quad (cwa)$$

In the case of a sound and complete derivability relation the condition $P \nvdash A$ is equivalent to $P \not\models A$. Thus, in the case of SLD-resolution (which is both sound and complete) the condition could be replaced by $A \notin M_P$. For instance, the formula $above(b, c)$ is not derivable from the program P by means of SLD-resolution, see Figure 4.1 (nor by any other sound system, since $above(b, c)$ is not a logical consequence of P). By the completeness of SLD-resolution it follows that $P \not\models above(b, c)$. Thus, using *cwa* it can be inferred that $\neg above(b, c)$.

However there are several problems with the *cwa*. One is that non-provability for definite programs is undecidable in the general case. Thus it is not possible to determine if the rule is applicable or not. A somewhat weaker version of the *cwa* rule therefore is to say that $\neg A$ is derivable from P if the goal $\leftarrow A$ has a *finitely failed SLD-tree w.r.t. P*:

$$\frac{\leftarrow A \text{ has a finitely failed SLD-tree}}{\neg A} \quad (naf)$$

This is called the *negation as (finite) failure* rule (*naf*). It should be contrasted with the *cwa* which may also be called the *negation as infinite failure* rule. To illustrate the difference between the two approaches the program above is extended with the following (obviously true) clause:

$$above(X, Y) \leftarrow above(X, Y).$$

[1]To be more precise, this would not really qualify as an inference rule in the traditional sense since $P \nvdash A$ is not a *logic formula*, but a statement of the meta language.

The SLD-tree of the goal $\leftarrow above(b, c)$ still contains no refutations but the tree is now infinite. Thus, it cannot be concluded that $\neg above(b, c)$ using *naf*. However, it still follows from the *cwa*.

A second, and more serious problem with the *cwa* (and the *naf*) is that it is unsound — $\neg above(b, c)$ is not a logical consequence of the program(s) above. In fact, any proof system that permits inferring a negative literal from a definite program is unsound! The reason is that the Herbrand base B_P (in which all ground atomic formulas are true) is always a model of P. There are two principal approaches to repair this problem. One is to view the program as a shorthand for another, larger, program from which the negative literal follows. The second approach is to redefine the notion of logical consequence so that only some of the models of program (e.g. the least Herbrand model) are considered. The effect in both cases is to discard some "uninteresting" models of the program. The first part of this chapter gives a logical justification of the *naf*-rule using the *completion* $comp(P)$ of a definite program P.

Once able to draw negative conclusions it is natural to extend the language of definite programs to permit the use of negative literals in the body of a clause. The final part of this chapter therefore introduces the language of general (logic) programs and introduces the notion of SLDNF-resolution that combines the SLD inference rule with the negation as finite failure rule. The two final sections of the chapter survey two alternative approaches to justify inference of negative conclusions from general programs. The first idea attempts to repair some problems with trying to justify negation as *finite* failure in terms of the program completion. The second approach, called the *well-founded semantics*, generalizes the closed world assumption (and thus, negation as *infinite* failure) from definite to general programs. Both approaches are based on an extension of classical logic from two into three truth-values.

4.2 The Completed Program

As pointed out above, any proof system that allows negative literals to be derived from a definite program is unsound. The objective of this section therefore is to give a logical justification of the *naf*-rule. The idea presented below is due to K. Clark (1978) and relies on the claim that when writing a definite program P the programmer really means something more than just a set of definite clauses. The "intended program" can be formalized and is called the *completion* of P. Consider the following definition:

$$\begin{array}{l} above(X, Y) \leftarrow on(X, Y). \\ above(X, Y) \leftarrow on(X, Z), above(Z, Y). \end{array}$$

The rules state that an object is above a second object if the first object is (1) on top of the second object *or* (2) on top of a third object which is above the second object. This could also be written thus:

$$above(X, Y) \leftarrow on(X, Y) \vee (on(X, Z), above(Z, Y))$$

Now what if the *if*-statement is instead replaced by an *if-and-only-if*-statement?

$$above(X, Y) \leftrightarrow on(X, Y) \vee (on(X, Z), above(Z, Y))$$

This formula states that X is above Y if and only if at least one of the conditions are true. That is, if none of the conditions hold it follows that X is *not* above Y! This is the intuition used to explain the negation as failure.

Unfortunately combining definite clauses as illustrated above is only possible if the clauses have identical heads. Thus, consider the following clauses:

$$\begin{array}{l} on(c, b). \\ on(b, a). \end{array}$$

By simple transformation the program can be rewritten as follows:

$$\begin{array}{l} on(X_1, X_2) \leftarrow X_1 \doteq c, X_2 \doteq b. \\ on(X_1, X_2) \leftarrow X_1 \doteq b, X_2 \doteq a. \end{array}$$

These clauses can be combined into one formula, where implication is replaced by the equivalence connective. In that way two if-statements have been combined into one if-and-only-if-statement:

$$on(X_1, X_2) \leftrightarrow (X_1 \doteq c, X_2 \doteq b) \vee (X_1 \doteq b, X_2 \doteq a)$$

The logical reading of this is that X_1 is on top of X_2 if and only if either $X_1 = c$ and $X_2 = b$ or $X_1 = b$ and $X_2 = a$.

The idea outline above will now be formalized as a special transformation of a definite program P. The resulting set of formulas is called the *completion of* P:

Definition 4.1 (Completed program) Let P be a definite program. The completion *comp*(P) of P is the formulas obtained from the following three transformation steps:

(i) For every predicate symbol p replace each clause C of the form:

$$p(t_1, \ldots, t_m) \leftarrow L_1, \ldots, L_n \qquad (n \geq 0)$$

by the formula:

$$p(X_1, \ldots, X_m) \leftarrow \exists Y_1, \ldots, Y_i (X_1 \doteq t_1, \ldots, X_m \doteq t_m, L_1, \ldots, L_n)$$

where $Y_1, \ldots, Y_i$ are all the variables in C and $X_1, \ldots, X_m$ are distinct variables which do not appear in C.

Remark: Note that the first step does not really change the logical understanding of the clauses provided that the appropriate definition of $\doteq$ is added (this is discussed in the third step).

(ii) For each predicate symbol p replace all formulas:

$$\begin{array}{c} p(X_1, \ldots, X_m) \leftarrow B_1 \\ \vdots \\ p(X_1, \ldots, X_m) \leftarrow B_j \end{array}$$

by the formula:

$$\begin{array}{ll} \forall X_1,\ldots,X_m(p(X_1,\ldots,X_m) \leftrightarrow B_1 \vee \cdots \vee B_j) & \text{if } j > 0 \\ \forall X_1,\ldots,X_m(\neg p(X_1,\ldots,X_m)) & \text{if } j = 0 \end{array}$$

(*iii*) Finally the program is extended with the following *free equality axioms* defining the equalities introduced in step (*i*) (To be more precise some of the axioms are axiom schemata making the resulting set of axioms infinite):

$$\forall(X \doteq X) \qquad (\mathcal{E}_1)$$
$$\forall(X \doteq Y \supset Y \doteq X) \qquad (\mathcal{E}_2)$$
$$\forall(X \doteq Y \wedge Y \doteq Z \supset X \doteq Z) \qquad (\mathcal{E}_3)$$
$$\forall(X_1 \doteq Y_1 \wedge \cdots \wedge X_n \doteq Y_n \supset f(X_1,\ldots,X_n) \doteq f(Y_1,\ldots,Y_n)) \qquad (\mathcal{E}_4)$$
$$\forall(X_1 \doteq Y_1 \wedge \cdots \wedge X_n \doteq Y_n \supset (p(X_1,\ldots,X_n) \supset p(Y_1,\ldots,Y_n))) \qquad (\mathcal{E}_5)$$
$$\forall(f(X_1,\ldots,X_n) \doteq f(Y_1,\ldots,Y_n) \supset X_1 \doteq Y_1 \wedge \cdots \wedge X_n \doteq Y_n) \qquad (\mathcal{E}_6)$$
$$\forall(\neg f(X_1,\ldots,X_m) \doteq g(Y_1,\ldots,Y_n)) \quad (\text{if } f/m \neq g/n) \qquad (\mathcal{E}_7)$$
$$\forall(\neg X \doteq t) \quad (\text{if } X \text{ is a proper subterm of } t) \qquad (\mathcal{E}_8)$$

▮

The free equality axioms enforce $\doteq$ to be interpreted as the identity relation in all Herbrand interpretations. Axioms $\mathcal{E}_1 - \mathcal{E}_3$ must be satisfied in order for $\doteq$ to be an equivalence relation. Axioms $\mathcal{E}_4 - \mathcal{E}_5$ enforcing $\doteq$ to be a congruence relation. Axioms $\mathcal{E}_1 - \mathcal{E}_5$ are sometimes dropped and replaced by the essential constraint that $\doteq$ always denotes the identity relation. Therefore the most interesting axioms are $\mathcal{E}_6 - \mathcal{E}_8$ which formalize the notion of unification. They are in fact similar to cases 1, 2 and 5a in the solved form algorithm. Axiom $\mathcal{E}_6$ states that if two compound terms with the same functor are equal, then the arguments must be pairwise equal. Axiom $\mathcal{E}_7$ states that two terms with distinct functors/constants are not equal and axiom $\mathcal{E}_8$ essentially states that no nesting of functions can return one of its arguments as its result.

Example 4.2 We now construct the completion of the *above*/2-program again. The first step yields:

$$\begin{array}{c} above(X_1,X_2) \leftarrow \exists X,Y(X_1 \doteq X, X_2 \doteq Y, on(X,Y)) \\ above(X_1,X_2) \leftarrow \exists X,Y,Z(X_1 \doteq X, X_2 \doteq Y, on(X,Z), above(Z,Y)) \\ on(X_1,X_2) \leftarrow (X_1 \doteq c, X_2 \doteq b) \\ on(X_1,X_2) \leftarrow (X_1 \doteq b, X_2 \doteq a) \end{array}$$

Step two yields:

$$\begin{array}{c} \forall X_1,X_2(above(X_1,X_2) \leftrightarrow \exists X,Y(\ldots) \vee \exists X,Y,Z(\ldots)) \\ \forall X_1,X_2(on(X_1,X_2) \leftrightarrow (X_1 \doteq c, X_2 \doteq b) \vee (X_1 \doteq b, X_2 \doteq a)) \end{array}$$

Finally the program is extended with the free equality axioms described above. ▮

Example 4.3 Consider the following program that describes a “world” containing two parents only, Mary and Kate, both of which are female:

$$\begin{aligned}&father(X) \leftarrow male(X), parent(X).\\&parent(mary).\\&parent(kate).\end{aligned}$$

Step one of the transformation yields:

$$\begin{gathered}father(X_1) \leftarrow \exists X(X_1 \doteq X, male(X), parent(X))\\parent(X_1) \leftarrow X_1 \doteq mary\\parent(X_1) \leftarrow X_1 \doteq kate\end{gathered}$$

Step two yields:

$$\begin{gathered}\forall X_1(father(X_1) \leftrightarrow \exists X(X_1 \doteq X, male(X), parent(X)))\\\forall X_1(parent(X_1) \leftrightarrow X_1 \doteq mary \vee X_1 \doteq kate)\\\forall X_1(\neg male(X_1))\end{gathered}$$

Note the presence of the formula $\forall X_1(\neg male(X_1))$ embodying the fact that there are no males in the world under consideration. ∎

The completion $comp(P)$ of a definite program P preserves all the positive literals entailed by P. Hence, if $P \models A$ then $comp(P) \models A$. It can also be shown that no information is lost when completing P, i.e. $comp(P) \models P$ (see exercise 4.4) and that no *positive* information is added, i.e. if $comp(P) \models A$ then $P \models A$. Thus, in transforming P into $comp(P)$ no information is removed from P and only negative information is added to the program. As concerns negative information it was previously concluded that no negative literal can be a logical consequence of a definite program. However, by replacing the implications in P by equivalences in $comp(P)$ it becomes possible to infer negative information from the *completion* of a definite program. This is the traditional way of justifying the *naf*-rule, whose soundness is due to Clark (1978):

Theorem 4.4 (Soundness of negation as finite failure) Let P be a definite program and $\leftarrow A$ a definite goal. If $\leftarrow A$ has a finitely failed SLD-tree then $comp(P) \models \forall(\neg A)$. ∎

Note that soundness is preserved even if A is not ground. For instance, since the goal $\leftarrow on(a, X)$ is finitely failed, it follows that $comp(P) \models \forall(\neg on(a, X))$.

The next theorem, due to Jaffar, Lassez and Lloyd (1983), shows that negation as finite failure is also complete (the proof of this theorem as well as the soundness theorem can also be found in Lloyd (1987), Apt (1990) or Doets (1994)):

Theorem 4.5 (Completeness of negation as finite failure) Let P be a definite program. If $comp(P) \models \forall(\neg A)$ then there exists a finitely failed SLD-tree of $\leftarrow A$. ∎

Note that the theorem only states the *existence* of a finitely failed SLD-tree. As already pointed out in Chapter 3 it may very well happen that the SLD-tree of a goal is finite under one computation rule, but infinite under another. In particular, the theorem does not hold if the computation rule is fixed to that of Prolog. However, the situation is not quite as bad as it may first seem.

An SLD-derivation is said to be *fair* if it is either finite or every occurrence of an atom (or its instance) in the derivation is eventually selected by the computation rule. An SLD-tree is said to be fair if all its derivations are fair. Jaffar, Lassez and Lloyd showed that the completeness result above holds for any fair SLD-tree. Clearly, derivations in Prolog are not fair so negation as failure as implemented in Prolog is not complete. Fair SLD-derivations can be implemented by always selecting the leftmost subgoal and appending new subgoals to the end of the goal. However, very few logic programming systems support fair derivations for efficiency reasons.

4.3 SLDNF-resolution for Definite Programs

In the previous chapter SLD-resolution was introduced as a means of proving that some instance of a positive literal is a logical consequence of a definite program (and its completion). Then, in the previous section, it was concluded that also negative literals can be derived from the completion of a definite program. By combining SLD-resolution with negation as finite failure it is possible to generalize the notion of goal to include both positive and negative literals. Such goals are called general goals:

Definition 4.6 (General goal) A *general goal* is a goal of the form:

$$\leftarrow L_1, \ldots, L_n. \qquad (n \geq 0)$$

where each L_i is a positive or negative literal. ▮

The combination of SLD-resolution, to resolve positive literals, and negation as (finite) failure, to resolve negative literals, is called *SLDNF-resolution*:

Definition 4.7 (SLDNF-resolution for definite programs) Let P be a definite program, G_0 a general goal and $\Re$ a computation rule. An *SLDNF-derivation* of G_0 (using P and $\Re$) is a finite or infinite sequence of general goals:

$$G_0 \overset{C_0}{\rightsquigarrow} G_1 \cdots G_{n-1} \overset{C_{n-1}}{\rightsquigarrow} G_n \cdots$$

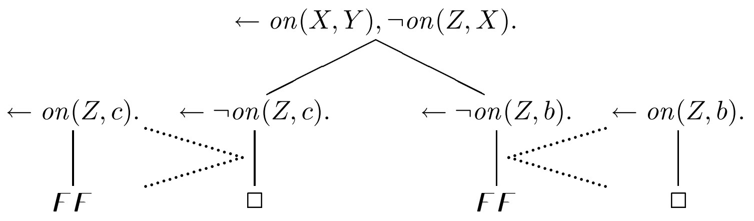
where $G_i \overset{C_i}{\rightsquigarrow} G_{i+1}$ if either:

(i) the $\Re$-selected literal in G_i is positive and G_{i+1} is derived from G_i and C_i by one step of SLD-resolution;

(ii) the $\Re$-selected literal in G_i is of the form $\neg A$, the goal $\leftarrow A$ has a finitely failed SLD-tree and G_{i+1} is obtained from G_i by removing $\neg A$ (in which case C_i is a special marker FF).

Each step of an SLDNF-derivation produces a substitution — in the case of (i) an mgu and in the case of (ii) the empty substitution. ▮

Thus, a negative literal $\neg A$ succeeds if $\leftarrow A$ has a finitely failed SLD-tree. Dually, $\neg A$ *finitely fails* if $\leftarrow A$ succeeds. It may also happen that $\leftarrow A$ has an infinite SLD-tree without refutations (i.e. infinite failure). Hence, apart from *refutations* and *infinite derivations* there are two more classes of complete SLDNF-derivations under a given computation rule:



$\leftarrow on(X,Y), \neg on(Z,X).$
$\leftarrow on(Z,c).$
$\leftarrow \neg on(Z,c).$
$\leftarrow \neg on(Z,b).$
$\leftarrow on(Z,b).$
FF
□
FF
□

Figure 4.2: SLDNF-derivations of $\leftarrow on(X,Y), \neg on(Z,X)$

- A derivation is said to be (*finitely*) *failed* if (1) the selected literal is positive and does not unify with the head of any clause or (2) the selected literal is negative and finitely failed;
- A derivation is said to be *stuck* if the selected subgoal is of the form $\neg A$ and $\leftarrow A$ is infinitely failed;

Example 4.8 Consider the following program describing a world where the block 'C' is piled on top of 'B' which is on top of 'A':

$$on(c,b).$$
$$on(b,a).$$

As shown in Figure 4.2 the goal $\leftarrow on(X,Y), \neg on(Z,X)$ has an SLDNF-refutation with the computed answer substitution $\{X/c, Y/b\}$. (In the figure failed derivations are terminated with the marker FF.) ▮

As might be expected SLDNF-resolution is sound. After all, both SLD-resolution and negation as finite failure are sound:

Theorem 4.9 (Soundness of SLDNF-resolution) Let P be a definite program and $\leftarrow L_1, \ldots, L_n$ a general goal. If $\leftarrow L_1, \ldots, L_n$ has an SLDNF-refutation with the computed answer substitution θ, then $comp(P) \models \forall(L_1\theta \wedge \cdots \wedge L_n\theta)$. ▮

However, contrary to what might be expected, SLDNF-resolution is not complete even though both SLD-resolution and negation as finite failure are complete. A simple counter-example is the goal $\leftarrow \neg on(X,Y)$ which intuitively corresponds to the query:

"Are there any blocks, X and Y, such that X is not on top of Y?" (†)

One expects several answers to this query. For instance, 'A' is not on top of any block. However, the only SLDNF-derivation of $\leftarrow \neg on(X,Y)$ fails since the goal $\leftarrow on(X,Y)$ succeeds. The root of the problem is that our notion of failed SLDNF-derivation is too conservative. The success of $\leftarrow on(X,Y)$ does not necessarily mean that there is *no* block which is not on top of another block — only that there *exists* at least one block which is on top of another block. Of course, if $\leftarrow on(X,Y)$ succeeds with the *empty computed answer substitution* then we can conclude that every block is on top of every block in which case (†) should be answered negatively.

The problem stems from the fact that negation as finite failure, in contrast to SLD-resolution, is only a test. Remember that according to the definition of SLDNF-resolution and soundness and completeness of negation as finite failure it holds that:

$$\begin{array}{lll} \neg on(X,Y) \text{ succeeds} & \text{iff} & \leftarrow on(X,Y) \text{ has a finitely failed SLD-tree} \\ & \text{iff} & comp(P) \models \forall(\neg on(X,Y)) \end{array}$$

Hence the goal $\leftarrow \neg on(X,Y)$ should not be read as an existential query but rather as a universal test:

"For all blocks, X and Y, is X not on top of Y?"

This query has a negative answer in the intended model, since e.g. 'B' is on top of 'A'. The problem above is due to the quantification of the variables in the negative literal. If the query above is rephrased as $\leftarrow \neg on(a,b)$ then SLDNF-resolution yields a refutation since $\leftarrow on(a,b)$ finitely fails. It is sometimes assumed that the computation rule is only allowed to select a negative literal $\neg A$ if A is ground or if $\leftarrow A$ has an empty computed answer substitution. Such computation rules are said to be *safe*. We will return to this issue when extending SLDNF-resolution to programs containing negative literals.

4.4 General Logic Programs

Once able to infer both negative and positive literals it is natural to extend the language of definite programs to include clauses that contain both positive and negative literals in bodies. Such formulas are called general clauses:[2]

Definition 4.10 (General clause) A general clause is a formula:

$$A_0 \leftarrow L_1, \ldots, L_n$$

where A_0 is an atomic formula and $L_1, \ldots, L_n$ are literals ($n \geq 0$). ▮

Thus, by analogy to definite clauses and definite programs:

Definition 4.11 (General program) A *general* (*logic*) *program* is a finite set of general clauses. ▮

By means of general clauses it is possible to extend the "blocks world" with the following relations:

$$\begin{array}{l} founding(X) \leftarrow on(Y,X), on\_ground(X). \\ on\_ground(X) \leftarrow \neg off\_ground(X). \\ off\_ground(X) \leftarrow on(X,Y). \\ on(c,b). \\ on(b,a). \end{array}$$

[2]General programs are sometimes also called normal logic programs in the literature.

The first clause states that a founding block is one which is on the ground and has another block on top of it. The second clause states that a block which is not off ground is on the ground and the third clause says that a block which is on top of another block is off ground.

The new language of general programs introduces a number of subtleties in addition to those already touched upon earlier in this chapter. For instance, it is not obviously clear how to understand a general program logically. Moreover, given a particular logical understanding, what kind of proof system should be used?

There is no single answer to these questions and only some of them will be addressed here. The remaining part of this chapter will mainly be devoted to the idea initiated above — that of program *completion* and *SLDNF-resolution*. However, some alternative approaches will also be discussed.

Although the language of programs has now been enriched it is still not possible for a negative literal to be a logical consequence of a general program P. The reason is the same as for definite programs — the Herbrand base is a model of P in which all negative literals are false. By analogy to definite programs question arises how to interpret general programs in order to allow for "sound" negative inferences to be made. Fortunately, the notion of program completion can be applied also to general programs. For instance, the completion of:

$$win(X) \leftarrow move(X, Y), \neg win(Y).$$

contains the formula:

$$\forall X_1 (win(X_1) \leftrightarrow \exists X, Y (X_1 \doteq X, move(X, Y), \neg win(Y)))$$

However, the completion of general programs sometimes leads to paradoxical situations. Consider the following general clause:

$$p \leftarrow \neg p.$$

Then the completed program contains the formula $p \leftrightarrow \neg p$. The inconsistency of the completed program is due to $p/0$ being defined in terms of its own complement. Such situations can be avoided by employing a special discipline when writing programs. The idea is to build the program in "layers" (called *strata*), thereby enforcing the programmer not to refer to the negation of a relation until the relation is fully defined (in a lower stratum). The following is a formal definition of the class of stratified programs (let the subset of all clauses in P with p in the head be denoted P^p):

Definition 4.12 (Stratified program) A general program P is said to be *stratified* iff there exists a partitioning $P_1 \cup \cdots \cup P_n$ of P such that:[3]

- if $p(\ldots) \leftarrow \ldots, q(\ldots), \ldots \in P_i$ then $P^q \subseteq P_1 \cup \cdots \cup P_i$;
- if $p(\ldots) \leftarrow \ldots, \neg q(\ldots), \ldots \in P_i$ then $P^q \subseteq P_1 \cup \cdots \cup P_{i-1}$.

∎

For instance, the following program is stratified:

[3]Note that there are often many partitionings of a program that satisfy the requirements.

P_2 :	$\mathit{founding}(X) \leftarrow \mathit{on}(Y,X), \mathit{on\_ground}(X).$
	$\mathit{on\_ground}(X) \leftarrow \neg \mathit{off\_ground}(X).$
P_1 :	$\mathit{off\_ground}(X) \leftarrow \mathit{on}(X,Y).$
	$\mathit{on}(c,b).$
	$\mathit{on}(b,a).$

It was shown by Apt, Blair and Walker (1988) that the completion of a stratified program is always consistent so that the situation described above cannot occur. However, stratification is only a sufficient condition for consistency; To determine if a general program is stratified is decidable, but the problem of determining if the completion of a general program is consistent or not is undecidable. Hence there are general programs which are not stratified but whose completion is consistent.

For stratified programs there is also a natural restatement of the least Herbrand model. It can be made in terms of the immediate consequence operator originally defined for definite programs. However, general clauses may contain negative literals. Thus, if I is a Herbrand interpretation we note that $I \models A$ iff $A \in I$ and $I \models \neg A$ iff $A \notin I$. The revised immediate consequence operator T_P is defined as follows:

$$T_P(I) \quad := \quad \{A_0 \mid A_0 \leftarrow L_1, \ldots, L_n \in \mathit{ground}(P) \wedge I \models L_1, \ldots, L_n\}$$

Then let $T_P \uparrow \omega(I)$ denote the limit of the sequence:

$$\begin{aligned} T_P \uparrow 0(I) \quad &:= \quad I \\ T_P \uparrow (n+1)(I) \quad &:= \quad T_P(T_P \uparrow n(I)) \cup T_P \uparrow n(I) \end{aligned}$$

Now, consider a program stratified by $P = P_1 \cup \cdots \cup P_n$. It is possible to define a canonical Herbrand model of P stratum-by-stratum as follows:

$$\begin{aligned} M_1 \quad &:= \quad T_{P_1} \uparrow \omega(\varnothing) \\ M_2 \quad &:= \quad T_{P_2} \uparrow \omega(M_1) \\ &\vdots \\ M_n \quad &:= \quad T_{P_n} \uparrow \omega(M_{n-1}) \end{aligned}$$

Apt, Blair and Walker (1988) showed that $M_P := M_n$ is a *minimal* Herbrand model — called the *standard model* — of P. It was also shown that the model does not depend on how the program is partitioned (as long as it is stratified). For instance, the standard model of the program $P_1 \cup P_2$ above may be constructed thus:

$$\begin{aligned} M_1 \quad &= \quad \{\mathit{on}(b,a), \mathit{on}(c,b), \mathit{off\_ground}(b), \mathit{off\_ground}(c)\} \\ M_2 \quad &= \quad \{\mathit{on\_ground}(a), \mathit{founding}(a)\} \cup M_1 \end{aligned}$$

The model conforms with our intuition of the intended model. However, in contrast to definite programs M_P is not necessarily the only minimal Herbrand model. For instance, the program:

$$\mathit{loops} \leftarrow \neg \mathit{halts}$$

has two minimal Herbrand models — the standard model $\{loops\}$ and a non-standard model $\{halts\}$. This is obviously a consequence of the fact that the clause is logically equivalent to $loops \vee halts$ (and $halts \leftarrow \neg loops$). However, by writing this as an implication with $loops/0$ in the consequent it is often argued that unless there is evidence for $halts/0$ from the rest of the program we should prefer to conclude $loops/0$. For instance, the following is an alternative (counter-intuitive) minimal model of the program $P_1 \cup P_2$ above:

$$\{on(b, a), on(c, b), off\_ground(a), off\_ground(b), off\_ground(c)\}$$

We return to the issue of canonical models of general programs when introducing the well-founded semantics in Section 4.7.

4.5 SLDNF-resolution for General Programs

In Section 4.3 the notion of SLDNF-resolution for definite programs and general goals was introduced. Informally speaking SLDNF-resolution combines the SLD-resolution principle with the following principles:

$$\neg A \text{ succeeds iff } \leftarrow A \text{ has a finitely failed SLD-tree}$$
$$\neg A \text{ finitely fails iff } \leftarrow A \text{ has an SLD-refutation}$$

When moving from definite to general programs the situation gets more complicated — in order to prove $\neg A$ there must be a finitely failed tree for $\leftarrow A$. But that tree may contain new negative literals which may either succeed or finitely fail. This complicates the definition of SLDNF-resolution for general programs quite a bit. For instance, paradoxical situations may occur when predicates are defined in terms of their own complement. Consider the non-stratified program:

$$p \leftarrow \neg p$$

Given an initial goal $\leftarrow p$ a derivation $\leftarrow p \rightsquigarrow \leftarrow \neg p$ can be constructed. The question is, whether the derivation can be completed. It can be extended into a refutation if $\leftarrow p$ finitely fails. Alternatively, if $\leftarrow p$ has a refutation then the derivation fails. Both cases are clearly impossible since $\leftarrow p$ cannot have a refutation and be finitely failed at the same time!

We now introduce the notions of SLDNF-derivation and SLDNF-tree, similar to the notions of SLD-derivation and SLD-tree used for SLD-resolution. Thus, an SLDNF-derivation is a sequence of general goals and an SLDNF-tree the combination of all possible SLDNF-derivations of a given initial goal under a fixed computation rule. It is difficult to introduce the notions separately since SLDNF-derivations have to be defined in terms of SLDNF-trees and vice versa. Instead both notions are introduced in parallel by the following notion of an *SLDNF-forest*. To simplify the definition the following technical definitions are first given: A *forest* is a set of *trees* whose nodes are labelled by general goals. A *subforest* of F is any forest obtained by removing some of the nodes (and all their children) in F. Two forests F_1 and F_2 are considered to be equivalent if they contain trees equal up to renaming of variables. Moreover, F_1 is said to be *smaller* than F_2 if F_1 is equivalent to a subforest of F_2. Now the SLDNF-forest of a goal is defined as follows:

Definition 4.13 (SLDNF-forest) Let P be a general program, G_0 a general goal and $\Re$ a computation rule. The *SLDNF-forest* of G_0 is the smallest forest (modulo renaming of variables) such that:

(i) G_0 is the root of a tree;

(ii) if G is a node in the forest whose selected literal is positive then for each clause C such that G' can be derived from G and C (with mgu θ), G has a child labelled G'. If there is *no* such clause then G has a single child labelled FF;

(iii) if G is a node in the forest whose selected literal is of the form $\neg A$ (that is, G is of the form $\leftarrow L_1, \ldots, L_{i-1}, \neg A, L_{i+1}, \ldots, L_{i+j}$), then:

- the forest contains a tree with root labelled $\leftarrow A$;
- if the tree with root $\leftarrow A$ has a leaf $\Box$ with the empty computed answer substitution, then G has a single child labelled FF;
- if the tree with the root labelled $\leftarrow A$ is finite and all leaves are labelled FF, then G has a single child labelled $\leftarrow L_1, \ldots, L_{i-1}, L_{i+1}, \ldots, L_{i+j}$ (the associated substitution is ϵ);

∎

Note that a selected negative literal $\neg A$ fails only if $\leftarrow A$ has a refutation with the *empty computed answer substitution*. As will be shown below, this condition, which was not needed when defining SLDNF-resolution for definite programs, is absolutely *vital* for the soundness of SLDNF-resolution of general programs.

The trees of the SLDNF-forest are called (complete) SLDNF-trees and the sequence of all goals in a branch of an SLDNF-tree with root G is called a complete SLDNF-derivation of G (under P and $\Re$). The tree labelled by G_0 is called the *main tree*. A tree with root $\leftarrow A$ is called *subsidiary* if $\neg A$ is a selected literal in the forest. (As shown below the main tree may also be a subsidiary tree.)

Example 4.14 Consider the following stratified program P:

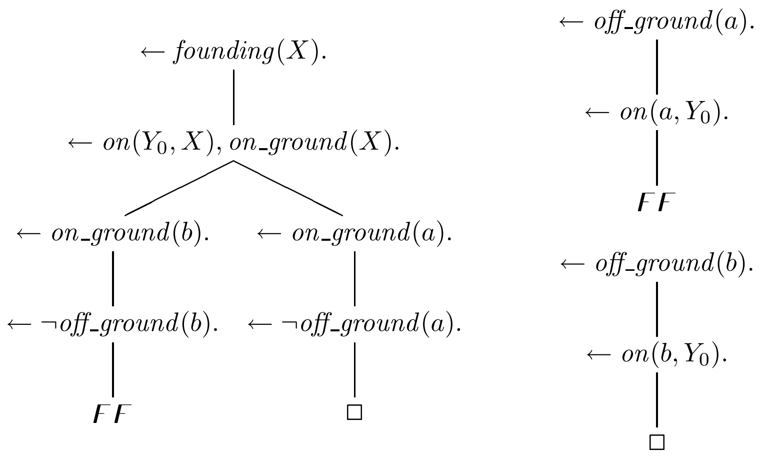
$$\begin{array}{l}
\mathit{founding}(X) \leftarrow \mathit{on}(Y,X), \mathit{on\_ground}(X).\\
\mathit{on\_ground}(X) \leftarrow \neg \mathit{off\_ground}(X).\\
\mathit{off\_ground}(X) \leftarrow \mathit{on}(X,Y).\\
\mathit{above}(X,Y) \leftarrow \mathit{on}(X,Y).\\
\mathit{above}(X,Y) \leftarrow \mathit{on}(X,Z), \mathit{above}(Z,Y).\\
\mathit{on}(c,b).\\
\mathit{on}(b,a).
\end{array}$$

The SLDNF-forest of $\leftarrow \mathit{founding}(X)$ is depicted in Figure 4.3. The main tree contains one failed derivation and one refutation with the computed answer substitution $\{X/a\}$.

∎

The branches of an SLDNF-tree in the SLDNF-forest represent all complete SLDNF-derivations of its root under the computation rule $\Re$. There are four kinds of *complete* SLDNF-derivations:

- *infinite* derivations;



$\leftarrow founding(X).$
$\leftarrow on(Y_0, X), on\_ground(X).$
$\leftarrow on\_ground(b).$
$\leftarrow on\_ground(a).$
$\leftarrow \neg off\_ground(b).$
$\leftarrow \neg off\_ground(a).$
FF
□
$\leftarrow off\_ground(a).$
$\leftarrow on(a, Y_0).$
FF
$\leftarrow off\_ground(b).$
$\leftarrow on(b, Y_0).$
□

Figure 4.3: SLDNF-forest of $\leftarrow founding(X)$

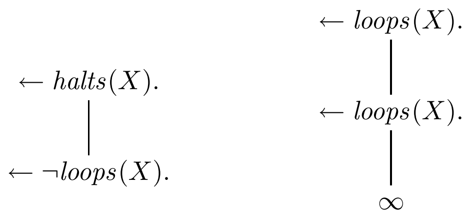


Figure 4.4: SLDNF-forest of $\leftarrow halts(X)$

- (*finitely*) *failed* derivations (that end in FF);
- *refutations* (that end in □);
- *stuck* derivations (if none of the previous apply).

Figure 4.3 contains only refutations and failed derivations. However, consider the following example:

$$\begin{aligned} &halts(X) \leftarrow \neg loops(X). \\ &loops(X) \leftarrow loops(X). \end{aligned}$$

The SLDNF-forest depicted in Figure 4.4 contains an infinite derivation (of the subsidiary goal $\leftarrow loops(X)$) and a stuck derivation (of $\leftarrow halts(X)$). This illustrates one cause of a stuck derivation — when a subsidiary tree contains only infinite or failed derivations. There are two more reasons why a derivation may get stuck. First consider the following program:

$$\begin{aligned} &paradox(X) \leftarrow \neg ok(X). \\ &ok(X) \leftarrow \neg paradox(X). \end{aligned}$$

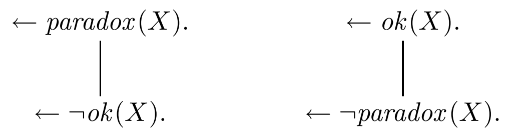


Figure 4.5: SLDNF-forest of $\leftarrow paradox(X)$

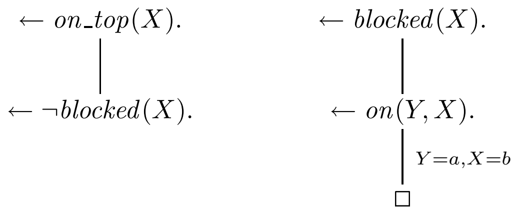


Figure 4.6: SLDNF-forest of $\leftarrow on\_top(X)$

Figure 4.5 depicts the SLDNF-forest of the goal $\leftarrow paradox(X)$. (The figure also illustrates an example where the main tree is also a subsidiary tree.) This forest contains two stuck derivations because the program contains a "loop through negation" — in order for $\leftarrow paradox(X)$ to be successful (resp. finitely failed) the derivation of $\leftarrow ok(X)$ must be finitely failed (resp. successful). However, in order for $\leftarrow ok(X)$ to be finitely failed (resp. successful) the derivation of $\leftarrow paradox(X)$ must be successful (resp. finitely failed).

Note that Definition 4.13 enforce the SLDNF-forest to be the *least* forest satisfying conditions (*i*)–(*iii*). If minimality of the forest is dropped, it is possible to get out of the looping situation above — for instance, it would be consistent with (*i*)–(*iii*) to extend $\leftarrow ok(X)$ into a refutation if $\leftarrow paradox(X)$ at the same time was finitely failed (or vice versa).

The last cause of a stuck derivation is demonstrated by the following example:

$$
\begin{aligned}
&on\_top(X) \leftarrow \neg blocked(X).\\
&blocked(X) \leftarrow on(Y, X).\\
&on(a, b).
\end{aligned}
$$

Clearly $on\_top(a)$ should be derived from the program. However, the SLDNF-tree of the goal $\leftarrow on\_top(X)$ in Figure 4.6 contains no refutation. Note that the derivation of $\leftarrow on\_top(X)$ is stuck even though $\leftarrow blocked(X)$ has a refutation. The reason why the goal $\leftarrow on\_top(X)$ is not finitely failed is that $\leftarrow blocked(X)$ does not have an *empty* computed answer substitution. Note also that it would be *very* counter-intuitive if $\leftarrow on\_top(X)$ had been finitely failed since it would have implied that no element was on top. This last case, when a subsidiary SLDNF-tree has at least one refutation, but none with the empty computed answer substitution is usually referred to as *floundering.* Floundering can sometimes be avoided by making sure that negative literals are selected by the computation rule only when they become ground. However, checking statically whether a negative literal ever becomes ground is undecidable. What is even more unfortunate: most Prolog systems do not even check dynamically if a derivation is floundering. That is, most Prolog systems assume that $\neg A$ is finitely failed if $\leftarrow A$ has *any* refutation. Consider the goal $\leftarrow \neg on\_top(X)$ corresponding to the query "is there an element which is not on top". We expect the answer b, but Prolog would answer incorrectly that all elements are not on top, i.e. $\forall(\neg on\_top(X))$. The reason is that Prolog considers $\leftarrow on\_top(X)$ to be finitely failed since $\leftarrow blocked(X)$ has a refutation. Thus, SLDNF-resolution as implemented in most Prolog systems is unsound. However, SLDNF-resolution as defined in Definition 4.13 is sound:

Theorem 4.15 (Soundness of SLDNF-resolution) If P is a general program and $\leftarrow L_1, \ldots, L_n$ a general goal then:

- If $\leftarrow L_1, \ldots, L_n$ has a computed answer substitution θ then:

$$comp(P) \models \forall(L_1\theta \wedge \cdots \wedge L_n\theta)$$

- If $\leftarrow L_1, \ldots, L_n$ has a finitely failed SLDNF-tree then:

$$comp(P) \models \forall(\neg(L_1 \wedge \cdots \wedge L_n))$$

▮

Example 4.16 Consider the following stratified program:

$$\begin{array}{l}
go\_well\_together(X, Y) \leftarrow \neg incompatible(X, Y) \\
incompatible(X, Y) \leftarrow \neg likes(X, Y) \\
incompatible(X, Y) \leftarrow \neg likes(Y, X) \\
likes(X, Y) \leftarrow harmless(Y). \\
likes(X, Y) \leftarrow eats(X, Y). \\
harmless(rabbit). \\
eats(python, rabbit).
\end{array}$$

As shown in Figure 4.7 $go\_well\_together(rabbit, rabbit)$ is a logical consequence of the completion. Also $\neg incompatible(rabbit, rabbit)$ is a logical consequence due to the finite failure of $\leftarrow incompatible(rabbit, rabbit)$. ▮

The definition of the SLDNF-forest should not be viewed as an implementation of SLDNF-resolution — it only represents an ideal computation space in which soundness can be guaranteed. The definition of the SLDNF-forest does not specify in which order the trees and the derivations should be constructed. This is similar to the notion of SLD-tree. Like in the case of SLD-resolution, Prolog uses a depth-first strategy when constructing an SLDNF-tree. If a negative literal ($\neg A$) is encountered the construction of the tree is temporarily suspended until either the tree for $\leftarrow A$ is finitely failed or a refutation of $\leftarrow A$ is found. As already pointed out, a major problem with most Prolog systems (and the ISO Prolog standard (1995)) is that a negative literal $\neg A$ is considered to be finitely failed if $\leftarrow A$ has a refutation — no check is made to see if the empty computed answer substitution was produced. This is yet another source of unsound conclusions of Prolog systems in addition to the lack of occur-check! An implementation of negation as finite failure as implemented in most Prolog systems is given in the next chapter.

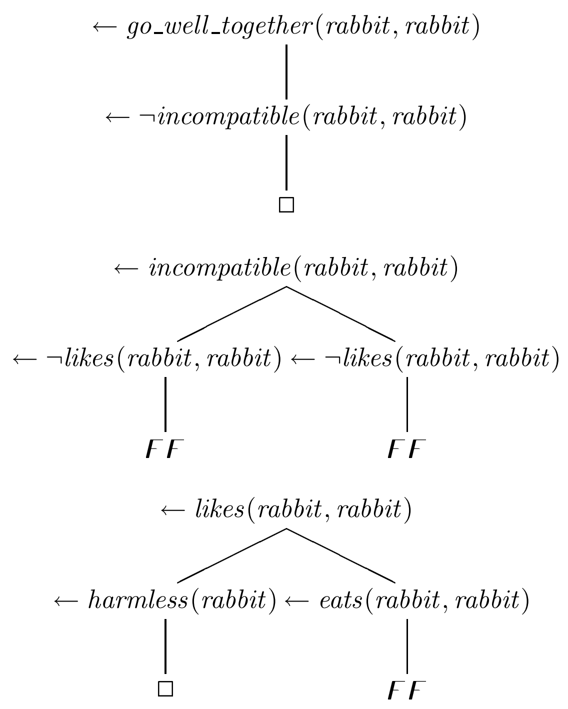


Figure 4.7: SLDNF-forest of $\leftarrow go\_well\_together(rabbit, rabbit)$

4.6 Three-valued Completion

Traditionally the SLDNF-resolution principle is justified within classical logic in terms of Clark's program completion. However, there are some problems with that approach — in particular, there are some noticeable mismatches between SLDNF-resolution and the program completion. For instance, there are consistent general programs which do not have a consistent completion:

$$p \leftrightarrow \neg p \in comp(p \leftarrow \neg p)$$

Thus, by classical logic, anything can be inferred from the completed program. However, the SLDNF-tree for $\leftarrow p$ is stuck, thus leading to incompleteness. Another anomaly of the program completion shows up in case of:

$$p \leftrightarrow (q \vee \neg q) \quad \in \quad comp\begin{pmatrix} p \leftarrow q. \\ p \leftarrow \neg q. \end{pmatrix}$$

By the law of the excluded middle, $comp(P) \models p$. This is certainly in accordance with our intuition if $\leftarrow q$ either succeeds or finitely fails, but what if it does not? For instance, what if q is defined by $q \leftarrow q$? Then $\leftarrow q$ has neither a refutation nor a finitely failed SLD(NF)-tree. Consequently the SLDNF-tree of $\leftarrow p$ is stuck.

Both of the problems above were repaired by Kunen (1987; 1989) and Fitting (1985) who introduced the notion of *three-valued* program completion of a general program.

In classical logic, formulas take on two truth-values — a formula is either true or false. In a three-valued (or partial) logic, formulas are also allowed to be undefined. The intuition behind the extra truth-value is to model diverging computations. Thus, in terms of SLDNF-resolution, "true" means successful, "false" means finitely failed and "undefined" means diverging. It is convenient to encode the truth-values as 0 (false), $\frac{1}{2}$ (undefined) and 1 (true) in which case the truth-value of compound formulas can then be defined as follows:

$$\begin{array}{lcl}
\Im_\sigma(\neg F) & := & 1 - \Im_\sigma(F) \\
\Im_\sigma(F \wedge G) & := & \min\{\Im_\sigma(F), \Im_\sigma(G)\} \\
\Im_\sigma(F \vee G) & := & \max\{\Im_\sigma(F), \Im_\sigma(G)\} \\
\Im_\sigma(F \leftarrow G) & := & \text{if } \Im_\sigma(F) < \Im_\sigma(G) \text{ then } 0 \text{ else } 1 \\
\Im_\sigma(F \leftrightarrow G) & := & \text{if } \Im_\sigma(F) = \Im_\sigma(G) \text{ then } 1 \text{ else } 0 \\
\Im_\sigma(\forall X F) & := & \min\{\Im_{\sigma[X \mapsto t]}(F) \mid t \in |\Im|\} \\
\Im_\sigma(\exists X F) & := & \max\{\Im_{\sigma[X \mapsto t]}(F) \mid t \in |\Im|\}
\end{array}$$

Most concepts from classical logic have natural counterparts in this three-valued logic: Let F be a closed formula. An interpretation $\Im$ is called a *model* of F iff $\Im(F) = 1$. By analogy to classical logic this is written $\Im \models_3 F$. Similarly, if P is a set of closed formulas, then F is a *logical consequence* of a P (denoted $P \models_3 F$) iff every model of P is a model of F.

Also by analogy to two-valued Herbrand interpretations, a three-valued or, as it will be called, *partial Herbrand interpretation* $\Im$ will be written as a set of literals with the restriction that not both A and $\neg A$ may be members of $\Im$. Hence, a literal L is true in $\Im$ iff $L \in \Im$. A ground atomic formula A is undefined in $\Im$ if neither $A \in \Im$ nor $\neg A \in \Im$. Thus, $\Im$ uniquely determines the truth-value of ground literals. A partial Herbrand interpretation such that either $A \in \Im$ or $\neg A \in \Im$ for each ground atom A is said to be *total* (or two-valued).

It can be shown that SLDNF-resolution is sound with respect to the three-valued completion:

Theorem 4.17 (Soundness of SLDNF-resolution revisited) Let P be a general program and $\leftarrow L_1, \ldots, L_n$ a general goal.

- If $\leftarrow L_1, \ldots, L_n$ has a computed answer substitution θ then:

$$comp(P) \models_3 \forall(L_1\theta \wedge \cdots \wedge L_n\theta)$$

- If $\leftarrow L_1, \ldots, L_n$ has a finitely failed SLDNF-tree then:

$$comp(P) \models_3 \forall(\neg(L_1 \wedge \cdots \wedge L_n))$$

∎

No *general* completeness result for SLDNF-resolution is available with respect to the three-valued completion. However, the situation is not as bad as in the case of two-valued completion. With a slightly modified notion of SLDNF-resolution it was shown by Drabent (1995a) that the only sources of incompleteness are floundering and unfair selection of literals in SLDNF-derivations. As already discussed floundering is an undecidable property but sufficient conditions may be imposed to guarantee the absence of floundering. One simple (and rather weak) sufficient condition is that every variable in a clause or in the goal occurs in a positive body literal. A general program P and a goal G are said to be *allowed* if all clauses of P and G satisfy this condition. Kunen (1989) showed the following completeness result for allowed programs:

Theorem 4.18 (Completeness of SLDNF-resolution) If P is an allowed program and $\leftarrow L_1, \ldots, L_n$ an allowed goal then:

- If $comp(P) \models_3 \forall((L_1 \wedge \cdots \wedge L_n)\theta)$ then $\leftarrow L_1, \ldots, L_n$ has a computed answer substitution θ.
- If $comp(P) \models_3 \forall(\neg(L_1 \wedge \cdots \wedge L_n))$ then $\leftarrow L_1, \ldots, L_n$ has a finitely failed SLDNF-tree.

∎

To illustrate the difference between two-valued and three-valued completion consider the following examples:

Example 4.19 Let P be the following (allowed) program:

$$\begin{array}{l} p \leftarrow \neg p. \\ q. \end{array}$$

Classically $comp(P)$ is inconsistent, but $\{q\}$ is a three-valued model of $comp(P)$. In fact, $comp(P) \models_3 q$. Consider also the following allowed program:

$$\begin{array}{l} p \leftarrow q \\ p \leftarrow \neg q \\ q \leftarrow q \end{array}$$

whose completion looks as follows:

$$\begin{array}{c} p \leftrightarrow (q \vee \neg q) \\ q \leftrightarrow q \end{array}$$

Classically $comp(P) \models p$. However, $\leftarrow p$ has a stuck SLDNF-tree. In the three-valued setting there is one interpretation — namely $\varnothing$ (where both p and q are undefined) — which is a model of $comp(P)$ but not a model of p. Thus, $comp(P) \not\models_3 p$. ∎

4.7 Well-founded Semantics

Program completion attempts to capture the intuition behind negation as *finite* failure — an imperfect alternative to the *closed world assumption* or negation as *infinite* failure. To illustrate the difference between the two, consider the following programs:

$$\begin{array}{llll} P_1: & halts(a). & P_2: & halts(a). \\ & & & halts(b) \leftarrow halts(b). \end{array}$$

Logically, both programs are equivalent. However, $comp(P_1)$ and $comp(P_2)$ are not equivalent. In particular:

$$comp(P_1) \models \neg halts(b) \quad \text{whereas} \quad comp(P_2) \not\models \neg halts(b)$$

On the other hand, under the closed world assumption defined as follows:

$$\begin{aligned} cwa(P) &:= P \cup \{\neg A \mid A \in B_P \text{ and } \leftarrow A \text{ has no SLD-refutation}\} && (9)\\ &= P \cup \{\neg A \mid A \in B_P \text{ and } P \not\models A\} && (10) \end{aligned}$$

the programs are indistinguishable:

$$cwa(P_1) \models \neg halts(b) \quad \text{and} \quad cwa(P_2) \models \neg halts(b)$$

(Since $\neg halts(b) \in cwa(P_i)$ for $i \in \{1,2\}$.) The question arises if the closed world assumption can be generalized to general programs. Definition (10) leads to problems as illustrated when P is of the form.

$$loops(a) \leftarrow \neg halts(a).$$

Since neither $P \models loops(a)$ nor $P \models halts(a)$ it follows that $cwa(P)$ is inconsistent. Definition (9) also makes little sense since SLD-resolution is only defined for definite programs. Note that it would not make sense to replace SLD-resolution by SLDNF-resolution:

$$cwa(P) := P \cup \{\neg A \mid A \in B_P \text{ and } \leftarrow A \text{ has no SLDNF-refutation}\}$$

since if P is extended by $halts(a) \leftarrow halts(a)$ then $cwa(P)$ is inconsistent. (Neither $loops(a)$ nor $halts(a)$ have an SLDNF-refutation.)

To avoid these inconsistency problems the closed world assumption will be identified with one particular Herbrand model of the program — called its *canonical model*. In the case of definite programs it is natural to adopt the *least Herbrand model* as the canonical model (as discussed on p. 60).[4] However for general programs it is not obvious which model to select. We first specify some properties that a canonical model should satisfy. Foremost, it has to be a model. Let T_P be defined as follows:

$$T_P(I) := \{H \mid H \leftarrow L_1, \ldots, L_n \in ground(P) \wedge I \models L_1, \ldots, L_n\}$$

It can be shown that:

Theorem 4.20 If P is a general program and I a Herbrand interpretation of P, then I is a model of P iff $T_P(I) \subseteq I$. ∎

Moreover, it is reasonable to assume that an atom A is true in the canonical model only if there is some constructive support for A:

[4]Note, also that a canonical model I may also be viewed as a (possibly infinite) program of ground literals:

$$\{A \mid A \in B_P \text{ and } I \models A\} \cup \{\neg A \mid A \in B_P \text{ and } I \models \neg A\}$$

Definition 4.21 (Supported interpretation) Let P be a general program. A Herbrand interpretation I of P is said to be *supported* iff for each $I \models A$ there exists some $A \leftarrow L_1, \ldots, L_n \in \mathit{ground}(P)$ such that $I \models L_1, \ldots, L_n$. ▮

Theorem 4.22 Let P be a general program and I a Herbrand interpretation. Then I is supported iff $I \subseteq T_P(I)$. ▮

Thus, a canonical model should be a fixed point of the T_P-operator. However, the program:

$$\mathit{loops}(a) \leftarrow \mathit{loops}(a).$$

has two supported models: $\{\mathit{loops}(a)\}$ and $\varnothing$. We therefore require that the canonical model is also *minimal*. This means that I is canonical if I is a minimal fixed point of T_P. Clearly this is satisfied by the least Herbrand model of a definite program.

Unfortunately there are general programs which have more than one minimal supported Herbrand model (see exercise 4.13) and, perhaps more seriously, there are programs which have no such model. For instance the general program $p \leftarrow \neg p$ has only one Herbrand model, $\{p\}$, which it is not supported and therefore not a fixed point of the T_P-operator. The crux is that $p/0$ is defined in terms of its own complement.

The problems just illustrated can be rectified by resorting to partial (or three-valued) Herbrand interpretations instead of two-valued ones. (Recall that a partial Herbrand interpretation I is a set of ground literals where not both $A \in I$ and $\neg A \in I$.) For instance, the partial interpretation $\varnothing$ is a model of the program above. In this interpretation $p/0$ is undefined. To deal with partial Herbrand interpretations we make the following modifications to our definitions:

$$T_P(I) := \{H \mid H \leftarrow L_1, \ldots, L_n \in \mathit{ground}(P) \wedge I \models_3 L_1, \ldots, L_n\}$$

A partial Herbrand interpretation I of P is said to be *supported* iff for each $I \models_3 A$ there exists some $A \leftarrow L_1, \ldots, L_n \in \mathit{ground}(P)$ such that $I \models_3 L_1, \ldots, L_n$.

Now consider a partial interpretation I. In order for I to be a model of P it is necessary that if $A \in T_P(I)$ then $A \in I$. Similarly, in order for I to be supported it is required that if $A \in I$ then $A \in T_P(I)$. Thus, in order for I to be a canonical partial model of P we require that:

$$A \in I \quad \text{iff} \quad A \in T_P(I) \tag{C_1}$$

As concerns false atoms the situation is more complicated and we have to introduce the auxiliary notion of an *unfounded set* to characterize atoms that must be false. Assume that a partial interpretation I is given describing literals which are *known to be true*. Informally, an atom A is false (i.e. is contained in an unfounded set) if each grounded clause $A \leftarrow L_1, \ldots, L_n$ either (1) contains a literal which is *false in* I or (2) contains a positive literal which is in an unfounded set. (As a special case A is false if there is no grounded clause with A as its head.)

Definition 4.23 (Unfounded set) Let I be a partial Herbrand interpretation. A subset U of the Herbrand base is called an *unfounded set* of P with respect to I if for each $A \in U$ at least one of the following holds for every $A \leftarrow L_1, \ldots, L_n \in \mathit{ground}(P)$:

- Some $L \in L_1, \ldots, L_n$ is false in I;
- Some positive literal $A \in L_1, \ldots, L_n$ is in U.

∎

For a given program P and partial interpretation I there is a unique greatest unfounded set which should be thought of as the set of all ground atomic formulas which are false provided that all literals in I are true. The *greatest unfounded set* of P w.r.t. I will be denoted $F_P(I)$.

Example 4.24 If P is a definite program then $F_P(\varnothing)$ is always equivalent to the complement of the least Herbrand model. That is, without any knowledge about the truth and falsity of body literals, the unfounded set is the set of all atoms which are false in the least Herbrand model. For instance, let P be:

$$\begin{array}{l} \mathit{odd}(s(0)). \\ \mathit{odd}(s(s(X))) \leftarrow \mathit{odd}(X). \end{array}$$

Then $F_P(\varnothing) = \{\mathit{odd}(s^{2n}(0)\}$. Next let P be a general program:

$$\begin{array}{l} \mathit{odd}(s(s(X))) \leftarrow \mathit{odd}(X). \\ \mathit{even}(X) \leftarrow \neg\mathit{odd}(X). \\ \mathit{odd}(s(0)). \end{array}$$

Then $F_P(\varnothing) = \{\mathit{odd}(s^{2n}(0))\}$. Thus, every atom $\mathit{odd}(s^{2n}(0))$ is false. Moreover, $F_P(\{\mathit{odd}(s(0))\}) = \{\mathit{odd}(s^{2n}(0)), \mathit{even}(s(0))\}$. Hence, if $\mathit{odd}(s(0))$ is known to be true, then both $\mathit{odd}(s^{2n}(0))$ and $\mathit{even}(s(0))$ must be false. ∎

We now require that a canonical model I satisfies:

$$\neg A \in I \text{ iff } A \in F_P(I) \qquad (C_2)$$

That is, A is false in the canonical model iff A is in the greatest unfounded set of P w.r.t. the canonical model itself. Partial Herbrand interpretations are partially ordered under set inclusion just like ordinary Herbrand interpretations, but the intuition is quite different. A minimal partial interpretation is maximally undefined whereas a minimal two-valued interpretation is maximally false. It was shown by Van Gelder, Ross and Schlipf (1991) that all general programs have a unique minimal partial model satisfying C_1 and C_2. The model is called the *well-founded model* of P:

Definition 4.25 (Well-founded model) Let P be a general program. The well-founded model of P is the least partial Herbrand interpretation I such that:

- if $A \in T_P(I)$ then $A \in I$;
- if $A \in F_P(I)$ then $\neg A \in I$.

∎

Example 4.26 The definite program:

$$\begin{array}{l} \mathit{odd}(s(0)). \\ \mathit{odd}(s(s(X))) \leftarrow \mathit{odd}(X). \end{array}$$

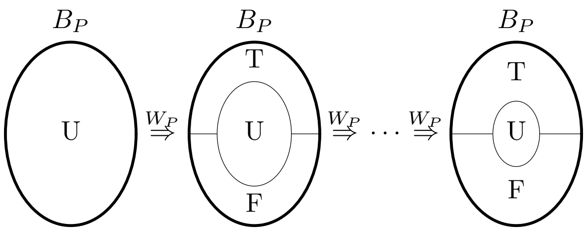


Figure 4.8: Approximation of the well-founded model

has the well-founded model $\{odd(s^{2n+1}(0)) \mid n \geq 0\} \cup \{\neg odd(s^{2n}(0)) \mid n \geq 0\}$. The program:

$$\begin{array}{l} loops(a) \leftarrow \neg halts(a). \\ loops(b) \leftarrow \neg halts(b). \\ halts(a) \leftarrow halts(a). \\ halts(b). \end{array}$$

has the well-founded model $\{halts(b), loops(a), \neg halts(a), \neg loops(b)\}$. Finally:

$$\begin{array}{l} p \leftarrow \neg p. \\ q \leftarrow q. \\ r. \end{array}$$

has the well-founded model $\{r, \neg q\}$. ∎

More formally the well-founded model of P can be characterized as the least fixed point of the operator:

$$W_P(I) \quad = \quad T_P(I) \cup \neg F_P(I)$$

where $\neg F_P(I)$ denotes the set $\{\neg A \mid A \in F_P(I)\}$.

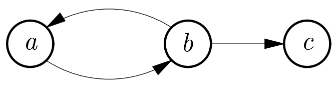
By known results from the theory of fixed points the least fixed point of W_P is a limit of a (possibly transfinite) iterative process. If the Herbrand universe is finite it is always possible to compute the well-founded model of P as the limit of the sequence:

$$\varnothing, \quad W_P(\varnothing), \quad W_P(W_P(\varnothing)), \quad \ldots$$

The iteration starts from the empty interpretation (where every literal is undefined). Each iteration of W_P then adds new positive and negative literals to the interpretation (cf. Figure 4.8) until the iteration converges (which happens in a finite number of steps if the Herbrand universe is finite).

Example 4.27 Consider a class of simple games consisting of a set of states and a set of moves between states. In all games there are two players who make moves in turn. A player loses if he is unable to make a move or has to move to a position where the opponent wins, and a player wins if he can move to a state where the opponent loses.

Then assume a particular instance in this class of games that has only three states, $\{a, b, c\}$, and the following moves:



The game can be formalized as follows:

$$\begin{array}{l} w(X) \leftarrow m(X,Y), \neg w(Y). \\ m(a,b). \\ m(b,a). \\ m(b,c). \end{array}$$

The well-founded model of the program can be computed as the limit of the iteration $W_P \uparrow n$ (abbreviated I_n). The first iteration yields:

$$\begin{array}{rcl} I_1 & = & T_P(\varnothing) \cup \neg F_P(\varnothing) \text{ where} \\ T_P(\varnothing) & = & \{m(a,b), m(b,a), m(b,c)\} \\ F_P(\varnothing) & = & \{m(a,a), m(a,c), m(b,b), m(c,a), m(c,b), m(c,c), w(c)\} \end{array}$$

Note that $w(c)$ is in the unfounded set since all ground instances of the partially instantiated clause $w(c) \leftarrow m(c,Y), \neg w(Y)$ contain a positive body literal which is in the unfounded set. The second iteration yields:

$$\begin{array}{rcl} I_2 & = & T_P(I_1) \cup \neg F_P(I_1) \text{ where} \\ T_P(I_1) & = & \{m(a,b), m(b,a), m(b,c), w(b)\} \\ F_P(I_1) & = & \{m(a,a), m(a,c), m(b,b), m(c,a), m(c,b), m(c,c), w(c)\} \end{array}$$

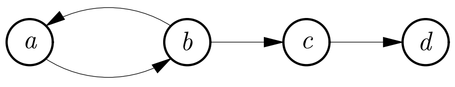
After which $w(b)$ is known to be true, since $I_1 \models m(b,c), \neg w(c)$. The third iteration yields:

$$\begin{array}{rcl} I_3 & = & T_P(I_2) \cup \neg F_P(I_2) \text{ where} \\ T_P(I_2) & = & \{m(a,b), m(b,a), m(b,c), w(b)\} \\ F_P(I_2) & = & \{m(a,a), m(a,c), m(b,b), m(c,a), m(c,b), m(c,c), w(c), w(a)\} \end{array}$$

After which the unfounded set is extended by $w(a)$ (since all ground instances of $w(a) \leftarrow m(a,Y), \neg w(Y)$ either contain a positive body literal which is in the unfounded set or a negative literal ($\neg w(b)$) which is false in I_2).

Now $W_P(I_3) = I_3$. Thus, in the well-founded model there are two losing and one winning state. Obviously, c is a losing state since no moves are possible. Consequently b is a winning state because the opponent can be put in a losing state. Finally a is a losing state since the only possible move leaves the opponent in a winning state. Incidentally, the well-founded model is total.

Next consider the following game:



This game can be formalized as follows:

$$
\begin{array}{l}
w(X) \leftarrow m(X,Y), \neg w(Y).\\
m(a,b).\\
m(b,a).\\
m(b,c).\\
m(c,d).
\end{array}
$$

The first iteration yields:

$$
\begin{array}{rcl}
I_1 & = & T_P(\varnothing) \cup \neg F_P(\varnothing)\\
T_P(\varnothing) & = & \{m(a,b), m(b,a), m(b,c), m(c,d)\}\\
F_P(\varnothing) & = & \{m(x,y) \mid m(x,y) \notin P\} \cup \{w(d)\}
\end{array}
$$

After the second iteration $w(c)$ is known to be true:

$$
\begin{array}{rcl}
I_2 & = & T_P(I_1) \cup \neg F_P(I_1)\\
T_P(I_1) & = & \{m(a,b), m(b,a), m(b,c), m(c,d), w(c)\}\\
F_P(I_1) & = & \{m(x,y) \mid m(x,y) \notin P\} \cup \{w(d)\}
\end{array}
$$

In fact, this is the well-founded model of P:

$$
I_3 \quad = \quad T_P(I_2) \cup \neg F_P(I_2) \quad = \quad I_2
$$

This time the well-founded model is partial. The state d is a losing and c is a winning state. However, nothing is known about a and b. This may appear startling. However, a player can clearly not win from state b by moving to c. Moreover he does not have to lose, since there is always the option of moving to state a. (From which the opponent can always move back to b.) Hence a and b are drawing states. ∎

The well-founded model coincides with the other canonical models discussed earlier in this chapter. For instance, the least Herbrand model in the case of definite programs:

Theorem 4.28 If P is a definite program then the well-founded model is total and coincides with the least Herbrand model. ∎

It is also coincides with the standard model of stratified programs:

Theorem 4.29 If P is stratified then the well-founded model is total and coincides with the standard model. ∎

Several other connections between the well-founded semantics and other semantics have also been established.

Several attempts have been made to define variants of SLDNF-resolutions which compute answers to goals using the well-founded semantics as the underlying declarative semantics. In general, no complete resolution mechanism can be found, but for restricted classes of general programs such resolution mechanisms exist. Most notably the notion of SLS-resolution of Przymusinski (1989).

Exercises

4.1 Consider the following definite program:

$$\begin{array}{l} p(X) \leftarrow q(Y,X), r(Y). \\ q(s(X),Y) \leftarrow q(X,Y). \\ r(0). \end{array}$$

Show that there is one computation rule such that $\leftarrow p(0)$ has a finitely failed SLD-tree and another computation rule such that $\leftarrow p(0)$ has an infinite SLD-tree.

4.2 Construct the completion of the program in the previous exercise. Show that $\neg p(0)$ is a logic consequence of $comp(P)$.

4.3 Construct the completion of the program:

$$\begin{array}{l} p(a) \leftarrow q(X). \\ p(b) \leftarrow r(X). \\ r(a). \\ r(b). \end{array}$$

Show that $\neg p(a)$ is a logical consequence of $comp(P)$.

4.4 Let P be a definite program. Show that $comp(P) \models P$.

4.5 Consider a general program:

$$\begin{array}{l} p(b). \\ p(a) \leftarrow \neg q(X). \\ q(a). \end{array}$$

Construct $comp(P)$ and show that $p(a)$ is a logical consequence of $comp(P)$.

4.6 Construct a fair SLD-tree for the program:

$$\begin{array}{l} p(s(X)) \leftarrow p(X). \\ q(X,Y) \leftarrow p(Y), r(X,0). \\ r(X,X). \end{array}$$

and the goal $\leftarrow p(X), q(X,Y)$.

4.7 Which of the following four programs are stratified?

$$P_1 \quad \begin{array}{l} p(X) \leftarrow q(X), r(X). \\ p(X) \leftarrow \neg r(X). \\ q(X) \leftarrow \neg r(X), s(X). \\ r(X) \leftarrow \neg s(X). \end{array} \qquad P_2 \quad \begin{array}{l} p(X) \leftarrow p(X), s(X). \\ s(X) \leftarrow r(X). \\ r(X) \leftarrow \neg p(X). \\ r(a). \end{array}$$

$$P_3 \quad \begin{array}{l} p(X) \leftarrow \neg q(X), r(X). \\ r(X) \leftarrow q(X). \\ q(X) \leftarrow \neg s(X). \end{array} \qquad P_4 \quad \begin{array}{l} p(X) \leftarrow r(X), p(X). \\ r(X) \leftarrow \neg p(X). \\ r(X) \leftarrow r(X). \end{array}$$

4.8 Construct the completion of the general program:

$$p(a) \leftarrow \neg q(b).$$

and show that $\{p(a)\}$ is a Herbrand model of the completion. Show also that the model is minimal.

4.9 Consider the general program:

$$\begin{array}{l}
\mathit{flies}(X) \leftarrow \mathit{bird}(X), \neg \mathit{abnormal}(X). \\
\mathit{bird}(\mathit{tom}). \\
\mathit{bird}(\mathit{sam}). \\
\mathit{bird}(\mathit{donald}). \\
\mathit{abnormal}(\mathit{donald}). \\
\mathit{abnormal}(X) \leftarrow \mathit{isa}(X, \mathit{penguin}). \\
\mathit{isa}(\mathit{sam}, \mathit{eagle}). \\
\mathit{isa}(\mathit{tom}, \mathit{penguin}). \\
\mathit{isa}(\mathit{donald}, \mathit{duck}).
\end{array}$$

Construct the SLDNF-forest for the goal $\leftarrow \mathit{flies}(X)$.

4.10 Consider the program in Example 4.16. Show that

$$\leftarrow \mathit{go\_well\_together}(\mathit{python}, \mathit{rabbit})$$

has a a finitely failed SLDNF-tree.

4.11 Prove theorem 4.20.

4.12 Prove theorem 4.22.

4.13 Show a general program which has more than one minimal supported Herbrand model.

4.14 What is the well-founded model of the program:

$$\begin{array}{l}
p \leftarrow \neg q \\
q \leftarrow \neg p \\
r \\
s \leftarrow p, \neg r.
\end{array}$$

Chapter 5

Towards Prolog: Cut and Arithmetic

Computations of logic programs require construction and traversal of SLD-trees. This is not necessarily the most efficient way of computing. Two extensions — the *cut* and *built-in arithmetic* — that are incorporated in the programming language Prolog to speed up computations will be presented separately in Sections 5.1 and 5.2. For the sake of simplicity the exposition covers only definite programs but all concepts carry over to general programs, SLDNF-derivations and SLDNF-trees. (In what follows, mentioning of Prolog refers to the ISO Prolog standard (1995) unless otherwise stated.)

5.1 Cut: Pruning the SLD-tree

An SLD-tree of a goal may have many failed branches and very few, or just one, success branch. Therefore the programmer may want to prevent the interpreter from constructing failed branches by adding control information to the program. However, such information relies on the operational semantics of the program. To give the required control information, the programmer has to know how the SLD-tree is constructed and traversed. However, for practical reasons this information has to be taken into account anyway — for the depth-first search employed in Prolog-interpreters, existence of an infinite branch in the SLD-tree may prevent the interpreter from finding an existing correct answer. To control the search the concept of *cut* is introduced in Prolog. Syntactically the cut is denoted by the nullary predicate symbol "!" and it may be placed in the body of a clause or a goal as one of its atoms. Its meaning can be best explained as a "shortcut" in the traversal of the SLD-tree. Thus, the presence of cut in a clause may avoid construction of some subtrees of the SLD-tree. For a more precise explanation some auxiliary notions are needed.

Every node n of an SLD-tree corresponds to a goal of an SLD-derivation and has

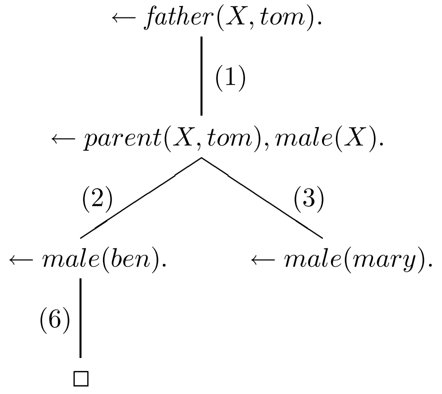


Figure 5.1: SLD-tree

a selected atom A. Assume that A is not an instance of a subgoal in the initial goal. Then A is an instance of a body atom B_i of a clause $B_0 \leftarrow B_1, \ldots, B_i, \ldots, B_n$ whose head unifies with the selected subgoal in some node n' between the root and n. Denote by $\mathit{origin}(A)$ the node n'.

Prolog interpreters traverse the nodes of the SLD-tree in a depth-first manner as depicted in Figure 3.6. The ordering of branches corresponds to the textual ordering of the clauses in the program. When a leaf of the tree is reached, *backtracking* takes place. The process terminates when no more backtracking is possible (that is, when all subtrees of the root are traversed). The atom "!" is handled as an ordinary atom in the body of a clause. However, when a cut is selected for resolution it succeeds immediately (with the empty substitution). The node where "!" is selected will be called the *cut-node*. A cut-node may be reached again during backtracking. In this case the normal order of tree-traversal illustrated in Figure 3.6 is altered — by definition of cut the backtracking continues *above* the node $\mathit{origin}(!)$ (if cut occurs in the initial goal the execution simply terminates). This is illustrated by the following simple example.

Example 5.1 The father of a person is its male parent. Assume that the following world is given:

$$
\begin{array}{ll}
(1) & \mathit{father}(X, Y) \leftarrow \mathit{parent}(X, Y), \mathit{male}(X).\\
(2) & \mathit{parent}(\mathit{ben}, \mathit{tom}).\\
(3) & \mathit{parent}(\mathit{mary}, \mathit{tom}).\\
(4) & \mathit{parent}(\mathit{sam}, \mathit{ben}).\\
(5) & \mathit{parent}(\mathit{alice}, \mathit{ben}).\\
(6) & \mathit{male}(\mathit{ben}).\\
(7) & \mathit{male}(\mathit{sam}).
\end{array}
$$

The SLD-tree of the goal $\leftarrow \mathit{father}(X, \mathit{tom})$ under Prolog's computation rule is shown in Figure 5.1. After first finding the solution $X = \mathit{ben}$, an attempt to find another solution will fail since Mary is not male. By the formulation of the problem it is clear that there may be at most one solution for this type of goal (that is, when the second

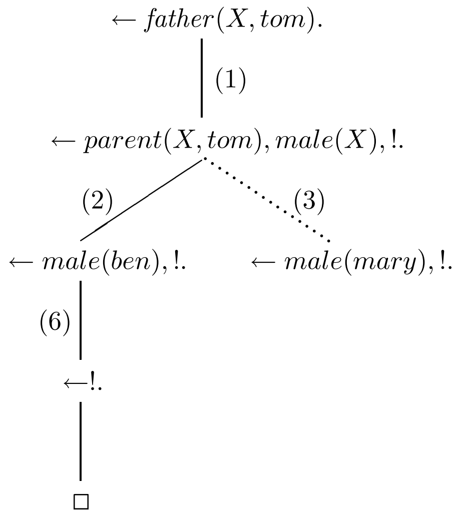

$\leftarrow father(X, tom).$
(1)
$\leftarrow parent(X, tom), male(X), !.$
(2)
(3)
$\leftarrow male(ben), !.$
$\leftarrow male(mary), !.$
(6)
$\leftarrow !.$
$\Box$

Figure 5.2: Pruning failing branches

argument is fully instantiated). When a solution is found the search can be stopped since no person has more than one father. To enforce this, cut may be inserted at the end of (1). The modified SLD-tree is shown in Figure 5.2 (The dashed line designates the branch cut off by "!"). The origin of the cut-node is the root of the tree so the search is completed after backtracking to the cut-node. Hence, the other branch of the tree is not traversed.

Notice that the modified version of (1) cannot be used for computing more than one element of the relation "... is the father of ...". The cut will stop the search after finding the first answer to the goal $\leftarrow father(X, Y)$ (consider the SLD-tree in Figure 5.3). ∎

It follows by the definition that the cut has the following effects:

- It divides the body into two parts where backtracking is carried out separately — after success of "!" no backtracking to the literals in the left-hand part is possible. However, in the right-hand part execution proceeds as usual;
- It cuts off unexplored branches directly below *origin*(!). In other words, there will be no further attempts to match the selected subgoal of *origin*(!) with the remaining clauses of the program.

Cut is, to put it mildly, a controversial construct. The intention of introducing cut is to control the execution of a program without changing its logical meaning. Therefore the logical reading of cut is "true". Operationally, if it removes only the failed branches of the SLD-tree it does not influence the meaning of the program. However, it may also cut off some success branches, thus destroying completeness of definite programs and soundness of general programs. To illustrate the latter, consider the following example:

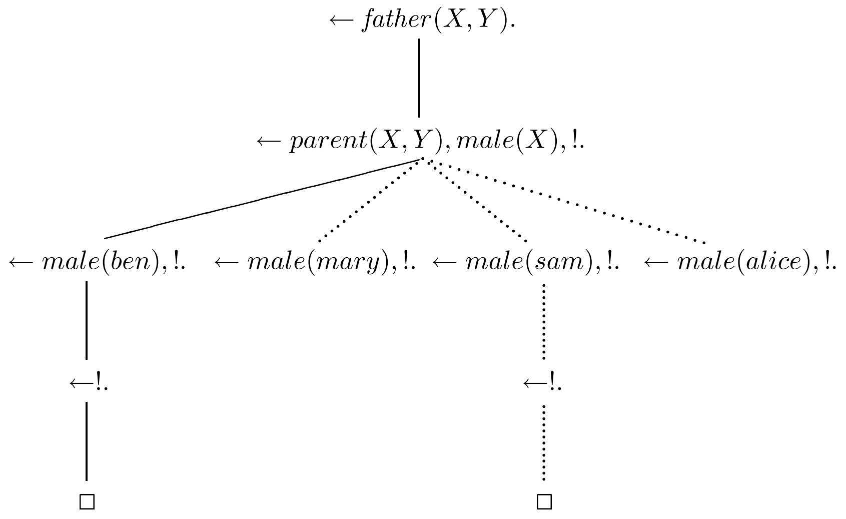

$\leftarrow father(X, Y).$
$\leftarrow parent(X, Y), male(X), !.$
$\leftarrow male(ben), !.$
$\leftarrow male(mary), !.$
$\leftarrow male(sam), !.$
$\leftarrow male(alice), !.$
$\leftarrow !.$
$\leftarrow !.$
$\Box$
$\Box$

Figure 5.3: Pruning success-branches

Example 5.2 It is a well known fact that fathers of newborn children are proud. This proposition is reflected by the following definite clause:

$$(1) \quad proud(X) \leftarrow father(X, Y), newborn(Y).$$

Take additionally the clauses:

$$\begin{array}{ll} (2) & father(X, Y) \leftarrow parent(X, Y), male(X). \\ (3) & parent(john, mary). \\ (4) & parent(john, chris). \\ (5) & male(john). \\ (6) & newborn(chris). \end{array}$$

The answer to the initial goal $\leftarrow proud(john)$ is “yes” since, as described, John is the father of Chris who is newborn.

Now, replace (2) by the version with cut used in Example 5.1:

$$(2') \quad father(X, Y) \leftarrow parent(X, Y), male(X), !.$$

This time the answer to the goal $\leftarrow proud(john)$ is “no”. It is so because the first “listed” child of John is Mary — the sister of Chris. After having found this answer there will be no more attempts to find any more children of John because of the cut.

This makes the program incomplete — some correct answer substitutions cannot be found. More seriously, this incompleteness may result in incorrect answers if negation is involved. For example, the goal $\leftarrow \neg proud(john)$ will succeed — implying that John is not proud. ▮

So far two principal uses of cut have been distinguished — to cut off failing branches of the SLD-tree and to prune succeeding branches. Cutting off failing branches is generally considered harmless since it does not alter the answers produced during the execution. Such cuts are sometimes referred to as “green cuts”. However, this restricted use of cut is usually tied to some particular use of the program. Thus, as illustrated in Figures 5.2 and 5.3, for some goals only failing branches are cut off whereas for other goals succeeding branches are also pruned.

In general, cutting succeeding branches *is* considered harmful. (Consequently such cuts are referred to as “red cuts”.) However, there are some cases when it is motivated. This section is concluded with two examples — in the first example the use of cut is sometimes (rightfully) advocated. The second example demonstrates a very harmful (albeit common) use of cut.

Consider the following (partial) program:

$$\begin{array}{l} proud(X) \leftarrow father(X,Y), newborn(Y). \\ \quad\quad \vdots \\ father(john, sue). \\ father(john, mary). \\ \quad\quad \vdots \\ newborn(sue). \\ newborn(mary). \end{array}$$

The SLD-tree of the goal $\leftarrow proud(X)$ has two success-leaves since John has two children both of which are newborn. However, both answers give the same binding for X — namely $X = john$. In general the user is not interested in getting the same answer twice or more. To avoid this, a cut may be inserted at the very end of the first clause (or possibly as the rightmost subgoal in the goal).

$$proud(X) \leftarrow father(X,Y), newborn(Y), !.$$

Next consider the following example[1] which describes the relation between two integers and their minimum:

$$\begin{array}{l} min(X,Y,X) \leftarrow X < Y, !. \\ min(X,Y,Y). \end{array}$$

At first glance this program may look correct. People used to imperative programming languages often reason as follows — “If X is less than Y then the minimum of X and Y is X, else it is Y”. Actually the program returns the expected answer both to the goal $\leftarrow min(2,3,X)$ and $\leftarrow min(3,2,X)$ — in both cases the answer $X = 2$ is obtained. However, the program is not correct. Consider the goal $\leftarrow min(2,3,3)$. This goal succeeds implying that “3 is the minimum of 2 and 3”! The program above is an

[1]Here $<$ is a binary predicate symbol written in infix notation designating the less-than relation over e.g. the integers. Intuitively it may be thought of as an infinite collection of facts:

$$\begin{array}{cccccc} \cdots & -1 < 0. & 0 < 1. & 1 < 2. & 2 < 3. & \cdots \\ \cdots & -1 < 1. & 0 < 2. & 1 < 3. & 2 < 4. & \cdots \\ \iddots & \vdots & \vdots & \vdots & \vdots & \ddots \end{array}$$

In Prolog $<$ is implemented as a so-called built-in predicate which will be discussed in the next section.

example of an incorrect program where (some of) the false answers are discarded by means of the cut. The intended model is simply not a model of the program since the second clause says that "For any two integers, X and Y, Y is their minimum". This use of cut is harmful. It may not only produce incorrect answers, but it also makes the program hard to read and understand. If cut is to be used it should be added to a program which is true in the intended model. Thus, the recommended version of the minimum program (with cut) would look as follows:

$$\begin{array}{l} min(X, Y, X) \leftarrow X < Y, !. \\ min(X, Y, Y) \leftarrow X \geq Y. \end{array}$$

This program is true in the intended model and the goal $\leftarrow min(2, 3, 3)$ does not succeed any longer.

As a final remark, cut may be used to implement negation in Prolog. Consider the following clauses (where *fail* is a Prolog predicate which lacks a definition and cannot be defined by the user):

$$\begin{array}{l} not(student(X)) \leftarrow student(X), !, fail. \\ not(student(X)). \end{array}$$

This definition relies entirely on the operational semantics of Prolog. That is, subgoals must be solved from left to right and clauses are searched in the textual order. If we want to know whether "John is not a student" the goal $\leftarrow not(student(john))$ may be given. Then there are two cases to consider — if the subgoal $student(john)$ succeeds (i.e. if John is a student), "!" will cut off the second clause and the negated goal will fail. That is, Prolog produces the answer "no". However, if the subgoal $student(john)$ finitely fails, the second clause will be tried (on backtracking) and the negated goal immediately succeeds.

To avoid having to write a separate definition for every predicate that the user may want to negate it is possible to use a predefined meta-predicate named $call/1$ which is available in standard Prolog. The argument of $call/1$ must not be a variable when the subgoal is selected and a call to the predicate succeeds iff the argument succeeds. In other words — the goal $\leftarrow call(G)$ succeeds iff the goal $\leftarrow G$ succeeds. Using this predicate $not/1$ may be defined for arbitrary goals:[2]

$$\begin{array}{l} not(X) \leftarrow call(X), !, fail. \\ not(X). \end{array}$$

Notice that the success of $call(t)$ may produce bindings for variables in t. Hence the implementation is not logically correct, as discussed in Chapter 4. However, it works as intended when the argument of $not/1$ is a *ground* atom.

In general it is possible to avoid using cut in most cases by sticking to negation instead. This is in fact advocated since unrestricted use of cut often leads to incorrect programs. It is not unusual that people — on their first contacts with Prolog and faced with a program that produces incorrect answers — clutter the program with cuts at random places instead of writing a logically correct program in the first place. In the following chapters the use of cut is avoided to make this point quite clear. However, this does not mean that cut should be abandoned altogether — correctly used, it can improve the efficiency of programs considerably.

[2]Standard Prolog uses the predicate \+ with a prefix notation to denote negation.

5.2 Built-in Arithmetic

It has been proved that definite programs can describe any computable relation. That is, any Turing machine can be coded as a logic program. This means that from a theoretical point of view logic programming is not less expressive than other programming paradigms. In other words, resolution and exhaustive search provide a universal tool for computation. But from a practical point of view it is not desirable to compute everything in that way. Take for example the arithmetic operations on natural numbers. They are efficiently implemented in the hardware of computers. Therefore, from a practical point of view, it is desirable to allow logic programs to access machine arithmetic. A similar argument concerns any other operation or procedure whose efficient implementation in hardware or software is available. The problem is whether it is possible to do that without destroying the declarative nature of logic programs that use these *external* features. This section discusses the problem for the example of arithmetic operations and shows the solution adopted in Prolog.

Notice first that arithmetic operations like plus or times can be easily described by a definite logic program. The natural numbers can be represented by ground terms. A standard way for that is to use the constant 0 for representing zero and the unary functor $s/1$ for representing the successor of a number. Thus, the consecutive natural numbers are represented by the following terms:

$$0, s(0), s(s(0)), \ldots$$

The operations of addition and multiplication are binary functions on natural numbers. Logic programs provide only a formalism for expressing relations. However, a binary function can be seen as a ternary relation consisting of all triples $\langle x, y, z\rangle$ such that z is the result of applying the function to the arguments x and y. It is well known that the operations of addition and multiplication on natural numbers can be characterized by the following Peano axioms:

$$\begin{array}{rcl} 0 + X & \doteq & X \\ s(X) + Y & \doteq & s(X + Y) \\ 0 * X & \doteq & 0 \\ s(X) * Y & \doteq & (X * Y) + Y \end{array}$$

These axioms relate arguments and results of the operations. In the relational notation of definite programs they can be reformulated as follows:

$plus(0, X, X).$
$plus(s(X), Y, s(Z)) \leftarrow plus(X, Y, Z).$

$times(0, X, 0).$
$times(s(X), Y, Z) \leftarrow times(X, Y, W), plus(W, Y, Z).$

This program can be used to add and multiply natural numbers represented by ground terms.

For example, to add 2 and 3 the goal $\leftarrow plus(s(s(0)), s(s(s(0))), X)$ can be given. The computed answer is $X = s(s(s(s(s(0)))))$. An SLD-refutation is used to construct it.

On the other hand, the program can be used also for subtraction and (a limited form of) division. For example, in order to subtract 2 from 5 the goal $\leftarrow plus(X, s(s(0)), s(s(s(s(s(0))))))$ can be used. The program can also perform certain symbolic computations. For example, one can add 2 to an unspecified natural number using the goal $\leftarrow plus(s(s(0)), X, Y)$. The computed answer is $Y = s(s(X))$. Thus, for any ground term t the result is obtained by adding two instances of the symbols s in front of t.

When comparing this with the usual practice in programming languages, the following observations can be made:

- the representation of numbers by compound terms is inconvenient for humans;
- the computations of the example program do not make use of arithmetic operations available in the hardware — therefore they are much slower. For instance, adding numbers N and M requires $N + 1$ procedure-calls;
- arithmetic expressions cannot be constructed, since the predicate symbols $plus/3$ and $times/3$ represent relations. For example, to compute $2 + (3 * 4)$ one has to introduce new temporary variables representing the values of subexpressions:

$$\leftarrow times(s(s(s(0))), s(s(s(s(0)))), X), plus(X, s(s(0)), Y).$$

The first problem can easily be solved by introducing some "syntactic sugar", like the convention that the decimal numeral for the natural number n represents the term $s^n(0)$ — for instance, 3 represents the term $s(s(s(0)))$. Techniques for compiling arithmetic expressions into machine code are also well known. Thus the main problem is how to incorporate arithmetic expressions into logic programs without destroying the declarative meaning of the programs.

Syntactically arithmetic expressions are terms built from numerals, variables and specific arithmetic functors, like "+", "∗", etc. usually written in infix notation. The intended meaning of a ground arithmetic expression is a number. It is thus essential that distinct expressions may denote the same number, take for example $2 + 2$, $2 * 1 + 4 - 2$ and 4. Thus, there is a binary relation on ground arithmetic expressions which holds between arbitrary expressions E_1 and E_2 iff E_1 and E_2 denote the same number. Clearly this relation is an equivalence relation. Every equivalence class includes one numeral which is the simplest representation of all terms in the class. The machine operations give a possibility of efficient reduction of a given ground arithmetic expression to this numeral.

Assume that arithmetic expressions may appear as terms in definite logic programs. The answers of such programs should take into account equivalence between the arithmetic expressions. For example, consider the following rule for computing tax — "if the annual income is greater than 150,000 then the tax is 50%, otherwise 25% of the income reduced by 30,000":

$$\begin{array}{l} tax(Income, 0.5 * Income) \leftarrow greater(Income, 150000). \\ tax(Income, 0.25 * (Income - 30000)) \leftarrow \neg greater(Income, 150000). \end{array}$$

A tax-payer received a decision from the tax department to pay 25,000 in tax from his income of 130,000. To check whether the decision is correct (s)he may want to use the rules above by giving the goal $\leftarrow tax(130000, 25000)$. But the rules cannot be used to find a refutation of the goal since none of the heads of the clauses unify with the subgoal in the goal. The reason is that standard unification is too weak to realize that the intended interpretations of the terms 25000 and $0.25 * (130000 - 30000)$ are the same. Thus, the equivalence must somehow be described by *equality* axioms for arithmetic. But they are not included in the program above.

This discussion shows the need for an extension of the concept of logic programs. For our example the program should consist of two parts — a set of definite clauses P and a set of equality axioms E describing the equivalences of terms. This type of program has been studied in the literature. The most important result is a concept of generalized unification associated with a given equality theory E and called E-unification. A brief introduction follows below. A more extensive account is provided in Chapter 13 and 14.

A *definite clause equality theory* is a (possibly infinite) set of definite clauses, where every atom is of the form $s \doteq t$ and s and t are terms. Sometimes the form of the clauses is restricted to facts.

A *definite program with equality* is a pair P, E where P is a definite program which contains no occurrences of the predicate symbol $\doteq/2$ and E is a definite clause equality theory.

Let E be a definite clause equality theory. A substitution θ is an *E-unifier* of the terms s and t iff $s\theta \doteq t\theta$ is a logical consequence of E.

Example 5.3 Let E be an equality theory describing the usual equivalence of arithmetic expressions. Consider the expressions:

$$t_1 := (2 * X) + 1 \quad \text{and} \quad t_2 := Y + 2$$

For instance, the substitution $\theta := \{Y/(2 * X - 1)\}$ is an E-unifier of t_1 and t_2. To check this, notice that $t_1\theta = t_1$ and that $t_2\theta = (2 * X - 1) + 2$ which is equivalent to t_1. ∎

Now, for a given program P, E and goal $\leftarrow A_1, \ldots, A_m$ the refutation of the goal can be constructed in the same way as for definite programs, with the only difference that E-unification is used in place of unification as presented in Chapter 3.

Finding E-unifiers can be seen as solving of equations in an algebra defined by the equality axioms. It is known that the problem of E-unification is in general undecidable. Even if it is decidable for some theory E there may be many different solutions of a given equation. The situation when there exists one most general unifier is rather unusual. This means that even if it is possible to construct all E-unifiers, a new dimension of nondeterminism is introduced.

Assume now that an equality theory E describes all external functions, including arithmetic operations, used in a logic program. This means that for any ground terms s and t whose main functors denote external functions, the formula $s \doteq t$ is a logical consequence of E iff the invocation of s returns the same result as the invocation of t. In other words, in the special case of ground terms their E-unifiability can be decided — they either E-unify with the identity substitution, if both reduce to the same result, or they are not E-unifiable, if their results are different. This can be exploited in the following way — whenever a call of an external function is encountered as a term to be E-unified, it is invoked and its reduced form is being unified instead by the usual unification algorithm. However, the external procedures can be invoked only with ground arguments. If some variables of the call are not instantiated, the computation cannot proceed and no E-unifier can be found. In this case a run time error may be reported.

This idea is incorporated in Prolog in a restricted form for arithmetic operations. Before explaining how, some syntactic issues should be mentioned.

The integers are represented in Prolog as integer numerals, for example 0, 1, 1989 and 17 etc. Prolog also supports a limited form of arithmetic over the reals using floating point numbers usually written as e.g. 3.14, 7.0, 0.3333 etc. Logically the numerals are constants. In addition, a number of predefined arithmetic functors for use in the infix notation is available. They denote standard arithmetic functions on integers and floats and refer to the operations of the computer. The most important operations are:

Functor	Operation
$+$	Addition
$-$	Subtraction
$*$	Multiplication
$/$	(Floating point) division
$//$	(Integer) division
mod	Remainder after division

Additionally unary minus is used to represent negative numbers. (For a full list see the ISO Prolog standard (1995).)

A ground term t constructed from the arithmetic functors and the numerals represents an integer or a floating point number, which can also be represented by a numeral n, possibly prefixed by "$-$". The machine operations of the computer make it possible to construct this term t' in an efficient way. The arithmetic operations *can* be axiomatized as an equational theory E such that $t \doteq t'$ is its logical consequence. Two predefined predicates of Prolog handle two specific cases of E-unification. They are $is/2$ and $=:=/2$ both of which are used in the infix notation.

The binary predicate $=:=/2$ *checks* if two ground arithmetic expressions are E-unifiable. For example the goal:

$$\leftarrow 2 + 3 =:= 1 + 4.$$

succeeds with the answer "yes" (corresponding to the empty substitution). If the arguments are not ground arithmetic expressions, the execution aborts with an error message in most Prolog implementations.

The binary predicate $is/2$ unifies its first argument with the reduced form of a term constructed from the arithmetic functors and numerals. For example the goal:

$$\leftarrow X \ is \ 2 + 2.$$

succeeds with the substitution $\{X/4\}$.

The first argument of this predicate need not be variable. Operationally the reduced form of the second argument, which is either a numeral or a numeral preceded by "$-$", is being unified with the first argument. If the latter is an arithmetic expression in the reduced form then this is a special case of E-unification handled also by

=:=/2. Otherwise the answer is "no". But an E-unifier may still exist. For example the goal:

$$\leftarrow X + 1 \; is \; 2 + 3.$$

will fail, although the terms $X + 1$ and $2 + 3$ have an E-unifier — namely $\{X/4\}$.

Another standard predicate =\=/2 (also in infix notation) checks whether two ground terms are not E-unifiable. Prolog also provides predefined predicates for comparing the number represented by ground arithmetic expressions. These are the binary infix predicates $<$, $>$, $\geq$ and $\leq$.

Exercises

5.1 Consider the following definite program:

$$\begin{array}{l} top(X,Y) \leftarrow p(X,Y). \\ top(X,X) \leftarrow s(X). \\ p(X,Y) \leftarrow true(1), q(X), true(2), r(Y). \\ p(X,Y) \leftarrow s(X), r(Y). \\ q(a). \\ q(b). \\ r(c). \\ r(d). \\ s(e). \\ true(X). \end{array}$$

Draw the SLD-tree of the goal $\leftarrow top(X,Y)$. Then show what branches are cut off:

- when $true(1)$ is replaced by cut;
- when $true(2)$ is replaced by cut.

5.2 Consider the following program:

$$\begin{array}{l} p(Y) \leftarrow q(X,Y), r(Y). \\ p(X) \leftarrow q(X,X). \\ q(a,a). \\ q(a,b). \\ r(b). \end{array}$$

Add cut at different places in the program above and determine the answers in response to the goal $\leftarrow p(Z)$.

5.3 Consider the definition of $not/1$ given on page 92. From a logical point of view, $\leftarrow p(X)$ and $\leftarrow not \; not \; p(X)$ are equivalent formulas. However, they behave differently when given to the program that consists of a single clause $p(a)$ — in what way?

5.4 Prolog implementations often incorporate a built-in predicate $var/1$ which succeeds (with the empty substitution) if the argument is an uninstantiated variable when the call is made and fails otherwise. That is:

$$\leftarrow var(X), X = a.$$

succeeds whereas:

$$\leftarrow X = a, var(X).$$

fails under the assumption that Prolog's computation rule is used. Define $var/1$ given the definition of $not/1$ on page 92.

5.5 Write a program which defines the relation between integers and their factorial. First use Peano arithmetic and then the built-in arithmetic predicates of Prolog.

5.6 Write a predicate $between(X, Y, Z)$ which holds if $X \leq Y \leq Z$. That is, given a goal $\leftarrow between(1, X, 10)$ the program should generate all integers in the closed interval (via backtracking).

5.7 Write a program that describes the relation between integers and their square using Peano arithmetic.

5.8 Implement the Euclidean algorithm for computing the greatest common divisor of two integers. Do this using both Peano arithmetic and built-in arithmetic.

5.9 The polynomial $c_n * x^n + \cdots + c_1 * x + c_0$ where $c_0, \ldots, c_n$ are integers may be represented by the term

$$c_n * x\hat{\ }n + \cdots + c_1 * x + c_0$$

where $\hat{\ }/2$ is written with infix notation and binds stronger than $*/2$ which in turn binds stronger than $+/2$. Now write a program which evaluates such polynomials given the value of x. For instance:

$$\leftarrow eval(2 * x\hat{\ }2 + 5, 4, X).$$

should succeed with answer $X = 37$. To solve the problem you may presuppose the existence of a predicate $integer/1$ which succeeds if the argument is an integer.

PART II
PROGRAMMING IN LOGIC

Chapter 6

Logic and Databases

This chapter discusses the relationship between logic programs and relational databases. It is demonstrated how logic can be used to represent — on a conceptual level — not only *explicit* data, but also *implicit* data (corresponding to *views* in relational database theory) and how it can be used as a *query language* for retrieval of information in a database. We do not concern ourselves with implementation issues but only remark that SLD-resolution does not necessarily provide the best inference mechanism for full logical databases. (An alternative approach is discussed in Chapter 15.) On the other hand, logic not only provides a uniform language for representation of databases — its additional expressive power also enables description, in a concise and intuitive way, of more complicated relations — for instance, relations which exhibit certain common properties (like transitivity) and relations involving structured data objects.

6.1 Relational Databases

As indicated by the name, the mathematical notion of relation is a fundamental concept in the field of relational databases. Let $\mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_n$ be collections of symbols called *domains*. In the context of database theory the domains are usually assumed to be finite although, for practical reasons, they normally include an infinite domain of numerals. In addition, the members of the domains are normally assumed to be atomic or indivisible — that is, it is not possible to access a proper part of a member.

A *database relation* R over the domains $\mathcal{D}_1, \ldots, \mathcal{D}_n$ is a subset of $\mathcal{D}_1 \times \cdots \times \mathcal{D}_n$. R is in this case said to be n-ary. A *relational database* is a finite number of such (finite) relations. Database relations and domains will be denoted by identifiers in capital letters.

Example 6.1 Let $\mathit{MALE} := \{\mathit{adam}, \mathit{bill}\}$, $\mathit{FEMALE} := \{\mathit{anne}, \mathit{beth}\}$ and finally $\mathit{PERSON} := \mathit{MALE} \cup \mathit{FEMALE}$. Then:

$$MALE \times PERSON = \left\{ \begin{array}{ll} \langle adam, adam \rangle & \langle bill, adam \rangle \\ \langle adam, bill \rangle & \langle bill, bill \rangle \\ \langle adam, anne \rangle & \langle bill, anne \rangle \\ \langle adam, beth \rangle & \langle bill, beth \rangle \end{array} \right\}$$

Now, let *FATHER*, *MOTHER* and *PARENT* be relations over the domains $MALE \times PERSON$, $FEMALE \times PERSON$ and $PERSON \times PERSON$ defined as follows:

$$\begin{array}{lcl} FATHER & := & \{\langle adam, bill \rangle, \langle adam, beth \rangle\} \\ MOTHER & := & \{\langle anne, bill \rangle, \langle anne, beth \rangle\} \\ PARENT & := & \{\langle adam, bill \rangle, \langle adam, beth \rangle, \langle anne, bill \rangle, \langle anne, beth \rangle\} \end{array}$$

∎

It is of course possible to imagine alternative syntactic representations of these relations. For instance in the form of tables:

FATHER:

C_1	C_2
adam	*bill*
adam	*beth*

MOTHER:

C_1	C_2
anne	*bill*
anne	*beth*

PARENT:

C_1	C_2
adam	*bill*
adam	*beth*
anne	*bill*
anne	*beth*

or as a collection of labelled tuples (that is, facts):

$father(adam, bill).$
$father(adam, beth).$
$mother(anne, bill).$
$mother(anne, beth).$
$parent(adam, bill).$
$parent(adam, beth).$
$parent(anne, bill).$
$parent(anne, beth).$

The table-like representation is the one found in most textbooks on relational databases whereas the latter is a logic program. The two representations are isomorphic if no notice is taken of the names of the columns in the tables. Such names are called *attributes* and are needed only to simplify the specification of some of the operations discussed in Section 6.3. It is assumed that the attributes of a table are distinct. In what follows the notation $R(A_1, A_2, \ldots, A_n)$ will be used to describe the name, R, and attributes, $\langle A_1, A_2, \ldots, A_n \rangle$, of a database table (i.e. relation). $R(A_1, A_2, \ldots, A_n)$ is sometimes called a *relation scheme*. When not needed, the attributes are omitted and a table will be named only by its relation-name.

A major difference between the two representations which is not evident above, is the set of values which may occur in each column/argument-position of the representations. Logic programs have only a single domain consisting of terms and the user is permitted to write:

$$father(anne, adam).$$

whereas in a relational database this is usually not possible since $anne \notin MALE$. To avoid such problems a notion of *type* is needed.

Despite this difference it should be clear that any relational database can be represented as a logic program (where each domain of the database is extended to the set of all terms) consisting solely of ground facts. Such a set of facts is commonly called the *extensional database* (EDB).

6.2 Deductive Databases

After having established the relationship between relational databases and a (very simple) class of logic programs, different extensions to the relational database-model are studied. We first consider the use of variables and a simple form of rules. By such extensions it is possible to describe — in a more succinct and intuitive manner — many database relations. For instance, using rules and variables the database above can be represented by the program:

$$
\begin{array}{l}
parent(X, Y) \leftarrow father(X, Y).\\
parent(X, Y) \leftarrow mother(X, Y).\\
father(adam, bill).\\
father(adam, beth).\\
mother(anne, bill).\\
mother(anne, beth).
\end{array}
$$

The part of a logic program which consists of rules and nonground facts is called the *intensional database* (IDB). Since logic programs facilitate definition of new atomic formulas which are ultimately *deduced* from explicit facts, logic programs are often referred to as *deductive databases.* The logic programs above are also examples of a class of logic programs called *datalog* programs. They are characterized by the absence of functors. In other words, the set of terms used in the program solely consists of constant symbols and variables. For the representation of relational databases this is sufficient since the domains of the relations are assumed to be finite and it is therefore always possible to represent the individuals with a finite set of constant terms. In the last section of this chapter logic programs which make also use of compound terms are considered, but until then our attention will be restricted to datalog programs.

Example 6.2 Below is given a deductive family-database whose extensional part consists of definitions of $male/1$, $female/1$, $father/2$ and $mother/2$ and whose intensional part consists of $parent/2$ and $grandparent/2$:

$$grandparent(X, Z) \leftarrow parent(X, Y), parent(Y, Z).$$

$$
\begin{array}{l}
parent(X, Y) \leftarrow father(X, Y).\\
parent(X, Y) \leftarrow mother(X, Y).
\end{array}
$$

$$
\begin{array}{ll}
father(adam, bill). & mother(anne, bill).\\
father(adam, beth). & mother(anne, beth).\\
father(bill, cathy). & mother(cathy, donald).\\
father(donald, eric). & mother(diana, eric).
\end{array}
$$

$$\begin{array}{ll}
female(anne). & male(adam).\\
female(beth). & male(bill).\\
female(cathy). & male(donald).\\
female(diana). & male(eric).
\end{array}$$

∎

In most cases it is possible to organize the database in many alternative ways. Which organization to choose is of course highly dependent on what information one needs to retrieve. Moreover, it often determines the size of the database. Finally, in the case of updates to the database, the organization is very important to avoid inconsistencies in the database — for instance, how should the removal of the labelled tuple $parent(adam, bill)$ from the database in Example 6.2 be handled? Although updates are essential in a database system they will not be discussed in this book.

Another thing worth noticing about Example 6.2 is that the unary definitions $male/1$ and $female/1$ can be seen as *type declarations*. It is easy to add another such type declaration for the domain of persons:

$$\begin{array}{l}
person(X) \leftarrow male(X).\\
person(X) \leftarrow female(X).
\end{array}$$

It is now possible to "type" e.g. the database on page 103 by adding to the body of every clause the type of each argument in the head of the clause:

$$\begin{array}{l}
parent(X, Y) \leftarrow person(X), person(Y), father(X, Y).\\
parent(X, Y) \leftarrow person(X), person(Y), mother(X, Y).\\
\\
father(adam, bill) \leftarrow male(adam), person(bill).\\
father(adam, beth) \leftarrow male(adam), person(beth).\\
\quad\vdots\\
\\
person(X) \leftarrow male(X).\\
person(X) \leftarrow female(X).\\
\quad\vdots
\end{array}$$

In this manner, "type-errors" like $father(anne, adam)$ may be avoided.

6.3 Relational Algebra vs. Logic Programs

In database textbooks one often encounters the concept of *views*. A view can be thought of as a relation which is not explicitly stored in the database, but which is created by means of operations on existing database relations and other views. Such implicit relations are described by means of some *query-language* which is often compiled into *relational algebra* for the purpose of computing the views. Below it will be shown that all standard operations of relational algebra can be mimicked in logic programming (with negation) in a natural way. The objective of this section is twofold — first it shows that logic programs have at least the computational power of relational algebra. Second, it also provides an alternative to SLD-resolution as the operational semantics of a class of logic programs.

The primitive operations of relational algebra are *union*, *set difference*, *cartesian product*, *projection* and *selection*.

Given two n-ary relations over the same domains, the *union* of the two relations, R_1 and R_2 (denoted $R_1 \cup R_2$), is the set:

$$\{\langle x_1, \ldots, x_n \rangle \mid \langle x_1, \ldots, x_n \rangle \in R_1 \vee \langle x_1, \ldots, x_n \rangle \in R_2\}$$

Using definite programs the union of two relations — represented by the predicate symbols r_1/n and r_2/n — can be specified by the two rules:

$$\begin{array}{l} r(X_1, \ldots, X_n) \leftarrow r_1(X_1, \ldots, X_n). \\ r(X_1, \ldots, X_n) \leftarrow r_2(X_1, \ldots, X_n). \end{array}$$

For instance, if the EDB includes the definitions *father*/2 and *mother*/2, then *parent*/2 can be defined as the union of the relations *father*/2 and *mother*/2:[1]

$$\begin{array}{l} parent(X, Y) \leftarrow father(X, Y). \\ parent(X, Y) \leftarrow mother(X, Y). \end{array}$$

The *difference* $R_1 \setminus R_2$ of two relations R_1 and R_2 over the same domains yields the new relation:

$$\{\langle x_1, \ldots, x_n \rangle \in R_1 \mid \langle x_1, \ldots, x_n \rangle \notin R_2\}$$

In logic programming it is not possible to define such relations without the use of negation; however, using negation it may be defined thus:

$$r(X_1, \ldots, X_n) \leftarrow r_1(X_1, \ldots, X_n), not\ r_2(X_1, \ldots, X_n).$$

For example, let *parent*/2 and *mother*/2 belong to the EDB. Now, *father*/2 can be defined as the difference of the relations *parent*/2 and *mother*/2:

$$father(X, Y) \leftarrow parent(X, Y), not\ mother(X, Y).$$

The *cartesian product* of two relations R_1 and R_2 (denoted $R_1 \times R_2$) yields the new relation:

$$\{\langle x_1, \ldots, x_m, y_1, \ldots, y_n \rangle \mid \langle x_1, \ldots, x_m \rangle \in R_1 \wedge \langle y_1, \ldots, y_n \rangle \in R_2\}$$

Notice that R_1 and R_2 may have both different domains and different arities. Moreover, if R_1 and R_2 contain disjoint sets of attributes they are carried over to the resulting relation. However, if the original relations contain some joint attribute the attribute of the two columns in the new relation must be renamed into distinct ones. This can be done e.g. by prefixing the joint attributes in the new relation by the relation where they came from. For instance, in the relation $R(A, B) \times S(B, C)$ the attributes are, from left to right, A, $R.B$, $S.B$ and C. Obviously, it is possible to achieve the same effect in other ways.

In logic programming the cartesian product is mimicked by the rule:

$$r(X_1, \ldots, X_m, Y_1, \ldots, Y_n) \leftarrow r_1(X_1, \ldots, X_m), r_2(Y_1, \ldots, Y_n).$$

[1]In what follows we will sometimes, by abuse of language, write "the relation p/n". Needless to say, p/n is not a relation but a predicate symbol which *denotes* a relation.

For instance, let *male*/1 and *female*/1 belong to the EDB. Then the set of all male-female couples can be defined by the rule:

$$couple(X, Y) \leftarrow male(X), female(Y).$$

Projection can be seen as the deletion and/or rearrangement of one or more "columns" of a relation. For instance, by projecting the F- and C-attributes of the relation $FATHER(F, C)$ on the F-attribute (denoted $\pi_F(FATHER(F, C))$) the new relation:

$$\{\langle x_1 \rangle \mid \langle x_1, x_2 \rangle \in FATHER\}$$

is obtained. The same can be achieved in Prolog by means of the rule:

$$father(X) \leftarrow father(X, Y).$$

The *selection* of a relation R is denoted $\sigma_F(R)$ (where F is a formula) and is the set of all tuples $\langle x_1, \ldots, x_n \rangle \in R$ such that "F is true for $\langle x_1, \ldots, x_n \rangle$". How to translate such an operation to a logic program depends on the appearance of the constraining formula F. In general F is only allowed to contain atomic objects, attributes, $\wedge$, $\vee$, $\neg$ and some simple comparisons (e.g. "$=$" and "$<$"). For instance, the database relation defined by $\sigma_{Y \geq 1{,}000{,}000}\ INCOME(X, Y)$ may be defined as follows in Prolog:

$$millionaire(X, Y) \leftarrow income(X, Y), Y \geq 1000000.$$

Some other operations (like intersection and composition) are sometimes encountered in relational algebra but they are usually all defined in terms of the mentioned, primitive ones and are therefore not discussed here. However, one of them deserves special attention — namely the *natural join*.

The natural join of two relations R and S can be computed only when the columns are named by attributes. Thus, assume that $T_1, \ldots, T_k$ are the attributes which appear both in R and in S. Then the natural join of R and S is defined thus:

$$R \bowtie S := \pi_A\ \sigma_{R.T_1 = S.T_1\ \wedge\ \cdots\ \wedge\ R.T_k = S.T_k}\ (R \times S)$$

where A is the list of all attributes of $R \times S$ with exception of $S.T_1, \ldots, S.T_k$. Thus, the natural join is obtained by (1) taking the cartesian product of the two relations, (2) selecting those tuples which have identical values in the columns with the same attribute and (3) filtering out the superfluous columns. Notice that if R and S have disjoint sets of attributes, then the natural join reduces to an ordinary cartesian product.

To illustrate the operation, consider the relation defined by $F(X, Y) \bowtie P(Y, Z)$ where $F(X, Y)$ and $P(Y, Z)$ are defined according to Figure 6.1(a) and 6.1(b) and denote the relation between fathers/parents and their children.

Now $F(X, Y) \bowtie P(Y, Z)$ is defined as $\pi_{X, F.Y, Z}\ \sigma_{F.Y = P.Y}\ (F(X, Y) \times P(Y, Z))$. Hence the first step consists in computing the cartesian product $F(X, Y) \times P(Y, Z)$ (cf. Figure 6.1(c)). Next the tuples with equal values in the columns named by $F.Y$ and $P.Y$ are selected (Figure 6.1(d)). Finally this is projected on the X, $F.Y$ and Z attributes yielding the relation in Figure 6.1(e).

If we assume that *father*/2 and *parent*/2 are used to represent the database relations F and P then the same relation may be defined with a single definite clause as follows:

X	*Y*
adam	*bill*
bill	*cathy*

(a)

Y	*Z*
adam	*bill*
bill	*cathy*
cathy	*dave*

(b)

X	*F.Y*	*P.Y*	*Z*
adam	*bill*	*adam*	*bill*
adam	*bill*	*bill*	*cathy*
adam	*bill*	*cathy*	*dave*
bill	*cathy*	*adam*	*bill*
bill	*cathy*	*bill*	*cathy*
bill	*cathy*	*cathy*	*dave*

(c)

X	*F.Y*	*P.Y*	*Z*
adam	*bill*	*bill*	*cathy*
bill	*cathy*	*cathy*	*dave*

(d)

X	*F.Y*	*Z*
adam	*bill*	*cathy*
bill	*cathy*	*dave*

(e)

Figure 6.1: Natural join

$$grandfather(X, Y, Z) \leftarrow father(X, Y), parent(Y, Z).$$

Notice that the standard definition of *grandfather*/2:

$$grandfather(X, Z) \leftarrow father(X, Y), parent(Y, Z).$$

is obtained by projecting $X, F.Y, Z$ on X and Z, that is, by performing the operation $\pi_{X,Z}(F(X, Y) \bowtie P(Y, Z))$.

6.4 Logic as a Query-language

In the previous sections it was observed that logic provides a uniform language for representing both explicit data and implicit data (so-called views). However, deductive databases are of little or no interest if it is not possible to retrieve information from the database. In traditional databases this is achieved by so-called *query-languages*. Examples of existing query-languages for relational databases are e.g. ISBL, SQL, QUEL and Query-by-Example.

By now it should come as no surprise to the reader that logic programming can be used as a query-language in the same way it was used to define views. For instance, to retrieve the children of Adam from the database in Example 6.2 one only has to give the goal clause:

$$\leftarrow parent(adam, X).$$

To this Prolog-systems would respond with the answers $X = bill$ and $X = beth$, or put alternatively — the unary relation $\{\langle bill\rangle, \langle beth\rangle\}$. Likewise, in response to the goal:

$$\leftarrow mother(X, Y).$$

Prolog produces four answers:

$$\begin{array}{ll} X = anne, & Y = bill \\ X = anne, & Y = beth \\ X = cathy, & Y = donald \\ X = diana, & Y = eric \end{array}$$

That is, the relation:

$$\{\langle anne, bill\rangle, \langle anne, beth\rangle, \langle cathy, donald\rangle, \langle diana, eric\rangle\}$$

Notice that a failing goal (e.g. $\leftarrow parent(X, adam)$) computes the empty relation as opposed to a succeeding goal without variables (e.g $\leftarrow parent(adam, bill)$) which computes a singleton relation containing a 0-ary tuple.

Now consider the following excerpt from a database:

$$\begin{array}{l} likes(X, Y) \leftarrow baby(Y). \\ baby(mary). \\ \quad\vdots \end{array}$$

Informally the two clauses say that "Everybody likes babies" and "Mary is a baby". Consider the result of the query "Is anyone liked by someone?". In other words the goal clause:

$$\leftarrow likes(X, Y).$$

Clearly Prolog will reply with $Y = mary$ and X being unbound. This is interpreted as "Everybody likes Mary" but what does it mean in terms of a database relation? One solution to the problem is to declare a type-predicate and to extend the goal with calls to this new predicate:

$$\leftarrow likes(X, Y), person(X), person(Y).$$

In response to this goal Prolog would enumerate all individuals of type $person/1$. It is also possible to add the extra literal $person(X)$ to the database rule. Another approach which is often employed when describing deductive databases is to adopt certain assumptions about the world which is modelled. One such assumption was mentioned already in connection with Chapter 4 — namely the closed world assumption (CWA). Another assumption which is usually adopted in deductive databases is the so-called *domain closure assumption* (DCA) which states that "the only existing individuals are those mentioned in the database". In terms of logic this can be expressed through the additional axiom:

$$\forall X(X = c_1 \lor X = c_2 \lor \cdots \lor X = c_n)$$

where $c_1, c_2, \ldots, c_n$ are all the constants occurring in the database. With this axiom the relation defined by the goal above becomes $\{\langle t, mary\rangle \mid t \in U_P\}$. However, this assumes that the database contains no functors and only a finite number of constants.

6.5 Special Relations

The main objective of this section is to show how to define relations that possess certain properties occurring frequently both in real life and in mathematics. This includes properties like *reflexivity*, *symmetry* and *transitivity*.

Let R be a binary relation over some domain $\mathcal{D}$. Then:

- R is said to be *reflexive* iff for all $x \in \mathcal{D}$, it holds that $\langle x, x\rangle \in R$;
- R is *symmetric* iff $\langle x, y\rangle \in R$ implies that $\langle y, x\rangle \in R$;
- R is *anti-symmetric* iff $\langle x, y\rangle \in R$ and $\langle y, x\rangle \in R$ implies that $x = y$;
- R is *transitive* iff $\langle x, y\rangle \in R$ and $\langle y, z\rangle \in R$ implies that $\langle x, z\rangle \in R$;
- R is *asymmetric* iff $\langle x, y\rangle \in R$ implies that $\langle y, x\rangle \notin R$.

To define an EDB which possesses one of these properties is usually a rather cumbersome task if the domain is large. For instance, to define a reflexive relation over a domain with n elements requires n tuples, or n facts in the case of a logic program. Fortunately, in logic programming, relations can be defined to be reflexive with a single clause of the form:

$$r(X, X).$$

However, in many cases one thinks of the Herbrand universe as the coded union of several domains. For instance, the Herbrand universe consisting of the constants *bill*, *kate* and *love* may be thought of as the coded union of persons and abstract notions. If — as in this example — the intended domain of $r/2$ (encoded as terms) ranges over proper subsets of the Herbrand universe and if the type predicate $t/1$ characterize this subset, a reflexive relation can be written as follows:

$$r(X, X) \leftarrow t(X).$$

For instance, in order to say that "every person looks like himself" we may write the following program:

$$\begin{array}{l} looks\_like(X, X) \leftarrow person(X). \\ person(bill). \\ person(kate). \\ abstract(love). \end{array}$$

In order to define a symmetric relation R it suffices to specify only one of the pairs $\langle x, y\rangle$ and $\langle y, x\rangle$ if $\langle x, y\rangle \in R$. Then the program is extended with the rule:

$$r(X, Y) \leftarrow r(Y, X).$$

However, as shown below such programs suffer from operational problems.

Example 6.3 Consider the domain:

$$\{sarah, diane, pamela, simon, david, peter\}$$

The relation "... is married to ..." clearly is symmetric and it may be written either as an extensional database:

$$\begin{array}{l} married(sarah, simon). \\ married(diane, david). \\ married(pamela, peter). \\ married(simon, sarah). \\ married(david, diane). \\ married(peter, pamela). \end{array}$$

or more briefly as a deductive database:

$$\begin{array}{l} married(X, Y) \leftarrow married(Y, X). \\ married(sarah, simon). \\ married(diane, david). \\ married(pamela, peter). \end{array}$$

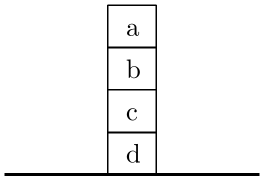
∎

Transitive relations can also be simplified by means of rules. Instead of a program P consisting solely of facts, P can be fully described by the clause:

$$r(X, Z) \leftarrow r(X, Y), r(Y, Z).$$

together with all $r(a, c) \in P$ for which there exists no b ($b \neq a$ and $b \neq c$) such that $r(a, b) \in P$ and $r(b, c) \in P$.

Example 6.4 Consider the world consisting of the "objects" a, b, c and d:



The relation "... is positioned over ..." clearly is transitive and may be defined either through a purely extensional database:

$$\begin{array}{ll} over(a, b). & over(a, c). \\ over(a, d). & over(b, c). \\ over(b, d). & over(c, d). \end{array}$$

or alternatively as the deductive database:

$$\begin{array}{l} over(X, Z) \leftarrow over(X, Y), over(Y, Z). \\ over(a, b). \\ over(b, c). \\ over(c, d). \end{array}$$

∎

The definitions above are declaratively correct, but they suffer from operational problems when executed by Prolog systems. Consider the goal $\leftarrow \mathit{married}(\mathit{diane}, \mathit{david})$ together with the deductive database of Example 6.3. Clearly $\mathit{married}(\mathit{diane}, \mathit{david})$ is a logical consequence of the program but any Prolog interpreter would go into an infinite loop — first by trying to prove:

$$\leftarrow \mathit{married}(\mathit{diane}, \mathit{david}).$$

Via unification with the rule a new goal clause is obtained:

$$\leftarrow \mathit{married}(\mathit{david}, \mathit{diane}).$$

When trying to satisfy $\mathit{married}(\mathit{david}, \mathit{diane})$ the subgoal is once again unified with the rule yielding a new goal, identical to the initial one. This process will obviously go on forever. The misbehaviour can, to some extent, be avoided by moving the rule textually after the facts. By doing so it may be possible to find some (or all) refutations before going into an infinite loop. However, no matter how the clauses are ordered, goals like $\leftarrow \mathit{married}(\mathit{diane}, \mathit{diane})$ always lead to loops.

A better way of avoiding such problems is to use an auxiliary anti-symmetric relation instead and to take the *symmetric closure* of this relation. This can be done by renaming the predicate symbol of the EDB with the auxiliary predicate symbol and then introducing two rules which define the symmetric relation in terms of the auxiliary one.

Example 6.5 The approach is illustrated by defining *married*/2 in terms of the auxiliary definition *wife*/2 which is anti-symmetric:

$$\begin{array}{l} \mathit{married}(X,Y) \leftarrow \mathit{wife}(X,Y). \\ \mathit{married}(X,Y) \leftarrow \mathit{wife}(Y,X). \\ \\ \mathit{wife}(\mathit{sarah}, \mathit{simon}). \\ \mathit{wife}(\mathit{diane}, \mathit{david}). \\ \mathit{wife}(\mathit{pamela}, \mathit{peter}). \end{array}$$

This program has the nice property that it never loops — simply because it is not recursive. ∎

A similar approach can be applied when defining transitive relations. A new auxiliary predicate symbol is introduced and used to rename the EDB. Then the *transitive closure* of this relation is defined by means of the following two rules (where $p/2$ denotes the transitive relation and $q/2$ the auxiliary one):

$$\begin{array}{l} p(X,Y) \leftarrow q(X,Y) \\ p(X,Y) \leftarrow q(X,Z), p(Z,Y). \end{array}$$

Example 6.6 The relation *over*/2 may be defined in terms of the predicate symbol *on*/2:

$$\begin{array}{l} \mathit{over}(X,Y) \leftarrow \mathit{on}(X,Y). \\ \mathit{over}(X,Z) \leftarrow \mathit{on}(X,Y), \mathit{over}(Y,Z). \end{array}$$

$$\begin{array}{l} on(a, b). \\ on(b, c). \\ on(c, d). \end{array}$$

Notice that recursion is not completely eliminated. It may therefore happen that the program loops. As shown below this depends on properties of the auxiliary relation. ∎

The transitive closure may be combined with the *reflexive closure* of a relation. Given an auxiliary relation denoted by $q/2$, its reflexive and transitive closure is obtained through the additional clauses:

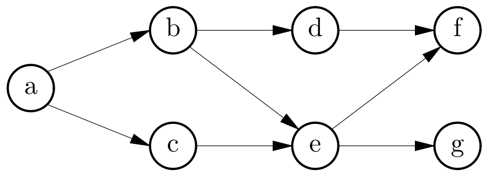
$$\begin{array}{l} p(X, X). \\ p(X, Y) \leftarrow q(X, Y). \\ p(X, Z) \leftarrow q(X, Y), p(Y, Z). \end{array}$$

Actually, the second clause is superfluous since it follows logically from the first and third clause: any goal, $\leftarrow p(a, b)$, which is refuted through unification with the second clause can be refuted through unification with the third clause where the recursive subgoal is unified with the first clause.

Next we consider two frequently encountered types of relations — namely *partial orders* and *equivalence relations*.

A binary relation is called a *partial order* if it is reflexive, anti-symmetric and transitive whereas a relation which is reflexive, symmetric and transitive is called an *equivalence relation*.

Example 6.7 Consider a directed, acyclic graph:



It is easy to see that the relation "there is a path from ... to ..." is a partial order given the graph above. To formally define this relation we start with an auxiliary, asymmetric relation (denoted by $edge/2$) which describes the edges of the graph:

$$\begin{array}{ll} edge(a, b). & edge(c, e). \\ edge(a, c). & edge(d, f). \\ edge(b, d). & edge(e, f). \\ edge(b, e). & edge(e, g). \end{array}$$

Then the reflexive and transitive closure of this relation is described through the two clauses:

$$\begin{array}{l} path(X, X). \\ path(X, Z) \leftarrow edge(X, Y), path(Y, Z). \end{array}$$

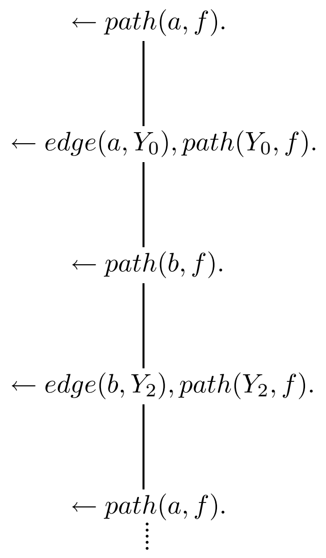

$\leftarrow path(a, f).$
$\leftarrow edge(a, Y_0), path(Y_0, f).$
$\leftarrow path(b, f).$
$\leftarrow edge(b, Y_2), path(Y_2, f).$
$\leftarrow path(a, f).$

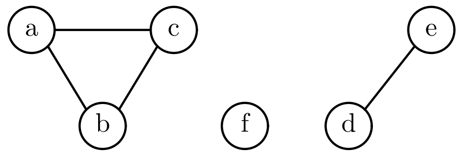
Figure 6.2: Infinite branch in the SLD-tree

This program does not suffer from infinite loops. In fact, no partial order defined in this way will loop as long as the domain is finite. However, if the graph contains a loop it may happen that the program starts looping — consider the addition of a cycle in the above graph. For instance, an additional edge from b to a:

$$edge(b, a).$$

Part of the SLD-tree of the goal $\leftarrow path(a, f)$ is depicted in Figure 6.2. The SLD-tree clearly contains an infinite branch and hence it may happen that the program starts looping without returning any answers. In Chapter 11 this problem will be discussed and a solution will be suggested. ▮

Example 6.8 Next consider some points on a map and bi-directed edges between the points:



This time the relation "there is a path from ... to ..." is an equivalence relation. To define the relation we may start by describing one half of each edge in the graph:

$$
\begin{array}{l}
edge(a, b).\\
edge(a, c).\\
edge(b, c).\\
edge(d, e).
\end{array}
$$

Next the other half of each edge is described by means of the symmetric closure of the relation denoted by $edge/2$:

$$
\begin{array}{l}
bi\_edge(X, Y) \leftarrow edge(X, Y).\\
bi\_edge(X, Y) \leftarrow edge(Y, X).
\end{array}
$$

Finally, $path/2$ is defined by taking the reflexive and transitive closure of this relation:

$$
\begin{array}{l}
path(X, X).\\
path(X, Z) \leftarrow bi\_edge(X, Y), path(Y, Z).
\end{array}
$$

Prolog programs defining equivalence relations usually suffer from termination problems unless specific measures are taken (cf. Chapter 11). ▮

6.6 Databases with Compound Terms

In relational databases it is usually required that the domains consist of atomic objects, something which simplifies the mathematical treatment of relational databases. Naturally, when using logic programming, nothing prevents us from using structured data when writing deductive databases. This allows for data abstraction and in most cases results in greater expressive power and improves readability of the program.

Example 6.9 Consider a database which contains members of families and the addresses of the families. Imagine that a family is represented by a ternary term $family/3$ where the first argument is the name of the husband, the second the name of the wife and the last a structure which contains the names of the children. The absence of children is represented by the constant *none* whereas the presence of children is represented by the binary term of the form $c(x, y)$ whose first argument is the name of one child and whose second argument recursively contains the names of the remaining children (intuitively *none* can be thought of as the empty set and $c(x, y)$ can be thought of as a function which constructs a set by adding x to the set represented by y). An excerpt from such a database might look as follows:

$$
\begin{array}{l}
address(family(john, mary, c(tom, c(jim, none))), main\_street(3)).\\
address(family(bill, sue, none), main\_street(4)).\\
\\
parent(X, Y) \leftarrow\\
\qquad address(family(X, Z, Children), Street),\\
\qquad among(Y, Children).\\
parent(X, Y) \leftarrow\\
\qquad address(family(Z, X, Children), Street),\\
\qquad among(Y, Children).
\end{array}
$$

$$
\begin{aligned}
&\mathit{husband}(X) \leftarrow \\
&\qquad \mathit{address}(\mathit{family}(X, Y, \mathit{Children}), \mathit{Street}). \\[1ex]
&\mathit{wife}(Y) \leftarrow \\
&\qquad \mathit{address}(\mathit{family}(X, Y, \mathit{Children}), \mathit{Street}). \\[1ex]
&\mathit{married}(X, Y) \leftarrow \\
&\qquad \mathit{address}(\mathit{family}(X, Y, \mathit{Children}), \mathit{Street}). \\
&\mathit{married}(Y, X) \leftarrow \\
&\qquad \mathit{address}(\mathit{family}(X, Y, \mathit{Children}), \mathit{Street}). \\[1ex]
&\mathit{among}(X, c(X, Y)). \\
&\mathit{among}(X, c(Y, Z)) \leftarrow \\
&\qquad \mathit{among}(X, Z).
\end{aligned}
$$

▮

The database above *can* be represented in the form of a traditional database by introducing a unique key for each family. For example as follows:

$$
\begin{aligned}
&\mathit{husband}(f1, \mathit{john}). \\
&\mathit{husband}(f2, \mathit{bill}). \\[1ex]
&\mathit{wife}(f1, \mathit{mary}). \\
&\mathit{wife}(f2, \mathit{sue}). \\[1ex]
&\mathit{child}(f1, \mathit{tom}). \\
&\mathit{child}(f1, \mathit{jim}). \\[1ex]
&\mathit{address}(f1, \mathit{main\_street}, 3). \\
&\mathit{address}(f2, \mathit{main\_street}, 4). \\[1ex]
&\mathit{parent}(X, Y) \leftarrow \mathit{husband}(\mathit{Key}, X), \mathit{child}(\mathit{Key}, Y). \\
&\qquad\qquad\qquad \vdots
\end{aligned}
$$

However, the latter representation is less readable and it may also require some extra book-keeping to make sure that each family has a unique key.

To conclude — the issues discussed in this chapter were raised to demonstrate the advantages of using logic as a uniform language for representing databases. Facts, rules and queries can be written in a single language. Moreover, logic supports definition of relations via recursive rules, something which is not allowed in traditional databases. Finally, the use of structured data facilitates definition of relations which cannot be made in traditional relational databases. From this stand-point logic programming provides a very attractive conceptual framework for describing relational databases. On the other hand we have not raised important issues like how to implement such databases let alone how to handle updates to deductive databases.

Exercises

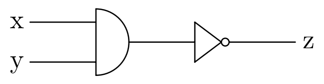
6.1 Reorganize the database in Example 6.2 so that *father*/2 and *mother*/2 become part of the intensional database.

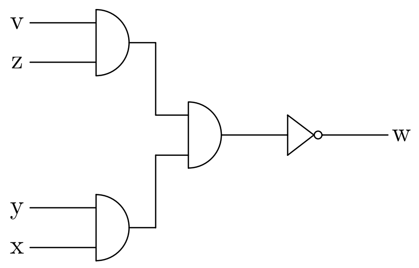
6.2 Extend Example 6.2 with some more persons. Then define the following predicate symbols (with obvious intended interpretations):

- *grandchild*/2
- *sister*/2
- *brother*/2
- *cousins*/2
- *uncle*/2
- *aunt*/2

6.3 Consider an arbitrary planar map of countries. Write a program which colours the map using only four colours so that no two adjacent countries have the same colour. NOTE: Two countries which meet only pointwise are not considered to be adjacent.

6.4 Define the input-output behaviour of AND- and inverter-gates. Then describe the relation between input and output of the following nets:





6.5 Translate the following relational algebra expressions into definite clauses.

- $\pi_{X,Y}(HUSBAND(Key, X) \bowtie WIFE(Key, Y))$
- $\pi_X(PARENT(X, Y) \cup \pi_X\ \sigma_{Y \leq 20{,}000}\ INCOME(X, Y))$

6.6 The following clauses define a binary relation denoted by $p/2$ in terms of the relations $q/2$ and $r/2$. How would you define the same relation using relational algebra?

$$p(X, Y) \leftarrow q(Y, X).$$
$$p(X, Y) \leftarrow q(X, Z), r(Z, Y).$$

6.7 Let R_1 and R_2 be subsets of $\mathcal{D} \times \mathcal{D}$. Define the composition of R_1 and R_2 using (1) definite programs; (2) relational algebra.

6.8 Let R_1 and R_2 be subsets of $\mathcal{D} \times \mathcal{D}$. Define the intersection of R_1 and R_2 using (1) definite programs; (2) relational algebra.

6.9 An ancestor is a parent, a grandparent, a great-grandparent etc. Define a relation $ancestor/2$ which is to hold if someone is an ancestor of somebody else.

6.10 Andrew, Ann, and Adam are siblings and so are Bill, Beth and Basil. Describe the relationships between these persons using as few clauses as possible.

6.11 Define a database which relates dishes and all of their ingredients. For instance, pancakes contain milk, flour and eggs. Then define a relation which describes the available ingredients. Finally define two relations:

- $can\_cook(X)$ which should hold for a dish X if all its ingredients are available;
- $needs\_ingredient(X, Y)$ which holds for a dish X and an ingredient Y if X contains Y.

6.12 Modify the previous exercise as follows — add to the database the quantity available of each ingredient and for each dish the quantity needed of each ingredient. Then modify the definition of $can\_cook/1$ so that the dish can be cooked if each of its ingredients is available in sufficient quantity.

Chapter 7

Programming with Recursive Data Structures

7.1 Recursive Data Structures

In the previous chapter we studied a class of programs that manipulate simple data objects — mostly constants. However, the last section of the chapter introduced the use of compound terms for representation of more complex worlds — like families and their members. Such data objects are typically used when there is a need to represent some collection of individuals where the size is not fixed or when the set of individuals is infinite. In the example a family may have indefinitely many children. Such objects are usually represented by means of so called *recursive data structures.* A recursive data structure is so called because its data objects may contain, recursively as substructures, objects of the same "type". In the previous chapter the functor $c/2$ was used to represent the children of a family — the first argument contained the name of one child and the second, recursively, a representation of the remaining children.

This chapter discusses some recursive data structures used commonly in logic programs and programming techniques for dealing with such structures.

7.2 Lists

Some well-known programming languages — for instance Lisp — use *lists* as the primary representation of data (and programs). Although logic programming only allows terms as representations of individuals, it is not very hard to *represent* lists as terms. Most Prolog systems even support the use of lists by means of special syntax. We will first introduce a precise concept of list.

Let $\mathcal{D}$ be some domain of objects. The set of all lists (over $\mathcal{D}$) is defined inductively as the smallest set satisfying the following conditions:

- the empty list (denoted ϵ) is a list (over $\mathcal{D}$);
- if T is a list (over $\mathcal{D}$) and $H \in \mathcal{D}$ then the pair $\langle H, T \rangle$ is a list (over $\mathcal{D}$).

For instance $\langle 1, \langle 2, \epsilon \rangle \rangle$ is a list over the domain of natural numbers.

In Prolog the empty list ϵ is usually represented by the constant $[\,]$ and a pair is represented using the binary functor $\mathtt{.}/2$. The list above is thus represented by the term $\mathtt{.}(1, \mathtt{.}(2, [\,]))$ and is said to have two *elements* — "1" and "2". The possibility of having different types of lists (depending on what domain $\mathcal{D}$ one uses) introduces a technical problem since logic programs lack a type-system. In general we will only consider types over a universal domain which will be represented by the Herbrand universe.

To avoid having to refer to the "representation of lists" every time such a term is referred to, it will simply be called a *list* in what follows. However, when the word "list" is used it is important to keep in mind that the object still is a term.

Every list (but the empty one) has a *head* and a *tail*. Given a list of the form $\mathtt{.}(H, T)$ the first argument is called the head and the second the tail of the list. For instance, $\mathtt{.}(1, \mathtt{.}(2, [\,]))$ has the head 1 and tail $\mathtt{.}(2, [\,])$. To avoid this rather awkward notation, most Prolog systems use an alternative syntax for lists. The general idea is to write $[H|T]$ instead of $\mathtt{.}(H, T)$. But since $[1|[2|[\,]]]$ is about as difficult to write (and read) as $\mathtt{.}(1, \mathtt{.}(2, [\,]))$ the following simplifications are allowed:

- $[s_1, \ldots, s_m|[t_1, \ldots, t_n|X]]$ is usually written $[s_1, \ldots, s_m, t_1, \ldots, t_n|X]$ $(m, n > 0)$;
- $[s_1, \ldots, s_m|[t_1, \ldots, t_n]]$ is usually written $[s_1, \ldots, s_m, t_1, \ldots, t_n]$ $(m > 0, n \geq 0)$.

Hence, instead of writing $[a|[b|[c|[\,]]]]$ the notation $[a, b, c]$ is used (note that $[c|[\,]]$ is written as $[c]$, $[b|[c]]$ is written as $[b, c]$ and $[a|[b, c]]$ is written as $[a, b, c]$). Similarly $[a, b|[c|X]]$ is written as $[a, b, c|X]$.

It is easy to write procedures which relate a list to its head and tail (cf. the functions CAR and CDR in Lisp):

$$\begin{array}{l} car(Head, [Head|Tail]). \\ cdr(Tail, [Head|Tail]). \end{array}$$

Presented with the goal $\leftarrow cdr(X, [a, b, c])$ Prolog answers $X = [b, c]$.

Now consider the definition of lists again. Looking more closely at the two statements defining what a list is, it is not very hard to see that both statements can be formulated as definite clauses — the first statement as a fact and the second as a recursive rule.

Example 7.1 Formally the definition of lists can be expressed as follows:

$$\begin{array}{l} list([\,]). \\ list([Head|Tail]) \leftarrow list(Tail). \end{array}$$

∎

This "type"-declaration has two different uses — it can (1) be used to *test* whether a term is a list or (2) to *enumerate/generate* all possible lists. In reply to the definite goal $\leftarrow list(X)$ — "Is there some X such that X is a list?" — Prolog starts enumerating all possible lists starting with $[\,]$ and followed by $[X_1], [X_1, X_2]$, etc. Remember that answers containing variables are understood to be universally quantified — that is, the second answer is interpreted as "For any X_1, $[X_1]$ is a list". (Of course the names of the variables may differ but are not important anyway.)

The next program considered is actually a version of the *among*/2 program from the previous chapter. Here it is called *member*/2 and it is used to describe membership in a list. An informal definition looks as follows:

- X is a member of any list whose head is X;
- if X is a member of *Tail* then X is a member of any list whose tail is *Tail*.

Again observe that the definition is directly expressible as a definite program!

Example 7.2

$$\begin{array}{l} member(X, [X \mid Tail]). \\ member(X, [Y \mid Tail]) \leftarrow member(X, Tail). \end{array}$$

∎

As a matter of fact, the first clause does not quite express what we intended. For instance, the goal $\leftarrow member(a, [a \mid b])$ has a refutation even though $[a \mid b]$ is not a list according to our definition. Such unwanted inferences could be avoided by strengthening the first clause into:

$$member(X, [X \mid Tail]) \leftarrow list(Tail).$$

Unfortunately the extra condition makes the program less efficient. Resolving a goal of the form:

$$\leftarrow member(t_m, [t_1, \ldots, t_m, \ldots, t_{m+n}]).$$

requires $n + 1$ extra resolution steps. Moreover, it is not necessary to have the extra condition if the program is used as expected, that is for examination of *list* membership only.

Just as *list*/1 has more than one use depending on how the arguments of the goal are instantiated, *member*/2 can be used either to test or to generate answers. For instance, the goal $\leftarrow member(b, [a, b, c])$ has a refutation whereas $\leftarrow member(d, [a, b, c])$ fails. By leaving the first argument uninstantiated the *member*/2-program will enumerate the elements of the list in the second argument. For instance, the goal $\leftarrow member(X, [a, b, c])$ has three refutations with three different answers — under Prolog's depth-first search strategy the first answer is $X = a$, followed by $X = b$ and finally $X = c$. The SLD-tree of the goal is shown in Figure 7.1.

Note that the program computes all the expected answers. Consider instead the goal $\leftarrow member(a, X)$ which reads "Is there some list which contains a?". The SLD-tree of the goal is depicted in Figure 7.2.

The first answer produced is $X = [a \mid Tail_0]$ which is interpreted as — "For any $Tail_0$, $[a \mid Tail_0]$ has a as a member" or less strictly "Any list starting with a contains a". The second success branch first binds X to $[Y_0 \mid Tail_0]$ and then binds $Tail_0$ to $[a \mid Tail_1]$. Hence the complete binding obtained for X is $[Y_0 \mid [a \mid Tail_1]\,]$ which is

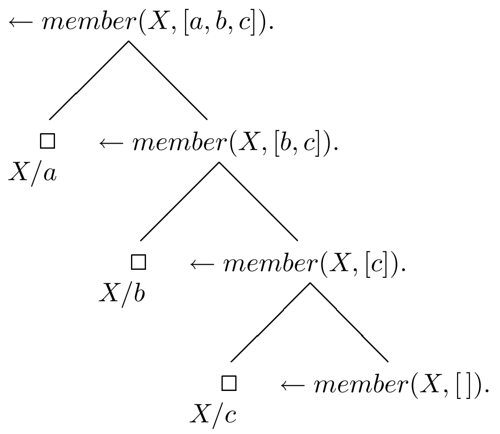

$\leftarrow member(X, [a, b, c])$.
$\square$
X/a
$\leftarrow member(X, [b, c])$.
$\square$
X/b
$\leftarrow member(X, [c])$.
$\square$
X/c
$\leftarrow member(X, [\,])$.

Figure 7.1: SLD-tree of the goal $\leftarrow member(X, [a, b, c])$

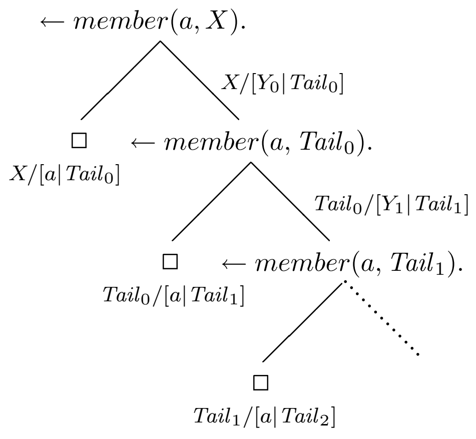


Figure 7.2: SLD-tree of the goal $\leftarrow member(a, X)$

equivalent to $[Y_0, a \mid Tail_1]$ and is interpreted as "Any list with a as the second element contains a". Similarly the third answer is interpreted as "Any list with a as the third element contains a". It is not hard to see that there are infinitely many answers of this kind and the SLD-tree obviously contains infinitely many success branches. This brings us to an important question discussed briefly in Chapter 3 — what impact has the textual ordering of clauses in Prolog?

What happens if the clauses in the *member*/2-program are swapped? Referring to Figure 7.1 one can see that instead of first traversing the leftmost branch in the SLD-tree the rightmost branch is traversed first. This branch will eventually fail, so the computation backtracks until the first answer (which is $X = c$) is found. Then the computation backtracks again and the answer $X = b$ is found followed by the final answer, $X = a$. Thus, nothing much happens — the SLD-tree is simply traversed in an alternative fashion which means that the answers show up in a different order. This may, of course, have serious impacts if the tree contains some infinite branch — consider the rightmost branch of the tree in Figure 7.2. Clearly no clause ordering will affect the size of the tree and it is therefore not possible to traverse the whole tree (that is, find all answers). However if the rightmost branch in the tree is always selected before the leftmost one the computation will loop for ever without reporting *any* answers (although there are answers to the goal).

The halting problem is of course undecidable (i.e. it is in general not possible to determine whether a program will loop or not), but it is good practice to put facts before recursive clauses when writing a recursive program. In doing so it is often possible to find all, or at least some, of the answers to a goal before going into an infinite loop. There is also another good reason for doing this which has to do with the implementation of modern Prolog compilers. If the rightmost subgoal in the last clause of a definition is a recursive call, the Prolog compiler is sometimes able to produce more efficient machine code.

The next program considered is that of "putting two lists together". The name commonly used for the program is *append*/3 although a more appropriate name would be *concatenate*/3. As an example, appending a list $[c, d]$ to another list $[a, b]$ yields the new list $[a, b, c, d]$. More formally the relation can be defined as follows:

- appending any list X to the empty list yields the list X;
- if appending Z to Y yields W, then appending Z to $[X|Y]$ yields $[X|W]$.

Again there is a direct translation of the definition into a definite program:

Example 7.3

$$\begin{array}{l} append([\,], X, X). \\ append([X|Y], Z, [X|W]) \leftarrow append(Y, Z, W). \end{array}$$

∎

Just like the previous programs, the *append*/3-program can be used in many different ways. Obviously, we can use it to test if the concatenation of two lists equals a third list by giving the goal:

$$\leftarrow append([a, b], [c, d], [a, b, c, d])$$

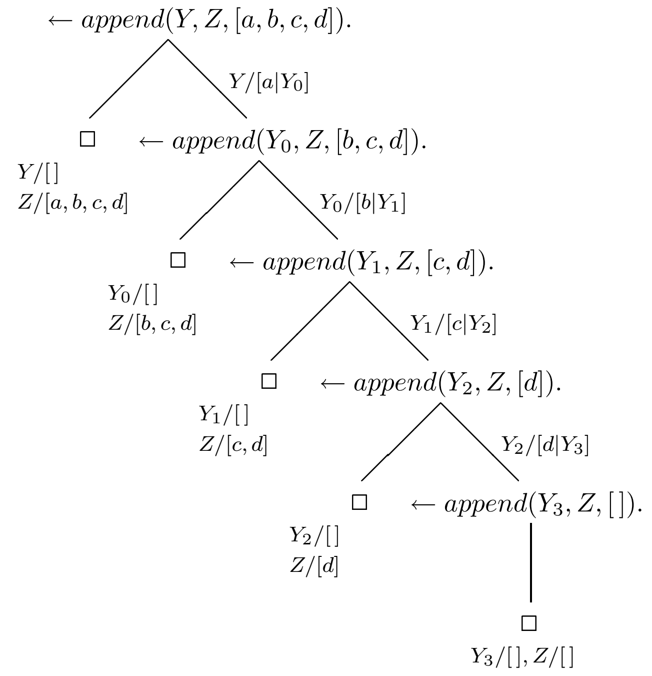

$\leftarrow append(Y, Z, [a, b, c, d])$.
$Y/[a|Y_0]$
$Y/[\,]$
$Z/[a, b, c, d]$
$\leftarrow append(Y_0, Z, [b, c, d])$.
$Y_0/[b|Y_1]$
$Y_0/[\,]$
$Z/[b, c, d]$
$\leftarrow append(Y_1, Z, [c, d])$.
$Y_1/[c|Y_2]$
$Y_1/[\,]$
$Z/[c, d]$
$\leftarrow append(Y_2, Z, [d])$.
$Y_2/[d|Y_3]$
$Y_2/[\,]$
$Z/[d]$
$\leftarrow append(Y_3, Z, [\,])$.
$Y_3/[\,], Z/[\,]$

Figure 7.3: SLD-tree of the goal $\leftarrow append(Y, Z, [a, b, c, d])$

It can also be used "as a function" to concatenate two lists into a third list:

$$\leftarrow append([a, b], [c, d], X)$$

in which case the computation succeeds with the answer $X = [a, b, c, d]$. However, it is also possible to give the goal:

$$\leftarrow append(Y, Z, [a, b, c, d])$$

which reads — "Are there two lists, Y and Z, such that Z appended to Y yields $[a, b, c, d]$?". Clearly there are two such lists — there are in fact five different possibilities:

$$\begin{array}{ll} Y = [\,] & Z = [a, b, c, d] \\ Y = [a] & Z = [b, c, d] \\ Y = [a, b] & Z = [c, d] \\ Y = [a, b, c] & Z = [d] \\ Y = [a, b, c, d] & Z = [\,] \end{array}$$

By now it should come as no surprise that all of these answers are reported by Prolog. The SLD-tree of the goal is depicted in Figure 7.3.

The program can actually be used for even more sophisticated tasks. For instance, the rule:

$$unordered(List) \leftarrow append(Front, [X, Y|End], List), X > Y.$$

describes the property of being an unordered list of integers. The clause expresses the fact that a list is unordered if there are two consecutive elements where the first is greater than the second.

Another example of the use of *append*/3 is shown in the following clause, which defines the property of being a list with multiple occurrences of some element:

$$multiple(List) \leftarrow append(L1, [X|L2], List), append(L3, [X|L4], L2).$$

The following is an alternative (and perhaps more obvious) definition of the same property:

$$\begin{array}{l} multiple([Head|\,Tail]) \leftarrow member(Head, Tail). \\ multiple([Head|\,Tail]) \leftarrow multiple(Tail). \end{array}$$

The *append*/3-program can also be used to define the membership-relation and the relation between lists and their last elements. (These are left as exercises.) One may be willing to compare a definition of the last element of the list based on *append*/3 with the following direct definition:

- X is the last element in the list $[X]$;
- if X is the last element in the list *Tail* then it is also the last element in the list $[Head|\,Tail]$.

This can be formalized as follows:

Example 7.4

$$\begin{array}{l} last(X, [X]). \\ last(X, [Head|\,Tail]) \leftarrow last(X, Tail). \end{array}$$

∎

All of the programs written so far have a similar structure — the first clause in each of them is a fact and the second clause is a recursive rule. In Examples 7.1 and 7.3 the resemblance is even closer: The program of Example 7.1 can be obtained from that of Example 7.3 by removing the second and third arguments of each atom in the program. This is no coincidence since (almost) every program that operates on lists has a uniform structure. Some programs differ slightly from the general pattern, like examples 7.2 and 7.4 — on the other hand, when removing the first argument from the atoms in these programs, they also closely resemble the *list*/1-program.

Almost all programs in this chapter (also those which follow) are defined by means of a technique which looks like that of *inductive definitions* of sets.

Remember that relations are sets of tuples. The propositions of an inductive definition describe which tuples are in, and outside of this set. The first proposition (usually called the *basic clause*) in the definition is normally unconditional or uses only already fully defined relation(s). It introduces some (one or more) initial tuples in the set.

The second proposition (called the *inductive clause*) states that if some tuples are in the set (and possibly satisfy some other, already defined relations) then some other tuples are also in the set. The inductive clause is used to repeatedly "pump up" the set as much as possible. That is, the basic clause gives a set S_0. The inductive clause then induces a new set S_1 from S_0. But since S_1 may contain tuples which do not appear in S_0, the inductive clause is used on S_1 to obtain the set S_2 and so on. The basic and inductive clause are sometimes called *direct* clauses.

The direct clauses specify that some tuples are *in* the set (the relation). But that does not exclude the set from also containing other tuples. For instance, saying that 1 is an integer does not exclude Tom and 17 from being integers. Hence, an inductive definition contains also a third clause (called the *extremal clause*) which states that no other tuples are in the set than those which belong to it as a result of the direct clauses. In the definitions above this last statement is omitted. A justification for this is that definite programs describe only positive information and it is not possible to express the extremal clause as a definite clause. However, taking into account the negation-as-failure rule, the extremal clause becomes explicit when considering the completion of the program. For instance the completion of Example 7.1 contains the formula:

$$\forall X(list(X) \leftrightarrow X = [\,] \vee \exists Head, Tail(X = [Head|Tail] \wedge list(Tail)))$$

The "if"-part of this formula corresponds to the direct clauses whereas the "only if"-part is the extremal clause which says that an individual is a list only if it is the empty list or a pair where the tail is a list.

The definition of *list*/1 is in some sense prototypical for inductive definitions of relations between lists and other objects. The basic clause states that something holds for the empty list and the inductive clause says that something holds for lists of length n given that something holds for lists of length m, $n > m$. This should be contrasted with the following programs:

$$\begin{array}{l} list([\,]).\\ list(Tail) \leftarrow list([Head|Tail]). \end{array}$$

and:

$$\begin{array}{l} list([\,]).\\ list(X) \leftarrow list(X). \end{array}$$

Declaratively, there is nothing wrong with them. All statements are true in the intended model. However, as inductive definitions they are incomplete. Both of them define the empty list to be the only list. They are not very useful as programs either since the goal $\leftarrow list([a])$ yields an infinite loop.

Next some other, more complicated, relations between lists are considered. The first relation is that between a list and its permutations. Informally speaking, a permutation of a list is a reordering of its elements. Consider a list with n elements. What possible reorderings are there? Clearly the first element can be put in n different positions. Consequently there are $n - 1$ positions where the second element may be put. More generally there are $n - m + 1$ positions where the m-th element may be put. From this it is easy to see that there are $n!$ different permutations of a list with n elements.

The relation between a list and its permutations is defined inductively as follows:

- the empty list is a permutation of itself;
- If W is a permutation of Y and Z is the result of inserting X into W then Z is a permutation of $[X|Y]$.

Or more formally:

Example 7.5

$$\begin{array}{l} \mathit{permutation}([\,],[\,]). \\ \mathit{permutation}([X|Y], Z) \leftarrow \mathit{permutation}(Y, W), \mathit{insert}(X, W, Z). \\ \\ \mathit{insert}(Y, XZ, XYZ) \leftarrow \mathit{append}(X, Z, XZ), \mathit{append}(X, [Y|Z], XYZ). \end{array}$$

∎

Operationally the rule of the program states the following — "Remove the first element, X, from the list to be permuted, then permute the rest of the list and finally insert X into the permutation". The insertion of an element Y into a list XZ is achieved by first splitting XZ into two parts, X and Z. Then the parts are put together with Y in-between.

Now the goal $\leftarrow \mathit{permutation}([a, b, c], X)$ may be given, to which Prolog replies with the six possible permutations:

$$\begin{array}{l} X = [a, b, c] \\ X = [b, a, c] \\ X = [b, c, a] \\ X = [a, c, b] \\ X = [c, a, b] \\ X = [c, b, a] \end{array}$$

Conceptually this relation is symmetric — that is, if A is a permutation of B then B is a permutation of A. In other words, the goal $\leftarrow \mathit{permutation}(X, [a, b, c])$ should return exactly the same answers. So it does (although the order of the answers is different) but after the final answer the program goes into an infinite loop. It turns out that recursive programs with more than one body-literal have to be used with some care. They cannot be called as freely as programs with only a single literal in the body.

If, for some reason, the need arises to call *permutation*/2 with the first argument uninstantiated and still have a finite SLD-tree, the body literals have to be swapped both in the rule of *permutation*/2 and in *insert*/3. After doing this the computation terminates, which means that the SLD-tree of the goal is finite. Hence, when changing the order of the body literals in a clause (or put alternatively, using another computation rule), a completely different SLD-tree is obtained, not just a different traversal of the same tree. As observed above, it is not unusual that one ordering leads to an infinite SLD-tree whereas another ordering results in a finite SLD-tree. Since Prolog uses a fixed computation rule, it is up to the user to make sure that the ordering is "optimal" for the intended use. In most cases this implies that the program only runs efficiently for certain types of goals — for other goals it may be very inefficient or even loop indefinitely (as shown above). If the user wants to run the program in different "directions", it is often necessary to have several versions of the program with different orderings of literals. Needless to say, some programs loop no matter how the body literals are ordered.

The *permutation*/2-program can be used to sort, for instance, lists of natural numbers. The classical specification of the relation between a list and its sorted version says that — "Y is a sorted version of X if Y is a sorted permutation of X". Together with the property *sorted*/1 (which holds if a list of integers is sorted in ascending order) the relation may be defined thus (*nsort*/2 stands for naive sort):

Example 7.6

$$nsort(X, Y) \leftarrow permutation(X, Y), sorted(Y).$$

$$\begin{array}{l} sorted([\,]).\\ sorted([X]).\\ sorted([X, Y|Z]) \leftarrow X \leq Y, sorted([Y|Z]). \end{array}$$

∎

The predicate symbol $\leq /2$ which is used to compare integers is normally predefined as a so called built-in predicate in most Prolog systems. Needless to say, this program is incredibly inefficient. For more efficient sorting programs the reader is advised to solve exercises 7.10 – 7.12. However, the program illustrates quite clearly why the order among the atoms in the body of a clause is important. Consider the goal:

$$\leftarrow nsort([2, 1, 3], X).$$

This reduces to the new goal:

$$\leftarrow permutation([2, 1, 3], X), sorted(X).$$

With the standard computation rule this amounts to finding a permutation of $[2, 1, 3]$ and then checking if this is a sorted list. Not a very efficient way of sorting lists but it is immensely better than first finding a sorted list and then checking if this is a permutation of the list $[2, 1, 3]$ which would be the effect of switching the order among the subgoals. Clearly there are only six permutations of a three-element list but there are infinitely many sorted lists.

The definition of *sorted*/1 differs slightly from what was said above — there are two basic clauses, one for the empty list and one for the list with a single element. It may also happen that there are two or more inductive clauses (cf. exercise 7.12).

The last example considered here is that of reversing a list. Formally the relation between a list and its reversal is defined as follows:

Example 7.7

$$\begin{array}{l} reverse([\,], [\,]).\\ reverse([X|Y], Z) \leftarrow reverse(Y, W), append(W, [X], Z). \end{array}$$

∎

Or more informally:

- the empty list is the reversal of itself;

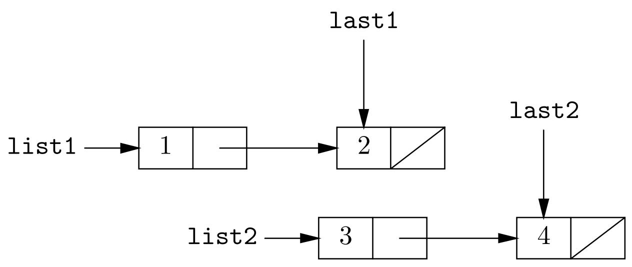


Figure 7.4: Representation of lists

- if W is the reversal of Y and Z is the concatenation of W and $[X]$ then Z is the reversal of $[X|Y]$.

Operationally, the second clause says the following — "to reverse $[X|Y]$, first reverse Y into W then concatenate W with $[X]$ to obtain Z". Just like in Example 7.5 the goal $\leftarrow \mathit{reverse}([a,b,c],X)$ has a finite SLD-tree whereas $\leftarrow \mathit{reverse}(X,[a,b,c])$ has an infinite one. However, when switching the order of the literals in the body of the recursive clause, the situation becomes the opposite.

7.3 Difference Lists

The computational cost of appending two lists in Prolog is typically proportional to the length of the first list. In general a linear algorithm is acceptable but other languages often facilitate concatenation of lists in constant time. The principal idea to achieve constant time concatenation is to maintain a pointer to the end of the list as shown in Figure 7.4. In order to append the two lists the following Pascal-like commands are needed:

```
  ⋮
last1^.pointer := list2;
last1 := last2;
  ⋮
```

In Prolog the same technique can be adopted by using "variables as pointers". Assume that, in the world of lists, there is a (partial) function which given two lists where the second is a suffix of the first, returns the list obtained by removing the suffix from the first (that is, the result is a prefix of the first list). Now let the functor "$-$" denote this function, then the term:

$$[t_1,\ldots,t_m,t_{m+1},\ldots,t_{m+n}] - [t_{m+1},\ldots,t_{m+n}]$$

denotes the same list as does $[t_1,\ldots,t_m]$. More generally, $[a,b,c|L]-L$ denotes the list $[a,b,c]$ for any list assigned to L. As a special case the term $L-L$ designates the empty list for any list assigned to L. It is now possible to use this to define concatenation of difference lists:

$$append(X - Y, Y - Z, X - Z).$$

Declaratively this stands for "Appending the difference of Y and Z to the difference of X and Y yields the difference of X and Z". The correctness of the statement is easier to see when written as follows:

$$\underbrace{t_1 \dots t_i\, \underbrace{t_{i+1} \dots t_j\, \underbrace{t_{j+1} \dots t_k}_{Z}}_{Y}}_{X}$$

Using the new definition of *append*/3 it is possible to refute the goal:

$$\leftarrow append([a, b|X] - X, [c, d|Y] - Y, Z)$$

in a single resolution-step and the computed answer substitution becomes:

$$\{X/[c, d|Y], Z/[a, b, c, d|Y] - Y\}$$

which implies that:

$$append([a, b, c, d|Y] - [c, d|Y], [c, d|Y] - Y, [a, b, c, d|Y] - Y)$$

is a logical consequence of the program.

The ability to concatenate lists in constant time comes in quite handy in some programs. Take for instance the program for reversing lists in Example 7.7. The time complexity of this program is $O(n^2)$ where n is the number of elements in the list given as the first argument. That is, to reverse a list of 100 elements approximately 10,000 resolution steps are needed.

However, using difference lists it is possible to write a program which does the same job in linear time:

Example 7.8

$$reverse(X, Y) \leftarrow rev(X, Y - [\,]).$$

$$\begin{aligned} &rev([\,], X - X).\\ &rev([X|Y], Z - W) \leftarrow rev(Y, Z - [X|W]). \end{aligned}$$

∎

This program reverses lists of n elements in $n + 2$ resolution steps.

Unfortunately, the use of difference lists is not without problems. Consider the goal:

$$\leftarrow append([a, b] - [b], [c, d] - [d], L).$$

One expects to get an answer that represents the list $[a, c]$. However, Prolog cannot unify the subgoal with the only clause in the program and therefore replies "no". It would be possible to write a new program which also handles this kind of goal, but it is not possible to write a program which concatenates lists in constant time (which was the main objective for introducing difference lists in the first place).

Another problem with difference lists is due to the lack of occur-check in most Prolog systems. A program that specifies the property of being an empty list may look as follows:

$$empty(L - L).$$

Clearly, $[a|Y] - Y$ is not an empty list (since it denotes the list $[a]$). However, the goal $\leftarrow empty([a|Y] - Y)$ succeeds with Y bound to an infinite term.

Yet another problem with difference lists stems from the fact that "$-$" designates a *partial* function. Thus far nothing has been said about the meaning of terms such as $[a, b, c] - [d]$. For instance the goal:

$$\leftarrow append([a, b] - [c], [c] - [b], L)$$

succeeds with the answer $L = [a, b] - [b]$. Again such problems can be solved with additional computational efforts. The *append*/3 program may for instance be written as follows:

$$append(X - Y, Y - Z, X - Z) \leftarrow suffix(Y, X), suffix(Z, Y).$$

But this means that concatenation of lists becomes linear again.

Exercises

7.1 Write the following lists as terms with "**.**" (dot) as functor and [ ] representing the empty list:

$[a, b]$	$[a \mid [b, c]]$
$[a \mid b]$	$[a, b \mid [\,]]$
$[a, [b, c], d]$	$[[\,] \mid [\,]]$
$[a, b \mid X]$	$[a \mid [b, c \mid [\,]]]$

7.2 Define a binary relation *last*/2 between lists and their last elements using only the predicate *append*/3.

7.3 Define the membership-relation by means of the *append*/3-program.

7.4 Define a binary relation *length*/2 between lists and their lengths (i.e. the number of elements in them).

7.5 Define a binary relation *lshift*/2 between lists and the result of shifting them (circularly) one step to the left. For example, so that the goal:

$$\leftarrow lshift([a, b, c], X)$$

succeeds with the answer $X = [b, c, a]$.

7.6 Define a binary relation *rshift*/2 between lists and the result of shifting them (circularly) one step to the right. For example, so that the goal:

$$\leftarrow rshift([a, b, c], X)$$

succeeds with the answer $X = [c, a, b]$.

7.7 Define a binary relation *prefix*/2 between lists and all its prefixes. Hint: [ ], $[a]$ and $[a, b]$ are prefixes of the list $[a, b]$.

7.8 Define a binary relation *suffix*/2 between lists and all its suffixes. Hint: $[\,]$, $[b]$ and $[a, b]$ are suffixes of the list $[a, b]$.

7.9 Define a binary relation *sublist*/2 between lists and their sublists.

7.10 Implement the insert-sort algorithm for integers in Prolog — informally it can be formulated as follows:

> Given a list, remove its first element, sort the rest, and insert the first element in its appropriate place in the sorted list.

7.11 Implement the quick-sort algorithm for integers in Prolog — informally it can be formulated as follows:

> Given a list, split the list into two — one part containing elements less than a given element (e.g. the first element in the list) and one part containing elements greater than or equal to this element. Then sort the two lists and append the results.

7.12 Implement the merge-sort algorithm for integers in Prolog — informally it can be formulated as follows:

> Given a list, divide the list into two halves. Sort the halves and "merge" the two sorted lists.

7.13 A nondeterministic finite automaton (NFA) is a tuple $\langle S, \Sigma, T, s_0, F\rangle$ where:

- S is a finite set of *states*;
- Σ is a finite *input alphabet*;
- $T \subseteq S \times S \times \Sigma$ is a *transition relation*;
- $s_0 \in S$ is an *initial state*;
- $F \subseteq S$ is a set of *final states.*

A string $x_1 x_2 \dots x_n \in \Sigma^n$ is *accepted* by an NFA if there is a sequence:

$$\langle s_0, s_1, x_1\rangle, \langle s_1, s_2, x_2\rangle, \cdots, \langle s_{n-1}, s_n, x_n\rangle \in T^*$$

such that $s_n \in F$. An NFA is often depicted as a *transition diagram* whose nodes are states and where the transition relation is denoted by labelled edges between nodes. The final states are indicated by double circles. Define the NFA depicted in Figure 7.5 as a logic program (let 1 be the initial state). Use the program to check if strings (represented by lists of a's and b's) are accepted.

7.14 Informally speaking, a *Turing machine* consists of an infinite tape divided into slots which may be read from/written into by a *tape-head.* In each slot there is exactly one of two symbols called "blank" (denoted by 0) and "nonblank" (denoted by 1). Initially the tape is almost blank — the number of slots from the "leftmost" to the "rightmost" nonblank of the tape is finite. This sequence (string) will be referred to as the *content of the tape.* The machine is always

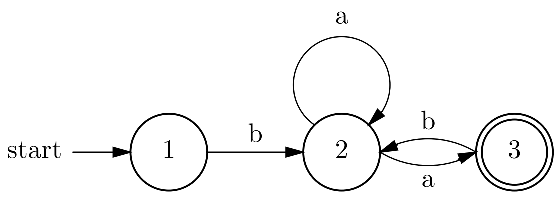


Figure 7.5: Transition diagram for NFA

in one of a finite number of *states*, some of which are called *final* and one of which is called *initial*.

The tape-head is situated at exactly one of the slots of the tape. Depending on the contents of this slot and the current state, the machine makes one of a number of possible *moves*:

- It writes 0 or 1 in the slot just read from, and
- changes state, and
- moves the tape-head one slot to the right or to the left.

This can be described through a function, which maps the current state and the symbol in the current slot of the tape into a triple which consists of a new state, the symbol written in the current slot and the direction of the move.

To start with the machine is in the initial state and the tape-head is situated at the "leftmost" nonblank of the tape. A string is said to be accepted by a Turing machine iff it is the initial contents of the tape and there is a finite sequence of moves that take the machine to one of its final states.

Write a Turing machine which accepts a string of n ($n > 0$) consecutive 1's and halts pointing to the leftmost nonblank in a sequence of $2 * n$ 1's. This illustrates another view of Turing machines, not only as language acceptors, but as a function which takes as input a natural number n represented by n consecutive 1's and produces some output represented by the consecutive 1's starting in the slot pointed to when the machine halts.

7.15 Write a program for multiplying matrices of integers. Obviously the program should succeed only if the number of columns of the first matrix equals the number of rows in the second. Then write a specialized program for multiplying matrices of fixed size.

7.16 Find some suitable representation of sets. Then define some standard operations on sets, like union, intersection, membership, set-difference.

7.17 Represent strings by difference lists. Define the property of being a palindrome.

7.18 Use the concept of difference list to implement the quick-sort algorithm in exercise 7.11.

7.19 Use difference lists to define queues. That is, a first-in-first-out stack. Write relations that describe the effects of adding new objects to and removing objects from queues.

7.20 Define the property of being a binary tree. Then define what it means for an element to be a member of a binary tree.

7.21 A binary tree is said to be *sorted* if for every node N in the tree, the nodes in the left subtree are all less than N and all nodes in the right subtree are greater than N. Define this by means of a definite program and then define relations for adding and deleting nodes in a sorted binary tree (so that it stays sorted).

7.22 Write a program for finding a minimal spanning tree in a weighted loop-free undirected connected graph.

Chapter 8

Amalgamating Object- and Meta-language

8.1 What is a Meta-language?

Generally speaking a *language* is a, usually infinite, collection of strings. For a given language L, some existing language has to be used to formulate and reason about the semantics and syntax of the language L. Sometimes a language is used to describe itself. This is often the case when reasoning about English (or any natural language for that matter). The following are examples of such sentences:

This sentence consists of fifty two letters excluding blanks
Rabbit is a noun

A language which is used to reason about another language (or possibly itself) is called a *meta-language* and the language reasoned about is called the *object-language*. Obviously the meta- and object-languages do not have to be natural languages — they can be any, more or less, formal languages. In the previous chapters English was used to describe the language of predicate logic and in forthcoming chapters logic will be used to formulate rules describing a subset of natural language. In this chapter special interest is paid to the use of logic programs as a meta-language for describing other languages, and in particular the use of logic programs to describe logic programming.

In computer science the word "meta" is used extensively in different contexts, and its meaning is not always very clear. In this book the word "meta-program" will be used for a program that manipulates other programs. With this rather broad definition, programs like compilers, interpreters, debuggers and even language specific editors are considered to be meta-programs. The main topic of this chapter is interpreters but in subsequent chapters other applications are considered.

An interpreter describes the operational semantics of a programming language — in a very general sense it takes as input a program and some data, and produces as output some (new) data. The language used to implement the interpreter is the meta-language and the language being interpreted is the object-language. In this chapter some alternative interpreters for “pure” Prolog (i.e. Prolog without built-in predicates) written in Prolog will be presented. Such interpreters are commonly called *meta-circular interpreters*[1] or simply *self-interpreters*.

When writing an interpreter for a language one of the most vital decisions is how to represent programs and data-objects of the object-language in the meta-language. In this chapter some advantages and drawbacks of various representations are discussed. First the object-language will be represented by ground terms of the meta-language but later we show a technique for integration of the object-language with the meta-language. This may lead to serious ambiguities since it is not always possible to tell whether a statement, or part of a statement, belongs to the object-language or the meta-language. For example, consider the following sentence of natural language:

Stockholm is a nine-letter word

If this is a sentence of a meta-language describing the form of words in an object language then “Stockholm” denotes itself, and the sentence is a true statement. However, if “Stockholm” denotes the capital, the statement is false.

8.2 Ground Representation

As discussed in previous chapters logic describes relations between individuals of some universe (domain). Clearly nothing prevents us from describing the world that consists of terms and formulas. For this, all constants, variables, compound terms and formulas of the object-language should be represented uniquely as terms of the meta-language. One possible representation of definite programs may look as follows:

- each constant of the object-language is represented by a unique constant of the meta-language;
- each variable of the object-language is represented by a unique constant of the meta-language;
- each n-ary functor of the object-language is represented by a unique n-ary functor of the meta-language;
- each n-ary predicate symbol of the object-language is represented by a unique n-ary functor of the meta-language;
- each connective of the object-language is represented by a unique functor of the meta-language (with the corresponding arity).

The representation can be given in terms of a bijective mapping ϕ from constants, variables, functors, predicate symbols and connectives of the object-language to a

[1]The term *meta*-interpreter is often used as an abbreviation for meta-circular interpreter. However, the word is somewhat misleading since *any* interpreter is a meta-interpreter.

subset of the constants and functors of the meta-language. The meta-language may of course also contain other symbols — in particular some predicate symbols. However, leaving them aside for the moment, the domain of the intended interpretation $\Im$ of the meta-language consists of terms and formulas of the object-language. Now the meaning of the constants and functors introduced above are given by the bijection ϕ, or rather by its inverse (ϕ^{-1}) as follows:

- The meaning $c_\Im$ of a constant c of the meta-language is the constant or variable $\phi^{-1}(c)$ of the object-language.
- The meaning $f_\Im$ of an n-ary functor f of the meta-language is an n-ary function which maps:
  - (i) the terms $t_1, \ldots, t_n$ to the term $\phi^{-1}(f)(t_1, \ldots, t_n)$ if $\phi^{-1}(f)$ is a functor of the object-language;
  - (ii) the terms $t_1, \ldots, t_n$ to the atom $\phi^{-1}(f)(t_1, \ldots, t_n)$ if $\phi^{-1}(f)$ is a predicate letter of the object-language;
  - (iii) the formulas $f_1, \ldots, f_n$ to the formula $\phi^{-1}(f)(f_1, \ldots, f_n)$ if $\phi^{-1}(f)$ is a connective of the object-language.

Example 8.1 Take as object-language a language with an alphabet consisting of the constants a and b, the predicate letters $p/1$ and $q/2$, the connectives $\wedge$ and $\leftarrow$ and an infinite but enumerable set of variables including X.

Now assume that the meta-language contains the constants a, b and x and the functors $p/1$, $q/2$, $and/2$ and $if/2$ with the obvious intended interpretation. Then the meaning of the meta-language term:

$$if(p(x), and(q(x, a), p(b)))$$

is the object-language formula:

$$p(X) \leftarrow q(X, a) \wedge p(b)$$

∎

It should be noted that the discussion above avoids some important considerations. In particular, the interpretation of meta-language functors consists of partial, not total, functions. For instance, nothing said about the meaning of "ill-formed" terms such as:

$$if(p(and(a, b)), q(if(a, b)))$$

Predicate logic requires that functors are interpreted as total functions and formally such terms must also have some kind of meaning. There are different methods to deal with the problem. However, they are not discussed here.

The coding of object-language expressions given above is of course only one possibility. In fact, in what follows we will not commit ourselves to any particular representation of the object-language. Instead $\ulcorner A \urcorner$ will be used to denote some particular representation of the object-language construction A. This makes it possible to present the idea without discussing technical details of a particular representation.

It is now possible to describe relations between terms and formulas of the object-language in the meta-language. In particular, our intention is to describe SLD-resolution and SLD-derivations. The first relation considered is that between consecutive goals, G_i and G_{i+1} $(i \geq 0)$, in an SLD-derivation:

$$G_0 \overset{C_0}{\rightsquigarrow} G_1 \cdots G_i \overset{C_i}{\rightsquigarrow} G_{i+1}$$

The relationship between two such goals can be expressed through the following "inference rule" discussed in Chapter 3:

$$\frac{\leftarrow A_1, \dots, A_{i-1}, A_i, A_{i+1}, \dots, A_n \qquad B_0 \leftarrow B_1, \dots, B_m}{\leftarrow (A_1, \dots, A_{i-1}, B_1, \dots, B_m, A_{i+1}, \dots, A_n)\theta}$$

where θ is the mgu of the two atoms A_i and B_0 and where $B_0 \leftarrow B_1, \dots, B_m$ is a renamed clause from the program. The relation between the two goals in the rule can be formulated as the following definite clause:

Example 8.2

$$\begin{array}{l}
\mathit{step}(\mathit{Goal}, \mathit{NewGoal}) \leftarrow \\
\qquad \mathit{select}(\mathit{Goal}, \mathit{Left}, \mathit{Selected}, \mathit{Right}), \\
\qquad \mathit{clause}(C), \\
\qquad \mathit{rename}(C, \mathit{Goal}, \mathit{Head}, \mathit{Body}), \\
\qquad \mathit{unify}(\mathit{Head}, \mathit{Selected}, \mathit{Mgu}), \\
\qquad \mathit{combine}(\mathit{Left}, \mathit{Body}, \mathit{Right}, \mathit{TmpGoal}), \\
\qquad \mathit{apply}(\mathit{Mgu}, \mathit{TmpGoal}, \mathit{NewGoal}).
\end{array}$$

∎

Informally the intended interpretation of the used predicate symbols is as follows:

- $\mathit{select}(A, B, C, D)$ describes the relation between a goal A and the selected subgoal C in A. B and D are the conjunctions of subgoals to the left and right of the selected one (obviously if A contains only C then B and D are empty);
- $\mathit{clause}(A)$ describes the property of being a clause in the object-language program;
- $\mathit{rename}(A, B, C, D)$ describes the relation between four formulas such that C and D are renamed variants of the head and body of A containing no variables in common with B;
- $\mathit{unify}(A, B, C)$ describes the relation between two atoms, A and B, and their mgu C;
- $\mathit{combine}(A, B, C, D)$ describes the relation between a goal D and three conjunctions which, when combined, form the conjunction D;
- $\mathit{apply}(A, B, C)$ describes the relation between a substitution A and two goals such that A applied to B yields C.

Some alternative approaches for representing the object-level program have been suggested in the literature. The most general approach is to explicitly carry the program around using an extra argument. Here a more pragmatic approach is employed where each clause C of the object-program is stored as a fact, $clause(\ulcorner C \urcorner)$, of the meta-language. For instance, $clause/1$ may consist of the following four facts:

Example 8.3

$$\begin{array}{l} clause(\ulcorner grandparent(X,Z) \leftarrow parent(X,Y), parent(Y,Z)\urcorner). \\ clause(\ulcorner parent(X,Y) \leftarrow father(X,Y)\urcorner). \\ clause(\ulcorner father(adam, bill)\urcorner). \\ clause(\ulcorner father(bill, cathy)\urcorner). \end{array}$$

∎

The task of completing $select/4$ and the other undefined relations in Example 8.2 is left as an exercise for the reader.

The relation $derivation/2$ between two goals G_0 and G_i of an SLD-derivation (i.e. if there is a derivation whose initial goal is G_0 and final goal is G_i) can be described as the reflexive and transitive closure of the inference rule above. Thus, an SLD-derivation can be described by means of the following two clauses:

Example 8.4

$$\begin{array}{l} derivation(G, G). \\ derivation(G0, G2) \leftarrow \\ \qquad step(G0, G1), \\ \qquad derivation(G1, G2). \end{array}$$

∎

If all undefined relations were properly defined it would be possible to give the goal clause:

$$\leftarrow derivation(\ulcorner \leftarrow grandparent(adam, X)\urcorner, \ulcorner \Box \urcorner).$$

(where $\ulcorner \Box \urcorner$ denotes the coding of the empty goal). This corresponds to the query "Is there a derivation from the object-language goal $\leftarrow grandparent(adam, X)$ to the empty goal?". The meta-language goal is reduced to:

$$\leftarrow step(\ulcorner \leftarrow grandparent(adam, X)\urcorner, G), derivation(G, \ulcorner \Box \urcorner).$$

The leftmost subgoal is then satisfied with G bound to the representation of the object-language goal $\leftarrow parent(adam, Y), parent(Y, X)$ yielding:

$$\leftarrow derivation(\ulcorner \leftarrow parent(adam, Y), parent(Y, X)\urcorner, \ulcorner \Box \urcorner).$$

Again this is unified with the second clause of $derivation/2$:

$$\leftarrow step(\ulcorner \leftarrow parent(adam, Y), parent(Y, X)\urcorner, G), derivation(G, \ulcorner \Box \urcorner).$$

The computation then proceeds until the goal:

$$\leftarrow derivation(\ulcorner \Box \urcorner, \ulcorner \Box \urcorner).$$

is obtained. This unifies with the fact of *derivation*/2 and Prolog produces the answer "yes". Note that the answer obtained to the initial goal is only "yes" since the goal contains no variables (only object-language variables which are represented as ground terms in the meta-language). The modified version of the program which also returns a substitution may look as follows:

Example 8.5

$$
\begin{array}{l}
\mathit{derivation}(G, G, \ulcorner\epsilon\urcorner).\\
\mathit{derivation}(G0, G2, S0) \leftarrow\\
\qquad \mathit{step}(G0, G1, \mathit{Mgu}),\\
\qquad \mathit{derivation}(G1, G2, S1),\\
\qquad \mathit{compose}(\mathit{Mgu}, S1, S0).\\
\\
\mathit{step}(\mathit{Goal}, \mathit{NewGoal}, \mathit{Mgu}) \leftarrow\\
\qquad \mathit{select}(\mathit{Goal}, \mathit{Left}, \mathit{Selected}, \mathit{Right}),\\
\qquad \mathit{clause}(C),\\
\qquad \mathit{rename}(C, \mathit{Goal}, \mathit{Head}, \mathit{Body}),\\
\qquad \mathit{unify}(\mathit{Head}, \mathit{Selected}, \mathit{Mgu}),\\
\qquad \mathit{combine}(\mathit{Left}, \mathit{Body}, \mathit{Right}, \mathit{TmpGoal}),\\
\qquad \mathit{apply}(\mathit{Mgu}, \mathit{TmpGoal}, \mathit{NewGoal}).
\end{array}
$$

∎

Now what is the point in having self-interpreters? Surely it must be better to let an interpreter run the object-language directly, instead of letting the interpreter run a self-interpreter which runs the object-program? The answer is that self-interpreters provide great flexibility for modifying the behaviour of the logical machinery. With a self-interpreter it is possible to write:

- interpreters which employ alternative search strategies — for instance, to avoid using Prolog's depth-first search, an interpreter which uses a breadth-first strategy may be written in Prolog and run on the underlying machine which uses depth-first search;
- debugging facilities — for instance, interpreters which emit traces or collect runtime statistics while running the program;
- interpreters which allow execution of a program which changes during its own execution — desirable in many A.I. applications or in the case of database updates;
- interpreters which collect the actual proof of a satisfied goal — something which is of utmost importance in expert-systems applications (see the next chapter);
- interpreters for nonstandard logics or "logic-like" languages — this includes fuzzy logic, nonmonotonic logic, modal logic, context-free grammars and Definite Clause Grammars (see Chapter 10).

Applications like compilers and language specific editors have already been mentioned. Furthermore, meta-circular interpreters play a very important role in the area of program *transformation*, *verification* and *synthesis* — topics which are outside the scope of this book.

8.3 Nonground Representation

Although the interpreters in the previous section are relatively clear and concise, they suffer severely from efficiency problems. The inefficiency is mainly due to the representation of object-language variables by constants in the meta-language. As a consequence, rather complicated definitions of *renaming*, *unification* and *application* of substitutions to terms/formulas are needed. In this section a less logical and more pragmatic approach is employed resulting in an extremely short interpreter. The idea is to represent object-language variables by meta-language variables — the whole approach seems straightforward at first sight, but it has severe semantical consequences some of which are raised below.

In addition to representing variables of the object-language by variables of the meta-language, an object-language clause of the form:

$$A_0 \leftarrow A_1, \ldots, A_n$$

will be represented in the meta-language by the term:[2]

$$\begin{array}{ll} A_0 \ \mathit{if} \ A_1 \ \mathit{and} \ \ldots \ \mathit{and} \ A_n & \text{when } n \geq 1 \\ A_0 \ \mathit{if} \ \mathit{true} & \text{when } n = 0 \end{array}$$

Similarly, a goal $\leftarrow A_1, \ldots, A_n$ of the object-language will be represented by the term A_1 *and* ... *and* A_n when $n \geq 1$ and *true* when $n = 0$.

The object-language program in Example 8.3 is thus represented by the following collection of meta-language facts:

Example 8.6

$$\begin{array}{l} \mathit{clause}(\mathit{grandparent}(X, Z) \ \mathit{if} \ \mathit{parent}(X, Y) \ \mathit{and} \ \mathit{parent}(Y, Z)). \\ \mathit{clause}(\mathit{parent}(X, Y) \ \mathit{if} \ \mathit{father}(X, Y)). \\ \mathit{clause}(\mathit{father}(\mathit{adam}, \mathit{bill}) \ \mathit{if} \ \mathit{true}). \\ \mathit{clause}(\mathit{father}(\mathit{bill}, \mathit{cathy}) \ \mathit{if} \ \mathit{true}). \end{array}$$

∎

The interpreter considered in this section simply looks as follows:

Example 8.7

$$\begin{array}{l} \mathit{solve}(\mathit{true}). \\ \mathit{solve}(X \ \mathit{and} \ Y) \leftarrow \mathit{solve}(X), \mathit{solve}(Y). \\ \mathit{solve}(X) \leftarrow \mathit{clause}(X \ \mathit{if} \ Y), \mathit{solve}(Y). \end{array}$$

∎

In what follows we will describe how operations which had to be explicitly spelled out in the earlier interpreters, are now performed automatically on the meta-level. Consider first *unification*:

Let $\leftarrow \mathit{parent}(\mathit{adam}, \mathit{bill})$ be an object-language goal. In order to find a refutation of this goal we may instead consider the following meta-language goal:

[2]Here it is assumed that *and*/2 is a right-associative functor that binds stronger than *if*/2. That is, the expression *a if b and c and d* is identical to the term *if*(*a*, *and*(*b*, *and*(*c*, *d*))).

$$\leftarrow solve(parent(adam, bill)).$$

The subgoal obviously unifies only with the head of the third clause in Example 8.7. The goal thus reduces to the new goal:

$$\leftarrow clause(parent(adam, bill)\ if\ Y_0), solve(Y_0).$$

Now the leftmost subgoal unifies with the second clause of Example 8.6, resulting in the new goal:

$$\leftarrow solve(father(adam, bill)).$$

This means that the unification of the object-language atom $parent(adam, bill)$ and the head of the object-language clause $parent(X, Y) \leftarrow father(X, Y)$ is performed automatically on the meta-level. There is no need to provide a definition of unification of object-language formulas as was needed in Example 8.2.

The same effect is achieved when dealing with *renaming* of variables in object-language clauses. Consider the goal:

$$\leftarrow solve(parent(X, bill)).$$

This reduces to the new goal:

$$\leftarrow clause(parent(X, bill)\ if\ Y_0), solve(Y_0).$$

Note that the goal and the second clause of Example 8.6 both contain the variable X. However, the variables of the rule are automatically renamed on the meta-level so that the next goal becomes:

$$\leftarrow solve(father(X, bill)).$$

Finally, *application* of a substitution to a goal is considered. The goal:

$$\leftarrow clause(father(adam, X)\ and\ father(X, Y)).$$

which represents the object-language goal $\leftarrow father(adam, X), father(X, Y)$ is resolved with the second clause of Example 8.7 and the goal:

$$\leftarrow solve(father(adam, X)), solve(father(X, Y))$$

is obtained. This goal is resolved with the third clause of Example 8.7 yielding:

$$\leftarrow clause(father(adam, X)\ if\ Y_1), solve(Y_1), solve(father(X, Y)).$$

Now the leftmost subgoal unifies with the clause:

$$clause(father(adam, bill)\ if\ true).$$

and the mgu $\{X/bill, Y_1/true\}$ is obtained. This substitution is used to construct the new goal:

$$\leftarrow solve(true), solve(father(bill, Y)).$$

Note that the mgu obtained in this step contains the "object-language substitution" $\{X/bill\}$ and that there is no need to apply it explicitly to the subgoal $father(X, Y)$ as was the case in the previous section.

Hence, by using the interpreter in Example 8.7 instead of that in the previous section, three of the most laborious operations (unification, renaming and application) are no longer explicitly needed. They are of course still performed but they now take place on the meta-level.

Yet another advantage of the interpreter in Example 8.7 is that there is no need to explicitly handle substitutions of the object program. By giving the goal:

$$\leftarrow solve(grandparent(adam, X)).$$

Prolog gives the answer $X = cathy$ since the object-language variable X is now represented as a meta-language variable. In the previous section object-language variables were represented by ground terms and to produce a computed answer substitution it was necessary to explicitly represent such substitutions.

Although the program in Example 8.7 works quite nicely, its declarative reading is far from clear. Its simplicity is due to the representation of object-language variables as meta-language variables. But this also introduces problems as pointed out by Hill and Lloyd (1988a). Namely, the variables in *solve*/1 and in *clause*/1 range over different domains — the variables in *solve*/1 range over formulas of the object-language whereas the variables in *clause*/1 range over individuals of the intended interpretation of the object-language program (intuitively the persons Adam, Bill and Cathy and possibly some others). The problem can, to some extent, be solved by using a *typed* language instead of standard predicate logic. However, this discussion is outside the scope of this book.

The interpreter in Example 8.7 can be used for, what is sometimes called, *pure* Prolog. This means that the object-program and goal are not allowed to contain constructs like negation. However, the interpreter may easily be extended to also take proper care of negation by including the rule:

$$solve(not\ X) \leftarrow not\ solve(X).$$

Similar rules can be added for most built-in predicates of Prolog. One exception is cut (!), which is difficult to incorporate into the self-interpreter above. However, with some effort it can be done (see e.g. O'Keefe (1990)).

8.4 The Built-in Predicate *clause*/2

To better support meta-programming, the Prolog standard provides a number of so called *built-in predicates*. In this section and the following two, some of them are briefly discussed. For a complete description of the built-in predicates the reader should consult his/her own Prolog user's manual or the ISO Prolog standard (1995).

For the interpreter in Example 8.7 to work, the object-program has to be stored as facts of the form $clause(\ulcorner C \urcorner)$, where $\ulcorner C \urcorner$ is the representation of an object-language clause. The built-in predicate *clause*/2 allows the object-program to be stored "directly as a meta-program". The effect of executing *clause*/2 can be described by the following "inference rule" (assuming that Prolog's computation rule is used and that t_1 and t_2 are terms):

$$\frac{\leftarrow clause(t_1, t_2), A_2, \ldots, A_m \qquad B_0 \leftarrow B_1, \ldots, B_n}{\leftarrow (A_2, \ldots, A_m)\theta}$$

where $B_0 \leftarrow B_1, \ldots, B_n$ is a (renamed) program clause and θ is an mgu of $clause(t_1, t_2)$ and $clause(B_0, (B_1, \ldots, B_n))$. Here comma is treated as a binary functor which is right-associative. That is, the expression (a, b, c) is the same term as $','(a, ','(b, c))$. For uniformity, a fact A, is treated as a rule of the form $A \leftarrow true$.

Note that there may be more than one clause which unifies with the arguments of $clause(t_1, t_2)$. Hence, there may be several possible derivations. For instance, consider the program:

$$\begin{array}{l} father(X, Y) \leftarrow parent(X, Y), male(X). \\ father(adam, bill). \end{array}$$

In case of the goal $\leftarrow clause(father(X, Y), Z)$ Prolog replies with two answers. The first answer binds X to X_0, Y to Y_0 and Z bound to $(parent(X_0, Y_0), male(X_0))$. The second answer binds X to *adam*, Y to *bill* and Z to *true*.

By using *clause*/2 it is possible to re-write Examples 8.6 and 8.7 as follows:

$$\begin{array}{l} solve(true). \\ solve((X, Y)) \leftarrow solve(X), solve(Y). \\ solve(X) \leftarrow clause(X, Y), solve(Y). \\ \\ grandparent(X, Z) \leftarrow parent(X, Y), parent(Y, Z). \\ \quad\vdots \end{array}$$

Note that it is no longer possible to distinguish the meta-language from the the object language.

The use of *clause*/2 is normally restricted in that the first argument of *clause*/2 must not be a variable at the time when the subgoal is selected. This is, for instance, stipulated in the ISO Prolog standard. Thus, the goal $\leftarrow solve(X)$ results in a run-time error in most Prolog systems.

8.5 The Built-in Predicates *assert*{*a,z*}/1

The Prolog standard also provides some built-in predicates that are used to modify the program during execution of a goal. For instance, the built-in predicates *asserta*/1 and *assertz*/1 are used to dynamically add new clauses to the program. The only difference between the two is that *asserta*/1 adds its argument textually first in the definition of a predicate whereas *assertz*/1 adds it argument textually at the end. We use *assert*/1 to stand for either of the two. From a proof-theoretic point of view the logical meaning of *assert*/1 can be described as follows (assuming that t is not a variable and that Prolog's computation rule is used):

$$\frac{\leftarrow assert(t), A_2, \ldots, A_n}{\leftarrow A_2, \ldots, A_n}$$

In other words, *assert*/1 can be interpreted as something which is always true when selected. However, the main effect of *assert*/1 is the addition of t to the database of clauses. Of course, t should be a well-formed "clause" not to cause a run-time error.[3]

Consider the trivial program:

$$parent(adam, bill).$$

Presented with the goal:

$$\leftarrow assertz(parent(adam, beth)), parent(adam, X).$$

Prolog replies with two answers — $X = bill$ and $X = beth$. Prolog first adds the clause $parent(adam, beth)$ to the program database and then tries to satisfy the second subgoal which now has two solutions. Changes made to Prolog's database by *assert*/1 are permanent. That is, they are not undone on backtracking. Moreover, it is possible to assert the same clause several times.

Unfortunately, the effect of using *assert*/1 is not always very clear. For instance, if the order among the subgoals in the previous goal is changed into:

$$\leftarrow parent(adam, X), assertz(parent(adam, beth)).$$

some Prolog systems would return the single answer $X = bill$, since when the call to *parent*/2 is made, Prolog records that its definition contains only one clause. Thus, when backtracking takes place the new clause added to the definition of *parent*/2 remains invisible to the call. This is in accordance with the ISO Prolog standard (1995). However, some (old) Prolog systems would return infinitely many answers. First $X = bill$, and thereafter an infinite repetition of the answer $X = beth$. This happens if the implementation does not "freeze" the definition of a predicate when a call is made to the predicate. Every time a solution is found to the leftmost subgoal a new copy of the clause $parent(adam, beth)$ is added to the definition of *parent*/2. Hence, there will always be one more clause for the leftmost subgoal to backtrack to. A similar problem occurs in connection with the clause:

$$void \leftarrow assertz(void), fail.$$

In some Prolog implementations the goal $\leftarrow void$ would succeed, whereas in others it would fail. However, in both cases the resulting program is:

$$\begin{array}{l} void \leftarrow assertz(void), fail. \\ void. \end{array}$$

This suggests that *assert*/1 should be used with great care. Just like cut, *assert*/1 is often abused in misguided attempts to improve the efficiency of programs. However, there are cases when usage of *assert*/1 can be motivated. For instance, if a subgoal is solved, the result can be stored in the database as a *lemma*. Afterwards the same subgoal can be solved in a single derivation step. This kind of usage does not cause any declarative problems since the lemma does not add to, or delete information from the program.

[3]The reason for quoting the word clause is that the argument of *assert*/1 formally is a term. However, in most Prolog systems clauses are handled just as if they were terms. That is, the logical connectives "$\leftarrow$" and "," are allowed to be used also as functors.

8.6 The Built-in Predicate *retract*/1

The built-in predicate *retract*/1 is used to delete clauses dynamically from Prolog's database during the execution of a goal. The logical meaning of *retract*/1 is similar to that of *clause*/2. It can be described by the inference rule:

$$\frac{\leftarrow \mathit{retract}((s \leftarrow t)), A_2, \ldots, A_m \qquad B_0 \leftarrow B_1, \ldots, B_n}{\leftarrow (A_2, \ldots, A_m)\theta}$$

where $B_0 \leftarrow B_1, \ldots, B_n$ is a renamed program clause such that $s \leftarrow t$ and $B_0 \leftarrow B_1, \ldots, B_n$ have an mgu θ. Like the case with *clause*/2 there may be more than one clause which unifies with $(s \leftarrow t)$. Hence, several derivations are possible. For uniformity, and by analogy to *clause*/2, a fact may be treated as a rule whose body consists of the literal *true*.

As a side-effect *retract*/1 removes the clause $B_0 \leftarrow B_1, \ldots, B_n$ from Prolog's internal database. The effect is permanent — that is, the clause is not restored when backtracking takes place. For instance, consider the Prolog program:

$$\begin{array}{l} \mathit{parent}(\mathit{adam}, \mathit{bill}). \\ \mathit{parent}(\mathit{adam}, \mathit{beth}). \\ \mathit{parent}(\mathit{bill}, \mathit{cathy}). \end{array}$$

In reply to the goal:

$$\leftarrow \mathit{retract}(\mathit{parent}(\mathit{adam}, X) \leftarrow \mathit{true}).$$

Prolog replies with two answers — $X = \mathit{bill}$ and $X = \mathit{beth}$. Then execution terminates and all that is left of the program is the clause:

$$\mathit{parent}(\mathit{bill}, \mathit{cathy}).$$

In most Prolog implementations (and according to the ISO standard) it is required that the argument of *retract*/1 is not a variable and if it is of the form $s \leftarrow t$ that s is not a variable.

Like *assert*/1, usage of *retract*/1 is controversial and the effect of using it may diverge in different implementations. In general there are both cleaner and more efficient methods for solving problems than resorting to these two. For example, naive users of Prolog often use *assert*/1 and *retract*/1 to implement a form of global variables. This usually has two effects — the program becomes harder to understand and it runs slower since asserting new clauses to the program involve considerable amount of work and book-keeping. This often comes as a big surprise to people who are used to programming in imperative programming languages.

Exercises

8.1 Complete the self-interpreter described in Examples 8.2 – 8.4. Either by using the suggested representation of the object-language or invent your own representation.

8.2 Extend Example 8.7 so that execution is aborted when the number of resolution-steps for solving a subgoal becomes too large.

8.3 Modify Example 8.7 so that it uses the *depth-first-iterative-deepening* search strategy. The general idea is to explore all paths of length n from the root (where n is initially 1) using a depth-first strategy. Then the whole process is repeated after incrementing n by 1. Using this technique every refutation in an SLD-tree is eventually found.

8.4 Write a program for differentiating formulas consisting of natural numbers and some variables (but not Prolog ones!). Implement the program by defining some of the most common differentiation-rules. For instance the following ones:

$$\frac{\partial m}{\partial x} = 0 \qquad \frac{\partial x}{\partial x} = 1 \qquad \frac{\partial x^n}{\partial x} = n * x^{n-1}$$

$$\frac{\partial(f+g)}{\partial x} = \frac{\partial f}{\partial x} + \frac{\partial g}{\partial x} \qquad \frac{\partial(f*g)}{\partial x} = g * \frac{\partial f}{\partial x} + f * \frac{\partial g}{\partial x}$$

where $m \geq 0$ and $n > 0$.

8.5 Write a Prolog program which determines if a collection of formulas of propositional logic is satisfiable.

8.6 Consider a small imperative language given by the following abstract syntax:

$$\begin{array}{lcl}
I & ::= & x \mid y \mid z \mid \ldots \\
N & ::= & 0 \mid 1 \mid 2 \mid \ldots \\
B & ::= & \mathit{true} \mid \mathit{false} \mid E > E \mid \ldots \\
E & ::= & I \mid N \mid E + E \mid E - E \mid E * E \mid \ldots \\
C & ::= & \mathit{skip} \mid \mathit{assign}(I, E) \mid \mathit{if}(B, C, C) \mid \mathit{while}(B, C) \mid \mathit{seq}(C, C)
\end{array}$$

For instance, the factorial program:

$$y := 1;\ \mathbf{while}\ x > 0\ \mathbf{do}\ y := y * x;\ x := x - 1\ \mathbf{od}$$

is represented by the abstract syntax:

$$\mathit{seq}(\mathit{assign}(y, 1), \mathit{while}(x > 0, \mathit{seq}(\mathit{assign}(y, y * x), \mathit{assign}(x, x - 1))))$$

Moreover, a binding environment is a mapping from identifiers to values (in our case integers) which could be represented as a list of pairs of variables and integers.

Now write an interpreter *eval*/3 which relates an initial binding environment and a command C to the new binding environment obtained by "executing" C. For instance, if x maps to 5 in σ and c is the program above the goal $\leftarrow \mathit{eval}(\sigma, c, X)$ should result in a new binding environment where x maps to 0 and y maps to 120 (i.e. the factorial of 5).

8.7 Write a tiny pure-Lisp interpreter which incorporates some of the primitive functions (like CAR, CONS etc.) and allows the definition of new functions in terms of these.

Chapter 9

Logic and Expert Systems

9.1 Expert Systems

Roughly speaking, an expert system is a program that guides the user in the solution of some problem which normally requires intervention of a human expert in the field. Tasks which typically call for expert level knowledge include, for instance, *diagnosis*, *control* and *planning*. Diagnosis means trying to find the cause of some malfunction, e.g. the cause of an illness. In control-applications the aim is to prevent a system, such as an industrial process, from entering abnormal states. Planning, finally, means trying to find a sequence of state transitions ending in a specified final state via a sequence of intermediate states given an initial one. A typical problem consists in finding a plan which assembles a collection of parts into a final product. This chapter illustrates the applicability of logic programming for expert systems and meta-level reasoning by a diagnosis example.

Usually an expert system exhibits a number of characteristics:

- It is divided into an *inference engine* and a *knowledge-base*. The knowledge-base contains rules which describe general knowledge about some problem domain. The inference engine is used to infer knowledge from the knowledge-base. Usually, the inference machine is generic in the sense that one can easily plug in a new knowledge-base without any major changes to the inference machine.
- It may contain rules which are subject to some *uncertainty*.
- The system often runs on modern workstations and much effort is put into the user interface and the dialogue with the user.
- It has the capability not only to infer new knowledge from existing knowledge, but also to explain how/why some conclusion was reached.
- It has support for incremental knowledge acquisition.

It is easy to see that the first point above coincides with the objectives of logic programming — namely to separate the logic component (*what* the problem is) from the control (*how* the problem should be solved). This can be expressed by the equation:

$$\text{Algorithm} = \text{Logic} + \text{Control}$$

That is, Kowalski's well-known paraphrase of Wirth's doctrine Program = Algorithm + Data Structure. In the spirit of Kowalski we could write:

$$\text{Expert System} = \text{Knowledge-base} + \text{Control} + \text{User Interface}$$

The last two terms are commonly called an expert-system *shell*.

The knowledge-base of an expert system typically consists of a set of so called *production rules* (or simply rules). Like definite clauses, they have a set of premises and a conclusion. Such rules say that whenever all the premises hold the conclusion also holds. A typical rule found in one of the earliest expert systems called MYCIN may look as follows:

IF	the stain of the organism is gram-positive
AND	the morphology of the organism is coccus
AND	the growth conformation of the organism is clumps
THEN	the identity of the organism is staphylococcus (0.7)

It is not very hard to express approximately the same knowledge in the form of a definite clause:

$$\begin{aligned}&\mathit{identity\_of\_organism}(\mathit{staphylococcus}) \leftarrow\\ &\qquad \mathit{stain\_of\_organism}(\mathit{gram\_positive}),\\ &\qquad \mathit{morphology\_of\_organism}(\mathit{coccus}),\\ &\qquad \mathit{growth\_conformation\_of\_organism}(\mathit{clumps}).\end{aligned}$$

The figure (0.7) given in the conclusion of the MYCIN-rule above is an example of uncertainty of the rule. It says that if the premises hold then the conclusion holds with probability 0.7. In the following we do not consider these figures of uncertainty but assume that they are always 1.0.

We consider an application which involves diagnosing starting problems of cars. The following two propositions seem to express general knowledge describing the cause of malfunctioning devices:

- if Y is a necessary component for X and Y is malfunctioning then X is also malfunctioning;
- if X exhibits a fault-symptom Z then either X is malfunctioning or there exists another malfunctioning component which is necessary for X.

In predicate logic this may be expressed as follows:

$$\forall X(\exists Y(\mathit{needs}(X,Y) \wedge \mathit{malfunctions}(Y)) \supset \mathit{malfunctions}(X))$$
$$\forall X, Z(\mathit{symptom}(Z,X) \supset (\mathit{malfunctions}(X) \vee \exists Y(\mathit{needs}(X,Y) \wedge \mathit{malfunctions}(Y))))$$

The first of these readily transforms into the definite clause:

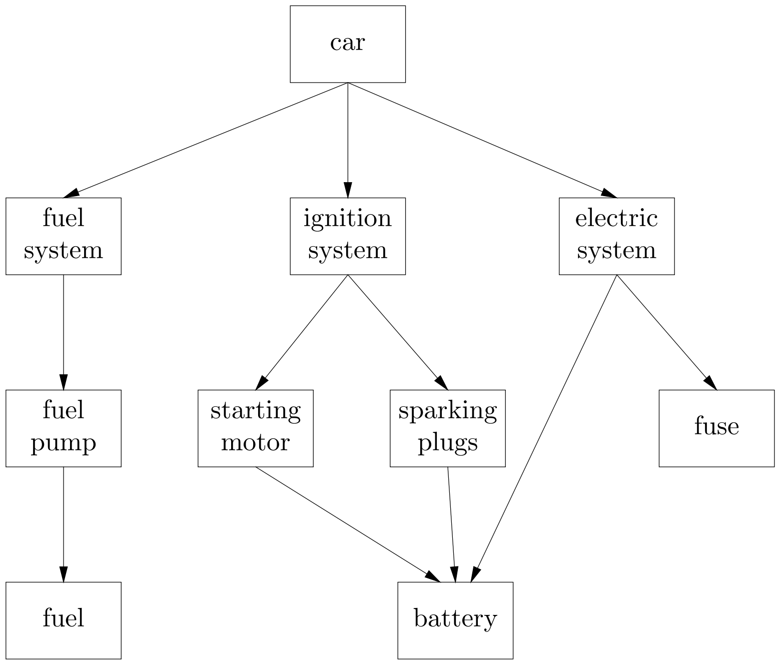


Figure 9.1: Taxonomy of a car-engine

$$\mathit{malfunctions}(X) \leftarrow \mathit{needs}(X, Y), \mathit{malfunctions}(Y).$$

However, in order to write the second formula as a definite clause some transformations are needed. First an auxiliary predicate symbol *indirect*/1 is introduced and defined as follows:

- X has an indirect fault if there exists a component which is necessary for X and which malfunctions.

Now the second formula can be replaced by the following two:

$$\forall X, Y(\mathit{symptom}(Y, X) \supset (\mathit{malfunctions}(X) \vee \mathit{indirect}(X)))$$
$$\forall X(\exists Y(\mathit{malfunctions}(Y) \wedge \mathit{needs}(X, Y)) \supset \mathit{indirect}(X))$$

These are straightforward to transform into general clauses:

$$\begin{aligned}&\mathit{malfunctions}(X) \leftarrow \mathit{symptom}(Y, X), \mathit{not}\ \mathit{indirect}(X).\\ &\mathit{indirect}(X) \leftarrow \mathit{needs}(X, Y), \mathit{malfunctions}(Y).\end{aligned}$$

We must now define the relation *needs*/2 which defines a hierarchy of components and dependencies between them. In this chapter a car-engine is abstracted into the components and dependencies in the taxonomy of Figure 9.1.

The relation described by the figure may be represented by the following definite program:

$needs(car, ignition\_system).$
$needs(car, fuel\_system).$
$needs(car, electric\_system).$
$needs(ignition\_system, starting\_motor).$
$needs(ignition\_system, sparking\_plugs).$
$needs(electric\_system, fuse).$
$needs(electric\_system, battery).$
$needs(fuel\_system, fuel\_pump).$
$needs(sparking\_plugs, battery).$
$needs(starting\_motor, battery).$
$needs(fuel\_pump, fuel).$

Finally the predicate *symptom*/2 which describes the symptoms of a car (or rather parts of the car) should be defined. However, the symptoms exhibited by a specific car depend on the particular car in a specific moment of time. The description of the symptoms of the car should therefore be added to the database when diagnosing the cause of malfunction of that particular car. How to cope with this is described below.

As shown above, the knowledge-base of an expert system can be described as a set of definite or general clauses. What about the inference engine?

The inference engine is used to infer new knowledge from existing knowledge. This can be done by using two different strategies — (1) either start from what is already known and infer new knowledge from this, or (2) start from the conclusion to be proved and reason backwards until the conclusion depends on what is already known. These methods are called *forward-* and *backward-chaining* respectively. Clearly, SLD-resolution is an example of a backward-chaining proof procedure. There are expert systems which rely on forward-chaining or a mixture of the two, but there are also expert systems which use backward-chaining only. MYCIN is an example of such an expert system.

We have established the close relationship between, on the one hand, the knowledge-base of an expert system and the set of clauses in logic programming and, on the other hand, the inference engines used in some expert systems and SLD-resolution. So what is the main difference between expert systems and logic programming?

Apart from the probabilities of rules and the user interface, an expert system differs from a logic program in the sense that its knowledge-base is usually *incomplete*. As explained above, the knowledge-base only contains general knowledge concerning different faults and symptoms. It does not contain information about the specific symptoms of a particular individual. This information has to be added to its knowledge-base whilst diagnosing the individual. That is, while inferring what fault the individual is suffering from, the inference engine asks questions which have to be filled in, in order to complete the knowledge-base. Thus, given a general description P of the world and a symptom or an observation F one can say that the aim of the expert system is to find a cause Δ such that $P \cup \Delta \vdash F$. This problem is commonly known under the name *abduction*.

Another major distinction between logic programming and expert systems is that expert systems have the capability to *explain* their conclusions. If an expert system draws a conclusion concerning the health of a patient it is likely that the doctor (or the patient) wants to know how the system came to that conclusion. Most Prolog systems are not automatically equipped with such a mechanism.

So clearly the knowledge-base may be described as a Prolog program but the Prolog inference engine does not satisfy the requirement needed in an expert system. In order to remedy this we are going to build a new inference engine based on the self-interpreter in Example 8.7 to provide these missing features.

9.2 Collecting Proofs

The first refinement made to the program in Example 8.7 is to add the capability of collecting the proof representing the refutation of a goal. As described in Section 3.3 a refutation may be represented as a *proof*- or *derivation-tree*. Referring to Example 8.7 we see that there are three types of goals:

- the empty goal (represented by the constant *true*);
- compound goals of the form X *and* Y;
- goals consisting of a single literal.

Now in the case of the goal *true* we can simply return the term *void* which represents an empty proof. Furthermore, under the assumption that X has a proof Px and that Y has a proof Py, the term Px & Py will represent a proof of the goal represented by X *and* Y. Finally, the single literal X has a proof represented by the term *proof*(X, Py) if there is a clause instance X *if* Y and Y has a proof represented by Py. It is straightforward to convert this into the following definite program:

Example 9.1

$$\begin{array}{l} solve(true, void). \\ solve((X \ and \ Y), (Px \ \& \ Py)) \leftarrow solve(X, Px), solve(Y, Py). \\ solve(X, proof(X, Py)) \leftarrow clause(X \ if \ Y), solve(Y, Py). \end{array}$$

∎

Given the goal $\leftarrow solve(grandparent(X, Y), Z)$ and the database:

$$\begin{array}{l} clause(grandparent(X, Y) \ if \ parent(X, Z) \ and \ parent(Z, Y)). \\ clause(parent(X, Y) \ if \ father(X, Y)). \\ clause(father(adam, bill) \ if \ true). \\ clause(father(bill, carl) \ if \ true). \end{array}$$

Prolog not only finds a refutation and produces the answer $X = adam$, $Y = carl$ but also returns the term:

$$\begin{array}{l} Z = proof(grandparent(adam, carl), \\ \qquad\qquad proof(parent(adam, bill), \\ \qquad\qquad\qquad proof(father(adam, bill), void)) \\ \qquad\qquad \& \\ \qquad\qquad proof(parent(bill, carl), \\ \qquad\qquad\qquad proof(father(bill, carl), void)) \\ \qquad) \end{array}$$

which represents the refutation of the goal $\leftarrow grandparent(X, Y)$.

As seen, when proving a positive goal it is rather easy to collect its proof. The situation is more complicated when trying to solve goals containing negative literals. Since the negation-as-failure rule says that *not* X succeeds if X finitely fails (that is, has no refutation) there is no proof to return (except possibly some kind of meta-proof). A naive solution is to add the clause:

$$solve(not\ X, proof(not\ X, void)) \leftarrow not\ solve(X, T).$$

to the interpreter. A more satisfactory solution would be to collect the meta-proof which proves that X has no proof but writing an interpreter for doing this is rather complicated. The discussion concerning the "proof-collecting" interpreter is now temporarily abandoned, but is resumed after the following section which suggests a solution to the problem of the incomplete knowledge-base.

9.3 Query-the-user

As explained above, the knowledge-base of an expert system is only capable of handling general information valid for every individual. In the case of diagnosing an illness, the symptoms of a specific patient have to be collected during "run-time". This means that when the inference engine encounters certain predicates it should not look for the definition in the knowledge-base but instead *query the user* for information. In the example above the predicate *symptom*/2 is used for this purpose. Such an approach can readily be implemented by means of the interpreter in Example 8.7 if these special predicate symbols are known before-hand.[1]

Example 9.2

```
solve(true).
solve(X and Y) ←
        solve(X), solve(Y).
solve(symptom(X, Y)) ←
        confirm(X, Y).
solve(X) ←
        clause(X if Y), solve(Y).
```

where *confirm*/2 is defined as follows:

```
confirm(X, Y) ←
        write('Is the '),
        write(Y), tab(1), write(X), write('? '),
        read(yes).
```

∎

Now consider the following (trivial) knowledge-base:

[1]This and subsequent examples require the use of input and output. The Prolog standard provides several built-in predicates. Thus, *write*/1 outputs a term on the current output stream. Similarly *read*/1 inputs a term from the current input stream (a call succeeds if the term unifies with the argument of the call). *nl*/0 outputs a newline character and *tab*/1 a specified number of blanks on the current output stream. Strings of characters enclosed by single quotes are taken to be constants.

$clause(malfunctions(X)\ if\ possible\_fault(Y, X)\ and\ symptom(Y, X)).$
$clause(possible\_fault(flat, tyre)\ if\ true).$

When given the goal $\leftarrow solve(malfunctions(X))$ Prolog will print the question "Is the tyre flat?". If the user replies with "yes" the execution succeeds with answer $X = tyre$; if the user answers anything but "yes" (or a variable) the goal fails.

9.4 Fixing the Car (Extended Example)

Now the principle of implementation of expert systems in Prolog is illustrated by continuing the example discussed in Section 9.1. We do not claim that the result is even close to a real expert system, which is, of course, considerably much more complicated than the program described below.

The first step is to describe the knowledge-base as a collection of facts in the meta-language. The predicate symbol *if*/2 is used for that purpose; the first argument represents the head of a rule and the second the conjunction of premises. To avoid having to treat facts of the object-language separately, they will be written in the form *X if true*. The knowledge-base described in Section 9.1 can now be described as follows:

$malfunctions(X)\ if\ needs(X, Y)\ and\ malfunctions(Y).$
$malfunctions(X)\ if\ symptom(Y, X)\ and\ not\ indirect(X).$

$indirect(X)\ if\ needs(X, Y)\ and\ malfunctions(Y).$

$needs(car, ignition\_system)\ if\ true.$
$needs(car, fuel\_system)\ if\ true.$
$needs(car, electric\_system)\ if\ true.$
$needs(ignition\_system, starting\_motor)\ if\ true.$
$needs(ignition\_system, sparking\_plugs)\ if\ true.$
$needs(electric\_system, fuse)\ if\ true.$
$needs(electric\_system, battery)\ if\ true.$
$needs(fuel\_system, fuel\_pump)\ if\ true.$
$needs(sparking\_plugs, battery)\ if\ true.$
$needs(starting\_motor, battery)\ if\ true.$
$needs(fuel\_pump, fuel)\ if\ true.$

To construct an inference engine, the interpreters from Examples 9.1 and 9.2 are "joined" into the following one:

```
$solve(true, void).$
$solve(X\ and\ Y, Px\ \&\ Py) \leftarrow$
        $solve(X, Px), solve(Y, Py).$
$solve(not\ X, proof(not\ X, void)) \leftarrow$
        $not\ solve(X, P).$
$solve(symptom(X, Y), proof(symptom(X, Y), void)) \leftarrow$
        $confirm(X, Y).$
$solve(X, proof(X, Py)) \leftarrow$
        $(X\ if\ Y), solve(Y, Py).$
```

The program is assumed to interact with a mechanic, exploiting the query-the-user facility. Hence, some easily spotted misbehaviours are characterized:

- one of the sparking plugs does not produce a spark;
- the fuel gauge indicates an empty tank;
- the fuel pump does not feed any fuel;
- a fuse (say number 13) is broken;
- the battery voltage is less than 11 volts;
- the starting motor is silent.

These can be formulated using the predicate symbol *confirm*/2 which poses questions to the mechanic and succeeds if he replies "yes". It may be defined as follows:

$$\begin{array}{l} \mathit{confirm}(X, Y) \leftarrow \\ \qquad \mathit{nl}, \mathit{ask}(X, Y), \mathit{read}(\mathit{yes}). \end{array}$$

where $\mathit{ask}(X, Y)$ prints a query asking if Y exhibits the misbehaviour X and is defined as follows:

$$\begin{array}{l}
\mathit{ask}(\mathit{worn\_out}, \mathit{sparking\_plugs}) \leftarrow \\
\qquad \mathit{write}(\text{'Do any of the sparking plugs fail to produce a spark?'}). \\
\mathit{ask}(\mathit{out\_of}, \mathit{fuel}) \leftarrow \\
\qquad \mathit{write}(\text{'Does the fuel gauge indicate an empty tank?'}). \\
\mathit{ask}(\mathit{broken}, \mathit{fuel\_pump}) \leftarrow \\
\qquad \mathit{write}(\text{'Does the fuel pump fail to feed any fuel?'}). \\
\mathit{ask}(\mathit{broken}, \mathit{fuse}) \leftarrow \\
\qquad \mathit{write}(\text{'Is fuse number 13 broken?'}). \\
\mathit{ask}(\mathit{discharged}, \mathit{battery}) \leftarrow \\
\qquad \mathit{write}(\text{'Is the battery voltage less than 11 volts?'}). \\
\mathit{ask}(\mathit{broken}, \mathit{starting\_motor}) \leftarrow \\
\qquad \mathit{write}(\text{'Is the starting motor silent?'}).
\end{array}$$

This more or less completes the knowledge-base and inference engine of the expert system. However, the program suffers from operational problems which become evident when giving the goal $\leftarrow \mathit{solve}(\mathit{malfunctions}(\mathit{car}), \mathit{Proof})$. If the user replies "no" to the question "Is the battery voltage less than 11 volts?", the system immediately asks the same question again. This happens because (see Figure 9.1):

- the car needs the ignition system;
- the ignition-system needs the sparking plugs and the starting motor;
- both the sparking plugs and the starting motor need the battery.

To avoid having to answer the same question several times the system must remember what questions it has already posed and the answers to those questions. This can be achieved by asserting the question/answer to the Prolog database. Before posing a query, the system should look into this database to see if an answer to the question is already there. To implement this facility the predicate *confirm*/2 must be redefined. For instance, in the following way:

```
$confirm(X, Y) \leftarrow$
        $known(X, Y, true).$
$confirm(X, Y) \leftarrow$
        $not\ known(X, Y, Z), nl, ask(X, Y),$
        $read(A), remember(X, Y, A), A = yes.$

$remember(X, Y, yes) \leftarrow$
        $assertz(known(X, Y, true)).$
$remember(X, Y, no) \leftarrow$
        $assertz(known(X, Y, false)).$
```

Note that the second clause is an example of unsafe use of negation. Given a selected subgoal $confirm(a, b)$, the first clause is used (and the subgoal is solved) if the question triggered by a and b was previously confirmed by the user (that is, if $confirm(a, b, true)$ is solved); the second clause is used if the question triggered by a and b was never posed before. This will be the effect since the selected subgoal in the next goal will be the negative literal $not\ known(a, b, Z)$ which succeeds if *there is no* Z such that $known(a, b, Z)$. If neither of these two clauses apply (that is, if the question triggered by a and b has been posed but denied) the call to $confirm/2$ fails.

When calling the program with the goal:

$$\leftarrow solve(malfunctions(car), X).$$

the system first prints the query:

Is the battery voltage less than 11 volts ?

If the user answers “no” the system asks:

Is the starting motor silent?

Under the assumption that the user replies “no”, the next question posed by the system is:

Do any of the sparking plugs fail to produce a spark?

If the reply to this question is “yes” the computation stops with the (rather awkward) answer:

```
$X = proof(malfunctions(car),$
        $proof(needs(car, ignition\_system), void)$
         $\&$
        $proof(malfunctions(ignition\_system),$
            $proof(needs(ignition\_system, sparking\_plugs), void)$
             $\&$
            $proof(malfunctions(sparking\_plugs),$
                $proof(symptom(worn\_out, sparking\_plugs), void)$
                 $\&$
                $proof(not\ indirect(sparking\_plugs), void)$
                $)$
            $)$
       $)$
```

Needless to say the answer is rather difficult to overview and there is need for routines that display the proof in a readable form. Some alternative approaches are possible, for instance, by printing the proof as a tree. However such a program would require rather complicated graphics routines. Instead a rather crude approach is employed where the rule-instances that the proof consists of are printed. The "top-loop" of the printing routine looks as follows:

```
print_proof(void).
print_proof(X & Y) ←
        print_proof(X), nl, print_proof(Y).
print_proof(proof(X, void)) ←
        write(X), nl.
print_proof(proof(X, Y)) ←
        Y ≠ void, write(X), write(' BECAUSE'), nl,
        print_children(Y), nl, print_proof(Y).
```

That is, in case of the empty proof nothing is done. If the proof consists of two or more proofs the proofs are printed separately. If the proof is of the form X *if* Y then two possibilities emerge — if Y is the empty proof then X is printed. If $Y \neq void$ then X is printed followed by the word "BECAUSE" and the "top nodes of the constituents of Y". Finally the subproofs are printed.

The program for printing the top-nodes of proofs looks as follows:

```
print_children(proof(X, Y) & Z) ←
        tab(8), write(X), write(' AND'), nl, print_children(Z).
print_children(proof(X, Y)) ←
        tab(8), write(X), nl.
```

That is, if it is a compound proof then the top of one constituent is printed followed by the word "AND", in turn followed by the top-nodes of the remaining constituents. If the proof is a singleton then the top of that proof is printed.

To complete the program the following "driver"-routine is added:

```
expert ←
        abolish(known/3),
        solve(malfunctions(car), X),
        print_proof(X).
```

where *abolish*/1 is a built-in predicate which removes all clauses whose heads have the same predicate symbol and arity as the argument. When faced with the goal ← *expert* the following dialogue may appear (with user input in bold-face):

```
Is the battery voltage less than 11 volts? no.

Is the starting motor silent? no.

Do any of the sparking plugs fail to produce a spark? yes.

malfunctions(car) BECAUSE
        needs(car, ignition_system) AND
        malfunctions(ignition_system)

needs(car, ignition_system)

malfunctions(ignition_system) BECAUSE
        needs(ignition_system, sparking_plugs) AND
        malfunctions(sparking_plugs)

needs(ignition_system, sparking_plugs)

malfunctions(sparking_plugs) BECAUSE
        symptom(worn_out, sparking_plugs) AND
        not indirect(sparking_plugs)

symptom(worn_out, sparking_plugs)

not indirect(sparking_plugs)
```

In conclusion, the complete listing of this tiny expert-system shell and the particular knowledge-base for diagnosing starting problems of cars is depicted below. (Lines preceded by the symbol "%" are comments.)

```
% Top level routine
expert ←
        abolish(known/3),
        solve(malfunctions(car), X),
        print_proof(X).

solve(true, void).
solve(X and Y, Px & Py) ←
        solve(X, Px), solve(Y, Py).
solve(not X, proof(not X, void)) ←
        not solve(X, P).
solve(symptom(X, Y), proof(symptom(X, Y), void)) ←
        confirm(X, Y).
solve(X, proof(X, Py)) ←
        (X if Y), solve(Y, Py).
```

```
% Query-the-user
confirm(X, Y) ←
        known(X, Y, true).
confirm(X, Y) ←
        not known(X, Y, Z), nl, ask(X, Y),
        read(A), remember(X, Y, A), A = yes.


% Queries
ask(worn_out, sparking_plugs) ←
        write('Do any of the sparking plugs fail to produce a spark?').
ask(out_of, fuel) ←
        write('Does the fuel gauge indicate an empty tank?').
ask(broken, fuel_pump) ←
        write('Does the fuel pump fail to feed any fuel?').
ask(broken, fuse) ←
        write('Is fuse number 13 broken?').
ask(discharged, battery) ←
        write('Is the battery voltage less than 11 volts?').
ask(broken, starting_motor) ←
        write('Is the starting motor silent?').


% Remember replies to queries
remember(X, Y, yes) ←
        assertz(known(X, Y, true)).
remember(X, Y, no) ←
        assertz(known(X, Y, false)).


% Knowledge-base
malfunctions(X) if needs(X, Y) and malfunctions(Y).
malfunctions(X) if symptom(Y, X) and not indirect(X).
indirect(X) if needs(X, Y) and malfunctions(Y).
needs(car, ignition_system) if true.
needs(car, fuel_system) if true.
needs(car, electric_system) if true.
needs(ignition_system, starting_motor) if true.
needs(ignition_system, sparking_plugs) if true.
needs(electric_system, fuse) if true.
needs(electric_system, battery) if true.
needs(fuel_system, fuel_pump) if true.
needs(sparking_plugs, battery) if true.
needs(starting_motor, battery) if true.
needs(fuel_pump, fuel) if true.
```

```
% Explanations
$print\_proof(void).$
$print\_proof(X \ \& \ Y) \leftarrow$
        $print\_proof(X), nl, print\_proof(Y).$
$print\_proof(proof(X, void)) \leftarrow$
        $write(X), nl.$
$print\_proof(proof(X, Y)) \leftarrow$
        $Y \neq void, write(X), write($' BECAUSE'$), nl,$
        $print\_children(Y), nl, print\_proof(Y).$

$print\_children(proof(X, Y) \ \& \ Z) \leftarrow$
        $tab(8), write(X), write($' AND'$), nl, print\_children(Z).$
$print\_children(proof(X, Y)) \leftarrow$
        $tab(8), write(X), nl.$
```

Exercises

9.1 Improve the printing of the proof. For instance, instead of printing the whole proof at once the system may print the top of the proof and then let the user decide which branch to explain further. Another possibility is the use of natural language. Thus a possible interaction may look as follows:

```
The car malfunctions BECAUSE
        (1) the car needs the ignition-system AND
        (2) the ignition-system malfunctions

Explore? 2.

The ignition-system malfunctions BECAUSE
        (1) the ignition-system needs the sparking-plugs AND
        (2) the sparking-plugs malfunction

Explore?
```

9.2 Write an inference engine which exploits probabilities of rules so that it becomes possible to draw conclusions together with some measurement of their belief. (See Shapiro (1983b).)

9.3 Extend the shell so that the user may give the query "why?" in reply to the system's questions. In such cases the system should explain the conclusions possible if the user gives a particular answer to the query.

Chapter 10

Logic and Grammars

10.1 Context-free Grammars

A *language* can be viewed as a (usually infinite) set of *sentences* or *strings* of finite length. Such strings are composed of symbols of some alphabet (not necessarily the Latin alphabet). However, not all combinations of symbols are well-formed strings. Thus, when defining a new, or describing an existing language, be it a natural or an artificial one (for instance, a programming language), the specification should contain only well-formed strings. A number of formalisms have been suggested to facilitate such systematic descriptions of languages — most notably the formalism of context-free grammars (CFGs). This section contains a brief recapitulation of basic definitions from formal language theory (for details see e.g. Hopcroft and Ullman (1979)).

Formally a context-free grammar is a 4-tuple $\langle N, T, P, S\rangle$, where N and T are finite, disjoint sets of identifiers called the nonterminal- and terminal-alphabets respectively. P is a finite subset of $N \times (N \cup T)^*$. S is a nonterminal symbol called the *start symbol*.

As usual $(N \cup T)^*$ denotes the set of all strings (sequences) of terminals and nonterminals. Traditionally the empty string is denoted by the symbol ϵ. Elements of the relation P are usually written in the form:

$$\begin{array}{l} A \rightarrow B_1 \ldots B_n \quad \text{when } n > 0 \\ A \rightarrow \epsilon \quad \text{when } n = 0 \end{array}$$

Each such element is called a *production rule*. To distinguish between terminals and nonterminals the latter are sometimes written within angle brackets. As an example, consider the production rule:

$$\langle \textit{statement} \rangle \quad \rightarrow \quad \textbf{if } \langle \textit{condition} \rangle \textbf{ then } \langle \textit{statement} \rangle$$

This rule states that a string is a statement if it begins with the symbol "**if**" followed in turn by; a string which is a condition, the symbol "**then**" and finally a string which is a statement. A CFG may contain several production rules with the same left-hand side.

Now let α, β and γ be arbitrary strings from the set $(N \cup T)^*$. We say that the string $\alpha\beta\gamma$ is *directly derivable* from $\alpha A\gamma$ iff $A \rightarrow \beta \in P$. The relation is denoted by $\alpha A\gamma \Rightarrow \alpha\beta\gamma$.

Let $\overset{*}{\Rightarrow}$ be the reflexive and transitive closure of the relation $\Rightarrow$ and let $\alpha, \beta \in (N \cup T)^*$. Then β is said to be *derived* from α iff $\alpha \overset{*}{\Rightarrow} \beta$. The sequence $\alpha \Rightarrow \cdots \Rightarrow \beta$ is called a *derivation.*

Example 10.1 Consider the following set of production rules:

$$\begin{array}{lcl}
\langle \textit{sentence} \rangle & \rightarrow & \langle \textit{noun-phrase} \rangle\ \langle \textit{verb-phrase} \rangle \\
\langle \textit{noun-phrase} \rangle & \rightarrow & \textbf{the}\ \langle \textit{noun} \rangle \\
\langle \textit{verb-phrase} \rangle & \rightarrow & \textbf{runs} \\
\langle \textit{noun} \rangle & \rightarrow & \textbf{engine} \\
\langle \textit{noun} \rangle & \rightarrow & \textbf{rabbit}
\end{array}$$

For instance, $\langle \textit{sentence} \rangle$ derives the string **the rabbit runs** since:

$$\begin{array}{lcl}
\langle \textit{sentence} \rangle & \Rightarrow & \langle \textit{noun-phrase} \rangle\ \langle \textit{verb-phrase} \rangle \\
& \Rightarrow & \textbf{the}\ \langle \textit{noun} \rangle\ \langle \textit{verb-phrase} \rangle \\
& \Rightarrow & \textbf{the rabbit}\ \langle \textit{verb-phrase} \rangle \\
& \Rightarrow & \textbf{the rabbit runs}
\end{array}$$

∎

The *language* of a nonterminal A is the set $\{\alpha \in T^* \mid A \overset{*}{\Rightarrow} \alpha\}$. The language of a CFG is the language of its start-symbol. However, no specific start-symbol will be used below.

Example 10.2 The language of $\langle \textit{sentence} \rangle$ in the previous example is the set:

$$\{\textbf{the rabbit runs}, \textbf{the engine runs}\}$$

∎

The derivation of a terminal string α from a nonterminal A can also be described by means of a so called derivation-tree (or parse-tree) constructed as follows:

- the root of the tree is labelled by A;
- the leaves of the tree are terminals and concatenation of the leaves from left to right yields the string α;
- an internal node X has the children $X_1, \ldots, X_n$ (from left to right) only if there is a production rule of the form $X \rightarrow X_1 \ldots X_n$.

For instance, the derivation in Example 10.1 is described in Figure 10.1. Notice that the derivation tree in general represents many derivations depending on which nonterminal is selected in each step. There is an analogy to the "independence of computation rule" which is reflected in the derivation- or proof-trees for definite programs (see Chapter 3).

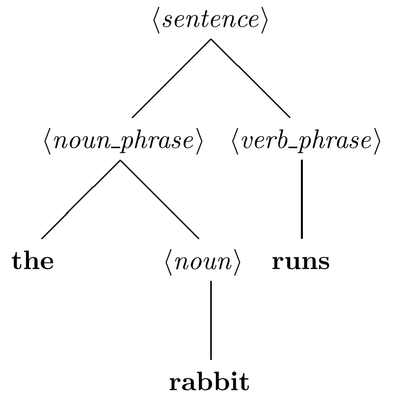


Figure 10.1: Derivation tree for CFG

By describing the two relations "$\Rightarrow$" and "$\overset{*}{\Rightarrow}$" it is possible to construct an "interpreter" for context-free grammars, which behaves as a parser that recognizes strings defined by the grammar. To do this, terminals and nonterminals will be represented by constants, strings will be represented by lists and each production rule will be represented by the clause:

$$prod\_rule(X, Y).$$

where X and Y represent the left- and right-sides of the rule.

Example 10.3 The CFG in Example 10.1 may be represented by the definite program:

```
prod_rule(sentence, [noun_phrase, verb_phrase]).
prod_rule(noun_phrase, [the, noun]).
prod_rule(verb_phrase, [runs]).
prod_rule(noun, [rabbit]).
prod_rule(noun, [engine]).
```

It is now straightforward to define *step*/2 denoting the relation "$\Rightarrow$":

```
step(X, Y) ←
        append(Left, [Lhs|Right], X),
        prod_rule(Lhs, Rhs),
        append(Left, Rhs, Tmp),
        append(Tmp, Right, Y).
```

Since "$\overset{*}{\Rightarrow}$" is the reflexive and transitive closure of "$\Rightarrow$" it is defined as follows (cf. Chapter 6):

```
derives(X, X).
derives(X, Z) ← step(X, Y), derives(Y, Z).
```

∎

Presented with the goal $\leftarrow derives([sentence], X)$ Prolog succeeds with all possible (non-)terminal strings which may be derived from $\langle sentence \rangle$ including the two terminal strings $X = [the, rabbit, runs]$ and $X = [the, engine, runs]$.

As shown in the next section there are more efficient ways of describing context-free languages than the program in Example 10.3. However, the program works as long as the grammar is not left-recursive and it has the following interesting properties:

- The program is quite general — to describe another context-free language it suffices to rewrite the definition of $prod\_rule/1$. The rest of the program may be used for any context-free grammar.
- When comparing Example 10.3 and Examples 8.2 – 8.4 one realizes that the programs are very similar. Both definitions of $step/2$ describe a relation between two expressions where the second is obtained by rewriting the first expression using some kind of rule. The relations $derives/2$ and $derivation/2$ are the reflexive and transitive closures of the "step"-relations. Finally, $prod\_rule/1$ and $clause/1$ are used to represent rules of the formalisms.
- The program operates either as a *top-down* or a *bottom-up* parser depending on how the subgoals in the clauses are ordered. As presented in Example 10.3 it behaves as a traditional recursive descent parser under Prolog's computation rule, but if the subgoals in $derives/2$ and $step/2$ are swapped the program behaves as a (quite inefficient) bottom-up parser.

10.2 Logic Grammars

Although Example 10.3 provides a very general specification of the derivability-relation for context-free grammars it is a rather inefficient program. In this section a more direct approach is discussed, whereby the extra interpreter-layer is avoided, resulting in parsers with better efficiency than the one above.

As already noted there is a close resemblance between logic programs and context-free grammars. It is not hard to see that logic programs can be used directly to specify exactly the same language as a CFG, without resorting to explicit definitions of the two relations "$\Rightarrow$" and "$\overset{*}{\Rightarrow}$". Consider the following clause:

$$sentence(Z) \leftarrow append(X, Y, Z), noun\_phrase(X), verb\_phrase(Y).$$

Declaratively it reads "For any X, Y, Z — Z is a sentence if X is a noun-phrase, Y is a verb-phrase and Z is the concatenation of X and Y". By representing strings as lists of ground terms the whole grammar in Example 10.1 can be formulated as the following Prolog program:

Example 10.4

$sentence(Z) \leftarrow append(X, Y, Z), noun\_phrase(X), verb\_phrase(Y).$
$noun\_phrase([the|X]) \leftarrow noun(X).$
$verb\_phrase([runs]).$
$noun([engine]).$
$noun([rabbit]).$

$$\begin{array}{l} append([\,], X, X). \\ append([X|Y], Z, [X|W]) \leftarrow append(Y, Z, W). \end{array}$$

∎

The program is able to refute goals like:

$$\begin{array}{l} \leftarrow sentence([the, rabbit, runs]). \\ \leftarrow sentence([the, X, runs]) \end{array}$$

In reply to the second goal Prolog would give the answers $X = rabbit$ and $X = engine$. It is even possible to give the goal:

$$\leftarrow sentence(X).$$

In this case Prolog returns all (i.e. both) sentences of the language before going into an infinite loop (incidentally, this loop can be avoided by moving the call to *append*/3 to the very end of the first clause).

Unfortunately the program in Example 10.4 is also rather inefficient. The *append*/3 procedure will blindly generate all partitions of the list and it may take some time to find the correct splitting (at least in the case when the list is very long). To remedy this problem the concept of *difference lists* may be used.

Using difference lists the first clause of Example 10.4 can be written as:

$$sentence(X_0 - X_2) \leftarrow noun\_phrase(X_0 - X_1), verb\_phrase(X_1 - X_2).$$

Declaratively it reads — "The difference between X_0 and X_2 is a sentence if the difference between X_0 and X_1 is a noun-phrase and the difference between X_1 and X_2 is a verb-phrase". The statement is evidently true — consider the string:

$$\underbrace{x_1 \ldots x_i \; \underbrace{x_{i+1} \ldots x_j \; \underbrace{x_{j+1} \ldots x_k}_{X_2}}_{X_1}}_{X_0}$$

If the difference between X_1 and X_2 (that is, the string $x_{i+1} \ldots x_j$) is a verb-phrase and the difference between X_0 and X_1 (the string $x_1 \ldots x_i$) is a noun-phrase then the string $x_1 \ldots x_i x_{i+1} \ldots x_j$ (that is the difference between X_0 and X_2) is a sentence.

Using this approach, Example 10.4 can be reformulated as follows:

Example 10.5

$$\begin{array}{l} sentence(X_0 - X_2) \leftarrow noun\_phrase(X_0 - X_1), verb\_phrase(X_1 - X_2). \\ noun\_phrase(X_0 - X_2) \leftarrow connects(X_0, the, X_1), noun(X_1 - X_2). \\ verb\_phrase(X_0 - X_1) \leftarrow connects(X_0, runs, X_1). \\ noun(X_0 - X_1) \leftarrow connects(X_0, engine, X_1). \\ noun(X_0 - X_1) \leftarrow connects(X_0, rabbits, X_1). \\ \\ connects([X|Y], X, Y). \end{array}$$

∎

Although the intended interpretation of the functor $-/2$ is a function which produces the difference between two lists, the Prolog interpreter has no knowledge about this particular interpretation. As a consequence the goal:

$$\leftarrow \mathit{sentence}([\mathit{the}, \mathit{rabbit}, \mathit{runs}]).$$

does not succeed (simply because $[\mathit{the}, \mathit{rabbit}, \mathit{runs}]$ does not unify with the term $X_0 - X_2$). In order to get a positive answer the goal must contain, as its argument, a difference list. For instance:

$$\leftarrow \mathit{sentence}([\mathit{the}, \mathit{rabbit}, \mathit{runs}] - [\,]).$$

Intuitively, this goal has the same declarative reading as the previous one since the difference of the list denoted by $[\mathit{the}, \mathit{rabbit}, \mathit{runs}]$ and the empty list is equivalent to the intended meaning of the term $[\mathit{the}, \mathit{rabbit}, \mathit{runs}]$.

Other examples of goals which succeed are:

$$\begin{array}{l}\leftarrow \mathit{sentence}([\mathit{the}, \mathit{rabbit}, \mathit{runs}|X] - X).\\ \leftarrow \mathit{sentence}([\mathit{the}, \mathit{rabbit}, \mathit{runs}, \mathit{quickly}] - [\mathit{quickly}]).\\ \leftarrow \mathit{sentence}(X - [\,]).\\ \leftarrow \mathit{sentence}(X).\end{array}$$

For instance, the third goal produces two answers: $X = [\mathit{the}, \mathit{rabbit}, \mathit{runs}]$ and $X = [\mathit{the}, \mathit{engine}, \mathit{runs}]$.

Notice the way terminals of the grammar are treated — an auxiliary predicate $\mathit{connects}(A, B, C)$ is used to check if the difference of A and C is equal to B. This auxiliary predicate can be eliminated using *unfolding* of the program — consider the clause:[1]

$$\mathit{noun\_phrase}(X_0 - X_2) \leftarrow \mathit{connects}(X_0, \mathit{the}, X_1), \mathit{noun}(X_1 - X_2).$$

Now resolve the leftmost atom in the body! Unification of $\mathit{connects}(X_0, \mathit{the}, X_1)$ with the clause $\mathit{connects}([X|Y], X, Y)$ yields an mgu:

$$\{X_0/[\mathit{the}|X_1], X/\mathit{the}, Y/X_1\}$$

Removing $\mathit{connects}(X_0, \mathit{the}, X_1)$ from the body of the clause and applying the mgu to the remaining atoms yields a specialized clause:

[1]Unfolding is a simple but powerful technique for transforming (logic) programs. Roughly speaking the idea can be formulated as follows — let C be the clause:

$$A_0 \leftarrow A_1, \ldots, A_{i-1}, A_i, A_{i+1}, \ldots, A_n$$

and let:

$$B_0 \leftarrow B_1, \ldots, B_m$$

be a clause whose head unifies with A_i (with the mgu θ). Then C may be replaced by the new clause:

$$(A_0 \leftarrow A_1, \ldots, A_{i-1}, B_1, \ldots, B_m, A_{i+1}, \ldots, A_n)\theta$$

It is easy to prove that this transformation does not add to the set of formulas which follow logically from the program. That is, the transformation is "sound". Under certain restrictions one can also prove that the technique is "complete" in the sense that the set of all logical consequences of the old program is exactly the same as the set of logical consequences of the new program.

$$noun\_phrase([the|X_1] - X_2) \leftarrow noun(X_1 - X_2).$$

Resolving all such subgoals from all the other clauses in Example 10.5 yields a new program which does not make use of the predicate $connects(A, B, C)$:

Example 10.6

$$\begin{array}{l}
sentence(X_0 - X_2) \leftarrow noun\_phrase(X_0 - X_1), verb\_phrase(X_1 - X_2).\\
noun\_phrase([the|X_1] - X_2) \leftarrow noun(X_1 - X_2).\\
verb\_phrase([runs|X_1] - X_1).\\
noun([engine|X_1] - X_1).\\
noun([rabbit|X_1] - X_1).
\end{array}$$

▮

10.3 Context-dependent Languages

Although context-free grammars are often used to specify the syntax of programming languages they have many limitations. It is well-known that the class of context-free languages is restricted. Even a simple language such as $\mathbf{a}^n\mathbf{b}^n\mathbf{c}^n$ where $n \in \{0, 1, 2, \ldots\}$ (i.e. all strings which are built from equal number of **a**'s, **b**'s and **c**'s) cannot be defined by a context-free grammar. Consider the grammar:

$$\begin{array}{lcl}
\langle abc \rangle & \rightarrow & \langle a \rangle\ \langle b \rangle\ \langle c \rangle\\
\langle a \rangle & \rightarrow & \epsilon\\
\langle a \rangle & \rightarrow & \mathbf{a}\ \langle a \rangle\\
\langle b \rangle & \rightarrow & \epsilon\\
\langle b \rangle & \rightarrow & \mathbf{b}\ \langle b \rangle\\
\langle c \rangle & \rightarrow & \epsilon\\
\langle c \rangle & \rightarrow & \mathbf{c}\ \langle c \rangle
\end{array}$$

The language of $\langle abc \rangle$ certainly contains the language $\mathbf{a}^n\mathbf{b}^n\mathbf{c}^n$ but also other strings — like **a b b c c c**. To describe languages like $\mathbf{a}^n\mathbf{b}^n\mathbf{c}^n$ more powerful formalisms are needed. For instance, the property of being a string in the language $\mathbf{a}^n\mathbf{b}^n\mathbf{c}^n$ is described by the following definite program:

Example 10.7

$$\begin{array}{l}
abc(X_0 - X_3) \leftarrow a(N, X_0 - X_1), b(N, X_1 - X_2), c(N, X_2 - X_3).\\
a(0, X_0 - X_0).\\
a(s(N), [a|X_1] - X_2) \leftarrow a(N, X_1 - X_2).\\
b(0, X_0 - X_0).\\
b(s(N), [b|X_1] - X_2) \leftarrow b(N, X_1 - X_2).\\
c(0, X_0 - X_0).\\
c(s(N), [c|X_1] - X_2) \leftarrow c(N, X_1 - X_2).
\end{array}$$

▮

Here the first clause reads "The string $X_0 - X_3$ is a member of $\mathbf{a}^n\mathbf{b}^n\mathbf{c}^n$ if $X_0 - X_1$ is a string of N **a**'s and $X_1 - X_2$ is a string of N **b**'s and $X_2 - X_3$ is a string of N **c**'s". The restriction to equal number of **a**'s, **b**'s and **c**'s is thus obtained through the extra argument of the predicate symbols $a/2$, $b/2$ and $c/2$.

As an additional example consider the following excerpt from a context-free natural language description:

$$
\begin{array}{lcl}
\langle \textit{sentence} \rangle & \rightarrow & \langle \textit{noun-phrase} \rangle \langle \textit{verb} \rangle \\
\langle \textit{noun-phrase} \rangle & \rightarrow & \langle \textit{pronoun} \rangle \\
\langle \textit{noun-phrase} \rangle & \rightarrow & \textbf{the } \langle \textit{noun} \rangle \\
\langle \textit{noun} \rangle & \rightarrow & \textbf{rabbit} \\
\langle \textit{noun} \rangle & \rightarrow & \textbf{rabbits} \\
\langle \textit{pronoun} \rangle & \rightarrow & \textbf{it} \\
\langle \textit{pronoun} \rangle & \rightarrow & \textbf{they} \\
\langle \textit{verb} \rangle & \rightarrow & \textbf{runs} \\
\langle \textit{verb} \rangle & \rightarrow & \textbf{run}
\end{array}
$$

Unfortunately the language of $\langle \textit{sentence} \rangle$ includes strings such as **the rabbit run**, **they runs** and **it run** — strings which should not be part of the language. Again, this can be repaired reasonably easy by defining the language by means of a logic program and by adding extra arguments to the predicate symbols corresponding to nonterminals:

Example 10.8 Consider the following logic program:

$$
\begin{array}{l}
\textit{sentence}(X_0 - X_2) \leftarrow \textit{noun\_phrase}(Y, X_0 - X_1), \textit{verb}(Y, X_1 - X_2). \\
\textit{noun\_phrase}(Y, X_0 - X_1) \leftarrow \textit{pronoun}(Y, X_0 - X_1). \\
\textit{noun\_phrase}(Y, X_0 - X_2) \leftarrow \textit{connects}(X_0, \textit{the}, X_1), \textit{noun}(Y, X_1 - X_2). \\
\textit{noun}(\textit{singular}(3), X_0 - X_1) \leftarrow \textit{connects}(X_0, \textit{rabbit}, X_1). \\
\textit{noun}(\textit{plural}(3), X_0 - X_1) \leftarrow \textit{connects}(X_0, \textit{rabbits}, X_1). \\
\textit{pronoun}(\textit{singular}(3), X_0 - X_1) \leftarrow \textit{connects}(X_0, \textit{it}, X_1). \\
\textit{pronoun}(\textit{plural}(3), X_0 - X_1) \leftarrow \textit{connects}(X_0, \textit{they}, X_1). \\
\textit{verb}(\textit{plural}(Y), X_0 - X_1) \leftarrow \textit{connects}(X_0, \textit{run}, X_1). \\
\textit{verb}(\textit{singular}(3), X_0 - X_1) \leftarrow \textit{connects}(X_0, \textit{runs}, X_1).
\end{array}
$$

A goal clause of the form:

$$\leftarrow \textit{sentence}([\textit{the}, \textit{rabbits}, \textit{runs}] - [\,]).$$

may be reduced to the compound goal:

$$\leftarrow \textit{noun\_phrase}(Y, [\textit{the}, \textit{rabbits}, \textit{runs}] - X_1), \textit{verb}(Y, X_1 - [\,]).$$

The leftmost goal eventually succeeds with the bindings $Y = \textit{plural}(3)$ and $X_1 = [\textit{runs}]$ but the remaining subgoal fails:

$$\leftarrow \textit{verb}(\textit{plural}(3), [\textit{runs}] - [\,]).$$

Thus, the string is not a member in the language defined by the program. ∎

The extra arguments added to some predicate symbols serve essentially two purposes — as demonstrated above, they can be used to propagate constraints between subgoals corresponding to nonterminals of the grammar and they may also be used to construct some alternative (structured) representation of the string being analysed. For instance, as a parse-tree or some other form of intermediate code.

10.4 Definite Clause Grammars (DCGs)

Many Prolog systems employ special syntax for language specifications. When such a description is encountered, the system automatically compiles it into a Prolog program. Such specifications are called *Definite Clause Grammars* (DCGs). There are two possible views of such grammars — either they are viewed as a "syntactic sugar" for Prolog. That is, the grammar is seen as a convenient shorthand for a Prolog program. Alternatively the notion of DCG is viewed as an independent formalism on its own. This book adopts the latter view.

Assume that an alphabet similar to that in Chapter 1 is given. Then a DCG is a triple $\langle N, T, P\rangle$ where:

- N is a possibly infinite set of atoms;
- T is a possibly infinite set of terms;
- $P \subseteq N \times (N \cup T)^*$ is a finite set of (production) rules.

By analogy to CFGs N and T are assumed to be disjoint and are called *nonterminals* and *terminals* respectively.

DCGs are generalizations of CFGs and it is therefore possible to generalize the concept of direct derivability. Let α, α', $\beta \in (N \cup T)^*$ and let $p(t_1, \ldots, t_n) \rightarrow \beta \in P$ (where variables are renamed so that no name clashes occur with variables in α). Then α' is *directly derivable* from α iff:

- α is of the form $\alpha_1 p(s_1, \ldots, s_n)\alpha_2$;
- $p(t_1, \ldots, t_n)$ and $p(s_1, \ldots, s_n)$ unify with mgu θ;
- α' is of the form $(\alpha_1\beta\alpha_2)\theta$.

Also the concept of *derivation* is a generalization of the CFG-counterpart. The derivability-relation for DCGs is the reflexive and transitive closure of the direct-derivability-relation. A string of terminals $\beta \in T^*$ is in the language of $A \in N$ iff $\langle A, \beta\rangle$ is in the derivability-relation.

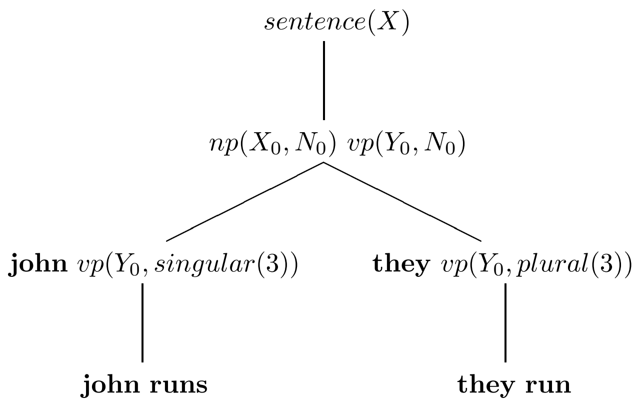
Example 10.9 Consider the following DCG:

$$\begin{array}{lcl}
sentence(s(X,Y)) & \rightarrow & np(X,N)\ vp(Y,N) \\
np(john, singular(3)) & \rightarrow & \mathbf{john} \\
np(they, plural(3)) & \rightarrow & \mathbf{they} \\
vp(run, plural(X)) & \rightarrow & \mathbf{run} \\
vp(runs, singular(3)) & \rightarrow & \mathbf{runs}
\end{array}$$

From the nonterminal $sentence(X)$ the following derivation may be constructed:

$$\begin{array}{lcl}
sentence(X) & \Rightarrow & np(X_0, N_0)\ vp(Y_0, N_0) \\
& \Rightarrow & \mathbf{john}\ vp(Y_0, singular(3)) \\
& \Rightarrow & \mathbf{john\ runs}
\end{array}$$

Thus, the string **john runs** is in the language of $sentence(X)$. But more than that, the mgu's of the derivation together produce a binding for the variables used in the



$sentence(X)$
$np(X_0, N_0)\ vp(Y_0, N_0)$
john $vp(Y_0, singular(3))$
they $vp(Y_0, plural(3))$
john runs
they run

Figure 10.2: "SLD-tree" for DCGs

derivation — in the first step X is bound to the term $s(X_0, Y_0)$. In the second step X_0 is bound to *john* and finally Y_0 is bound to *runs*. Composition of the mgu's yields the "answer" $X/s(john, runs)$ to the initial nonterminal. ∎

However, the derivation in the example is not the only one that starts with the nonterminal $sentence(X)$. For instance, in derivation step two, the second nonterminal may be selected instead, and in the same step the third production rule may be used instead of the second. It turns out that the choice of nonterminal is of no importance, but that the choice of production rule is. This is yet another similarity between DCGs and logic programs. The choice of nonterminal corresponds to the selection of subgoal. In fact, the collection of all derivations starting with a nonterminal under a fixed "computation rule" can be depicted as an "SLD-tree". For instance, all possible derivations originating from the nonterminal $sentence(X)$ (where the leftmost nonterminal is always selected) is depicted in Figure 10.2.

As discussed above many Prolog systems support usage of DCGs by automatically translating them into Prolog programs. In order to discuss combining DCGs and Prolog we need to settle some notational conventions for writing DCGs:

- since Prolog systems cannot distinguish terms from atoms, terminals are enclosed by list-brackets;
- nonterminals are written as ordinary compound terms or constants except that they are not allowed to use certain reserved symbols (e.g. ./2) as principal functors;
- the functor ','/2 (comma) separates terminals and nonterminals in the right-hand side of rules;
- the functor '-->'/2 separates the left- and right-hand sides of a production rule;

- the empty string is denoted by the empty list.

This means that DCGs can be represented as terms. For instance, the rule:

$$np(X) \to \textbf{the}\, noun(X)$$

will be written as the term:

$$np(X) \texttt{-->} [the], noun(X)$$

or using standard syntax:

$$\text{'-->'}(np(X), \text{','}([the], noun(X)))$$

In addition Prolog often allows special treatment of nonterminals which always derive the empty string. Consider the language where each string consists of an even number of **a**'s followed by the same number of **b**'s in turn followed by the same number of **c**'s. This language can be specified as follows using DCGs with Prolog syntax:

$$\begin{array}{lll}
abc & \texttt{-->} & a(N), b(N), c(N), even(N). \\
a(0) & \texttt{-->} & [\,]. \\
a(s(N)) & \texttt{-->} & [a], a(N). \\
b(0) & \texttt{-->} & [\,]. \\
b(s(N)) & \texttt{-->} & [b], b(N). \\
c(0) & \texttt{-->} & [\,]. \\
c(s(N)) & \texttt{-->} & [c], c(N). \\
even(0) & \texttt{-->} & [\,]. \\
even(s(s(N))) & \texttt{-->} & even(N).
\end{array}$$

In this example the occurrence of $even(N)$ in the first rule always derives the empty string. In this respect it can be removed from the rule. However, the primary function of this nonterminal is to constrain bindings of N to terms representing even numbers. That is, it not only defines the language consisting solely of the empty string but also defines a relation (in this case the property of being an even natural number). To distinguish such nonterminals from those which derive nonempty strings they are written within curly brackets, and the definition of the relation is written directly in Prolog. Thus, in Prolog it is possible to write the rule:

$$abc \;\texttt{-->}\; a(N), b(N), c(N), \{even(N)\}.$$

together with the definite program:

$$\begin{array}{l}
even(0). \\
even(s(s(N))) \leftarrow even(N).
\end{array}$$

replacing the first and the two final production rules of the previous DCG.

Now since calls to Prolog may be inserted into a DCG it is also possible to utilize the built-in predicates of Prolog in a DCG.

Example 10.10 This idea is illustrated by the following example, which is a grammar that recognizes arithmetic expressions but, more than that, also computes the value of the expression:

$$
\begin{array}{lll}
expr(X) & \text{-->} & term(Y), [+], expr(Z), \{X \; is \; Y + Z\}.\\
expr(X) & \text{-->} & term(Y), [-], expr(Z), \{X \; is \; Y - Z\}.\\
expr(X) & \text{-->} & term(X).\\
term(X) & \text{-->} & factor(Y), [\,*\,], term(Z), \{X \; is \; Y * Z\}.\\
term(X) & \text{-->} & factor(Y), [\,/\,], term(Z), \{X \; is \; Y/Z\}.\\
term(X) & \text{-->} & factor(X).\\
factor(X) & \text{-->} & [X], \{integer(X)\}.
\end{array}
$$

For instance, the last rule states that any string which consists of a single terminal which is an integer is a factor with the same value as the terminal. Similarly, the first rule states that any string which starts with a term of value Y followed by "+" followed by an expression of value Z is an expression of value $Y + Z$. ▮

As already discussed, when a DCG is loaded into a Prolog system it is usually compiled into a Prolog program similar in style to those in Section 10.2. This transformation is quite simple and it is discussed in the next section. Some implementations of Prolog do not include this feature. Fortunately, it is not very hard to write an *interpreter* for DCGs similar to the Prolog-interpreter in Example 8.7.

For this purpose, view a DCG-rule as a binary fact with predicate symbol '-->'/2 where the first and second arguments consist of the left- and right-side of the production rule. The relationship between strings of terminals/nonterminals and the derivable terminal strings can then be defined as follows:

$$
\begin{array}{l}
derives([\,], S - S).\\
derives([X], [X|S] - S).\\
derives(\{X\}, S - S) \leftarrow\\
\qquad call(X).\\
derives((X, Y), S_0 - S_2) \leftarrow\\
\qquad derives(X, S_0 - S_1), derives(Y, S_1 - S_2).\\
derives(X, S_0 - S_1) \leftarrow\\
\qquad (X \text{ --> } Y), derives(Y, S_0 - S_1).
\end{array}
$$

The interpreter is surprisingly simple. Declaratively the clauses state the following:

- the empty string derives itself. That is, the difference between S and S for any S;
- the second clause says that the terminal string $[X]$ derives itself. That is, the difference between $[X|S]$ and S for any S;
- a nonterminal X in curly brackets derives the empty string if the goal $\leftarrow X$ has a refutation;
- if the string X derives the terminal string $S_0 - S_1$ and the string Y derives the terminal string $S_1 - S_2$ then the string (X, Y) (that is, Y appended to X) derives the terminal string $S_0 - S_2$;
- if there is a rule $(X \text{ --> } Y)$ such that Y derives the terminal string $S_0 - S_1$ then the nonterminal X derives the same terminal string.

For instance, in the presence of the DCG of Example 10.9 the goal:

$$\leftarrow \mathit{derives}(\mathit{sentence}(X), [\mathit{john}, \mathit{runs}] - [\,]).$$

succeeds with answer $X = s(\mathit{john}, \mathit{runs})$. Similarly the grammar in Example 10.10 and the goal:

$$\leftarrow \mathit{derives}(\mathit{expr}(X), [2, +, 3, *, 4] - [\,]).$$

result in the answer $X = 14$.

10.5 Compilation of DCGs into Prolog

The standard treatment of DCGs in most Prolog systems is to compile them directly into Prolog clauses. Since each production rule translates into one clause, the transformation is relatively simple. The clause obtained as a result of the transformations described below may differ slightly from what is obtained in some Prolog systems but the principle is the same.

The general idea is the following — consider a production rule of the form:

$$p(t_1, \ldots, t_n) \text{ --> } T_1, \ldots, T_m$$

Assume that $X_0, \ldots, X_m$ are distinct variables which do not appear in the rule. Then the production rule translates into the definite clause:

$$p(t_1, \ldots, t_n, X_0, X_m) \leftarrow A_1, \ldots, A_m$$

where:

- if T_i is of the form $q(s_1, \ldots, s_j)$, then A_i is $q(s_1, \ldots, s_j, X_{i-1}, X_i)$;
- if T_i is of the form $[T]$, then A_i is $\mathit{connects}(X_{i-1}, T, X_i)$;[2]
- if T_i is of the form $\{T\}$, then A_i is $T, X_{i-1} \doteq X_i$;
- if T_i is of the form $[\,]$, then A_i is $X_{i-1} \doteq X_i$.

For instance, the first rule of Example 10.10 is transformed as follows:

$$\begin{array}{lll}
\mathit{expr}(X) \text{ -->} & & \mathit{expr}(X, X_0, X_4) \leftarrow \\
\quad \mathit{term}(Y), & & \quad \mathit{term}(Y, X_0, X_1), \\
\quad [+], & \Rightarrow & \quad \mathit{connects}(X_1, +, X_2), \\
\quad \mathit{expr}(Z), & & \quad \mathit{expr}(Z, X_2, X_3), \\
\quad \{X \ \mathit{is}\ Y + Z\}. & & \quad X \ \mathit{is}\ Y + Z, X_3 \doteq X_4.
\end{array}$$

Some simplifications can be made to the final result — in particular, subgoals of the form $X \doteq Y$ may be omitted if all occurrences of Y are replaced by the variable X. This means that the result obtained above can be simplified into:

[2]Many Prolog systems use instead a built-in predicate 'C'(A, B, C) with the same semantics as $\mathit{connects}(A, B, C)$.

$$\begin{aligned}&expr(X, X_0, X_3) \leftarrow\\ &\qquad term(Y, X_0, X_1),\\ &\qquad connects(X_1, +, X_2),\\ &\qquad expr(Z, X_2, X_3),\\ &\qquad X \ is \ Y + Z.\end{aligned}$$

When all rules are translated in this way and the program is extended by the definition $connects([X|Y], X, Y)$, the resulting program can be used to refute goals like $\leftarrow expr(X, [2, +, 3, *, 4], [\,])$, with the expected answer $X = 14$.

A CFG (which is a special case of a DCG) like the one in Example 10.1 translates into the program of Example 10.5 except that arguments of the form $X - Y$ are split into two arguments and that the names of the variables may differ.

Example 10.11 As a final example the translation of Example 10.9 results in the following program:

$$\begin{aligned}&sentence(s(X, Y), X_0, X_2) \leftarrow\\ &\qquad np(X, N, X_0, X_1), vp(Y, N, X_1, X_2).\\[4pt] &np(john, singular(3), X_0, X_1) \leftarrow\\ &\qquad connects(X_0, john, X_1).\\ &np(they, plural(3), X_0, X_1) \leftarrow\\ &\qquad connects(X_0, they, X_1).\\[4pt] &vp(run, plural(X), X_0, X_1) \leftarrow\\ &\qquad connects(X_0, run, X_1).\\ &vp(runs, singular(3), X_0, X_1) \leftarrow\\ &\qquad connects(X_0, runs, X_1).\\[4pt] &connects([X|Y], X, Y).\end{aligned}$$

Given the goal $\leftarrow sentence(X, [john, runs], [\,])$ Prolog replies with the answer $X = s(john, runs)$. ∎

Exercises

10.1 Write a DCG which describes the language of strings of octal numbers. Extend the grammar so that the decimal value of the string is returned. For instance, the goal $\leftarrow octal(X, [4, 6], [\,])$ should succeed with $X = 38$.

10.2 Write a DCG which accepts strings in the language $\mathbf{a}^m\mathbf{b}^n\mathbf{c}^m\mathbf{d}^n$, $(n,\ m \geq 0)$.

10.3 Consider the following CFG:

$$\begin{array}{lcl}\langle bleat\rangle & \rightarrow & \mathbf{b}\ \langle aaa\rangle\\ \langle aaa\rangle & \rightarrow & \mathbf{a}\\ \langle aaa\rangle & \rightarrow & \mathbf{a}\ \langle aaa\rangle\end{array}$$

Describe the same language using DCG notation. Then "compile" the specification into a Prolog program and write an SLD-refutation which proves that the string "**b a a**" is in the language of $\langle bleat\rangle$.

10.4 Explain the usage of the following DCG:

$$\begin{array}{lll} x([\,], X, X) & \texttt{-->} & [\,]. \\ x([X|Y], Z, [X|W]) & \texttt{-->} & x(Y, Z, W). \end{array}$$

10.5 Write an interface to a database that facilitates communication in natural language.

10.6 Define a concrete syntax for the imperative language outlined in exercise 8.6. Then write a compiler which translates the concrete syntax (i.e. strings of characters) into the given abstract syntax.

It is probably a good idea to split the translation into two phases. In the first phase the string is translated into a list of lexical items representing identifiers, reserved words, operators etc, after which the string is translated into the abstract syntax.

The following context-free grammar may serve as a starting point for the compiler:

$$\begin{array}{lcl} \langle cmnd \rangle & \rightarrow & \textbf{skip} \\ & | & \langle var \rangle \ := \ \langle expr \rangle \\ & | & \textbf{if } \langle bool \rangle \textbf{ then } \langle cmnd \rangle \textbf{ else } \langle cmnd \rangle \textbf{ fi} \\ & | & \textbf{while } \langle bool \rangle \textbf{ do } \langle cmnd \rangle \textbf{ od} \\ & | & \langle cmnd \rangle \ ; \ \langle cmnd \rangle \\ \langle bool \rangle & \rightarrow & \langle expr \rangle > \langle expr \rangle \\ & | & \ldots \\ \langle expr \rangle & \rightarrow & \langle var \rangle \\ & | & \langle nat \rangle \\ & | & \langle expr \rangle \ \langle op \rangle \ \langle expr \rangle \\ \langle var \rangle & \rightarrow & \mathbf{x} \mid \mathbf{y} \mid \mathbf{z} \mid \ldots \\ \langle nat \rangle & \rightarrow & \mathbf{0} \mid \mathbf{1} \mid \mathbf{2} \mid \ldots \\ \langle op \rangle & \rightarrow & + \mid - \mid * \mid \ldots \end{array}$$

Hint: Most Prolog systems permit using the syntax "prolog" as an alternative for the list of ASCII-characters $[112, 114, 111, 108, 111, 103]$.

Chapter 11

Searching in a State-space

11.1 State-spaces and State-transitions

Many problems in computer science can be formulated as a possibly infinite set S of *states* and a binary transition-relation $\rightsquigarrow$ over this *state-space*. Given some *start*-state $s_0 \in S$ and a set $G \subseteq S$ of *goal*-states such problems consist in determining whether there exists a sequence:

$$s_0 \rightsquigarrow s_1, s_1 \rightsquigarrow s_2, s_2 \rightsquigarrow s_3, \cdots s_{n-1} \rightsquigarrow s_n$$

such that $s_n \in G$ (i.e. to determine if $\langle s_0, s_n \rangle$ is in the transitive and reflexive closure, $\overset{*}{\rightsquigarrow}$, of $\rightsquigarrow$). More informally the states can be seen as nodes in a graph whose edges represent the pairs in the transition-relation. Then the problem reduces to that of finding a path from the start-state to one of the goal-states.

Example 6.7 embodies an instance of such a problem — the state-space consisted of a finite set of states named by a, b, c, d, e, f and g. The predicate symbol $edge/2$ was used to describe the transition relation and $path/2$ described the transitive and reflexive closure of the transition relation. Hence, the existence of a path from, for instance, the state a to e is checked by giving the goal $\leftarrow path(a, e)$. Now this is by no means the only example of such a problem. The following ones are all examples of similar problems:

- *Planning* amounts to finding a sequence of worlds where the initial world is transformed into some desired final world. For instance, the initial world may consist of a robot and some parts. The objective is to find a world where the parts are assembled in some desirable way. Here the description of the world is a state and the transformations which transform one world to another can be seen as a transition relation.

- The derivation of a string of terminals α from a nonterminal A can also be viewed in this way. The state-space consists of all strings of terminals/nonterminals. The string A is the start-state and α the goal-state. The relation "$\Rightarrow$" is the transition relation and the problem amounts to finding a sequence of derivation steps $A \Rightarrow \cdots \Rightarrow \alpha$.

- Also SLD-derivations may be formulated in this way — the states are goals and the transition relation consists of the SLD-resolution principle which produces a goal G_{i+1} out of another goal G_i and some program clause C_i. In most cases the start-state is the initial goal and the goal-state is the empty goal.

Consider the following two clauses of Example 6.7 again:

$$\begin{array}{l} path(X, X). \\ path(X, Z) \leftarrow edge(X, Y), path(Y, Z). \end{array}$$

Operationally the second clause reads as follows provided that Prolog's computation rule is employed — "To find a path from X to Z, first find an edge from X to Y and then find a path from Y to Z". That is, first try to find a node adjacent to the start-state, and then try to find a path from the new node to the goal-state. In other words, the search proceeds in a forward direction — from the start-state to the goal-state. However, it is easy to modify the program to search in the opposite direction assuming that Prolog's computation rule is used — simply rewrite the second clause as:

$$path(X, Z) \leftarrow edge(Y, Z), path(X, Y).$$

The decision whether to search in a forward or backward direction depends on what the search space looks like. Such considerations will not be discussed here, but the reader is referred to the AI-literature

The $path/2$-program above does not work without modifications if there is more than one goal-state. For instance, if both f and g are goal-states one has to give two goals. An alternative solution is to extend the program with the property of being a goal-state:

$$\begin{array}{l} goal\_state(f). \\ goal\_state(g). \end{array}$$

Now the problem of finding a path from a to one of the goal-states reduces to finding a refutation of the goal:

$$\leftarrow path(a, X), goal\_state(X).$$

The program above can be simplified if the goal-state is known in advance. In this case it is not necessary to use the second argument of $path/2$. Instead the program may be simplified into:

$$\begin{array}{l} path(\ulcorner goal \urcorner). \\ path(X) \leftarrow edge(X, Y), path(Y). \end{array}$$

where $\ulcorner goal \urcorner$ is a term representing the goal-state (if there are several goal-states there will be one such fact for each state).

11.2 Loop Detection

One problem mentioned in connection with Example 6.7, appears when the graph defined by the transition relation is cyclic.

Example 11.1 Consider the program:

$$
\begin{array}{l}
path(X, X). \\
path(X, Z) \leftarrow edge(X, Y), path(Y, Z).
\end{array}
$$

$$
\begin{array}{lll}
edge(a, b). & edge(b, a). & edge(a, c). \\
edge(b, d). & edge(b, e). & edge(c, e). \\
edge(d, f). & edge(e, f). & edge(e, g).
\end{array}
$$

∎

As pointed out in Chapter 6 the program may go into an infinite loop for certain goals — from state a it is possible to go to state b and from this state it is possible to go back to state a via the cycle in the transition relation. One simple solution to such problems is to keep a *log* of all states already visited. Before moving to a new state it should be checked that the new state has not already been visited.

Example 11.2 The following program extends Example 11.1 with a log:

$$
\begin{array}{l}
path(X, Y) \leftarrow \\
\qquad path(X, Y, [X]). \\
\\
path(X, X, \mathit{Visited}). \\
path(X, Z, \mathit{Visited}) \leftarrow \\
\qquad edge(X, Y), \\
\qquad not\ member(Y, \mathit{Visited}), \\
\qquad path(Y, Z, [Y \mid \mathit{Visited}]). \\
\\
member(X, [X|Y]). \\
member(X, [Y|Z]) \leftarrow \\
\qquad member(X, Z).
\end{array}
$$

∎

Declaratively the recursive clause of *path*/3 says that — "there is a path from X to Z if there is an edge from X to Y and a path from Y to Z such that Y has not already been visited".

At first glance the solution may look a bit inelegant and there certainly *are* more sophisticated solutions around. However, carrying the log around is not such a bad idea after all — in many problems similar to the one above, it is not sufficient just to answer "yes" or "no" to the question of whether there is a path between two states. Often it is necessary that the actual path is returned as an answer to the goal. As an example, it is not much use to know that there *is* a plan which assembles some pieces of material into a gadget; in general one wants to see the actual plan.

Example 11.3 This extension can be implemented through the following modification of Example 11.2:

$$
\begin{array}{l}
path(X, Y, Path) \leftarrow \\
\qquad path(X, Y, [X], Path). \\
\\
path(X, X, \mathit{Visited}, \mathit{Visited}). \\
path(X, Z, \mathit{Visited}, Path) \leftarrow \\
\qquad edge(X, Y), \\
\qquad not\ member(Y, \mathit{Visited}), \\
\qquad path(Y, Z, [Y \mid \mathit{Visited}], Path).
\end{array}
$$

∎

With these modifications the goal $\leftarrow path(a, d, X)$ succeeds with the answer $X = [d, b, a]$ which says that the path from a to d goes via the intermediate state b. Intuitively, $path(A, B, C, D)$ can be interpreted as follows — "The difference between D and C constitutes a path from A to B".

11.3 Water-jug Problem (Extended Example)

The discussion above will be illustrated with the well-known water-jug problem often encountered in the AI-literature. The problem is formulated as follows:

> Two water jugs are given, a 4-gallon and a 3-gallon jug. Neither of them has any type of marking on it. There is an infinite supply of water (a tap?) nearby. How can you get exactly 2 gallons of water into the 4-gallon jug? Initially both jugs are empty.

The problem can obviously be described as a state-space traversal — a state is described by a pair $\langle x, y\rangle$ where x represents the amount of water in the 4-gallon jug and y represents the amount of water in the 3-gallon jug. The start-state then is $\langle 0, 0\rangle$ and the goal-state is any pair where the first component equals 2. First of all some transformations between states must be formulated. The following is by no means a complete set of transformations but it turns out that there is no need for additional ones:

- empty the 4-gallon jug if it is not already empty;
- empty the 3-gallon jug if it is not already empty;
- fill up the 4-gallon jug if it is not already full;
- fill up the 3-gallon jug if it is not already full;
- if there is enough water in the 3-gallon jug, use it to fill up the 4-gallon jug until it is full;
- if there is enough water in the 4-gallon jug, use it to fill up the 3-gallon jug until it is full;
- if there is room in the 4-gallon jug, pour all water from the 3-gallon jug into it;
- if there is room in the 3-gallon jug, pour all water from the 4-gallon jug into it.

It is now possible to express these actions as a binary relation between two states. The binary functor :/2 (written in infix notation) is used to represent a pair:

$$\begin{array}{l}
action(X:Y, 0:Y) \leftarrow X > 0.\\
action(X:Y, X:0) \leftarrow Y > 0.\\
action(X:Y, 4:Y) \leftarrow X < 4.\\
action(X:Y, X:3) \leftarrow Y < 3.\\
action(X:Y, 4:Z) \leftarrow X < 4, Z \ is \ Y - (4 - X), Z \geq 0.\\
action(X:Y, Z:3) \leftarrow Y < 3, Z \ is \ X - (3 - Y), Z \geq 0.\\
action(X:Y, Z:0) \leftarrow Y > 0, Z \ is \ X + Y, Z \leq 4.\\
action(X:Y, 0:Z) \leftarrow X > 0, Z \ is \ X + Y, Z \leq 3.
\end{array}$$

The definition of a path is based on the program in Example 11.3. However, since the goal-state is known to be $\langle 2, X\rangle$ for any value of X (or at least $0 \leq X \leq 3$) there is no need for the second argument of *path*/4. With some minor additional changes the final version looks as follows:

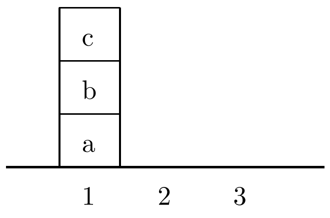
$$\begin{array}{l}
path(X) \leftarrow\\
\qquad path(0:0, [0:0], X).\\
\\
path(2:X, Visited, Visited).\\
path(State, Visited, Path) \leftarrow\\
\qquad action(State, NewState),\\
\qquad not\ member(NewState, Visited),\\
\qquad path(NewState, [NewState|\,Visited], Path).\\
\\
member(X, [X|Y]).\\
member(X, [Y|Z]) \leftarrow\\
\qquad member(X, Z).
\end{array}$$

Given this program and the goal $\leftarrow path(X)$ several answers are obtained some of which are rather naive. One answer is $X = [2:0, 0:2, 4:2, 3:3, 3:0, 0:3, 0:0]$. That is, first fill the 3-gallon jug and pour this water into the 4-gallon jug. Then the 3-gallon jug is filled again, and the 4-gallon jug is filled with water from the 3-gallon jug. The last actions are to empty the 4-gallon jug and then pour the content of the 3-gallon jug into it. Another answer is $X = [2:0, 0:2, 4:2, 3:3, 3:0, 0:3, 4:3, 4:0, 0:0]$. In all 27 answers are produced.

11.4 Blocks World (Extended Example)

A similar problem is the so-called *blocks world*. Consider a table with three distinct positions. On the table are a number of blocks which may be stacked on top of each other. The aim is to move the blocks from a given start-state to a goal-state. Only blocks which are free (that is, with no other block on top of them) can be moved.

The first step is to determine how to represent the state. A reasonable solution is to use a ternary functor *state*/3 to represent the three positions of the table. Furthermore, use the constant *table* to denote the table. Finally represent by $on(X, Y)$ the fact that X is positioned on top of Y. That is, $state(on(c, on(b, on(a, table))), table, table)$ represents the state:



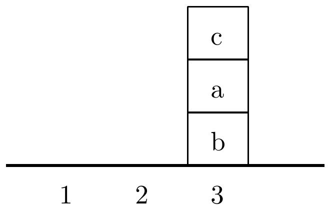
The following are all possible actions that transform the state:

- if the first position is nonempty the topmost block can be moved to either the second or the third position;
- if the second position is nonempty the topmost block can be moved to either the first or the third position;
- if the third position is nonempty the topmost block can be moved to either the first or the second position.

The first action may be formalized as follows:

$$move(state(on(X, NewX), OldY, Z), state(NewX, on(X, OldY), Z)).$$
$$move(state(on(X, NewX), Y, OldZ), state(NewX, Y, on(X, OldZ))).$$

The remaining two actions may be formalized in a similar way. Finally the program is completed by adding the path-program from Example 11.3 (where *edge*/2 is renamed into *move*/2). It is now possible to find the path from the start-state above to the following goal-state:



by giving the goal:

$$\begin{aligned} \leftarrow path(\quad & state(on(c, on(b, on(a, table))), table, table), \\ & state(table, table, on(c, on(a, on(b, table)))), \quad X). \end{aligned}$$

One answer to the goal is:

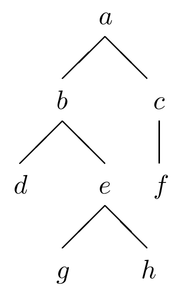
$$\begin{aligned} X = [\quad & state(table, table, on(c, on(a, on(b, table)))), \\ & state(table, on(c, table), on(a, on(b, table))), \\ & state(on(a, table), on(c, table), on(b, table)), \\ & state(on(b, on(a, table)), on(c, table), table), \\ & state(on(c, on(b, on(a, table))), table, table) \quad] \end{aligned}$$

That is, first move c to position 2. Then move b to position 3 and a on top of b. Finally move c on top of a.

11.5 Alternative Search Strategies

For many problems the depth-first traversal of a state-space is sufficient as shown above — the depth-first strategy is relatively simple to implement and the memory requirements are relatively modest. However, sometimes there is need for alternative search strategies — the depth-first traversal may be stuck on an infinite path in the state-space although there are finite paths which lead to (one of) the goal-states. Even worse, sometimes the branching of the state-space is so huge that it is simply not feasible to try all possible paths — instead one has to rely on heuristic knowledge to reduce the number of potential paths. When describing such problems by means of logic programming there are two solutions to this problem — either the logic program (for instance that in Example 11.1) is given to an inference system which employs the desired search strategy; or one writes a Prolog program which solves the problem using the desired strategy (however, this program is of course executed using the standard depth-first technique of Prolog). In this section an example of a Prolog program which searches a (simple) tree using a breadth-first traversal is shown.

Example 11.4 Consider the following tree:



To look for a path in a tree (or a graph) using a breadth-first strategy means first looking at all paths of length 1 from the start-state (or *to* the goal-state). Then all paths of length 2 are investigated. The process is repeated until a complete path from the start- to a goal-state is found.

The tree above will be represented using a binary predicate symbol $children/2$ where the first argument is the name of a node of the tree and the second argument is the names of the children of that node. Hence the tree above is represented as follows:

$$\begin{array}{l} children(a, [b, c]). \\ children(b, [d, e]). \\ children(c, [f]). \\ children(e, [g, h]). \end{array}$$

In order to realize the breadth-first strategy paths will be represented by “reversed” lists of nodes. For instance, $[d, b, a]$ represents the path which starts at the root of the tree and proceeds to d via the intermediate node b. Given all paths of length n from a start-state the problem of finding a path to a goal-state reduces to finding a path from the end of one of these paths to the goal-state. Initially this amounts to finding a path from the empty branch $[X]$ (where X is the start-state) to a goal-state. That is:

$$path(X,Y) \leftarrow bf\_path([[X]],Y).$$

The first argument of *bf_path*/2 consists of a collection of paths (represented by a list) and the second argument is the goal-state (this may easily be generalized to several goal-states):

$$
\begin{array}{l}
bf\_path([\,[Leaf|Branch]\mid Branches\,], Leaf).\\
bf\_path([\,[Leaf|Branch]\mid Branches\,], Goal) \leftarrow\\
\quad children(Leaf, Adjacent),\\
\quad expand([Leaf|Branch], Adjacent, Expanded),\\
\quad append(Branches, Expanded, NewBranches),\\
\quad bf\_path(NewBranches, Goal).\\
bf\_path([\,[Leaf|Branch]\mid Branches\,], Goal) \leftarrow\\
\quad not\ children(Leaf, Leaves),\\
\quad bf\_path(Branches, Goal).
\end{array}
$$

The last clause exploits unsafe use of negation and applies when a path cannot be expanded any further. Notice, that in order to implement a breadth-first search, it is vital that *Expanded* is appended *to Branches*. The other way around would lead to a depth-first search. Thus, the first argument of *bf_path*/2 behaves as a FIFO-queue where a prefix contains paths of length n and where the rest of the queue contains paths of length $n+1$. *expand*/3 describes the relation between a path X, the children $X_1, \ldots, X_n$ of the final node in X and the paths obtained by adding X_1, X_2, etc. to the end of X:

$$
\begin{array}{l}
expand(X, [\,], [\,]).\\
expand(X, [Y|Z], [\,[Y|X]\mid W\,]) \leftarrow\\
\quad expand(X, Z, W).
\end{array}
$$

For instance, the goal $\leftarrow expand([b,a],[d,e],X)$ succeeds with the answer $X = [[d,b,a], [e,b,a]]$.

When extended with the usual definition of *append*/3 the goal $\leftarrow path(a,X)$ yields all eight possible solutions. That is, a, b, c, d, e, f, g and h. ∎

Exercises

11.1 A chessboard of size $N \times N$ is given — the problem is to move a knight across the board in such a way that every square on the board is visited exactly once. The knight may move only in accordance with the standard chess rules.

11.2 A farmer, a wolf and a goat are standing on the same river-bank accompanied by a cabbage-head (a huge one!). A boat is available for transportation. Unfortunately, it has room for only two individuals including the cabbage-head. To complicate things even more (1) the boat can be operated only by the farmer and (2) if the goat is left alone with the wolf it will be eaten. Similarly if the cabbage-head is left alone with the goat. Is there some way for them to cross the river without anyone being eaten?

11.3 Three missionaries and three cannibals are standing on the same side of a river. A boat with room for two persons is available. If the missionaries on either side of the river are outnumbered by cannibals they will be done away with. Is there some way for all missionaries and cannibals to cross the river without anyone being eaten?

First solve the problem using a depth-first search strategy. (In which case a log must be used to prune infinite paths.) Then solve the same problem using a breadth-first strategy similar to that on page 186.

11.4 (Towers of Hanoi) Three pins are available together with N disks of different sizes. Initially all disks are stacked (smaller on top of bigger) on the leftmost pin. The task is to move all disks to the rightmost pin. However, at no time may a disk be on top of a smaller one. Hint: there is a very simple and efficient algorithmic solution to this puzzle. However, it may also be solved with the techniques described above.

11.5 How can the program in Example 11.4 be modified to avoid the use of negation?

PART III

ALTERNATIVE LOGIC PROGRAMMING SCHEMES

Chapter 12

Logic Programming and Concurrency

12.1 Algorithm = Logic + Control

The construction of a computer program can be divided into two phases which are usually intertwined — the formulation of the actual problem (*what* the problem is) and the description of *how* to solve the problem. Together they constitue an algorithm. This idea is the heart of logic programming — the logic provides a description of the problem and SLD-resolution provides the means for executing the description. However, the logic has a meaning in itself — its declarative semantics — which is independent of any particular execution strategy. This means that, as long as the inference mechanism is sound, the behaviour of the algorithm may be altered by selecting an alternative inference mechanism. We do not have to go as far as abandoning SLD-resolution — the behaviour of the execution can be altered simply by choosing different computation rules as illustrated by the following example:

Example 12.1 The following execution trace illustrates the impact of a more versatile computation rule than the one used in Prolog. First consider the program:

$$\begin{array}{ll}
(1) & \mathit{append}([\,], X, X). \\
(2) & \mathit{append}([X|Y], Z, [X|W]) \leftarrow \mathit{append}(Y, Z, W). \\
 & \\
(3) & \mathit{succlist}([\,], [\,]). \\
(4) & \mathit{succlist}([X|Y], [Z|W]) \leftarrow \mathit{succlist}(Y, W), Z \ \mathit{is}\ X + 1.
\end{array}$$

Then consider the following SLD-derivation whose initial goal consists of two components (subgoals). Each of the subsequent goals in the derivation can be divided into two halves originating from the components of the initial goal as visualized by the frames:

$$\leftarrow \boxed{append([4,5],[3],X)}\,\boxed{succlist(X,Res)} \qquad (G_0)$$

Resolving $append([4,5],[3],X)$ using (2) yields the binding $[4|W_0]$ for X and the new goal:

$$\leftarrow \boxed{append([5],[3],W_0)}\,\boxed{succlist([4|W_0],Res)} \qquad (G_1)$$

Resolving $succlist([4|W_0],Res)$ using (4) yields the binding $[Z_1|W_1]$ for Res and the goal:

$$\leftarrow \boxed{append([5],[3],W_0)}\,\boxed{succlist(W_0,W_1), Z_1\ is\ 4+1} \qquad (G_2)$$

Resolving $append([5],[3],W_0)$ using (2) yields the binding $[5|W_2]$ for W_0:

$$\leftarrow \boxed{append([\,],[3],W_2)}\,\boxed{succlist([5|W_2],W_1), Z_1\ is\ 4+1} \qquad (G_3)$$

Selection of $Z_1\ is\ 4+1$ binds Z_1 to 5. Consequently Res is bound to $[5|W_1]$:

$$\leftarrow \boxed{append([\,],[3],W_2)}\,\boxed{succlist([5|W_2],W_1)} \qquad (G_4)$$

Resolving $succlist([5|W_2],W_1)$ using (4) yields the binding $[Z_4|W_4]$ for W_1:

$$\leftarrow \boxed{append([\,],[3],W_2)}\,\boxed{succlist(W_2,W_4), Z_4\ is\ 5+1} \qquad (G_5)$$

Resolving $append([\,],[3],W_2)$ using (1) binds W_2 to $[3]$:

$$\leftarrow \boxed{}\,\boxed{succlist([3],W_4), Z_4\ is\ 5+1} \qquad (G_6)$$

Selection of $Z_4\ is\ 5+1$ binds Z_4 to 6 and Res is bound to $[5,6|W_4]$:

$$\leftarrow \boxed{}\,\boxed{succlist([3],W_4)} \qquad (G_7)$$

In the next step W_4 is bound to $[Z_7|W_7]$ yielding:

$$\leftarrow \boxed{}\,\boxed{succlist([\,],W_7), Z_7\ is\ 3+1} \qquad (G_8)$$

Selection of $succlist([\,],W_7)$ binds W_7 to $[\,]$:

$$\leftarrow \boxed{}\,\boxed{Z_7\ is\ 3+1} \qquad (G_9)$$

Finally Z_7 is bound to 4 yielding a refutation where the binding for Res is the list $[5,6,4]$. ∎

Thus, 10 SLD-steps are needed to refute the initial goal. Notice that it is not possible to improve on this by choosing an alternative computation rule — no matter what rule is used, the refutation will have the length 10.

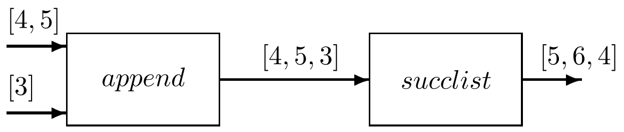


Figure 12.1: Process-interpretation of G_0

With this versatile computation rule the two subgoals in G_0 may be viewed as two *processes* which communicate with each other using the (shared) variable X. The frames in the derivation capture the internal behaviour of the processes and the shared variable acts as a "communication channel" where a stream of data flows — namely the elements of the list to which the shared variable X is (incrementally) bound to (first 4 then 5 and finally 3). See Figure 12.1.

Notice that there is no real parallelism in this example. The executions of the two processes are only interleaved with each other. The control is merely shifted between them and there is no real gain in performance. This type of control is commonly known as *coroutining*.

The possibility of viewing subgoals as *processes* and goals as *nets of communicating processes* connected by means of shared variables implies yet another interpretation of logic programs in addition to its operational and declarative meaning. This new view of logic programming extends the possible application areas of logic programming to include also process programming (like operating systems, simulators or industrial process control systems).

12.2 And-parallelism

Instead of solving the subgoals in a goal in sequence (using SLD-resolution) it is possible to use an operational semantics where some of the subgoals are solved in parallel. This is commonly called AND-parallelism. However, since the subgoals may contain shared variables it is not always feasible to solve all of them independently. Consider the following program:

$$\begin{array}{l} \mathit{do\_this}(a). \\ \mathit{do\_that}(b). \end{array}$$

A goal of the form:

$$\leftarrow \mathit{do\_this}(X), \mathit{do\_that}(X).$$

would fail using SLD-resolution. However, the two subgoals are solvable separately. The leftmost subgoal binds X to a and the rightmost binds X to b. When two subgoals contain a shared variable special care must be taken so that different occurrences of the variable do not get bound to inconsistent values. This calls for some form of *communication/synchronization* between the subgoals. However there are some special cases when two or more derivation-steps can be carried out independently:

- when the subgoals have no shared variable, and

- when at most one subgoal binds each shared variable.

Consider the derivation in Example 12.1 again. Note that both subgoals in G_1 can be resolved in parallel since the shared variable (W_0) is bound only by $\mathit{append}([5],[3],W_0)$. Similarly, all subgoals in G_3, G_6 and G_8 may be resolved in parallel. Thus, by exploiting AND-parallelism the goal may be solved in only five steps ($G_0, G_1, G_3, G_6, G_8, G_{10}$) reducing the (theoretical) time of execution by 50%.

12.3 Producers and Consumers

One point worth noticing about Example 12.1 is that the execution is completely determinate — no selected subgoal unifies with more than one clause-head. However, this is not necessarily the case if some other computation rule is employed. For instance, the rightmost subgoal in G_0 unifies with two different clauses. To reduce the search space it is desirable to have a computation rule which is "as determinate as possible". Unfortunately it is rather difficult (if at all possible) to implement a computation rule which always selects a determinate subgoal. However, the programmer often has some idea how the program should be executed to obtain good efficiency (although not always optimal). Hence the programmer may be allowed to provide additional information describing *how* the program should be executed.

The subgoal $\mathit{append}([4,5],[3],X)$ in G_0 may be viewed as a *process* which consumes input from two streams ($[4,5]$ and $[3]$) and acts as a *producer* of bindings for the variable X. Similarly $\mathit{succlist}(X, \mathit{Res})$ may be viewed as a *process* which *consumes* bindings for the variable X and produces a stream of output for the variable Res. Since, in general, it is not obvious which subgoals are intended to act as consumers and producers of shared variable-occurrences the user normally has to provide a declaration. For instance, that the first two arguments of $\mathit{append}/3$ act as consumers and the third as producer. There are several ways to provide such information. In what follows we will use a notion of *read-only* variable which very closely resembles that employed in languages such as Concurrent Prolog (one of the first and most influential languages based on a concurrent execution model).

Roughly speaking, each clause (including the goal) may contain several occurrences of a variable. On the other hand, variables can be bound at most once in an SLD-derivation. This implies that at most one of the atoms in a clause acts as a producer of a binding for that variable whereas all remaining atoms containing some occurrence of the variable act as consumers. The idea employed in Concurrent Prolog is that the user annotates variable-occurrences appearing in consumer atoms by putting a question mark immediately after each occurrence. For instance, G_0 may be written as follows:

$$\leftarrow \mathit{append}([4,5],[3],X), \mathit{succlist}(X?, \mathit{Res}).$$

This means that the call to $\mathit{append}/3$ acts as a producer of values for X and that $\mathit{succlist}/2$ acts as a consumer of values for X and a producer of values for Res. Variables annotated by '?' are called *read-only* variables. Variables which are not annotated are said to be *write-enabled*.

Now what is the meaning of a read-only variable? From a declarative point of view they are not different from write-enabled occurrences of the same variable. Consequently, the question-mark can be ignored in which case X and $X?$ denote the same variable. However, from an operational point of view X and $X?$ behave differently. The role of $X?$ is to *suspend* unification temporarily if it is not possible to unify a subgoal with a clause head without producing a binding for $X?$. The unification can be resumed only when the variable X is bound to a non-variable by some other process containing a write-enabled occurrence of the variable. For instance, unification of $p(X?)$ and $p(f(Y))$ suspends whereas unification of $p(Y?)$ and $p(X)$ succeeds with mgu $\{X/Y?\}$.

Application of a substitution θ to a term or a formula is defined as before except that $X?\theta = (X\theta)?$. Consequently, a read-only annotation may appear after non-variable terms. In this case the annotation has no effect and can simply be removed. For instance:

$$\begin{aligned} p(X?, Y, Z?)\{X/f(W), Y/f(W?)\} &= p(f(W)?, f(W?), Z?) \\ &= p(f(W), f(W?), Z?) \end{aligned}$$

Example 12.2 Consider the following (nonterminating) program describing the *producer-consumer* problem with an unbounded buffer. That is, there is a producer which produces data and a consumer which consumes data and we require that the consumer does not attempt to consume data which is not there:

$$\begin{aligned} &\mathit{producer}([X|Y]) \leftarrow \mathit{get}(X), \mathit{producer}(Y). \\ &\mathit{consumer}([X|Y]) \leftarrow \mathit{print}(X?), \mathit{consumer}(Y?). \end{aligned}$$

We do not specify exctly how $\mathit{get}/1$ and $\mathit{print}/1$ are defined but only assume that the call $\mathit{get}(X)$ suspends until some data (e.g. a text file) is available from the outside whereas $\mathit{print}(X)$ is a printer-server which prints the file X. To avoid some technical problems we also assume that a call to $\mathit{producer}/1$ (resp. $\mathit{consumer}/1$) does not go ahead until $\mathit{get}/1$ (resp. $\mathit{print}/1$) succeeds.

Now consider the goal:

$$\leftarrow \mathit{producer}(X), \mathit{consumer}(X?).$$

Because of the read-only annotation the second subgoal suspends. The first subgoal unifies with $\mathit{producer}([X_0|Y_0])$ resulting in the mgu $\{X/[X_0|Y_0]\}$ and the new goal:

$$\leftarrow \mathit{get}(X_0), \mathit{producer}(Y_0), \mathit{consumer}([X_0|Y_0]).$$

At this point only the third subgoal may proceed. (The first subgoal suspends until some external data becomes available and the second subgoal suspends until the first subgoal succeeds.)

$$\leftarrow \mathit{get}(X_0), \mathit{producer}(Y_0), \mathit{print}(X_0?), \mathit{consumer}(Y_0?).$$

This goal suspends until an external job arrives (to avoid having to consider how jobs are represented we just denote them by job_n). Assume that job_1 eventually arrives in which case $\mathit{get}(X_0)$ succeeds with X_0 bound to job_1:

$$\leftarrow \mathit{producer}(Y_0), \mathit{print}(\mathit{job}_1), \mathit{consumer}(Y_0?).$$

Then assume that the $\mathit{producer}/1$-process is reduced to:

$$\leftarrow \mathit{get}(X_1), \mathit{producer}(Y_1), \mathit{print}(\mathit{job}_1), \mathit{consumer}([X_1|Y_1]).$$

and that a new job arrives from outside while the first is being printed:

$$\leftarrow producer(Y_1), print(job_1), consumer([job_2 \,|Y_1]).$$

The $producer/1$-process can now be reduced to:

$$\leftarrow get(X_2), producer(Y_2), print(job_1), consumer([job_2, X_2|Y_2]).$$

The whole goal suspends again until either (1) a new job arrives (in which case the new job is enqueued after the second job) or (2) printing of the first job ends (in which case the second job can be printed). As pointed out above the program does not terminate. ∎

Notice that it may happen that all subgoals in a goal become suspended forever. A trivial example is the goal $\leftarrow consumer(X?)$. This situation is called *deadlock*.

12.4 Don't Care Nondeterminism

The Prolog computation consists of a traversal of the SLD-tree. The branching of the tree occurs when the selected subgoal matches several clause heads. To be sure that no refutations are disregarded, a backtracking strategy is employed. Informally the system "does not know" how to obtain the answers so all possibilities are tried (unless, of course, the search gets stuck on some infinite branch). This is sometimes called "don't know nondeterminism".

The traversal of the tree may be carried out in parallel. This is commonly called *OR-parallelism.* Notice that OR-parallelism does not necessarily speed up the discovery of a particular answer.

Although full OR-parallelism may be combined with AND-parallelism this is seldom done because of implementation difficulties. Instead a form of limited OR-parallelism is employed. The idea is to *commit* to a single clause as soon as possible when trying to solve a literal. Informally this means that all parallel attempts to solve a subgoal are immediately surrendered when the subgoal unifies with the head of some clause and certain subgoals in that clause are solved. To make this more precise the concept of *commit operator* is introduced. The commit operator divides the body of a clause into a *guard*- and *body*-part. The commit-operator may be viewed as a generalized cut operator in the sense that it cuts off all other attempts to solve a subgoal. This scheme is usually called "don't care nondeterminism". Intuitively this can be understood as follows — assume that a subgoal can be solved using several different clauses all of which lead to the same solution. Then it does not matter which clause to pick. Hence, it suffices to pick *one* of the clauses not caring about the others. Of course, in general it is not possible to tell whether all attempts will lead to the same solution and the responsibility has to be left to the user.

12.5 Concurrent Logic Programming

The concepts discussed above provide a basis for a class of programming languages based on logic programming. They are commonly called Concurrent Logic Programming languages or Committed Choice Languages. For the rest of this chapter the principles of these languages are discussed. To illustrate the principles we use a language similar to Shapiro's Concurrent Prolog (1983a).

By analogy to definite programs, the programs considered here are finite sets of *guarded clauses*. The general scheme of a guarded clause is as follows:

$$H \leftarrow G_1, \ldots, G_m \mid B_1, \ldots, B_n \qquad (m \geq 0, n \geq 0)$$

where $H, G_1, \ldots, G_m, B_1, \ldots, B_n$ are atoms (possibly containing read-only annotations). H is called the *head* of the clause. $G_1, \ldots, G_m$ and $B_1, \ldots, B_n$ are called the *guard* and the *body* of the clause. The symbol "|" which divides the clause into a guard- and body-part is called the *commit operator*. If the guard is empty the commit operator is not written out. To simplify the operational semantics of the language, guards are only allowed to contain certain predefined *test-predicates* — typically arithmetic comparisons. Such guards are usually called *flat* and the restriction of Concurrent Prolog which allows only flat guards is called Flat Concurrent Prolog (FCP).

Like definite programs, FCP-programs are used to produce bindings for variables in goals given by the user. The initial goal is not allowed to contain any guard.

Example 12.3 The following are two examples of FCP-programs for merging lists and deleting elements from lists:

$$\begin{array}{l}
merge([\,],[\,],[\,]).\\
merge([X|Y], Z, [X|W]) \leftarrow merge(Y?, Z, W).\\
merge(X, [Y|Z], [Y|W]) \leftarrow merge(X, Z?, W).\\
\\
delete(X, [\,], [\,]).\\
delete(X, [X|Y], Z) \leftarrow delete(X, Y?, Z).\\
delete(X, [Y|Z], [Y|W]) \leftarrow X \neq Y \mid delete(X, Z?, W).
\end{array}$$

∎

Like definite clauses, guarded clauses have a logical reading:

- all variables in a guarded clause are implicitly universally quantified — the read-only annotations have no logical meaning;
- "←" denotes logical implication;
- "|" and "," denote logical conjunctions.

Each clause of the program must contain exactly one commit operator (although usually not explicitly written when the guard-part is empty). Operationally it divides the right-hand side of a clause into two parts which are solved strictly in sequence. Before starting solving the body the whole guard must be solved. Literals in the guard and body are separated by commas. Operationally this means that the literals may be solved in parallel.

The notion of derivation basically carries over from definite programs. However, the read-only annotations and commit operators impose certain restrictions on the selection of a subgoal in a derivation step. This is because some subgoals may be temporarily suspended. There are three reasons for this — either because (1) unification of a subgoal with a clause head cannot be performed without binding read-only variables or (2) the subgoal appears in the body of a guarded clause whose guard is not yet satisfied or (3) the subgoal is a non-ground test-predicate.

To describe the basic derivation-step taking these restrictions into account the goal will be partitioned into groups of guards and bodies. To emphasize this the goal will be written as follows:

$$\leftarrow \boxed{G_1 \mid B_1}, \ldots, \boxed{G_i \mid B_i}, \ldots, \boxed{G_n \mid B_n}.$$

where both G_j and B_j, $(1 \leq j \leq n)$, are possibly empty conjunctions of atoms (in case of G_j containing only test-predicates). A single reduction of the goal then amounts to selecting some subgoal A such that either:

(*i*) A is a test-predicate in G_i or B_i (if G_i is empty) which is both ground and true. The new goal is obtained by removing A from the goal.

(*ii*) G_i is empty, A appears in B_i and is a user-defined predicate and there is a (renamed) guarded clause of the form:

$$H \leftarrow G_m \mid B_m.$$

such that A and H unify (with mgu θ) without binding any read-only variables. The new goal obtained is:

$$(\leftarrow \boxed{G_1 \mid B_1}, \ldots, \boxed{B_i \setminus A}, \boxed{G_m \mid B_m}, \ldots, \boxed{G_n \mid B_n}\ )\theta.$$

where $B_i \setminus A$ denotes the result of removing A from B_i.

A successful derivation is one where the final goal is empty.

Like SLD-resolution this scheme contains several nondeterministic choices — many subgoals may be selected and if the subgoal selected is user-defined there may be several guarded clauses which unify with it. In the latter case the commit operator has an effect similar to that of cut. In order to solve a subgoal several clauses are tried in parallel. However, as soon as the subgoal unifies with *one* of the clauses and succeeds in solving its guard, all other attempts to solve the subgoal are immediately surrendered. Thus, the commit operator behaves as a kind of symmetric cut. For instance, take Example 12.3 and the goal:

$$\leftarrow merge([a, b], [c, d], X).$$

This goal has many solutions in Prolog. In FCP there is only one solution to the goal. The result depends on what clauses the refutation commits to.

Since each clause is required to contain exactly one commit operator no goal can have more than one solution. Thus, it is not possible to use *append*/3 to generate splittings of a list. At most one solution will be found. This is one of the main disadvantages of this class of languages. However, this is the price that has to be paid in order to be able to implement these languages efficiently. Note that it is vital to test for inequality in the guard of the last clause of the *delete*/3-program for this reason. If the test is moved to the body it may happen that goals fail because of committing to the third clause instead of the second.

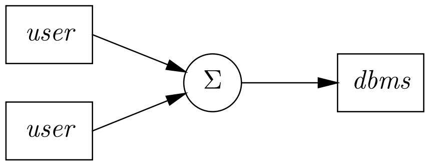


Figure 12.2: Transaction system

The execution model given above is somewhat simplified since at each step only one subgoal is selected. As already mentioned, languages like FCP support AND-parallelism which means that several subgoals may be selected simultaneously. However, incorporating this extra dimension into the execution model above makes it rather complicated and we will therefore stick to the sequential version which simulates parallelism through coroutining.

The chapter is concluded with an example of a CLP program that implements a simple database system with a fixed number of clients.

Example 12.4 Consider an application involving a database transaction system. Such a system consists of some processes where customers (users) input transactions and a database management system (DBMS) performs the transactions using a database and outputs the results to the user. See Figure 12.2.

To start up such a system of processes the following goal may be given (if we restrict ourselves to two users of the database system):

$$\leftarrow user(tty_1, X), user(tty_2, Y), merge(X?, Y?, Z), dbms(Z?, [\,]).$$

A formal definition of the $user/2$-process will not be provided. Informally the process $user(tty_n, X)$ is assumed to behave as follows:

(i) It suspends until the arrival of a message from the terminal named tty_n;

(ii) When a message M arrives it binds X to the pair $[M|Msgs]$ where $Msgs$ is a new variable. (For a description of all possible messages see Figure 12.3.);

(iii) Then the process suspends until M becomes ground;

(iv) When M becomes ground it prints a message on tty_n;

(v) Finally it calls itself with $user(tty_n, Msgs)$.

Thus, the two $user/2$-processes generate two (infinite) streams of transactions which are merged nondeterministically by the $merge/3$-process and the resulting stream is processed (and grounded) by the $dbms/2$-system.

For the sake of simplicity, assume that the database consists of a list of pairs of the form $item(key, value)$ where key is a unique identifier and $value$ is the data associated with the key. Three different transactions are to be considered — a pair may be (1) added to the database, (2) deleted from the database, and (3) retrieved from the database. On the top level the database management system may be organized as follows (the first clause is not needed in this version of the program but is added as a hint to exercise 12.3):

Transactions	Description
$add(key, value, Reply)$	This message represents an external request to add a new item of the form $item(key, value)$ to the database. The first two arguments are ground and *Reply* a variable which eventually is grounded by the DBMS.
$del(key, Reply)$	This message represents an external request to delete an item of the form $item(key, \_)$ from the database. The first argument is ground and *Reply* a variable which eventually is grounded by the DBMS.
$in(key, Reply)$	This message represents an external request to retrieve the value stored in the item of the form $item(key, \_)$ from the database. The first argument is ground and *Reply* a variable which eventually is grounded by the DBMS.

Figure 12.3: Description of database transactions

$$
\begin{array}{l}
dbms([kill|Nxt], Db).\\
dbms([in(Key, Reply)|Nxt], Db) \leftarrow\\
\qquad retrieve(Key, Db, Reply),\\
\qquad dbms(Nxt?, Db).\\
dbms([add(Key, Val, Reply)|Nxt], Db) \leftarrow\\
\qquad insert(Key, Val, Db?, NewDb, Reply),\\
\qquad dbms(Nxt?, NewDb?).\\
dbms([del(Key, Reply)|Nxt], Db) \leftarrow\\
\qquad delete(Key, Db?, NewDb, Reply),\\
\qquad dbms(Nxt?, NewDb?).
\end{array}
$$

If the first transaction appearing in the stream is a request to retrieve information from the database, $dbms/2$ invokes the procedure $retrieve/3$. The first argument is the key sought for, the second argument is the current database and the third argument is the value associated with the key (or *not_found* if the key does not appear in the database).

$$
\begin{array}{l}
retrieve(Key, [\,], not\_found).\\
retrieve(Key, [item(Key, X)|Db], X).\\
retrieve(Key, [item(K, Y)|Db], X) \leftarrow\\
\qquad Key \neq K \mid retrieve(Key, Db, X).
\end{array}
$$

If the first transaction is a request to add a new key/value-pair to the database the data is stored in the database by means of the predicate $insert/5$. The first and second arguments are the key and the associated value, the third argument is the current database, the fourth argument is the new database after adding the pair and the final argument returns a reply to the user (since the operation always succeeds the reply is always *done*).

$$\begin{array}{l} \mathit{insert}(\mathit{Key}, X, [\,], [\mathit{item}(\mathit{Key}, X)], \mathit{done}). \\ \mathit{insert}(\mathit{Key}, X, [\mathit{item}(\mathit{Key}, Y)|\mathit{Db}], [\mathit{item}(\mathit{Key}, X)|\mathit{Db}], \mathit{done}). \\ \mathit{insert}(\mathit{Key}, X, [\mathit{item}(K, Y)|\mathit{Db}], [\mathit{item}(K, Y)|\mathit{NewDb}], \mathit{Reply}) \leftarrow \\ \qquad \mathit{Key} \neq K \mid \mathit{insert}(\mathit{Key}, X, \mathit{Db}, \mathit{NewDb}, \mathit{Reply}). \end{array}$$

Notice that *insert*/5 either adds the pair at the very end of the database or, if the key is already used in the database, replaces the old value associated with the key by the new value. If the latter is not wanted an error message may be returned instead by simple modifications of the second clause.

The last transaction supported is the removal of information from the database. This is taken care of by the predicate *delete*/4. The first argument is the key of the pair to be removed from the database — the second argument. The third argument will be bound to the new database and the fourth argument records the result of the transaction (*done* if the key was found and *not_found* otherwise):

$$\begin{array}{l} \mathit{delete}(\mathit{Key}, [\,], [\,], \mathit{not\_found}). \\ \mathit{delete}(\mathit{Key}, [\mathit{item}(\mathit{Key}, Y)|\mathit{Db}], \mathit{Db}, \mathit{done}). \\ \mathit{delete}(\mathit{Key}, [\mathit{item}(K, Y)|\mathit{Db}], [\mathit{item}(K, Y)|\mathit{NewDb}], \mathit{Reply}) \leftarrow \\ \qquad \mathit{Key} \neq K \mid \mathit{delete}(\mathit{Key}, \mathit{Db}, \mathit{NewDb}, \mathit{Reply}). \end{array}$$

We conclude the example by considering an outline of an execution trace of the goal:

$$\leftarrow \mathit{user}(\mathit{tty}_1, X), \mathit{user}(\mathit{tty}_2, Y), \mathit{merge}(X?, Y?, Z), \mathit{dbms}(Z?, [\,]).$$

Initially, all subgoals are suspended. Now assume that the first *user*/2-process binds X to $[\mathit{add}(\mathit{k10}, \mathit{john}, R)|X_0]$:

$$\leftarrow \ldots, \mathit{merge}([\mathit{add}(\mathit{k10}, \mathit{john}, R)|X_0], Y?, Z), \mathit{dbms}(Z?, [\,]).$$

Then *merge*/3 can be resumed binding Z to $[\mathit{add}(\mathit{k10}, \mathit{john}, R)|W_1]$ and the new goal becomes:

$$\leftarrow \ldots, \mathit{merge}(X_0?, Y?, W_1), \mathit{dbms}([\mathit{add}(\mathit{k10}, \mathit{john}, R)|W_1], [\,]).$$

At this point *dbms*/2 is resumed reducing the goal to:

$$\leftarrow \ldots, \mathit{merge}(\ldots), \mathit{insert}(\mathit{k10}, \mathit{john}, [\,], D, R), \mathit{dbms}(W_1?, D?).$$

The call to *insert*/5 succeeds binding D to $[\mathit{item}(\mathit{k10}, \mathit{john})]$ and R to *done* (the reply *done* is echoed on tty_1):

$$\leftarrow \ldots, \mathit{merge}(X_0?, Y?, W_1), \mathit{dbms}(W_1?, [\mathit{item}(\mathit{k10}, \mathit{john})]).$$

At this point both *merge*/3 and *dbms*/2 are suspended waiting for new messages from one of the terminals. Assume that the second user wants to know the value associated with the key *k10*. Then Y is bound to $[\mathit{in}(\mathit{k10}, R)|Y_2]$:

$$\leftarrow \ldots, \mathit{merge}(X_0?, [\mathit{in}(\mathit{k10}, R)|Y_2], W_1), \mathit{dbms}(W_1?, [\mathit{item}(\mathit{k10}, \mathit{john})]).$$

Next W_1 is bound to $[\mathit{in}(\mathit{k10}, R)|W_3]$ and the goal is reduced to:

$$\leftarrow \ldots, \mathit{merge}(X_0?, Y_2?, W_3), \mathit{dbms}([\mathit{in}(\mathit{k10}, R)|W_3], [\mathit{item}(\mathit{k10}, \mathit{john})]).$$

Thereafter *dbms*/2 is resumed and unified with the second clause yielding the goal:

$$\leftarrow \ldots, merge(\ldots), retrieve(k10, [item(k10, john)], R), dbms(W_3?, \ldots).$$

The call to *retrieve*/3 succeeds with R bound to *john*. The answer is echoed on tty_2 and the goal is reduced to:

$$\leftarrow \ldots, merge(X_0?, Y_2?, W_3), dbms(W_3?, [item(k10, john)]).$$

At this point the whole system is suspended until one of the users supplies another transaction. ▮

The example above illustrates one fundamental difference between sequential SLD-resolution for definite programs and concurrent execution. In the former case computations are normally finite and the program computes relations. However, in the latter case, computations may be infinite and the meaning of the program is not so easily defined in terms of relations. For instance, when giving a goal:

$$\leftarrow A_1, \ldots, A_n$$

we normally want this goal to succeed with some answer substitution. However, the goal in Example 12.4 does not terminate, yet the execution results in some useful output via side-effects (supplying transactions to the terminal and obtaining answers echoed on the screen). This fundamental difference makes more complicated to give a declarative semantics to concurrent logic programming languages like FCP.

Exercises

12.1 Write a concurrent logic program for checking if two binary trees have the same set of labels associated with the nodes of the tree. Note that the labels associated with corresponding nodes do not have to be the same.

12.2 Write a concurrent program for multiplying $N \times N$-matrices of integers (for arbitrary N's).

12.3 Suggest a way of including a "kill"-process in Example 12.4. Such a process is initially suspended but should, when it is activated, terminate all other processes in the transaction system in a controlled way.

12.4 Write a concurrent program which takes as input a stream of letters (represented by constants) and replaces all occurrences of the sequence "aa" by "a" and all occurrences of "–" by the empty string. All other letters should appear as they stand in the input.

12.5 Give a solution to the producer-consumer problem with a bounded buffer.

Chapter 13

Logic Programs with Equality

As emphasized in the previous chapters, logic programs describe relations. Of course, since a function may be viewed as a special case of a relation it is also possible to define functions as relations using logic programs. However, in this case it is usually not clear whether the described relation is a function or not (cf. Section 5.2). Furthermore, this kind of description associates functions with predicate symbols, while it would be more desirable to have functions associated with functors.

In this chapter we present a mechanism that allows us to incorporate such functional definitions into logic programming. The idea is to introduce a special binary predicate symbol "$\doteq$" — called the equality — which is to be interpreted as the identity relation on the domain of any interpretation of logic programs.

The notion of equality thus makes it possible to restrict attention to interpretations where certain terms are identified. For instance the factorial function may be defined by the following (implicitly universally quantified) equations:

$$\begin{array}{rcl}
\mathit{fac}(0) & \doteq & s(0). \\
\mathit{fac}(s(X)) & \doteq & s(X) * \mathit{fac}(X). \\
0 * X & \doteq & 0. \\
s(X) * Y & \doteq & X * Y + Y. \\
0 + X & \doteq & X. \\
s(X) + Y & \doteq & s(X + Y).
\end{array}$$

In any model of these formulas the meanings of the terms $\mathit{fac}(0)$ and $s(0)$ are the same. The use of such equations may be exploited to extend the notion of unification. Consider a definite program:

$$\begin{array}{l}
\mathit{odd}(s(0)). \\
\mathit{odd}(s(s(X))) \leftarrow \mathit{odd}(X).
\end{array}$$

The formula $\mathit{odd}(\mathit{fac}(0))$ certainly is true in the intended interpretation. However, since SLD-resolution is based on the notion of syntactic equality it is not powerful enough to produce an SLD-refutation from the definite goal:

$$\leftarrow odd(fac(0)).$$

In Section 13.3 we will see how definite programs with equality can be used to extend the notion of unification into so-called E-unification. However, before involving definite programs we study the meaning of equality axioms similar to those used above.

13.1 Equations and E-unification

In what follows an *equation* will be a formula of the form $s \doteq t$ where s and t are terms from a given alphabet. This kind of unconditional equation may be extended to *conditional* ones. That is, formulas of the form:

$$F \supset (s \doteq t)$$

where s and t are terms and F is some formula possibly containing other predicate symbols than "$\doteq$". In this chapter attention is restricted to unconditional equations. At a first glance the restriction to unconditional equations may seem to be a serious limitation, but from a theoretical point of view unconditional equations are sufficient to define any computable function (e.g. Rogers (1967)). Hence, the restriction is solely syntactic.

The intuition behind introducing the new predicate symbol "$\doteq$", is to identify terms which denote the same individual in the domain of discourse, regardless of the values of their variables. Hence, for two terms s and t, the formula $s \doteq t$ is true in an interpretation $\Im$ and valuation φ iff s and t have identical interpretations in $\Im$ and φ (that is, if $\varphi_\Im(s) = \varphi_\Im(t)$). An interpretation $\Im$ is said to be a *model* of $\forall(s \doteq t)$ if $s \doteq t$ is true in $\Im$ under any valuation φ. This extends to sets E of equations: $\Im$ is a model of a set E of equations iff $\Im$ is a model of each equation in E.

The concept of logical consequence carries over from Chapter 1 with the modification that the only interpretations that are considered are those that associate "$\doteq$" with the identity relation. Hence, given a set of equations E, $s \doteq t$ is said to be a *logical consequence* of E (denoted $E \models s \doteq t$) iff $s \doteq t$ is true in any model of E.

One of the main objectives of any logic is to permit *inference* of new formulas from old ones using a system of rewrite rules. The inference rules in Figure 13.1 defines the relation between a set of equational hypohesis E and new derived equations. (The notation $E \vdash s \doteq t$ should be read "$s \doteq t$ is derived from E".) The derivability relation induces an equivalence relation, $\equiv_E$, on the set of all terms, defined by $s \equiv_E t$ iff $E \vdash s \doteq t$. The relation is called an *equality theory*. The inference rules just introduced were shown to be both sound and complete by Birkhoff (1935):

Theorem 13.1 (Soundness and Completeness)

$$E \models s \doteq t \quad \text{iff} \quad E \vdash s \doteq t \quad \text{iff} \quad s \equiv_E t$$

∎

The notion of E-unification is defined relative to the equality theory, $\equiv_E$, induced by E and $\vdash$.

Hypothesis:

$$E \vdash s \doteq t \quad (\text{if } s \doteq t \in E)$$

Reflexivity:

$$E \vdash s \doteq s$$

Symmetry:

$$\frac{E \vdash s \doteq t}{E \vdash t \doteq s}$$

Transitivity:

$$\frac{E \vdash r \doteq s \quad E \vdash s \doteq t}{E \vdash r \doteq t}$$

Stability:

$$\frac{E \vdash s \doteq t}{E \vdash s\theta \doteq t\theta}$$

Congruence:

$$\frac{E \vdash s_1 \doteq t_1 \quad \cdots \quad E \vdash s_n \doteq t_n}{E \vdash f(s_1, \ldots, s_n) \doteq f(t_1, \ldots, t_n)}$$

Figure 13.1: Inference rules for equality

Definition 13.2 (E-unifier) Two terms, s and t, are said to be *E-unifiable* if there exists some substitution θ such that $s\theta \equiv_E t\theta$. The substitution θ is called an *E-unifier* of s and t. ▮

Example 13.3 Let E be the following equalities defining addition of natural numbers:

$$\begin{aligned} sum(0, X) &\doteq X. \\ sum(s(X), Y) &\doteq s(sum(X, Y)). \end{aligned}$$

Consider the problem of finding an E-unifier of the two terms $sum(s(X), Y)$ and $s(s(0))$. As formally shown in Figure 13.2, the two terms have at least one E-unifier — namely $\{X/0, Y/s(0)\}$. ▮

Note that in the case of the empty equality theory, $\equiv_\varnothing$ relates every term only to itself. This implies that two terms, s and t, are $\varnothing$-unifiable iff there is some substitution θ such that $s\theta$ is identical to $t\theta$. Hence, the notion of E-unification encompasses "standard" unification as a special case.

13.2 More on E-unification

As observed above, standard unification as defined in Chapter 3 is a special case of E-unification for the degenerate case when $E = \varnothing$. This suggests that it may be possible to generalize SLD-resolution into something more powerful by replacing

$$\dfrac{\dfrac{E \vdash sum(s(X),Y) \doteq s(sum(X,Y))}{E \vdash sum(s(0),s(0)) \doteq s(sum(0,s(0)))} \quad \dfrac{\dfrac{E \vdash sum(0,X) \doteq X}{E \vdash sum(0,s(0)) \doteq s(0)}}{E \vdash s(sum(0,s(0))) \doteq s(s(0))}}{E \vdash sum(s(0),s(0)) \doteq s(s(0))}$$

Figure 13.2: Proof of $E \vdash sum(s(0),s(0)) \doteq s(s(0))$

standard unification by E-unification. Such an extension is discussed in the next section but first a number of questions are raised concerning the practical problems of E-unification.

First of all, an E-unification algorithm must be provided. For the case when $E = \varnothing$ there are efficient unification algorithms available as discussed in Chapter 3. The algorithm given there has some nice properties — it always terminates and if the terms given as input to the algorithm are unifiable, it returns a most general unifier of the terms; otherwise it fails. For arbitrary sets of equations these properties are not carried over. For instance, E-unification is undecidable. That is, given an arbitrary set E of equations and two terms s and t, it is not in general possible to determine whether $s \equiv_E t$.

In addition, the algorithm of Chapter 3 is *complete* in the sense that if s and t are unifiable, then any of their unifiers can be obtained by composing the output of the algorithm with some other substitution. This is because existence of a unifier implies the existence of a *most general* one. This is not true for arbitrary sets of equations. Instead a *set* of unifiers must be considered. Before resorting to an example, some preliminaries are needed to formulate this more precisely.

A term t is said to *subsume* the term s iff there is a substitution σ such that $t\sigma \equiv_E s$. This is denoted by $s \preceq_E t$. The relation can be extended to substitutions as follows — let V be a set of variables and σ, θ substitutions. Then θ subsumes σ relative to V (denoted $\sigma \preceq_E \theta[V]$) iff $X\sigma \preceq_E X\theta$ for all $X \in V$. If V is the set of all variables in s and t, then the set S of substitutions is a *complete set of E-unifiers* of s and t iff:

- every $\theta \in S$ is an E-unifier of s and t;
- for every E-unifier σ of s and t, there exists $\theta \in S$ such that $\sigma \preceq_E \theta[V]$.

For the case when $E = \varnothing$ the standard unification algorithm produces a complete set of E-unifiers. This set is either empty (if the terms are not unifiable) or consists of a single mgu. Unfortunately, for nonempty sets E of equations, complete sets of E-unifiers may be arbitrary large. In fact, there are cases when two terms only have an infinite complete set of E-unifiers.

The following example shows two terms with two E-unifiers where the first E-unifier is not subsumed by the other and vice versa.

Example 13.4 Consider the equations in Example 13.3 again. As shown in Figure 13.2 the substitution $\theta := \{X/0, Y/s(0)\}$ is an E-unifier of the terms $sum(s(X), Y)$ and $s(s(0))$. However, also $\sigma := \{X/s(0), Y/0\}$ is a unifier of the terms. (The proof is left as an exercise). It can be shown that neither $\theta \preceq_E \sigma[\{X, Y\}]$ nor $\sigma \preceq_E \theta[\{X, Y\}]$. It can also be shown that any other E-unifier of the two terms is subsumed by one of these two substitutions. Thus, the set $\{\theta, \sigma\}$ constitutes a complete set of E-unifiers of the two terms. ∎

An E-unification algorithm is said to be *sound* if, for arbitrary terms s and t, its output is a set of E-unifiers of s and t. The algorithm is *complete* if the set in addition is a complete set of E-unifiers. Needless to say, it is desirable to have an E-unification algorithm which is at least sound and preferably complete. However, as already pointed out, there are sets of equations and pairs of terms which do not have finite sets of E-unifiers. For such cases we cannot find a complete E-unification algorithm. Thus, one must weaken the notion of completeness by saying that an algorithm is complete if it *enumerates* a complete set of E-unifiers (for arbitrary pairs of terms). Under this definition there are both sound and complete E-unification algorithms for arbitrary sets E of equations. Unfortunately they are of little practical interest because of their tendency to loop.

Thus, instead of studying general-purpose algorithms, research has concentrated on trying to find algorithms for restricted classes of equations, much like research on logic programming started with the restricted form of definite programs. Standard unification is a trivial example where no equations whatsoever are allowed.

The most well-known approach based on restricted forms of equations is called *narrowing* which, in many ways, resembles SLD-resolution. It has been shown to be both sound and complete for a nontrivial class of equational theories. Characterizing this class more exactly is outside the scope of this book.

Unfortunately narrowing also suffers from termination problems. The reason is that the algorithm does not know when it has found a complete set of unifiers. It may of course happen that this set is infinite in which case there is no hope for termination whatsoever. But even if the set is finite, the algorithm often loops since it is not possible to say whether the set found so far is a complete set of E-unifiers. Hence, in practice one has to impose some sort of restrictions not only on the form of the equations but also on the terms to be E-unified. One simple case occurs when both terms are ground. In this case either $\varnothing$ or the singleton $\{\epsilon\}$ is a complete set of E-unifiers of the terms.

13.3 Logic Programs with Equality

In this section we review the integration of definite programs and equations. It turns out that the proof-theoretic and model-theoretic semantics of this language are natural extensions of the corresponding concepts for definite programs alone. But before describing the nature of these extensions the syntax of definite programs with equations is given. Thereafter weaknesses of definite programs alone are discussed to motivate the extensions.

A *definite program with equality* is a pair P, E where:

- P is a finite set of definite clauses not containing the predicate symbol "$\doteq$";
- E is a possibly infinite set of equations.

One sometimes sees different extensions of this idea where E may contain e.g. conditional equations or where "$\doteq$" may appear in the bodies of clauses in P. What is described below can also be generalized to such programs with some additional effort.

Now, consider the following definite program P where the symbols have their natural intended interpretations:

$$\begin{array}{l} odd(1). \\ odd(X+2) \leftarrow odd(X). \end{array}$$

Although $odd(2+1)$ is true in the intended model it is not a logical consequence of the program because the program has at least one model (for instance the least Herbrand model M_P) where $odd(2+1)$ is false. It may thus be argued that the least Herbrand model is "incompatible" with the intended interpretation since the two terms $1+2$ and $2+1$ have distinct interpretations in M_P — recall that any ground term denotes itself in any Herbrand interpretation.

As pointed out above equations may be used to focus attention on certain models — namely those where some terms denote the same object. For instance, by adding to P the equation E:

$$2+1 \doteq 1+2$$

(or more generally $X+Y \doteq Y+X$) it is possible to exclude certain unwanted interpretations from being models of P and E. In particular, M_P is no longer a model of both P and E. (In fact, no Herbrand interpretation of P is a model of P and E since the terms $1+2$ and $2+1$ denote distinct objects.)

We recall that the model-theoretic semantics of definite programs without equality enjoys some attractive properties: To characterize the meaning of a program (i.e. its set of ground, atomic logical consequences) it is sufficient to consider the set of all Herbrand models. In fact, attention may be focused on a single *least* Herbrand model. Evidently, this is not applicable to definite programs with equality. However, there is a natural extension of these ideas: Instead of considering interpretations where the domain consists of ground terms one may consider interpretations where the domain consists of *sets* of equivalent ground terms. More precisely one may consider the quotient set of U_P with respect to a congruence relation. Such a set will be called an E-universe. In what follows, it will be clear from the context what congruence relation is intended, and we will just write $\overline{s}$ to denote the equivalence class which contains s.

By analogy to definite programs the E-base will be the set:

$$\{p(t_1,\dots,t_n) \mid t_1,\dots,t_n \in E\text{-universe and } p/n \text{ is a predicate symbol}\}$$

and an E-interpretation will be a subset of the E-base. The intuition behind an E-interpretation is as follows: (1) the meaning of a ground term t is the equivalence class $\overline{t}$ and (2) if s and t are ground terms, then $s \doteq t$ is true in the interpretation iff s and t are members in the same equivalence class of the domain (i.e. if $\overline{s} = \overline{t}$).

To characterize the set of all ground, atomic logical consequences of a program P, E we first define a set of E-interpretations which are models of E. Then we consider E-interpretations which are also models of P. The following theorem shows that it is reasonable to restrict attention to E-interpretations whose domain is $U_P/{\equiv_E}$ (the set of all equivalence-classes of U_P w.r.t. the relation $\equiv_E$), since they characterize the set of all ground equations which are logical consequences of E:

Theorem 13.5 Let E be a set of equations, s and t ground terms and $\Im$ an E-interpretation whose domain is $U_P/\!\equiv_E$. Then:

$$\begin{aligned} \Im \models s \doteq t \quad & \text{iff} \quad \overline{s} = \overline{t} \\ & \text{iff} \quad s \equiv_E t \\ & \text{iff} \quad E \models s \doteq t \end{aligned}$$

∎

Such E-interpretations are called *canonical.* Notice that if $E = \varnothing$ then $\overline{s} = \{s\}$ for any ground term s, and $\Im$ reduces to a Herbrand interpretation (except that the domain consists of singleton sets of ground terms).

Example 13.6 Consider the following set E of equations:

$$\begin{array}{l} \mathit{father}(\mathit{sally}) \doteq \mathit{robert}. \\ \mathit{father}(\mathit{bruce}) \doteq \mathit{adam}. \\ \mathit{father}(\mathit{simon}) \doteq \mathit{robert}. \end{array}$$

Then $U_P/\!\equiv_E$ contains elements such as:

$$\begin{aligned} \overline{\mathit{robert}} &= \{\mathit{robert}, \mathit{father}(\mathit{sally}), \mathit{father}(\mathit{simon})\} \\ \overline{\mathit{adam}} &= \{\mathit{adam}, \mathit{father}(\mathit{bruce})\} \\ \overline{\mathit{sally}} &= \{\mathit{sally}\} \\ \overline{\mathit{bruce}} &= \{\mathit{bruce}\} \end{aligned}$$

∎

Most of the results from Chapter 2 can be carried over to canonical E-interpretations. For instance (see Jaffar, Lassez and Maher (1986) or (1984) for details):

- if P, E has a model then it also has a canonical E-model;
- the intersection of all canonical E-models of P, E is a canonical E-model;
- there is a least canonical E-model (denoted by $M_{P,E}$).

Moreover, $M_{P,E}$ characterizes the set of all ground, atomic logical consequences of P, E. In what follows let $\overline{p(t_1, \ldots, t_n)}$ be an abbreviation of $p(\overline{t_1}, \ldots, \overline{t_n})$. If $t_1, \ldots, t_n$ are ground terms, then:

$$P, E \models p(t_1, \ldots, t_n) \quad \text{iff} \quad \overline{p(t_1, \ldots, t_n)} \in M_{P,E}$$

An alternative characterization of this set can be given by a fixed point-operator similar to the T_P-operator for definite programs. The operator — denoted by $T_{P,E}$ — is defined as follows:

$$T_{P,E}(x) := \{\overline{A} \mid A \leftarrow A_1, \ldots, A_n \in \mathit{ground}(P) \land \overline{A_1}, \ldots, \overline{A_n} \in x\}$$

Jaffar, Lassez and Maher (1984) showed that $M_{P,E} = T_{P,E} \uparrow \omega$.

Example 13.7 Let P, E be the program:

$$\begin{array}{l} proud(father(X)) \leftarrow newborn(X). \\ newborn(sally). \\ newborn(bruce). \\ \\ father(sally) \doteq robert. \\ father(bruce) \doteq adam. \\ father(simon) \doteq robert. \end{array}$$

In this case:

$$M_{P,E} = \{proud(\overline{robert}), proud(\overline{adam}), newborn(\overline{sally}), newborn(\overline{bruce})\}$$

∎

As observed above the model-theoretic semantics of definite programs with equality is a generalization of the model-theoretic semantics of definite programs. This is not particularly strange since a definite program also has a set of equations, albeit empty. One might expect that a similar situation would crop up for the proof-theoretic semantics and, indeed, it does. In principle, the only modification which is needed to SLD-resolution is to replace ordinary unification by E-unification. In what follows we presuppose the existence of a complete E-unification algorithm. However, we do not spell out how it works. The following informal description describes the principles of the proof-theory.

Let P be a definite program, E a set of equations and G the definite goal:

$$\leftarrow A_1, \ldots, A_{m-1}, A_m, A_{m+1}, \ldots, A_n$$

Now assume that C is the (renamed) program clause:

$$B_0 \leftarrow B_1, \ldots, B_j \qquad (j \geq 0)$$

and that A_m and B_0 have a nonempty, complete set of E-unifiers Θ. Then G and C resolve into the new goal:

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

if $\theta \in \Theta$.

To avoid confusing this with ordinary SLD-resolution it will be called the *SLDE-resolution* principle. The notion of SLDE-derivation, refutation etc. are carried over from Chapter 3. SLDE-resolution introduces one extra level of nondeterminism — since two atoms may have several E-unifiers none of which subsume the others, it may happen that a given computation rule, a goal and a clause with a head that E-unifies with the selected subgoal, result in several new goals. This was not the case for SLD-resolution since the existence of a unique mgu allowed only one new goal to be derived.

Example 13.8 Consider again the following definite program and equations:

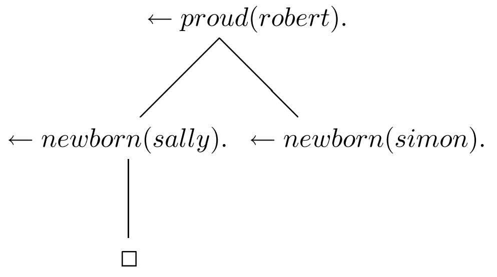

$\leftarrow proud(robert).$
$\leftarrow newborn(sally).$
$\leftarrow newborn(simon).$
$\square$

Figure 13.3: SLDE-tree for the goal $\leftarrow proud(robert)$

$$
\begin{array}{l}
proud(father(X)) \leftarrow newborn(X).\\
newborn(sally).\\
newborn(bruce).\\
\\
father(sally) \doteq robert.\\
father(bruce) \doteq adam.\\
father(simon) \doteq robert.
\end{array}
$$

Let G_0 be the goal:

$$\leftarrow proud(robert).$$

Since $\{\{X_0/sally\}, \{X_0/simon\}\}$ is a complete set of E-unifiers of $proud(robert)$ and $proud(father(X_0))$, G_1 is either of the form:

$$\leftarrow newborn(sally).$$

which results in a refutation; or of the form:

$$\leftarrow newborn(simon).$$

which fails since *simon* E-unifies neither with *sally* nor with *bruce*. ∎

By analogy to SLD-resolution all (complete) SLDE-derivations under a given computation rule may be depicted in a single SLDE-tree (cf. Figure 13.3). Notice in contrast to SLD-trees, that the root of the tree has two children despite the fact that the definition of $proud/1$ contains only one clause. Since the complete set of E-unifiers of two terms may be infinite, a node in the SLDE-tree may have infinitely many children — something which is not possible in ordinary SLD-trees. This, of course, may cause operational problems since a breadth-first traversal is, in general, not sufficient for finding all refutations in the SLDE-tree. However, soundness and completeness results similar to those for SLD-resolution can and have been proved also for SLDE-resolution described above.

Exercises

13.1 Consider the equations in Example 13.3. Prove that $\{X/s(0), Y/0\}$ is an E-unifier of $sum(s(X), Y)$ and $s(s(0))$.

13.2 Show that the inference rules in Figure 13.1 are sound. Try to prove that they are complete!

13.3 Consider the following equations:

$$\begin{array}{rcl} append(nil, X) & \doteq & X \\ append(cons(X, Y), Z) & \doteq & cons(X, append(Y, Z)) \end{array}$$

Prove that $append(X, cons(b, nil))$ and $cons(a, cons(b, nil))$ are E-unifiable.

13.4 Prove that two terms, s and t, are $\varnothing$-unifiable iff there is some substitution θ such that $s\theta$ and $t\theta$ are syntactically identical.

Chapter 14

Constraint Logic Programming

The following program (which describes the property of being a list whose elements are sorted in ascending order) is intended to illustrate some shortcomings of SLD-resolution as previously presented:

$$\begin{array}{l} \mathit{sorted}([\,]). \\ \mathit{sorted}([X]). \\ \mathit{sorted}([X, Y|Xs]) \leftarrow X \leq Y, \mathit{sorted}([Y|Xs]). \end{array}$$

Consider the query "Are there integers X, Y and Z, such that the list $[X, Y, Z]$ is sorted?". The query may be formalized as a goal:

$$\leftarrow \mathit{sorted}([X, Y, Z]). \qquad (G_0)$$

SLD-resolution, as described in Chapter 3, attempts to construct a counter-example — a substitution θ such that $P \cup \{\forall(\neg \mathit{sorted}([X, Y, Z])\theta)\}$ is unsatisfiable. If such a counter-example can be constructed, then most Prolog systems present the answer as a set of variable bindings, e.g. $\{X/1, Y/2, Z/3\}$. This may be interpreted as follows:

$$\forall(X \doteq 1 \land Y \doteq 2 \land Z \doteq 3 \supset \mathit{sorted}([X, Y, Z]))$$

Now the goal G_0 may be reduced to:

$$\leftarrow X \leq Y, \mathit{sorted}([Y, Z]). \qquad (G_1)$$

In a Prolog implementation this would lead to a run-time error since arithmetic tests can be made only if the arguments are instantiated. This could easily be repaired by imagining an infinite set of facts of the form $0 \leq 0, 0 \leq 1, \ldots$ Unfortunately this would lead to an infinite number of answers to the original query. Assume instead that the interpreter is clever enough to realize that $X \leq Y$ has at least one solution. Then the second subgoal of G_1 may be selected instead, in which case G_1 is reduced to:

$$\leftarrow X \leq Y, Y \leq Z, \mathit{sorted}([Z]). \qquad (G_2)$$

Under the assumption that the interpreter is intelligent enough to realize that $\exists(X \leq Y \wedge Y \leq Z)$ is satisfiable, the final recursive call can be eliminated:

$$\leftarrow X \leq Y, Y \leq Z. \qquad (G_3)$$

If $\exists(X \leq Y \wedge Y \leq Z)$ is satisfiable, then G_3 — i.e. $\forall\neg(X \leq Y \wedge Y \leq Z)$ — is unsatisfiable. Thus, G_0 has a refutation. Moreover, $X \leq Y, Y \leq Z$ may be viewed as an answer to G_0:

$$\forall(X \leq Y \wedge Y \leq Z \supset \mathit{sorted}([X, Y, Z]))$$

The example illustrates two desirable extensions of logic programming. First the use of dedicated predicate symbols (in our case $\leq$) whose semantics are built into the interpreter rather than defined by the user. Second the extension of the notion of an answer not just to include equations. These extensions have spawned a number of new logic programming languages incorporating various built-in domains, operations and relations. They are all instances of a more general scheme known as *constraint logic programming* (CLP). This chapter surveys the theoretical foundations of constraint logic programming languages and discusses some specific instances of the scheme.

14.1 Logic Programming with Constraints

In the previous chapter logic programming was extended with a dedicated predicate symbol $\doteq$, always interpreted as the identity relation. Constraint logic programming languages generalize this idea by allowing also other dedicated *interpreted* predicates, function symbols and constants (in addition to the uninterpreted symbols). Hence, a CLP language $\mathit{CLP}(\mathcal{D})$ is parameterized by an interpretation $\mathcal{D}$ of certain symbols in the language.

Definition 14.1 (Constraint logic program) A constraint logic program is a finite set of clauses:

$$A_0 \leftarrow C_1, \ldots, C_m, A_1, \ldots, A_n \qquad (m, n \geq 0)$$

where $C_1, \ldots, C_m$ are formulas built from the interpreted alphabet (including variables, quantifiers and logical connectives) and $A_0, \ldots, A_n$ are atoms with uninterpreted predicate symbols. ∎

The formulas $C_1, \ldots, C_m$ are called *constraints*. In all our examples, a constraint will be an atomic formula with an interpreted predicate symbols. All CLP languages are assumed to contain the predicate $\doteq$ which is *always* interpreted as identity.

Let $\mathcal{Z}$ be an interpretation of numerals such as $0, 1, \ldots$, function symbols such as $+, -, \ldots$ and predicates such as $\doteq, <, \leq, \ldots$ Assume that $|\mathcal{Z}| = \mathbb{Z}$ (the integers) and that all symbols have their usual interpretation. (For instance, $+_{\mathcal{Z}}$ is integer addition.) Then the following is a $\mathit{CLP}(\mathcal{Z})$ program with the uninterpreted constant $[\,]$, the functor $./2$ and the predicate symbol $\mathit{sorted}/1$:

$$\begin{array}{l}\mathit{sorted}([\,]).\\ \mathit{sorted}([X]).\\ \mathit{sorted}([X, Y|Xs]) \leftarrow X \leq Y, \mathit{sorted}([Y|Xs]).\end{array}$$

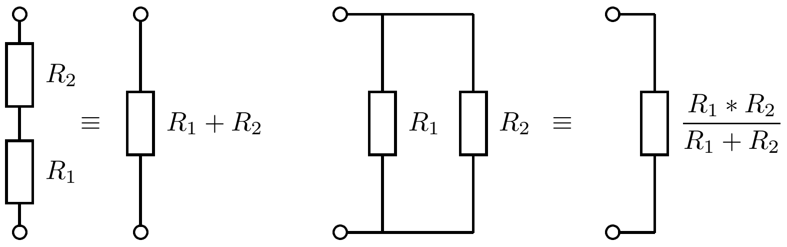


Figure 14.1: Equivalent nets

Similarly, let $\mathcal{R}$ be an interpretation of the standard arithmetic operators and relations (including $\doteq$) over the reals (i.e. $|\mathcal{R}| = \mathbb{R}$). Then the following is a $CLP(\mathcal{R})$ program that describes the relation between some simple electrical circuits and their resistance:

$$
\begin{aligned}
&res(r(R), R).\\
&res(cell(E), 0).\\
&res(series(X, Xs), R + Rs) \leftarrow\\
&\qquad res(X, R), res(Xs, Rs).\\
&res(parallel(X, Xs), R * Rs/(R + Rs)) \leftarrow\\
&\qquad res(X, R), res(Xs, Rs).
\end{aligned}
$$

The program uses the uninterpreted functors $r/1, cell/1, series/2, parallel/2$ and the predicate symbol $res/2$. The clauses express elementary electrical laws such as those depicted in Figure 14.1.

Finally, let $\mathcal{H}$ be a Herbrand interpretation. That is, $|\mathcal{H}| = U_P$ and each ground term is interpreted as itself. The only pre-defined predicate symbol is the equality. The following is a $CLP(\mathcal{H})$ program:

$$
\begin{aligned}
&append(X, Y, Y) \leftarrow X \doteq [\,].\\
&append(X, Y, Z) \leftarrow X \doteq [U|V], Z \doteq [U|W], append(V, Y, W).
\end{aligned}
$$

In Section 14.4 more examples of CLP languages and CLP programs will be presented. However, first we consider the *declarative* and *operational* semantics of the general CLP scheme.

14.2 Declarative Semantics of *CLP*

Given a specific CLP language $CLP(\mathcal{D})$, there are several alternative approaches to define the declarative meaning of a $CLP(\mathcal{D})$ program P. One possibility is to assume that there is a so-called background theory $Th_{\mathcal{D}}$ with a unique model, $\mathcal{D}$, in which case the declarative semantics of P can be defined in terms of classical models of $P \cup Th_{\mathcal{D}}$. Unfortunately, there are interpretations $\mathcal{D}$ which cannot be uniquely axiomatized in predicate logic. Another approach that is sometimes used is to define the declarative meaning of P in terms of a unique classical model (similar to the least Herbrand model) defined by a generalized T_P-operator. The approach used here is an intermediate one — similar to that of the previous chapter — where a subset of *all* classical interpretations are taken into account. Namely those that interpret the pre-defined symbols in accordance with $\mathcal{D}$.

Consider a CLP language $CLP(\mathcal{D})$. A $\mathcal{D}$-*interpretation* is an interpretation I which complies with $\mathcal{D}$ on the interpreted symbols of the language:

Definition 14.2 ($\mathcal{D}$-interpretation) A $\mathcal{D}$-*interpretation* is an interpretation I such that $|\mathcal{D}| \subseteq |I|$ and for all interpreted constants c, functors f/n and predicates p/n (except $\doteq /2$):

- $c_I = c_{\mathcal{D}}$;
- $f_I(d_1, \ldots, d_n) = f_{\mathcal{D}}(d_1, \ldots, d_n)$ for all $d_1, \ldots, d_n \in |\mathcal{D}|$;
- $\langle d_1, \ldots, d_n \rangle \in p_I$ iff $\langle d_1, \ldots, d_n \rangle \in p_{\mathcal{D}}$.

∎

As pointed out above $\doteq$ is always assumed to be contained in the language and always denotes the identity relation. The notion of logical consequence can now be modified taking into account $\mathcal{D}$-interpretations only:

Definition 14.3 ($\mathcal{D}$-model) A $\mathcal{D}$-interpretation which is a model of a set P of closed formulas is called a $\mathcal{D}$-*model* of P. ∎

Definition 14.4 (Logical consequence) A formula F is a logical $\mathcal{D}$-consequence of a $CLP(\mathcal{D})$-program P (denoted $P, \mathcal{D} \models F$) iff every $\mathcal{D}$-model of P is a $\mathcal{D}$-model of F. ∎

If P is the *sorted*/1-program then:

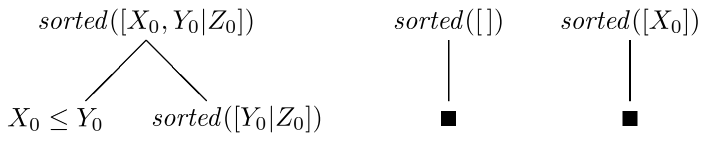
$$P, \mathcal{Z} \models \forall (X \leq Y \wedge Y \leq Z \supset \mathit{sorted}([X, Y, Z]))$$

The notation $\mathcal{D} \models F$ is used when F is true in every $\mathcal{D}$-model. For instance:

$$\begin{array}{l} \mathcal{Z} \models \exists (X \leq Y \wedge Y \leq Z) \\ \mathcal{H} \models \exists (X \doteq f(Y) \wedge Y \doteq a) \end{array}$$

14.3 Operational Semantics of *CLP*

This section describes a class of abstract execution strategies of CLP programs based on the notion of *derivation tree* introduced in Section 3.6. Recall that a derivation tree is a tree built from elementary trees that represent (renamed) program clauses (and an atomic goal) such as:



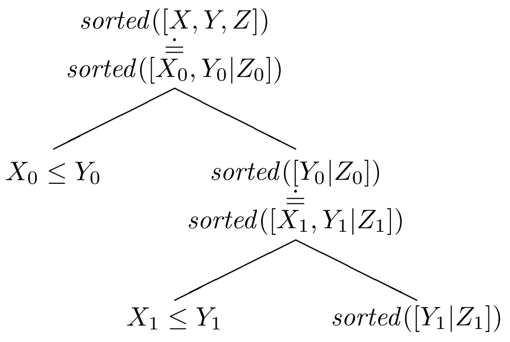
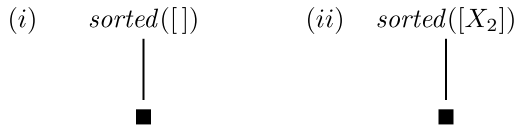


Figure 14.2: Incomplete derivation tree of $\leftarrow sorted([X, Y, Z])$

The notion of derivation tree will have to be extended to permit the use of constraints, such as $X_0 \leq Y_0$, in programs. Since interpreted predicate symbols are not allowed in the head of any clause a constraint will always appear in the position of a leaf in a tree. A derivation tree is said to be *complete* if all of its leaves are of the form ■ or labelled by constraints. A complete derivation tree is also called a *proof tree*. Figure 14.2 depicts an (incomplete) derivation tree of the goal $\leftarrow sorted([X, Y, Z])$. The tree can be made complete by combining the rightmost leaf with one of the following elementary trees:



A derivation tree has an associated set of equalities (the labels of the internal nodes) and constraints. This set will be called the *constraint store*. For instance, the derivation tree in Figure 14.2 has the following constraint store:

$$\{[X, Y, Z] \doteq [X_0, Y_0|Z_0], X_0 \leq Y_0, [Y_0|Z_0] \doteq [X_1, Y_1|Z_1], X_1 \leq Y_1\} \qquad (\dagger)$$

We are particularly interested in constraint stores that are *satisfiable*:

Definition 14.5 (Satisfiable constraint store) A constraint store $\{C_1, \ldots, C_n\}$ is said to be *satisfiable* iff $\mathcal{D} \models \exists(C_1 \wedge \cdots \wedge C_n)$. ▮

The constraint store (†) is satisfiable. So is the constraint store obtained by combining the derivation tree in Figure 14.2 with the elementary tree (*ii*). However, when combined with (*i*) the resulting constraint store is unsatisfiable (since $[X_1] \doteq [\,]$ is not true in every $\mathcal{Z}$-interpretation).

In what follows we assume the existence of a (possibly imperfect) decision procedure $sat(\cdot)$ which checks if a constraint store is satisfiable. If $sat(\{C_1, \ldots, C_n\})$ succeeds (i.e. returns *true*) then the store is said to be *consistent*. Ideally, satisfiability and consistency coincide in which case $sat(\cdot)$ is said to be *complete* (or *categorical*). However, in the general case it is not possible or even desirable to employ a complete

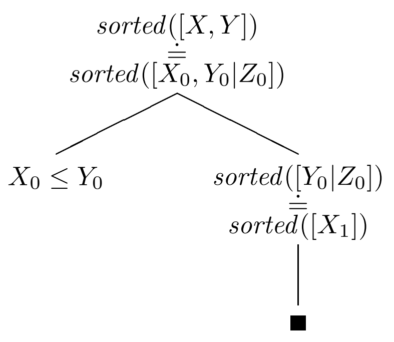


Figure 14.3: Proof tree of $\leftarrow sorted([X, Y])$

satisfiability check. In such cases, it is usually required that the check is *conservative.* That is to say:

$$\text{if } \mathcal{D} \models \exists(C_1 \wedge \cdots \wedge C_n) \text{ then } sat(\{C_1, \ldots, C_n\}) \text{ succeeds}$$

Thus, if the procedure fails (i.e. returns *false*) then the constraint store must be unsatisfiable. This is typically achieved by checking the satisfiability only of a *subset* of the constraints in the store. This is the case in many CLP languages involving arithmetic constraints — the satisfiability check is complete as long as the arithmetic constraints are linear. Non-linear constraints (such as $X^2 \doteq Y$) are not checked at all unless they can be simplified into linear constraints.

Complete derivation trees with satisfiable constraint stores are of particular interest since they represent answers to the initial goal. However, the constraint store usually contains a large number of free variables which do not appear in the initial goal. Such variables are called *local* variables. Since the bindings of local variables are of no direct relevance they can be projected away (cf. computed answer substitutions) by existential quantification:

Definition 14.6 (Answer constraint) Let $\leftarrow A$ be an atomic goal with a complete derivation tree and a satisfiable constraint store $\{C_1, \ldots, C_n\}$. If $\mathbf{X}$ is the set of all local variables in $\{C_1, \ldots, C_n\}$ then $\exists\mathbf{X}(C_1 \wedge \cdots \wedge C_n)$ is called a (computed) *answer constraint.* ▮

The answer constraint (or an equivalent simplified expression) can be viewed as an answer to the initial goal. However, in practice the satisfiability check may be incomplete, in which case the following weaker notion of conditional answer may be used instead:

Definition 14.7 (Conditional answer) Let $\leftarrow A$ be an atomic goal with a complete derivation tree and a satisfiable constraint store $\{C_1, \ldots, C_n\}$. Then $\forall((C_1 \wedge \cdots \wedge C_n) \supset A)$ is called a *conditional answer.* ▮

The goal $\leftarrow sorted([X, Y])$ has a complete derivation tree depicted in Figure 14.3. The associated constraint store is satisfiable. Hence, the following is a conditional answer:

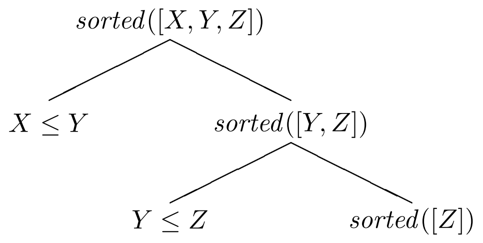


Figure 14.4: Simplified derivation tree of $\leftarrow sorted([X, Y, Z])$

$$\forall(([X, Y] \doteq [X_0, Y_0|Z_0] \wedge X_0 \leq Y_0 \wedge [Y_0|Z_0] \doteq [Y_0]) \supset sorted([X, Y]))$$

This is obviously not a very informative answer. Fortunately it is usually possible to eliminate or simplify the constraints. First note that a conditional answer may also be written as follows:

$$\forall(\exists \mathbf{X}(C_1 \wedge \cdots \wedge C_n) \supset A)$$

where $\mathbf{X}$ is the set of all local variables. This means that all local variables are existentially quantified on the level of the constraints. It is usually possible to eliminate or simplify at least constraints involving local variables. Simplification is needed not only for the purpose of producing readable answers, but it is often *necessary* to simplify the store in order to avoid having to check the satisfiability of the *whole* store when new constraints are added. Simplification, usually amounts to transforming the constraints into some kind of *canonical form*, such as solved form. It is normally required that the new store S' is equivalent to the old store S. That is:

$$\mathcal{D} \models \forall(S \leftrightarrow S')$$

Many of the transformations are of course domain dependent (e.g. that $X \leq Y, Y \leq X$ can be replaced by $X \doteq Y$). However, some simplifications are always possible — the constraint store can be divided into equalities over uninterpreted symbols (and variables) and other constraints involving interpreted symbols (and variables). The former are called *free constraints* and the latter *built-in constraints.* (Equations such as $X \doteq [3 + 3]$ which mix interpreted and uninterpreted symbols may be viewed as shorthand for the free constraint $X \doteq [Y]$ *and* the built-in constraint $Y \doteq 3 + 3$.) If a constraint store is satisfiable, it is always possible to transform the free constraints into *solved form.* For instance, (†) can be transformed into the equivalent constraint store:

$$\{X_0 \doteq X, Y_0 \doteq Y, Z_0 \doteq [Z], X_1 \doteq Y, Y_1 \doteq Z, Z_1 \doteq [\,], X_0 \leq Y_0, X_1 \leq Y_1\}$$

Such a store can be simplified further; all equations $X \doteq t$ where X is a local variable can be removed provided that all occurrences of X in the store and in the tree are replaced by t. Hence, the derivation tree in Figure 14.2 can be simplified to the derivation tree in Figure 14.4 and the constraint store (†) can be simplified into:

$$\{X \leq Y, Y \leq Z\}$$

which is clearly satisfiable in $\mathcal{Z}$ (and thus, consistent provided that the check is conservative). Similarly, the conditional answer:

$$\forall(([X,Y] \doteq [X_0, Y_0|Z_0] \wedge X_0 \leq Y_0 \wedge [Y_0|Z_0] \doteq [Y_0]) \supset \mathit{sorted}([X,Y]))$$

may be simplified into:

$$\forall(X \leq Y \supset \mathit{sorted}([X,Y]))$$

The notions of *tree construction*, *checking of satisfiability* (i.e. *consistency*) and *simplification* are the basis of the operational semantics of many CLP languages. From an abstract point of view a computation may be seen as a sequence of transitions:

$$\langle T_0, S_0\rangle \leadsto \langle T_1, S_1\rangle \leadsto \langle T_2, S_2\rangle \leadsto \cdots$$

over a space of computation states consisting of pairs of derivation trees and constraint stores (and the additional computation state *fail* representing a failed derivation). A computation starts with a computation state where T_0 is a derivation tree representing the initial (atomic) goal $\leftarrow A$ and S_0 is an empty constraint store. The only available transitions are the following ones:

- $\langle T, S\rangle \overset{E}{\leadsto} \langle T', S'\rangle$ if T can be *extended* into T' by combining a leaf labelled by A_i with an elementary tree whose root is B_0. Moreover, $S' = S \cup \{A_i \doteq B_0\}$;
- $\langle T, S\rangle \overset{C}{\leadsto} \langle T, S\rangle$ if $\mathit{sat}(S) = \mathit{true}$;
- $\langle T, S\rangle \overset{C}{\leadsto} \mathit{fail}$ if $\mathit{sat}(S) = \mathit{false}$;
- $\langle T, S\rangle \overset{S}{\leadsto} \langle T', S'\rangle$ if T and S can be *simplified* into T' and S'.

Different CLP systems use different strategies when applying these transitions. At one end of the spectrum one may imagine a strategy where extension-transitions are applied until the tree is complete, after which simplification and satisfiability-checking are performed. Hence, computations are of the form:

$$\bullet \overset{E}{\leadsto} \bullet \overset{E}{\leadsto} \cdots \overset{E}{\leadsto} \bullet \overset{E}{\leadsto} \bullet \overset{S}{\leadsto} \bullet \overset{C}{\leadsto} \bullet$$

Such a *lazy* approach runs the risk of entering infinite loops since failure due to an unsatisfiable constraint store might not be detected. On the other hand, the constraint store is checked for satisfiability only once.

At the other end of the spectrum one may imagine a strategy where simplification and satisfiability-checking take place after each extension of the tree:

$$\bullet \overset{E}{\leadsto} \bullet \overset{S}{\leadsto} \bullet \overset{C}{\leadsto} \bullet \overset{E}{\leadsto} \bullet \overset{S}{\leadsto} \bullet \overset{C}{\leadsto} \cdots \overset{E}{\leadsto} \bullet \overset{S}{\leadsto} \bullet \overset{C}{\leadsto} \bullet$$

Such an *eager* strategy has a much improved termination behaviour compared to the lazy approach above. However, checking for satisfiability is usually more expensive than in the lazy strategy. With the eager strategy it is vital that the constraint store can be checked incrementally to avoid computational overhead. Most CLP systems seem to employ an eager strategy. This is often combined with a Prolog-like depth

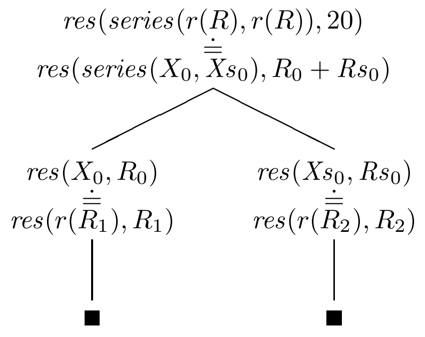


Figure 14.5: Proof tree of $\leftarrow res(series(r(10), r(R)), 30)$

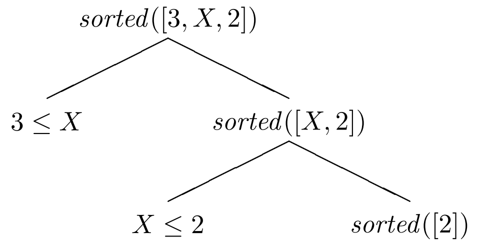


Figure 14.6: Derivation tree with unsatisfiable constraints

first search using chronological backtracking in order to handle the non-determinism that arise due to the extension-transition.

To illustrate the lazy strategy consider the goal $\leftarrow res(series(r(R), r(R)), 20)$ and derivation tree in Figure 14.5. The tree can be constructed by means of three extension-transitions. The associated constraint store of the tree looks as follows:

$$\left\{ \begin{array}{c} series(r(R), r(R)) \doteq series(X_0, Xs_0), \\ 20 \doteq R_0 + Rs_0, X_0 \doteq r(R_1), R_0 \doteq R_1, Xs_0 \doteq r(R_2), Rs_0 \doteq R_2 \end{array} \right\}$$

This store can be simplified into:

$$\{20 \doteq R + R\}$$

Which is clearly satisfiable. Using domain knowledge the store can be further simplified yielding the conditional answer:

$$\forall (R \doteq 10 \supset res(series(r(R), r(R)), 20))$$

Next consider the goal $\leftarrow sorted([3, X, 2])$. By an eager strategy it can be detected already after two "cycles" that the tree has an unsatisfiable constraint store $\{3 \leq X, X \leq 2\}$ (cf. Figure 14.6). A lazy strategy may get stuck in an infinite process of tree construction and may never get to the point of checking the satisfiability of the store.

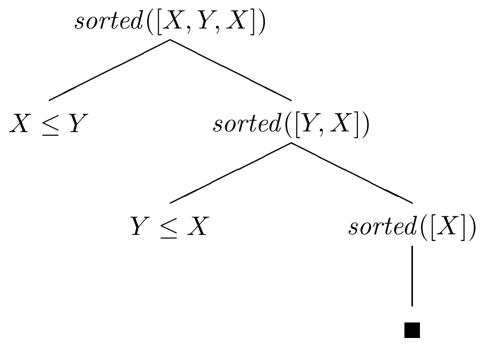


Figure 14.7: Proof tree of $\leftarrow \mathit{sorted}([X, Y, X])$

Finally consider the goal $\leftarrow \mathit{sorted}([X, Y, X])$. Using either a lazy or an eager strategy the simplified proof tree in Figure 14.7 can be constructed. The corresponding constraint store $\{X \leq Y, Y \leq X\}$ is clearly satisfiable. The store can be further simplified into the conditional answer:

$$\forall(X \doteq Y \supset \mathit{sorted}([X, Y, X]))$$

by exploiting domain knowledge.

The execution scheme described above can be shown to be sound:

Theorem 14.8 (Soundness) Let P be a $\mathit{CLP}(\mathcal{D})$-program and $\leftarrow A$ an atomic goal. If $\forall((C_1 \wedge \cdots\cdots C_n) \supset A)$ is a conditional answer of $\leftarrow A$ then $P, \mathcal{D} \models \forall((C_1 \wedge \cdots \wedge C_n) \supset A)$. ▮

The execution scheme can also be shown to be complete and generalized to negation as finite failure. However, for this discussion, which requires the introduction of a number of auxiliary definitions, see Jaffar and Maher (1994).

14.4 Examples of CLP-languages

This section surveys some of the most popular constraint domains found in existing CLP systems:

- Boolean constraints;
- Numerical constraints over integers, rational numbers or reals;
- String constraints;
- Finite and infinite tree constraints.

It should be noted that most CLP systems come with more than one constraint domain. For instance, Prolog III supports constraints over Booleans, rational numbers, strings and infinite trees. CHIP contains constraints over finite domains, Booleans and rational numbers. CLP(BNR) handles constraints involving Booleans and intervals of natural numbers and reals.

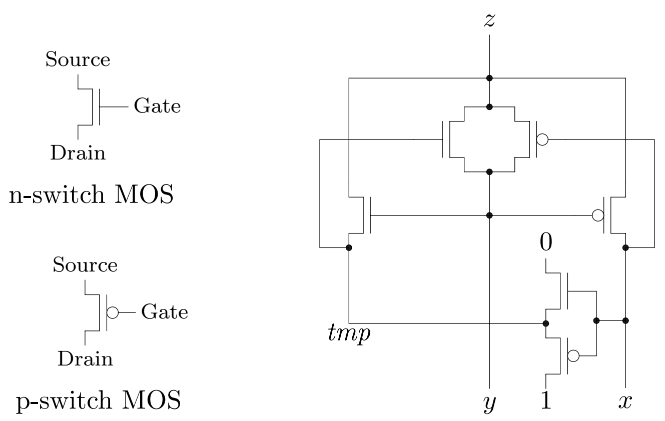


Figure 14.8: Xor-gate in MOS technology

Boolean constraints

Many CLP systems (e.g. CHIP, CAL, CLP(BNR), Prolog III and SICStus Prolog) are equipped with Boolean constraints. Such a *CLP*($\mathcal{B}$) language has a binary domain $|\mathcal{B}| = \{true, false\}$ and the language typically provides:

- two interpreted constants $0, 1$ such that $0_\mathcal{B} = false$ and $1_\mathcal{B} = true$;
- function symbols such as $\wedge, \neg, \vee, \ldots$ with the standard interpretation;
- predicates such as $\doteq, \neq, \ldots$

Boolean constraints with more than two values are also available in some systems.

Boolean constraints are useful in many contexts, e.g. for modelling digital circuits. The following example is due to Dincbas et.al. (1988).

Consider the circuit depicted in Figure 14.8. The circuit is a schematic model of an xor-gate with inputs x and y and the output z. The gate is built from mos-transistors of which there are two kinds — n-mos and p-mos (also depicted in Figure 14.8). The transistors have three connections usually referred to as the drain (D), source (S) and gate (G). From the logical point of view the transistors may be viewed as binary switches with the following (Boolean) relationship between D, S and G:

$$\begin{array}{lrcl} \text{n-mos:} & D \wedge G & \doteq & G \wedge S \\ \text{p-mos:} & D \wedge \neg G & \doteq & \neg G \wedge S \end{array}$$

In a *CLP*($\mathcal{B}$) language the circuit in Figure 14.8 can be described as follows:

$$\begin{array}{l} xor(X, Y, Z) \leftarrow \\ \quad\quad pswitch(Tmp, 1, X), \\ \quad\quad nswitch(0, Tmp, X), \\ \quad\quad pswitch(Z, X, Y), \\ \quad\quad nswitch(Z, Tmp, Y), \\ \quad\quad nswitch(Z, Y, Tmp), \\ \quad\quad pswitch(Z, Y, X). \end{array}$$

$$
\begin{aligned}
& nswitch(S, D, G) \leftarrow \\
& \qquad D \wedge G \doteq G \wedge S. \\
& pswitch(S, D, G) \leftarrow \\
& \qquad D \wedge \neg G \doteq \neg G \wedge S.
\end{aligned}
$$

This program may be used to verify that the design in Figure 14.8 *is* an xor-gate by giving the goal clause:

$$\leftarrow xor(X, Y, Z).$$

A system such as SICStus Prolog (version 2.1) which has an interpreted functor $\oplus$ denoting the operation of exclusive or, returns the conditional answer:

$$\forall(X \doteq Y \oplus Z \supset xor(X, Y, Z))$$

which proves that the circuit is indeed correct.

Numerical constraints

Most CLP systems support constraints over a numerical domain. However, constraints over the integers are difficult to handle efficiently and, as a consequence, most systems support constraints only over the reals ($CLP(\mathcal{R})$) or the rational numbers ($CLP(\mathcal{Q})$). Different systems support different operations and relations — some CLP languages, such as CLP(BNR), support arithmetic operations over intervals. Some systems support only linear equalities, inequalities and disequalities. That is, constraints that may be written:

$$c_n x_n + c_{n-1} x_{n-1} + \cdots + c_1 x_1 + c_0 \quad \Box \quad 0$$

where $\Box \in \{\doteq, <, \leq, \neq\}$ and c_i are constants. Systems that permit non-linear constraints usually employ an incomplete satisfiability check which does not take non-linear constraints into account unless they can be simplified into linear equations. The algorithms for checking satisfiability and for simplification of the built-in constraints often rely on techniques from Operations Research — such as Gauss elimination and the simplex method — and Artificial Intelligence. For an introduction to implementation of various CLP languages, see Jaffar and Maher (1994).

Consider the problem to position five queens on a five-by-five chess-board so that no queen attacks another queen. (Two queens attack one another iff they are on the same row, column or diagonal.) The program given below should be expressible in any CLP system that supports linear equations and disequations over the reals, the integers or the rational numbers.

Before presenting the solution some general observations should be made:

- There must be exactly one queen in each column of the board. Hence, all solutions can be represented by a list of length five. Moreover, there must be exactly one queen in each row of the board. This restriction can be imposed by requiring that the solution is a permutation of the list $[1, 2, 3, 4, 5]$. (For instance, the solution in Figure 14.9 can be represented by the list $[5, 3, 1, 4, 2]$.) With these restrictions it is sufficient to check that each diagonal of the board contains at most one piece;

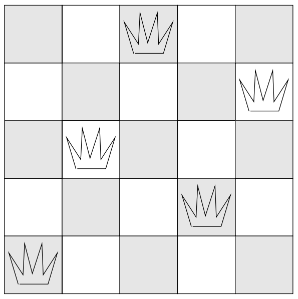


Figure 14.9: Five-queens problem

- If a queen in column $m + n$ attacks a queen in column m, then the attack is mutual. Hence, a board is "attack-free" if no queen is on the same diagonal as one of the queens to its right.

The two observations can now be formalized as follows:

$$
\begin{array}{l}
\mathit{five\_queens}(\mathit{Board}) \leftarrow \\
\qquad \mathit{safe}(\mathit{Board}), \mathit{perm}(\mathit{Board}, [1,2,3,4,5]). \\
\\
\mathit{safe}([\,]). \\
\mathit{safe}([X|Y]) \leftarrow \\
\qquad \mathit{noattack}(X,1,Y), \mathit{safe}(Y). \\
\\
\mathit{noattack}(X,N,[\,]). \\
\mathit{noattack}(X,N,[Y|Z]) \leftarrow \\
\qquad Y \neq X+N, Y \neq X-N, \mathit{noattack}(X,N+1,Z). \\
\\
\mathit{perm}([\,],[\,]). \\
\mathit{perm}([X|Y],W) \leftarrow \\
\qquad \mathit{perm}(Y,Z), \mathit{insert}(X,Z,W). \\
\\
\mathit{insert}(X,Y,[X|Y]). \\
\mathit{insert}(X,[Y|Z],[Y|W]) \leftarrow \\
\qquad \mathit{insert}(X,Z,W).
\end{array}
$$

To solve the puzzle the user may give the goal clause:

$$\leftarrow \mathit{five\_queens}([A,B,C,D,E]).$$

The goal has several conditional answers one of which is:

$$\forall((A \doteq 5 \land B \doteq 3 \land C \doteq 1 \land D \doteq 4 \land E \doteq 2) \supset \mathit{five\_queens}([A,B,C,D,E]))$$

The five queens program illustrates an interesting strategy for solving complex problems. Provided that subgoals are expanded from left to right, the program first collects a store of disequations by expanding *safe*/1. Note that this process is completely deterministic and the objective is to *constrain* the possible values of the variables *A*–*E*. Then *perm*/2 *generates* bindings for the constrained variables. Our solution is an instance of a more general strategy sometimes called *constrain-and-generate*:

$$\begin{array}{l} \mathit{solution}(X) \leftarrow \\ \qquad \mathit{constrain}(X), \mathit{generate}(X). \end{array}$$

The constrain-and-generate strategy takes a completely different direction than the *generate-and-test* strategy usually employed in Prolog:

$$\begin{array}{l} \mathit{five\_queens}(\mathit{Board}) \leftarrow \\ \qquad \mathit{perm}(\mathit{Board}, [1, 2, 3, 4, 5]), \mathit{safe}(\mathit{Board}). \end{array}$$

Here potential solutions are first generated and then tested to check if the constraints can be completely *solved*.

Monoids

In addition to Boolean and numerical constraints, Prolog III also supports (a limited form of) constraints over the monoid $\mathcal{S}$ of strings equipped with concatenation and a neutral element. A *CLP*$(\mathcal{S})$ language typically includes:

- a set of constants denoting string elements;
- an binary functor $\sharp$ denoting the associative operation of concatenation;
- a neutral element $[\,]$.

In a *CLP*$(\mathcal{S})$ language the reverse program may be written as follows:

$$\begin{array}{l} \mathit{reverse}([\,], [\,]). \\ \mathit{reverse}([X] \sharp Y, Z \sharp [X]) \leftarrow \mathit{reverse}(Y, Z). \end{array}$$

Equational constraints

The previous CLP language is interesting in that it can easily be axiomatized as a set of equations. A monoid is completely characterized by the following equations:

$$\begin{array}{rcl} X \sharp (Y \sharp Z) & \doteq & (X \sharp Y) \sharp Z \\ [\,] \sharp X & \doteq & X \\ X \sharp [\,] & \doteq & X \end{array}$$

Hence, a *CLP*$(\mathcal{S})$ program P may also be defined as a logic program with the above equality axioms. Conversely, *any* logic program P with an equality theory E may be viewed as a constraint logic program *CLP*(E) where:

- $|E| = (U_P/\!\equiv_E)$;
- every ground term t is interpreted as $\overline{t}$ (i.e. $\{s \in U_P \mid s \equiv_E t\}$);
- $\doteq$ is the identity relation.

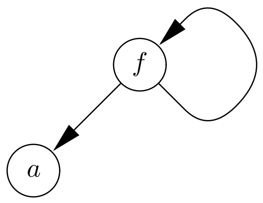
As already mentioned, the special case when $E = \varnothing$ is of course also a CLP language where:

- $|E| = U_P$ (the Herbrand universe);
- every ground term is interpreted as itself;

- $\doteq$ is the identity relation.

Hence, definite programs may be viewed as constraint logic programs with equational constraints over the Herbrand universe. The solved form algorithm (with occur-check) provides a complete satisfiability check for definite programs.

The terms of the Herbrand domain may be viewed as *ordered finite trees*. Related to finite trees are *rational trees*. A rational tree is a possibly infinite tree with a finite set of subtrees. For instance, $f(a, f(a, f(a, \ldots)))$ has only two distinct subtrees, a and the tree itself. Rational trees can be finitely represented as a (possibly) cyclic graph:



One of the first CLP languages, Prolog II, employed constraints (equations and disequations) over the rational trees. The solved form algorithm without the occur-check provides a complete satisfiability check for rational trees.

Exercises

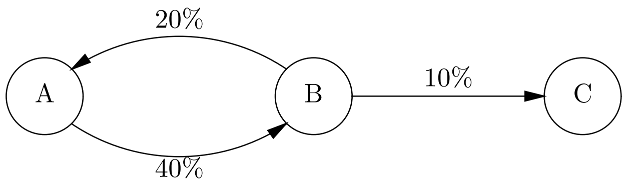
14.1 Solve the "SEND MORE MONEY" puzzle using logic programs with constrains similar to those above. The puzzle consists in associating with each letter a distinct number from 0 to 9 such that the equation:

$$\begin{array}{ccccc} & S & E & N & D \\ + & M & O & R & E \\ \hline M & O & N & E & Y \end{array}$$

is satisfied.

14.2 Modify the five-queens program so that it handles chess boards of arbitrary (quadratic) size.

14.3 Three jars A, B and C contain a total of 1 liter of water. In each time unit water flows as follows between the jars:



Describe the relation between the initial amount of liquid in the three jars and the amount after n units of time.

Chapter 15

Query-answering in Deductive Databases

One of the great virtues of logic programming is that programs have a declarative semantics which can be understood independently of any particular operational semantics. SLD-resolution is one example of a class of interpreters that can be used to compute the logical consequences of a definite program. But also other strategies can be used. In fact, SLD-resolution has several weaknesses also in the case of an ideal interpreter. For instance, consider the following definition of the Fibonacci numbers (for convenience $X + n$ and n abbreviate the terms $s^n(X)$ and $s^n(0)$):

$$\begin{array}{l} \mathit{fib}(0, 1). \\ \mathit{fib}(1, 1). \\ \mathit{fib}(X + 2, Y) \leftarrow \mathit{fib}(X + 1, Z), \mathit{fib}(X, W), \mathit{add}(Z, W, Y). \end{array}$$

Now assume that the following goal clause is given:

$$\leftarrow \mathit{fib}(10, X).$$

The goal reduces to the following:

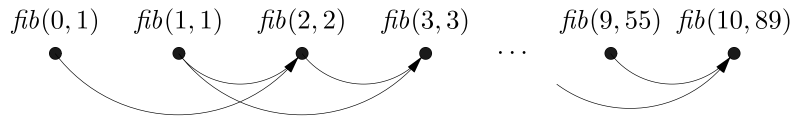
$$\leftarrow \mathit{fib}(9, Z_0), \mathit{fib}(8, W_0), \mathit{add}(Z_0, W_0, X).$$

And selection of the leftmost subgoal yields:

$$\leftarrow \mathit{fib}(8, Z_1), \mathit{fib}(7, W_1), \mathit{add}(Z_1, W_1, Z_0), \mathit{fib}(8, W_0), \mathit{add}(Z_0, W_0, X).$$

This goal contains two subgoals which are identical up to variable renaming: $\mathit{fib}(8, Z_1)$ and $\mathit{fib}(8, W_0)$. In order to resolve the whole goal, both subgoals have to be resolved leading to duplicate work. As a matter of fact, the number of recursive calls to $\mathit{fib}/2$ grows exponentially with the size of the input.

In this particular case it is better to compute answers to $\leftarrow \mathit{fib}(10, X)$ starting from the base cases: Since $\mathit{fib}(0, 1)$ and $\mathit{fib}(1, 1)$ it must hold that $\mathit{fib}(2, 2)$ and so forth:



Another problem with SLD-resolution has to do with termination. Even when no function symbols are involved and there are only a finite number of answers to a goal, SLD-resolution may loop. Consider the goal $\leftarrow \mathit{married}(X, Y)$ and the program:

$$\begin{array}{l} \mathit{married}(X, Y) \leftarrow \mathit{married}(Y, X). \\ \mathit{married}(\mathit{adam}, \mathit{anne}). \end{array}$$

There are only two answers to the goal. However, the SLD-tree is infinite and Prolog would not even find the answers unless the clauses were swapped.

In Chapter 6 logic programming was advocated as a representation language for relational databases. But as illustrated above, SLD-resolution is not always the best mechanism for a query-answering system. In a database system it is particularly important to guarantee termination of the query-answering process whenever that is possible. This chapter considers an alternative inference mechanism for logic programs which has a much improved termination behaviour than SLD-resolution. It may also avoid unnecessary recomputations such as those above. Some assumptions which are often made in the deductive database literature are consciously avoided in our exposition. (In particular, the division of the program into an extensional and intensional part.)

15.1 Naive Evaluation

SLD-resolution is goal-directed — the starting point of the computation is the goal and the aim of the reasoning process is to derive a contradiction by rewritings of the goal. This has several advantages:

- The tree-like structure of the search space lends itself to efficient implementations, both space- and time-wise;
- By focusing attention on the goal it is possible to avoid some inferences which are of no importance for the answers to the goal.

A completely different approach is to start from what is known to be true — the facts and the rules of the program — and to (blindly) generate consequences of the program until the goal can be refuted. (For the sake of simplicity, it will be assumed that all goals are of the form $\leftarrow A$.) For instance, let P be the program:

$$\begin{array}{l} \mathit{married}(X, Y) \leftarrow \mathit{married}(Y, X). \\ \mathit{married}(\mathit{adam}, \mathit{anne}). \end{array}$$

Clearly, $P \models \mathit{married}(\mathit{adam}, \mathit{anne})$. Moreover, since $\mathit{married}(\mathit{adam}, \mathit{anne})$ is true in any model of the program it must hold that $P \models \mathit{married}(\mathit{anne}, \mathit{adam})$. This idea resembles the immediate consequence operator originally introduced in Chapter 3:

```
fun naive(P)
begin
   x := facts(P);
   repeat
      y := x;
      x := S_P(y);
   until x = y;
   return x;
end
```

Figure 15.1: Naive evaluation

$$T_P(I) = \{A_0 \mid A_0 \leftarrow A_1, \ldots, A_n \in \mathit{ground}(P) \text{ and } A_1, \ldots, A_n \in I\}$$

Recall that the least fixed point of this operator (which can be obtained as the limit of $T_P \uparrow n$ where $n \geq 0$) characterizes the set of all *ground* atomic consequences of the program. Hence, the T_P-operator *can* be used for query-answering. However, for computational reasons it is often more practical to represent Herbrand interpretations by sets of atoms (ground or non-ground). A "non-ground T_P-operator" may be defined as follows:[1]

$$S_P(I) = \{A_0\theta \mid A_0 \leftarrow A_1, \ldots, A_n \in P \text{ and } \theta \in \mathit{solve}((A_1, \ldots, A_n), I)\}$$

where $\theta \in \mathit{solve}((A_1, \ldots, A_n), I)$ if θ is an mgu of $\{A_1 \doteq B_1, \ldots, A_n \doteq B_n\}$ and $B_1, \ldots, B_n$ are members in I renamed apart form each other and $A_0 \leftarrow A_1, \ldots, A_n$.

It can be shown that S_P is closed under logical consequence. That is, if every atom in I is a logical consequence of P then so is $S_P(I)$. In particular, every atom in $\varnothing$ is clearly a logical consequence of P. Thus, every atom in $S_P(\varnothing)$ is a logical consequence of P. Consequently every atom in $S_P(S_P(\varnothing))$ is a logical consequence of P and so forth. This iteration — which may be denoted:

$$S_P \uparrow 0, \quad S_P \uparrow 1, \quad S_P \uparrow 2, \quad \ldots$$

yields larger and larger sets of atomic consequences. By analogy to the immediate consequence operator it can be shown that there exists a least set I of atomic formulas such that $S_P(I) = I$ and that this set equals the limit of $S_P \uparrow n$. An algorithmic formulation of this iteration can be found in Figure 15.1. (Let $\mathit{facts}(P)$ denote the set of all facts in P.) The algorithm is often referred to as *naive evaluation*.

The result of the naive evaluation can be used to answer queries to the program: If B is an atom in $\mathit{naive}(P)$ renamed apart from A and θ is an mgu of $A \doteq B$. Then θ is an answer to the goal $\leftarrow A$.

Example 15.1 Consider the following transitive closure program:

[1] For the sake of simplicity it is assumed that $S_P(I)$ never contains two atoms which are renamings of each other.

```
fun semi-naive(P)
begin
  $\Delta x := facts(P)$;
  $x := \Delta x$;
  repeat
    $\Delta x := \Delta S_P(x, \Delta x)$;
    $x := x \cup \Delta x$;
  until $\Delta x = \varnothing$;
  return $x$;
end
```

Figure 15.2: Semi-naive evaluation

$path(X, Y) \leftarrow edge(X, Y).$
$path(X, Y) \leftarrow path(X, Z), edge(Z, Y).$

$edge(a, b).$
$edge(b, a).$

Let x_i denote the value of x after i iterations. Then the iteration of the naive-evaluation algorithm looks as follows:

$$\begin{aligned}
x_0 &= \{edge(a,b), edge(b,a)\}\\
x_1 &= \{edge(a,b), edge(b,a), path(a,b), path(b,a)\}\\
x_2 &= \{edge(a,b), edge(b,a), path(a,b), path(b,a), path(a,a), path(b,b)\}\\
x_3 &= \{edge(a,b), edge(b,a), path(a,b), path(b,a), path(a,a), path(b,b)\}
\end{aligned}$$

The goal $\leftarrow path(a, X)$ has two answers: $\{X/a\}$ and $\{X/b\}$. ∎

15.2 Semi-naive Evaluation

As suggested by the name, naive evaluation can be improved in several respects. In particular, each iteration of the algorithm recomputes everything that was computed in the previous iteration. That is $x_i \subseteq x_{i+1}$. The revised algorithm in Figure 15.2 avoids this by keeping track of the difference $x_i \setminus x_{i-1}$ in the auxiliary variable Δx. Note first that the loop of the naive evaluation may be replaced by:

```
repeat
  $\Delta x := S_P(x) \setminus x$;
  $x := x \cup \Delta x$;
until $\Delta x = \varnothing$;
```

The expensive operations here is $S_P(x) \setminus x$ which renders the modified algorithm even more inefficient than the original one. However, the new auxiliary function call $\Delta S_P(x, \Delta x)$ computes the difference more efficiently:

$$\Delta S_P(I, \Delta I) = \{A_0\theta \notin I \mid \begin{array}{l} A_0 \leftarrow A_1, \ldots, A_n \in P \text{ and} \\ \theta \in \mathit{solve}((A_1, \ldots, A_n), I, \Delta I)\} \end{array}$$

where $\theta \in \mathit{solve}((A_1, \ldots, A_n), I, \Delta I)$ if θ is an mgu of $\{A_1 \doteq B_1, \ldots, A_n \doteq B_n\}$ and $B_1, \ldots, B_n$ are atoms in I renamed apart form each other and $A_0 \leftarrow A_1, \ldots, A_n$ and at least one $B_i \in \Delta I$.

It can be shown that the algorithm in Figure 15.2 — usually called *semi-naive evaluation* — is equivalent to naive evaluation:

Theorem 15.2 (Correctness of semi-naive evaluation) Let P be a definite program, then $\mathit{naive}(P) = \mathit{semi\text{-}naive}(P)$. ∎

Example 15.3 The following is a trace of the semi-naive evaluation of the programs in Example 15.1:

$$\begin{aligned} \Delta x_0 &= \{\mathit{edge}(a, b), \mathit{edge}(b, a)\} \\ \Delta x_1 &= \{\mathit{path}(a, b), \mathit{path}(b, a)\} \\ \Delta x_2 &= \{\mathit{path}(a, a), \mathit{path}(b, b)\} \\ \Delta x_3 &= \varnothing \end{aligned}$$

∎

The main advantage of the naive and semi-naive approach compared to SLD-resolution is that they terminate for some programs where SLD-resolution loops. In particular when no function symbols are involved (i.e. datalog programs). For instance, the goal $\leftarrow \mathit{path}(a, X)$ loops under SLD-resolution. On the other hand, there are also examples where the (semi-) naive approach loops and SLD-resolution terminates. For instance, consider the goal $\leftarrow \mathit{fib}(5, X)$ and the following program (extended with the appropriate definition of $\mathit{add}/3$):

$$\begin{array}{l} \mathit{fib}(0, 1). \\ \mathit{fib}(1, 1). \\ \mathit{fib}(X + 2, Y) \leftarrow \mathit{fib}(X + 1, Z), \mathit{fib}(X, W), \mathit{add}(Z, W, Y). \end{array}$$

The SLD-derivation terminates but both the naive and the semi-naive evaluation loop. The reason is that both naive and semi-naive evaluation blindly generate consequences without taking the goal into account. However, the fact $\mathit{fib}(5, 8)$ is obtained early on in the iteration. (In fact, if it was not for the addition it would be computed in the fourth iteration.)

Both naive and semi-naive evaluation also lend themselves to *set-oriented operations* in contrast to SLD-resolution which uses a tuple-at-a-time strategy. The set-oriented approach is often advantageous in database applications where data may reside on secondary storage and the number of disk accesses must be minimized.

15.3 Magic Transformation

This section presents a query-answering approach which combines the advantages of semi-naive evaluation with goal-directedness. The approach amounts to transforming the program P and a goal $\leftarrow A$ into a new program $\mathit{magic}(P \cup \{\leftarrow A\})$ which may be executed by the naive or semi-naive algorithm.

One of the problems with the semi-naive evaluation is that it blindly generates consequences which are not always needed to answer a specific query. This can be repaired by inserting a "filter" (an extra condition) into the body of each program clause $A_0 \leftarrow A_1, \ldots, A_n$ so that (an instance of) A_0 is a consequence of the program only if it is *needed* in order to compute an answer to a specific atomic goal.

For the purpose of defining such filters, the alphabet of predicate symbols is extended with one new predicate symbol *call_p* for each original predicate symbol p. If A is of the form $p(t_1, \ldots, t_n)$ then $\mathit{call}(A)$ will be used to denote the atom $\mathit{call\_p}(t_1, \ldots, t_n)$. Such an atom is called a *magic template*. The basic transformation scheme may be formulated as follows:

Definition 15.4 (Magic transformation) Let $\mathit{magic}(P)$ be the smallest program such that if $A_0 \leftarrow A_1, \ldots, A_n \in P$ then:

- $A_0 \leftarrow \mathit{call}(A_0), A_1, \ldots, A_n \in \mathit{magic}(P)$;
- $\mathit{call}(A_i) \leftarrow \mathit{call}(A_0), A_1, \ldots, A_{i-1} \in \mathit{magic}(P)$ for each $1 \leq i \leq n$.

∎

Given an initial goal $\leftarrow A$ a transformed clause of the form:

$$A_0 \leftarrow \mathit{call}(A_0), A_1, \ldots, A_n$$

can be interpreted as follows:

A_0 is true if A_0 is needed (to answer $\leftarrow A$) and $A_1, \ldots, A_n$ are true.

The statement "... is needed (to answer $\leftarrow A$)" can also be read as "... is called (in a goal-directed computation of $\leftarrow A$)". Similarly a clause of the form:

$$\mathit{call}(A_i) \leftarrow \mathit{call}(A_0), A_1, \ldots, A_{i-1}$$

can then be understood as follows:

A_i is called if A_0 is called and $A_1, \ldots, A_{i-1}$ are true.

Hence, the first clause extends each clause of the original program with a filter as described above and the second clause defines when a filter is true. The magic transformation can be said to encode a top-down computation with Prolog's computation rule. In fact, as will be illustrated below there is a close correspondence between the semi-naive evaluation of the magic program and the SLD-derivations of the original program.

Example 15.5 Let P be the program in Example 15.1. Then $\mathit{magic}(P)$ is the following program:

$$\begin{array}{l}
\mathit{path}(X, Y) \leftarrow \mathit{call\_path}(X, Y), \mathit{edge}(X, Y). \\
\mathit{path}(X, Y) \leftarrow \mathit{call\_path}(X, Y), \mathit{path}(X, Z), \mathit{edge}(Z, Y). \\
\mathit{edge}(a, b) \leftarrow \mathit{call\_edge}(a, b). \\
\mathit{edge}(b, a) \leftarrow \mathit{call\_edge}(b, a). \\
\\
\mathit{call\_edge}(X, Y) \leftarrow \mathit{call\_path}(X, Y). \\
\mathit{call\_path}(X, Z) \leftarrow \mathit{call\_path}(X, Y). \\
\mathit{call\_edge}(Z, Y) \leftarrow \mathit{call\_path}(X, Y), \mathit{path}(X, Z).
\end{array}$$

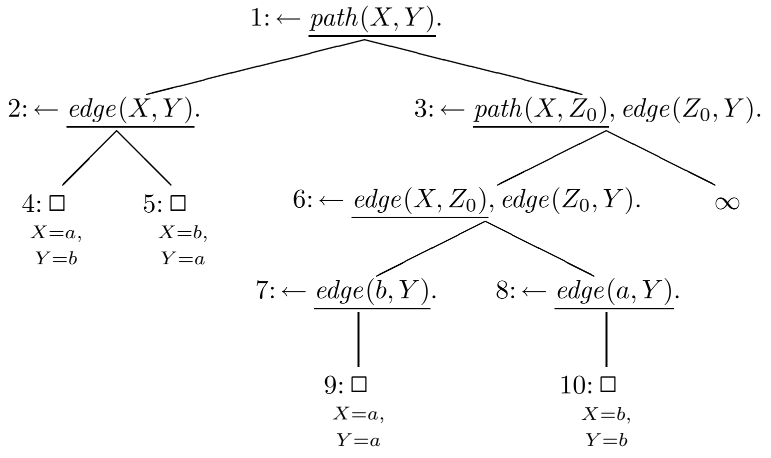


Figure 15.3: SLD-tree of $\leftarrow path(X, Y)$

For instance, note that the last clause may be read: "$edge(Z, Y)$ is called if $path(X, Y)$ is called and $path(X, Z)$ is true". Now compare this with the recursive clause of the original program in Example 15.1! ▮

Note that the program in the example does not contain any facts. Hence, no atomic formula can be a logical consequence of the program. In order to be able to use the magic program for answering a query the program has to be extended with such a fact. More precisely, in order to answer an atomic goal $\leftarrow A$ the transformed program must be extended with the fact $call(A)$. The fact may be read "A is called".

Example 15.6 Consider a goal $\leftarrow path(X, Y)$ to the program in Example 15.1. The semi-naive evaluation of the transformed program looks as follows:

$$\begin{aligned}
\Delta x_0 &= \{call\_path(X, Y)\}\\
\Delta x_1 &= \{call\_edge(X, Y)\}\\
\Delta x_2 &= \{edge(a, b), edge(b, a)\}\\
\Delta x_3 &= \{path(a, b), path(b, a)\}\\
\Delta x_4 &= \{call\_edge(a, Y), call\_edge(b, Y), path(a, a), path(b, b)\}\\
\Delta x_5 &= \varnothing
\end{aligned}$$

Hence, the evaluation terminates and produces the expected answers: $path(a, a)$, $path(a, b)$, $path(b, a)$ and $path(b, b)$. ▮

It is interesting to compare the semi-naive evaluation of the magic program with the SLD-tree of the goal $\leftarrow path(X, Y)$ with respect to the original program.

The selected subgoal in the root of the SLD-tree in Figure 15.3 is $path(X, Y)$. Conceptually this amounts to a call to the procedure $path/2$. In the magic computation this corresponds to the state before the first iteration. The first iteration generates the fact $call\_edge(X, Y)$ which corresponds to the selection of the subgoal $edge(X, Y)$ in node 2 of the SLD-tree. Simultaneously, $path(X, Z_0)$ is selected in node 3. However,

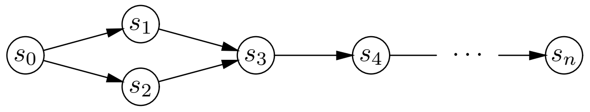


Figure 15.4: Duplicate paths

this is not explicitly visible in Δx_1 since *call_path*(X, Y) and *call_path*(X, Z_0) are renamings of each other. Iteration two yields two answers to the call to *edge*(X, Y); namely *edge*(a, b) and *edge*(b, a) corresponding to nodes 4 and 5 in the SLD-tree. These nodes also provide answers to the call *path*(X, Y) (and *path*(X, Z_0)) and correspond to the result of iteration three, and so forth.

The magic approach can be shown to be both sound and complete:

Theorem 15.7 (Soundness of magic) Let P be a definite program and $\leftarrow A$ an atomic goal. If $A\theta \in naive(magic(P \cup \{\leftarrow A\}))$ then $P \models \forall(A\theta)$. ▮

Theorem 15.8 (Completeness of magic) Let P be a definite program and $\leftarrow A$ an atomic goal. If $P \models \forall(A\theta)$ then there exists $A\sigma \in naive(magic(P \cup \{\leftarrow A\}))$ such that $A\theta$ is an instance of $A\sigma$. ▮

The magic approach combines advantages of naive (and semi-naive) evaluation with goal-directedness. In particular, it has a much improved termination behaviour over both SLD-resolution and naive (and semi-naive) evaluation of the original program P:

- If the SLD-tree of $\leftarrow A$ is finite then $naive(magic(P \cup \{\leftarrow A\}))$ terminates;
- If $naive(P)$ terminates, then $naive(magic(P \cup \{\leftarrow A\}))$ terminates;

Moreover, the magic approach sometimes avoids repeating computations. Consider the following program and graph in Figure 15.4:

$$\begin{array}{l} path(X,Y) \leftarrow edge(X,Y). \\ path(X,Y) \leftarrow edge(X,Z), path(Z,Y). \end{array}$$

Even if the SLD-tree of $\leftarrow path(s_0, X)$ is finite the tree contains two branches which are identical up to variable renaming — one that computes all paths from s_3 via s_0 and s_1 and one branch that computes all paths from s_3 via s_0 and s_2. By using semi-naive evaluation of the transformed program this is avoided since the algorithm computes a *set* of magic templates and answers to the templates.

15.4 Optimizations

The magic transformation described in the previous section may be modified in various ways to optimize the query-answering process. We give here a brief account of some potential optimizations without going too much into technical detail.

Supplementary magic

Each iteration of the naive evaluation amounts to computing:

$$\mathit{solve}((A_1, \ldots, A_n), I)$$

for each program clause $A_0 \leftarrow A_1, \ldots, A_n$. This in turn amounts to finding an mgu of sets of equations of the form:

$$\{A_1 \doteq B_1, \ldots, A_n \doteq B_n\}$$

If the program contains several clauses with common subgoals it means that the same unification steps are repeated in each iteration of the evaluation. (The same can be said about semi-naive evaluation.) This is a crucial observation for evaluation of magic programs as the magic transformation of a clause $A_0 \leftarrow A_1, \ldots, A_n$ gives rise to the following sub-program:

$$\begin{array}{rcl}
\mathit{call}(A_1) & \leftarrow & \mathit{call}(A_0). \\
\mathit{call}(A_2) & \leftarrow & \mathit{call}(A_0), A_1. \\
\mathit{call}(A_3) & \leftarrow & \mathit{call}(A_0), A_1, A_2. \\
 & \vdots & \\
\mathit{call}(A_n) & \leftarrow & \mathit{call}(A_0), A_1, A_2, \ldots, A_{n-1}. \\
A_0 & \leftarrow & \mathit{call}(A_0), A_1, A_2, \ldots, A_{n-1}, A_n.
\end{array}$$

Hence, naive and semi-naive evaluation run the risk of having to repeat a great many unification steps in each iteration of the evaluation.

It is possible to *factor* out the common subgoals using so-called *supplementary* predicates $\triangledown_0, \ldots, \triangledown_n$; Let $\mathbf{X}$ be the sequence of all variables in the original clause and let the supplementary predicates be defined as follows:

$$\begin{array}{rcl}
\triangledown_0(\mathbf{X}) & \leftarrow & \mathit{call}(A_0) \\
\triangledown_1(\mathbf{X}) & \leftarrow & \triangledown_0(\mathbf{X}), A_1 \\
 & \vdots & \\
\triangledown_n(\mathbf{X}) & \leftarrow & \triangledown_{n-1}(\mathbf{X}), A_n
\end{array}$$

Intuitively, $\triangledown_i(\mathbf{X})$ describes the state of the computation in a goal-directed computation of $A_0 \leftarrow A_1, \ldots, A_n$ after the success of A_i (or before A_1 if $i = 0$). For instance, the last clause states that "if $\triangledown_{n-1}(\mathbf{X})$ is the state of the computation before calling A_n and A_n succeeds then $\triangledown_n(\mathbf{X})$ is the state of the computation after A_n".

Using supplementary predicates the magic transformation may be reformulated as follows:

$$\begin{array}{rcll}
\mathit{call}(A_{i+1}) & \leftarrow & \triangledown_i(\mathbf{X}). & \qquad (0 \leq i < n) \\
A_0 & \leftarrow & \triangledown_n(\mathbf{X}). &
\end{array}$$

The transformation increases the number of clauses in the transformed program. It also increases the number of iterations in the evaluation. However, the amount of work in each iteration decreases dramatically since the clause bodies are shorter and avoids much of the redundancy due to duplicate unifications.

Note that no clause in the new sub-program contains more than two body literals. In fact, the supplementary transformation is very similar in spirit to Chomsky Normal Form used to transform context-free grammars (see Hopcroft and Ullman (1979)).

Subsumption

In general we are only interested in "most general answers" to a goal. That is to say, if the answer A_1 is a special case of A_2 then we are only interested in A_2. (This is why the definition of SLD-resolution involves only most general unifiers.) In a naive or semi-naive evaluation it may happen that the set being computed contains two atoms, A_1 and A_2, where A_1 is an instance of A_2 (i.e. there is a substitution θ such that $A_1 = A_2\theta$). Then A_1 is said to be *subsumed* by A_2. In this case A_1 is redundant and may be removed from the set without sacrificing completeness of the query-answering process. Moreover, keeping the set as small as possible also improves performance of the algorithms. In the worst case, redundancy due to subsumption may propagate leading to an explosion in the size of the set.

From a theoretical perspective it is easy to extend both naive and semi-naive evaluation with a normalization procedure which removes redundancy from the set. However, checking for subsumption may be so expensive from the computational point of view (and is so rarely needed), that it is often not used in practice.

Sideways information passing

As commented the magic transformation presented in Definition 15.4 encodes a goal-directed computation where the subgoals are solved in the left to right order. Hence, given a clause:

$$A_0 \leftarrow A_1, \ldots, A_n$$

the transformed program contains a clause:

$$\mathit{call}(A_i) \leftarrow \mathit{call}(A_0), A_1, \ldots, A_{i-1}$$

In addition to imposing an ordering on the body atoms, the clause also propagates bindings of $\mathit{call}(A_0)$ and $A_1, \ldots, A_{i-1}$ to $\mathit{call}(A_i)$. Now two objections may be raised:

- The left to right goal ordering is not necessarily the most efficient way of answering a query;
- Some of the bindings may be of no use for $\mathit{call}(A_i)$. And even if they are of use, we may not necessarily want to propagate all bindings to it.

Consider the following sub-program which checks if two nodes (e.g. in a tree) are on the *same depth*. That is, if they have a common ancestor the same number of generations back.

$$\begin{array}{l} sd(X, X). \\ sd(X, Y) \leftarrow \mathit{child}(X, Z), \mathit{child}(Y, W), sd(Z, W). \end{array}$$

Note that there is no direct flow of bindings between $\mathit{child}(X, Z)$ and $\mathit{child}(Y, W)$ in a goal-directed computation. Hence the two subgoals may be solved in parallel. However, the recursive call $sd(X, Z)$ relies on bindings from the previous two, and should probably await the success of the other subgoals. Now, the magic transformation imposes a linear ordering on the subgoals by generating:

$$
\begin{aligned}
&\mathit{call\_child}(X, Z) \leftarrow \mathit{call\_sd}(X, Y).\\
&\mathit{call\_child}(Y, W) \leftarrow \mathit{call\_sd}(X, Y), \mathit{child}(X, Z).\\
&\mathit{call\_sd}(Z, W) \leftarrow \mathit{call\_sd}(X, Y), \mathit{child}(X, Z), \mathit{child}(Y, W).
\end{aligned}
$$

In this particular case it would probably be more efficient to emit the clauses:

$$
\begin{aligned}
&\mathit{call\_child}(X, Z) \leftarrow \mathit{call\_sd}(X, Y).\\
&\mathit{call\_child}(Y, W) \leftarrow \mathit{call\_sd}(X, Y).\\
&\mathit{call\_sd}(Z, W) \leftarrow \mathit{call\_sd}(X, Y), \mathit{child}(X, Z), \mathit{child}(Y, W).
\end{aligned}
$$

Intuitively this means that the two calls to *child*/2 are carried out "in parallel" as soon as *sd*/2 is called. The recursive call, on the other hand, goes ahead only if the two calls to *child*/2 succeed.

This example illustrates that there are variations of the magic transformation which potentially yield more efficient programs. However, in order to exploit such variations the transformation must be parameterized by the strategy for solving the subgoals of each clause. Which strategy to use relies on the flow of data between atoms in the program and may require global analysis of the program. Such strategies are commonly called sip's (sideways information passing strategies) and the problem of generating efficient strategies is an active area of research (see Ullman (1989) for further reading).

Exercises

15.1 Transform the following program using Definition 15.4:

$$
\begin{aligned}
&\mathit{expr}(X, Z) \leftarrow \mathit{expr}(X, [+|Y]), \mathit{expr}(Y, Z).\\
&\mathit{expr}([\mathit{id}|Y], Y).
\end{aligned}
$$

Then use naive and semi-naive evaluation to "compute" answers to the goal:

$$\leftarrow \mathit{expr}([\mathit{id}, +, \mathit{id}], X).$$

What happens if the goal is evaluated using SLD-resolution and the original program?

15.2 Consider the program *sd*/2 on p. 238 and the following "family tree":

$$
\begin{array}{lllll}
\mathit{child}(b, a). & \mathit{child}(c, a). & \mathit{child}(d, b). & \mathit{child}(e, b). & \mathit{child}(f, c).\\
\mathit{child}(g, d). & \mathit{child}(h, d). & \mathit{child}(i, e). & \mathit{child}(j, f). & \mathit{child}(k, f).
\end{array}
$$

Transform the program using (a) magic templates (b) supplementary magic. Then compute the answers to the goal:

$$\leftarrow sd(d, X)$$

Appendix A

Bibliographical Notes

A.1 Foundations

LOGIC: Logic is the science of valid reasoning and its history dates back to 300-400 B.C. and the work of Aristotle. His work predominated for over 2000 years until logic finally begun to take its current shape around 100 years ago. That is, long before the era of electronic computers. In this historical perspective it is not surprising that logic programming, in many ways, builds on fundamentally different principles than most existing (algorithmic) programming languages. In fact, many of the results which constitute the core of logic programming (including some used in this book) actually date back to the early 20th century. For instance, the name *Herbrand interpretation* is given in honour to the French logician Herbrand. However, the ideas were first introduced around 1920 by Skolem and Löwenheim. Theorem 2.12 is a consequence of the so-called Skolem-Löwenheim theorem.

There are a large number of introductory readings on mathematical logic. Books by Shoenfield (1967), van Dalen (1983), Galton (1990) and Mendelson (1987) are recommended. For readers already familiar with the basic concepts of predicate logic the book by Boolos and Jeffrey (1980) provides a good starting point for further studies. An account by Davis of the early history of mathematical logic and its influence on computer science can be found in the collection of Siekmann and Wrightson (1983a).

With the introduction of electronic computers it was widely believed that it was only a matter of time before computers were able to reason intelligently by means of logic. Much research was devoted to this field (commonly called *automated theorem proving* or *automated reasoning*) during the 1960s. Many of the most influential papers from this era are collected by Siekmann and Wrightson (1983a; 1983b). Good introductions to theorem proving and different kinds of resolution methods are provided by Chang and Lee (1973) and Robinson (1979). The basis of both of these books, and for logic programming, is the work of Robinson (1965) where he introduced the notion of *unification* and the *resolution principle* for predicate logic.

DEFINITE PROGRAMS: Logic programming emerged with the motivation to improve the efficiency of theorem proving. As already stated the proposed solution was to take a subset of the language of predicate logic called definite clauses and to use the specialized inference rule known as the SLD-resolution principle. Definite clauses are sometimes also called Horn clauses after the French logician Horn.

The observation that every definite programs has a unique minimal Herbrand model (expressed in the Theorems 2.14, 2.16 and 2.20) originates from van Emden and Kowalski's landmark paper (1976). Proofs similar to those provided in Chapter 2 can be found also in Apt (1990) and Lloyd (1987).

The name *immediate consequence operator* was coined by Clark (1979) and uses results of the theory of fixed points due to Tarski (1955) and Scott (1976). It was shown by van Emden and Kowalski (1976) that the least fixed point of T_P is identical to the model-theoretic meaning of definite programs (Theorem 2.20). A corresponding result relating the greatest fixed point of T_P to the subset of the Herbrand base whose members finitely fails (that is, has a finite SLD-tree without any refutations) was provided by Apt and van Emden (1982). For a comprehensive account of the fixed point semantics of definite programs, see Apt (1990) and Lloyd (1987). Several alternative fixed point characterizations of logic programs have also been proposed. A survey is provided by Bossi et al. (1994).

SLD-RESOLUTION: SLD-resolution is an offspring from SL-resolution which was first described by Kowalski and Kuehner (1972). SLD-resolution was originally called LUSH-resolution (Linear resolution with Unrestricted Selection for Horn clauses) by Hill (1974). However, it was first described (without being named) by Kowalski (1974). The formulation of soundness for SLD-resolution (Theorem 3.20) is due to Clark (1979). Versions of this proof can be found in Apt (1990), Doets (1994) and Lloyd (1987). The first completeness theorem for SLD-resolution was reported by Hill (1974) but the formulation of the stronger result given in Theorem 3.22 is due to Clark (1979). The actual proof is not very difficult and can be found in Apt (1990) and Lloyd (1987). A much simplified proof was subsequently presented by Stärk (1990). See also Doets (1994). The proof is made possible by representing an SLD-refutation as a tree closely reminiscent of the derivation trees of Section 3.6.

The core of the resolution principle is the *unification* algorithm. The algorithm as we know it today is usually attributed to the landmark paper of Robinson (1965). Although its origin is not entirely clear the concept of unification goes back to results of Herbrand (1967) and Prawitz (1960). However, several alternative definitions and algorithms have been proposed as discussed by Lassez, Maher and Marriott (1988). Some important results related to unification are also described by Eder (1985). A general introduction to unification is provided by Siekmann (1984). Various applications, implementation techniques and generalizations are discussed by Knight (1989).

Several attempts have been made to come up with unification algorithms that are not (worst case) exponential. Some of these are reported to have (worst case) linear time complexity (see Martelli and Montanari (1982) or Paterson and Wegman (1978)). However, the ISO Prolog standard (1995) and most Prolog implementations "solve the problem" simply by omitting the occur-check. This may in rare cases lead to unsound conclusions. Unfortunately, the problem of checking if occur-check is actually needed is undecidable. Much research has therefore focused on finding sufficient conditions when occur-check can be safely omitted (for details see Apt and Pellegrini (1992), Beer (1988), Chadha and Plaisted (1994), Deransart, Ferrand and Téguia (1991), Marriott and Søndergaard (1989) or Plaisted (1984)).

NEGATION: As discussed in Chapter 2, definite programs and SLD-resolution cannot be used to infer negative knowledge. The solution to this shortcoming is similar to that adopted in relational databases — if something is not inside the definition of a relation it is assumed to be outside of the relation. This idea, commonly called the *closed world assumption*, is due to Reiter (1978). Clark suggested a weaker notion called the *negation as finite failure* rule (1978). Clark provided a logical justification of the this rule by introducing the notion of *completion* and showed the soundness (Theorem 4.4) of the rule. Completeness of negation as finite failure (Theorem 4.5) was provided by Jaffar, Lassez and Lloyd (1983).

There seems to be no general agreement whether to use the term *general* or *normal* program for logic programs containing negative body literals. Moreover, several definitions of SLDNF-resolution can be found in the literature. Many definitions of SLDNF-derivations and SLDNF-trees are problematic. For instance, the definitions of Lloyd (1987) and Nilsson and Małuszynski (1990) apply only to a restricted class of general programs. By a construction similar to the search forest of Bol and Degerstedt (1993b) Definition 4.13 avoids such problems. The definition is similar to that of Apt and Doets (1994) (see also Doets (1994) or Apt and Bol (1994)), but is less operational in nature. The proof of the soundness of SLDNF-resolution was provided by Clark (1978) and can also be found in Lloyd (1987).

As pointed out, SLDNF-resolution is not complete. However, there are several completeness results for restricted classes of programs. Cavedon and Lloyd (1989) showed the completeness for a limited class of stratified programs. Later Stroetmann reported a more general result (1993). Fitting (1985) and Kunen (1987; 1989) suggested a three-valued interpretation of the completion semantics which better corresponds to the intuition of negation as finite failure. With the three-valued completion as a basis Kunen (1989) proved completeness of SLDNF-resolution for allowed programs. Later Stärk (1992; 1993) and Drabent (1995a) strengthened these results.

The notion of *stratified* programs was introduced by Apt, Blair and Walker (1988). The notion was also independently put forward by Van Gelder (1988). Both of these papers build on results by Chandra and Harel (1985). Apt, Blair and Walker showed that every stratified program has a well-defined minimal model called the *standard* model and in the case of definite programs it coincides with the least Herbrand model. Later Przymusinski (1988a; 1988b) extended the class of stratified programs into *locally* stratified programs and the standard model into the *perfect model semantics.* Roughly speaking, a program is locally stratified if the Herbrand base can be partitioned in such a way that for every ground instance of each program clause, the head appears in higher or equal stratum than all positive literals in the body and strictly higher stratum than negative literals in the body. Unfortunately, the property of being locally stratified is undecidable whereas ordinary stratification is not.

The notion of *well-founded models* was introduced by van Gelder, Ross and Schlipf (1991). The well-founded model coincides with the standard model (resp. the perfect model) in the case of stratified (resp. locally stratified) programs. By allowing partial (or three-valued) interpretations it applies to arbitrary general programs. Several alternative characterizations of the well-founded semantics can be found in the literature (see Apt and Bol (1994)). An alternative to the well-founded semantics was proposed by Gelfond and Lifschitz (1988) who suggested a notion of *stable models*. The stable model semantics coincides with perfect model semantics (and thus, well-founded semantics) in the case of locally stratified programs. However, in the general case it assigns a *set* of minimal models to a programs.

There is no well-established equivalent of SLDNF-resolution for computing answers to goals with the well-founded semantics (or stable model semantics) as the underlying declarative semantics. Przymusinski (1989) and Ross (1992) suggested a notion of (global) SLS-resolution which is an idealistic and non-effective procedural semantics for computing answers to goals using well-founded semantics as the underlying declarative semantics. An alternative approach based on the notion of search forest was proposed by Bol and Degerstedt (1993a). A similar idea was independently suggested by Chen and Warren (1993).

One of the main shortcomings of the negation as finite failure rule is the disability to fully handle existentially quantified negative subgoals — that is, queries which intuitively read "is there an X such that $\neg p(X)$?". This restriction has motivated a number of techniques with the ability of producing answers (roughly speaking, bindings for X) to this type of goals. This area of research is commonly called *constructive* negation. For further reading see Chan (1988), Drabent (1995b) or Małuszyński and Näslund (1989).

As pointed out in Chapter 3 negation as implemented in Prolog, is unsound. In particular, the subgoal $\neg A$ fails when the goal $\leftarrow A$ succeeds. In a sound implementation it is necessary to check that the computed answer substitution for $\leftarrow A$ is empty. Still there are implementations such as NU-Prolog (cf. Thom and Zobel (1987)), which incorporate sound versions of negation. In addition NU-Prolog allows variables to be existentially quantified within the scope of "$\neg$". This, and other features of NU-Prolog, are reported by Naish (1985) and (1986).

For a fine and extensive survey of negation in logic programming see Apt and Bol (1994). Surveys are also provided e.g. by Shepherdson (1988).

CUT AND ARITHMETIC: Cut and built-in arithmetic are part of the ISO Prolog standard (1995). Our discussion on the effects of cut is somewhat simplified since it does not take into account the effects of combining cut with other built-in predicates of Prolog. For an extensive treatment of different uses and effects of cut see O'Keefe (1990). Several logic programming languages have tried to introduce a cleaner approach to built-in arithmetic than that employed in the Prolog standard. For details, see e.g. Małuszyński et al. (1993).

A.2 Programming in Logic

DEDUCTIVE DATABASES: As pointed out in Chapter 6 there are many similarities between relational databases (as first defined by Codd (1970)) and logic programming in that they are both used to describe relations between objects. There has been a growing interest in the database community to use logic programs as a *language* for representing *data*, *integrity constraints*, *views* and *queries* in a single uniform framework. Several survey articles of the field of deductive databases are available. For instance, both Gallaire, Minker and Nicolas (1984) and Reiter (1984) provide extensive comparison between logic and relational databases, but use a richer logical language than that normally found in the logic programming literature. Ullman (1988) and (1989) provides a thorough introduction both to traditional database theory and the use of logic programming for describing relational databases. Minker (1988) gives a historical account of the field of deductive databases and discusses its relation to negation.

Several suggestions have been put forward on how to increase the expressive power of deductive databases. In a series of papers Lloyd and Topor (1984; 1985; 1986) suggest several extensions. The main idea is to extend logic programs to include a notion of *program clauses* which are formulas of the form $\forall(A \leftarrow F)$ where A is an atom and F is an arbitrary *typed* formula of predicate logic. It is shown how to compile program clauses into Prolog programs. They also raise the problem of how to handle *integrity constraints*. Roughly speaking, an integrity constraint is a formula which constrains the information which may be stored in the database. The validity of the constraints must be checked every time updates are made to the database. Lloyd, Sonenberg and Topor (1987) provide a method for checking the validity of integrity constraints in the case of stratified databases. A recapitulation of these results is also available in Lloyd (1987).

The notion of integrity constraints concerns *updates* in databases. The semantics of database updates is a major problem, not only in logic programming systems, but in any system which must maintain consistent information. The problems become particularly difficult when the updates are made while making deductions from the database, since adding information to or deleting information from the database may invalidate conclusions already made. These are problems which have engaged quite a number of researchers in different fields. A common suggestion is to treat the database as a collection of theories and to specify, explicitly, in which theory to prove subgoals. Some alternative approaches have been suggested by Bacha (1987), Bowen (1985), Bowen and Kowalski (1982), Hill and Lloyd (1988b) and Warren (1984).

RECURSIVE DATA-STRUCTURES: Most of the programs in Chapter 7 (together with programs operating on other recursive data structures) can be found in Prolog monographs such as Clocksin and Mellish (1994), Sterling and Shapiro (1994) and O'Keefe (1990). *Difference lists* were discussed together with several other data-structures by Clark and Tärnlund (1977). Ways of transforming programs operating on lists into programs operating on difference lists were discussed by Hansson and Tärnlund (1981), Zhang and Grant (1988) and Marriott and Søndergaard (1988).

META-LOGICAL REASONING: The idea to describe a language in itself is not new. Many of the most important results on computability and incompleteness of predicate logic are based on this idea. For instance, Gödel's incompleteness theorem and the undecidability of the halting problem (see e.g. Boolos and Jeffrey (1980) for a comprehensive account of these results). Bowen and Kowalski (1982) raised the possibility of amalgamating the object- and meta-language in the case of logic programming. The self-interpreter based on the ground representation presented in Chapter 8 is influenced by their interpreter. Extensions of Bowen's and Kowalski's ideas are reported by Bowen and Weinberg (1985) and Bowen (1985).

The self-interpreter in Example 8.7 which works on a nonground representation seems to have appeared for the first time in Pereira, Pereira and Warren (1979). O'Keefe (1985) described an extension of the self-interpreter which handles cut. See also O'Keefe (1990).

Hill and Lloyd (1988a) pointed out several deficiencies of existing attempts to formalize meta-level reasoning in Prolog. Their solution is to use a language of typed predicate logic and a ground representation of the object language. The solution makes it possible to give a clean logical meaning to some of Prolog's built-in predicates. Hill and Lloyd (1988b) also provide a clean semantics for the predicates *assert*/1 and *retract*/1. These ideas were incorporated in the logic programming language Gödel (see Hill and Lloyd (1994) for details).

Meta-logical reasoning is closely related to the area of *program transformation*. On the one hand program transformation is a special case of meta-level reasoning. On the other hand program transformation can be used to improve the efficiency of meta-level programs. In particular *partial evaluation* plays an important role here. The notion of partial evaluation was suggested in the 1970s and was introduced into logic programming by Komorowski (1981). The approach has gained considerable interest because of its ability to "compile away" the overhead introduced by having one extra level of interpretation. Several papers on partial evaluation of logic programs are collected in Ershov (1988) and annotated bibliographies are available in Bjørner, Ershov and Jones (1988).

EXPERT SYSTEMS: Expert systems as described in Chapter 9, are particular applications of meta-logical reasoning. An early account of the application of logic programming in the field of expert systems was provided by Clark and McCabe (1982). The example in Chapter 9 is based on the technique of composing self-interpreters suggested by Sterling and Lakhotia (1988). Similar expert-system shells are described e.g. by Sterling and Beer (1989) and Sterling and Shapiro (1994). To incorporate probabilities see Shapiro (1983b).

For a survey of abductive reasoning in logic programming see Kakas, Kowalski and Toni (1992).

DEFINITE CLAUSE GRAMMARS: One of the first applications of logic programming was that of formalizing natural language. In fact, the very first implementation of Prolog — made by Colmerauer's group in Marseilles (see Roussel (1975)) — was primarily used for processing of natural language (e.g. Colmerauer et al. (1973)). Since then several results relating logic programming and various grammatical formalisms have been published. For an extensive account of the area see Deransart and Małuszyński (1993).

The notion of Definite Clause Grammar (DCG) was introduced by Warren and Pereira (1980) and incorporated into the DEC-10 Prolog system developed at the University of Edinburgh. However, the basic idea is an adaptation of Colmerauer's *Metamorphosis Grammars* (1978). The form of DCGs described in Chapter 10 may deviate somewhat from that implemented in most Prolog systems. In most implementations DCGs are viewed merely as a syntactic sugar for Prolog and, as a consequence, all of Prolog built-in features (including cut, negation etc.) may be inserted into the grammar rules. Any user's manual of specific Prolog systems that support DCGs can fill in the remaining gaps.

The simple translation of DCGs into Prolog clauses shown in Section 10.5 is by no means the only possibility. Matsumoto et al. (1983) describe a left-corner bottom-up strategy. Nilsson (1986) showed how to translate an arbitrary DCG into a Prolog program which embodies the LR(k) parsing technique. Finally, Cohen and Hickey (1987) describe a whole range of parsing techniques and their use in compiler construction.

A large number of formalisms similar to DCGs have been suggested. Some of the most noteworthy are Abramson's *Definite Clause Translation Grammars* (1984) (which are closely related to attribute grammars) and *Gapping Grammars* by Dahl and Abramson (1984). These, and other formalisms, are surveyed by Dahl and Abramson (1989). For an extensive account of the use of Prolog in natural language processing, see Dahl (1994) and monographs by Pereira and Shieber (1987) (who also make extensive use of partial evaluation techniques) and Gazdar and Mellish (1989).

SEARCHING: Chapter 11 presents some fundamental concepts related to the problem of searching in a state space. Several other, more advanced techniques can be found in textbooks by Bratko (1990), Sterling and Shapiro (1994), Clocksin and Mellish (1994) and O'Keefe (1990).

A.3 Alternative Logic Programming Schemes

CONCURRENCY: There exists a number of logic programming languages based on a concurrent execution model. Three of the most influential are *PARLOG*, *Guarded Horn Clauses* and *Concurrent Prolog*, but several others have been suggested. They all originate from the experimental language *IC-Prolog* developed around 1980 by Clark, McCabe and Gregory (1982) at Imperial College, London. The main feature of this language was its execution model based on pseudo-parallelism and coroutining as suggested by Kowalski (1979a; 1979b). In IC-Prolog this was achieved by associating control-annotations to variables in the clauses. The language also had a concept of *guards* but no *commit operator* in the sense of Chapter 12. IC-Prolog was succeeded by the *Relational Language* of Clark and Gregory (1981) which introduced guards and the commit operator inspired by Dijkstra's guarded commands (1976) and Hoare's CSP (1985). The synchronization between subgoals was specified by means of *mode-declarations* — that is, annotations which describe how arguments of calls to predicates are to be instantiated in order for the call to go ahead. Unfortunately, the modes were so restricted that programs in the language more or less behaved as functional programs with relational syntax. However, some of these restrictions were relaxed in the successor language called PARLOG (for details, see Clark and Gregory (1986) and Gregory (1987)).

Concurrent Prolog was developed by Shapiro (1983a; 1986) as a direct descendant of the Relational Language. The language is closely related to PARLOG but differs in some respects. In particular, in Concurrent Prolog synchronization between subgoals is achieved by means of *read-only* annotations on variables as opposed to PARLOG where the same effect is obtained by means of mode-declarations.

Guarded Horn Clauses (GHC) suggested by Ueda (1985) is also based on the same idea as PARLOG and Concurrent Prolog. In contrast to the previous two, GHC does not provide any explicit declarations for synchronization. Instead a subgoal suspends if it is unable to commit to a clause without binding variables in the call.

A large number of papers on concurrent logic programming languages are collected by Shapiro (1988). Shapiro (1989) also provides a survey of most existing concurrent logic programming languages.

EQUATIONAL LOGIC PROGRAMMING: A large number of logic programming languages attempt to combine logic programming with *functional* programming. This may be achieved in three different ways:

- integration of logic programming on top of some existing functional language. LOGLISP of Robinson and Sibert (1982), Komorowski's QLOG (1982) and POPLOG of Mellish and Hardy (1984) are examples of this approach;
- a logic programming language able to call functions defined in arbitrary languages through a well-defined interface (cf. Małuszyński et al (1993));
- defining a new language in which it is possible to write both logic and functional programs. This approach is represented by languages such as LEAF (cf. Barbuti et al. (1986), FUNLOG (cf. Subrahmanyam and You (1986)). BABEL (cf. Moreno-Navarro and Rodriguez-Artalejo (1992)) and ALF (Hanus (1992)).

All of these have their own merits depending on whether one is interested in efficiency or logical clarity. The first approach is usually the most efficient whereas the third is probably the most attractive from a logical point of view and also the one that is most closely related to what was said in Chapter 13.

The third approach usually consists of extending logic programming with equational theories. Operationally, it amounts to extending SLD-resolution with some form of equation solving — for instance, using different adaptations of *narrowing* (cf. Slagle (1974)). For further reading see the work of Hullot (1980), Dershowitz and Plaisted (1988) or the excellent survey by Huet and Oppen (1980). However, as pointed out in Chapter 13, equation solving without restrictions is likely to end up in infinite loops. Much of the research is therefore directed towards finding more efficient methods and special cases when the technique is more likely to halt.

Surveys describing integration of logic and functional programming are provided by Bellia and Levi (1985) and Hanus (1994). Some of the papers cited above are collected by DeGroot and Lindstrom (1986). The generalization of model- and proof-theoretic semantics from definite programs into definite programs with equality is due to Jaffar, Lassez and Maher (1984; 1986). Gallier and Raatz (1986) provide basic soundness and completeness results for SLD-resolution extended with E-unification.

A slightly different, but very powerful, approach to combining logic programming with functional programming can be obtained by exploiting higher-order unification of λ-terms (cf. Huet (1975) or Snyder and Gallier (1990)). This idea has been exploited in λ-Prolog developed by Miller et al. at the University of Pennsylvania (see Nadathur and Miller (1995)).

CONSTRAINTS: The use of constraints (see Leler (1988) and Steele (1980)) in logic programming is closely related to the integration of logical and functional languages. Colmerauer's Prolog II (1982; 1984), now succeeded by Prolog III (1990), seems to be the first logic programming language that makes extensive use of constraints. In the case of Prolog II the constraints are restricted to equalities and disequalities over rational trees. Jaffar and Lassez (1987) lay the foundation for combining logic programming with other constraint domains by providing a parameterized framework $CLP(X)$, where X may be instantiated to various domains. An instance of the scheme — $CLP(\mathcal{R})$ where $\mathcal{R}$ stands for the domain of real numbers — was implemented at Monash University, Australia (see Heintze, Michaylov and Stuckey (1987a) and Jaffar and Michaylov (1987)). Several applications of the system have been demonstrated. For instance, in electrical engineering (Heintze et al. (1987b)) and in option trading (see Lassez, McAloon and Yap (1987)).

Several other constraint logic programming systems have been proposed. *CAL*, by Aiba et al. (1988), supports (non-)linear algebraic polynomial equations, boolean equations and linear inequalities. The language *CHIP* supports equations over finite domains, Booleans and rational numbers (see Dincbas et al. (1988) and Van Hentenryck (1989)). CHIP subsequently split into several successor languages (such as Sepia and cc(FD)). Prolog III provides constraints over the binary Boolean algebra, strings and linear equations over the reals and rational numbers (see Colmerauer (1990)). CLP(BNR) contains constraints over real and integer intervals, finite domains and the Boolean algebra. The language CLP(Σ^*) supports constraints over domains of regular sets (see Walinsky (1989)). The language LIFE supports constraints over a domain of order-sorted feature trees (see Aït-Kaci and Podelski (1993)).

Imbert, Cohen and Weeger (1993) describe an incremental and efficient algorithm for testing the satisfiability of linear constraints. They also describe how to incorporate the algorithm into a Prolog meta-interpreter.

Constraints may also be combined with concurrent logic programming as shown by Maher (1987). Saraswat (1993) proposed a family of concurrent constraint (logic) programming languages. The language AKL (Agents Kernel Language)[1] is a multi-paradigm language which combines constraints with don't-care non-deteminism of concurrent logic programming and (a restricted form of) don't-know non-determinism of Prolog (see Janson (1994)). AKL provides constraints over finite domains and is in several respects similar to the language Oz developed by Smolka (see Schulte, Smolka and Würtz (1994)).

Jaffar and Maher (1994) provide an excellent survey of the theory, implementation and applications of constraint logic programming.

QUERY-ANSWERING IN DEDUCTIVE DATABASES: Two main streams can be singled out in query-processing of deductive databases. One approach is based on naive- or semi-naive evaluation of a transformed program. A large number of transformations have been proposed for various classes of programs. Most notably magic sets of Bancilhon, Maier, Sagiv and Ullman (1986) and magic templates of Ramakrishnan (1988). For an introduction to these techniques and other methods such as *counting*, *envelopes* see Bancilhon and Ramakrishnan (1988).

[1]Formerly called the ANDORRA Kernel Language (see Haridi and Brand (1988)).

The other main approach is to extend SLD-resolution with tabulation or dynamic programming techniques. That is, methods were results are tabulated and re-used when needed instead of being recomputed. Examples of this approach are Earley deduction of Pereira and Warren (1983), OLDT-resolution of Tamaki and Sato (1986), SLD-AL-resolution of Vieille (1989) and the search forest approach of Bol and Degerstedt (1993b). As illustrated by the example in Chapter 15 and as shown by Bry (1990) there is a strong correspondence between both methods. See also Warren (1992) for a survey of tabulation techniques.

For an introduction to goal-ordering see Ullman (1985) or (1989). The notion of sideways information passing (*sip*) was formally defined by Beeri and Ramakrishnan (1987) who also introduced the transformation based on supplementary predicates. Both Ullman (1988; 1989) as well as Abiteboul, Hull and Vianu (1995) provide extensive introductions to foundations of deductive databases.

Several deductive database systems have been developed including Coral, NAIL, Aditi and LDL. For an introduction to deductive database systems and applications of deductive databases, see the collection of Ramakrishnan (1995).

Appendix B

Basic Set Theory

This appendix contains a brief summary of basic set theoretic notions used in the book. It is not intended to be an introduction. To this end we recommend reading Gill (1976) or Grimaldi (1994) or some other introductory textbook on discrete mathematics.

B.1 Sets

By a *set* we mean an aggregate, or a collection, of objects. The objects that belong to a set are called the *elements* of the set. The notation $x \in S$ is used to express the fact that x is an element in S. Similary $x \notin S$ expresses the fact that x is *not* an element in S. There are several ways of writing a set: a set with a small, finite number of elements is usually written out in its entirety. For instance, $\{0, 1, 2, 3\}$ denotes the set of all natural numbers that are less than 4. For larger sets, such as the set of all natural numbers less than 1000, the notation $\{0, 1, 2, \ldots, 999\}$ or $\{x \mid 0 \leq x < 1000\}$ may be used. Note that a set of the form $\{x_1, \ldots, x_n\}$ is always assumed to be finite in contrast to a set of the form $\{x_1, x_2, \ldots\}$ which is allowed to be infinite. The empty set is written $\varnothing$. The set of all natural numbers $\{0, 1, 2, \ldots\}$ is denoted by $\mathbb{N}$. Similarly $\mathbb{Z}$, $\mathbb{Q}$ and $\mathbb{R}$ denote the sets of integers, rational numbers and real numbers respectively.

The *union*, $S_1 \cup S_2$, of two sets, S_1 and S_2, is the set of all elements that belong either to S_1 or to S_2. The *intersection*, $S_1 \cap S_2$, is the set of all elements that belong both to S_1 and S_2 and the *difference*, $S_1 \setminus S_2$, is the set of all elements that belong to S_1 but not to S_2. A set S_1 is said to be a *subset* of S_2 (denoted $S_1 \subseteq S_2$) if every element of S_1 is also an element of S_2. The set that consists of all subsets of a set, S, is called the *powerset* of S. This set is denoted $\wp(S)$.

The *cartesian product*, $S_1 \times S_2$, of two sets is the set of all pairs $\langle x_1, x_2 \rangle$ such that $x_1 \in S_1$ and $x_2 \in S_2$. This can be extended to any number of sets. Thus, $S_1 \times \cdots \times S_n$ denotes the set of all n-tuples $\langle x_1, \ldots, x_n \rangle$ such that $x_i \in S_i$ $(1 \leq i \leq n)$. If $S_i = S$ (for all $1 \leq i \leq n$) we usually write S^n instead of $S \times \cdots \times S$.

Let S be a set. By S^* we denote the set of all *strings* (sequences) of elements from S. That is, $S^* = \{x_1, \ldots, x_n \mid n \geq 0 \land x_i \in S \ (1 \leq i \leq n)\}$. Note that S^* contains the string of length 0, denoted ϵ and called the *empty string*.

B.2 Relations

Let $S_1, \ldots, S_n$ be sets and let $R \subseteq S_1 \times \cdots \times S_n$. Then R is said to be a *relation* (over $S_1, \ldots, S_n$). The fact that $\langle x_1, \ldots, x_n \rangle \in R$ is usually written $R(x_1, \ldots, x_n)$. We say that R is an n-ary relation. When $n = 0, 1, 2, 3$ we say that R is nullary, unary, binary and ternary respectively.

A binary relation over S, is said to be *reflexive* if $R(x, x)$ for every $x \in S$. The relation is said to be *symmetric* if $R(x, y)$ whenever $R(y, x)$ and *transitive* if $R(x, z)$ whenever both $R(x, y)$ and $R(y, z)$. The relation is said to be *anti-symmetric* if $R(x, y)$ and $R(y, x)$ imply that $x = y$. A relation that is reflexive, transitive and anti-symmetric is called a *partial order* and a relation that is reflexive, transitive and symmetric is called an *equivalence relation*.

A binary relation $R \subseteq S \times S$ such that $R(x, y)$ iff $x = y$ is called the *identity relation*.

B.3 Functions

A binary relation $f \subseteq S_1 \times S_2$ is called a *function* (or a *mapping*) if whenever $f(x, z)$ and $f(y, z)$ then $x = y$. We say that the function f *assigns* the value z to x (or *maps* x on z) and write this as $f(x) = z$. The set S_1 is called the *domain* of the function and S_2 is called its *codomain*. It is common practice to abbreviate this as $f: S_1 \to S_2$.

A function $f: S_1 \to S_2$ is said to be *total* if for every $x \in S_1$ there exists an element y in S_2 such that $f(x) = y$. Otherwise f is said to be *partial*.

The *composition*, $f_2 \circ f_1$, of two functions $f_1: S_1 \to S_2$ and $f_2: S_2 \to S_3$, is itself a function — with domain S_1 and codomain S_3 — with the property that $(f_2 \circ f_1)(x) = f_2(f_1(x))$, for any $x \in S_1$.

A function $f: S_1 \to S_2$ is called a *bijection* if it is total and $x = y$ whenever $f(x) = f(y)$. Bijections are sometimes called *one-to-one*-mappings. Every bijection has a invers $f^{-1}: S_2 \to S_1$ which satisfies $f(f^{-1}(x)) = f^{-1}(f(x)) = x$.

Appendix C

Answers to Selected Exercises

1.1 The following is a possible solution (but not the only one):

$$\begin{gathered}\forall X(\mathit{natural}(X) \supset \exists Y(\mathit{equal}(s(X), Y))) \\ \neg\exists X\, \mathit{better}(X, \mathit{taking\_a\_nap}) \\ \forall X(\mathit{integer}(X) \supset \neg \mathit{negative}(X)) \\ \forall X, Y(\mathit{name}(X, Y) \wedge \mathit{innocent}(X) \supset \mathit{changed}(Y)) \\ \forall X(\mathit{area\_of\_cs}(X) \supset \mathit{important\_for}(\mathit{logic}, X) \\ \forall X(\mathit{renter}(X) \wedge \mathit{in\_accident}(X) \supset \mathit{pay\_deductible}(X))\end{gathered}$$

1.2 The following is a possible solution (but not the only one):

$$\begin{gathered}\mathit{better}(\mathit{bronze\_medal}, \mathit{nothing}) \\ \neg\exists X\, \mathit{better}(X, \mathit{gold\_medal}) \\ \mathit{better}(\mathit{bronze\_medal}, \mathit{gold\_medal})\end{gathered}$$

1.3 Let MOD(X) denote the set of all models of the formulas X. Then:

$$\begin{array}{lll} P \models F & \text{iff} & \mathrm{MOD}(P) \subseteq \mathrm{MOD}(F) \\ & \text{iff} & \mathrm{MOD}(P) \cap \mathrm{MOD}(\neg F) = \varnothing \\ & \text{iff} & \mathrm{MOD}(P \cup \{\neg F\}) = \varnothing \\ & \text{iff} & P \cup \{\neg F\} \text{ is unsatisfiable}\end{array}$$

1.4 Take for instance, $F \supset G \equiv \neg F \vee G$. Let $\Im$ and φ be an arbitrary interpretation and valuation respectively. Then:

$$\begin{array}{lll} \Im \models_\varphi F \supset G & \text{iff} & \Im \models_\varphi G \text{ whenever } \Im \models_\varphi F \\ & \text{iff} & \Im \models_\varphi G \text{ or } \Im \not\models_\varphi F \\ & \text{iff} & \Im \models_\varphi \neg F \text{ or } \Im \models_\varphi G \\ & \text{iff} & \Im \models_\varphi \neg F \vee G\end{array}$$

1.6 Let MOD(X) denote the set of all models of the formula X. Then:

$$\begin{array}{lll} F \equiv G & \text{iff} & \text{MOD}(F) = \text{MOD}(G) \\ & \text{iff} & \text{MOD}(F) \subseteq \text{MOD}(G) \text{ and } \text{MOD}(G) \subseteq \text{MOD}(F) \\ & \text{iff} & \{F\} \models G \text{ and } \{G\} \models F \end{array}$$

1.8 Hint: Assume that there is a finite interpretation and establish a contradiction using the semantics of formulas.

1.12 Hints:

$E(\theta\sigma) = (E\theta)\sigma$: by the definition of application it suffices to consider the case when E is a variable.

$(\theta\sigma)\gamma = \theta(\sigma\gamma)$: it suffices to show that the two substitutions give the same result when applied to an arbitrary variable. The fact that $E(\theta\sigma) = (E\theta)\sigma$ can be used to complete the proof.

1.15 Only the last one. (Look for counter-examples of the first two!)

2.1 Definite clauses:

$$\begin{array}{ll} (1) & p(X) \leftarrow q(X). \\ (2) & p(X) \leftarrow q(X,Y), r(X). \\ (3) & r(X) \leftarrow p(X), q(X). \\ (4) & p(X) \leftarrow q(X), r(X). \end{array}$$

2.3 The Herbrand universe:

$$U_P = \{a, b, f(a), f(b), g(a), g(b), f(g(a)), f(g(b)), f(f(a)), f(f(b)), \ldots\}$$

The Herbrand base:

$$B_P = \{q(x,y) \mid x, y \in U_P\} \cup \{p(x) \mid x \in U_P\}$$

2.4 $U_P = \{0, s(0), s(s(0)), \ldots\}$ and $B_P = \{p(x,y,z) \mid x, y, z \in U_P\}$.

2.5 Formulas 2, 3 and 5. Hint: Consider ground instances of the formulas.

2.6 Use the immediate consequence operator:

$$\begin{array}{lll} T_P \uparrow 0 & = & \varnothing \\ T_P \uparrow 1 & = & \{q(a, g(b)), q(b, g(b))\} \\ T_P \uparrow 2 & = & \{p(f(b))\} \cup T_P \uparrow 1 \\ T_P \uparrow 3 & = & T_P \uparrow 2 \end{array}$$

That is, $M_P = T_P \uparrow 3$.

2.7 Use the immediate consequence operator:

$$\begin{array}{lll} T_P \uparrow 0 & = & \varnothing \\ T_P \uparrow 1 & = & \{p(0,0,0), p(0, s(0), s(0)), p(0, s(s(0)), s(s(0))), \ldots\} \\ T_P \uparrow 2 & = & \{p(s(0), 0, s(0)), p(s(0), s(0), s(s(0))), \ldots\} \cup T_P \uparrow 1 \\ & \vdots & \\ T_P \uparrow \omega & = & \{p(s^x(0), s^y(0), s^z(0)) \mid x + y = z\} \end{array}$$

3.1 $\{X/a, Y/a\}$, not unifiable, $\{X/f(a), Y/a, Z/a\}$ and the last pair is not unifiable because of occur-check.

3.2 Let σ be a unifier of s and t. By the definition of mgu there is a substitution δ such that $\sigma = \theta\delta$. Now since ω is a renaming it follows also that $\sigma = \theta\omega\omega^{-1}\delta$. Thus, for every unifier σ of s and t there is a substitution $\omega^{-1}\delta$ such that $\sigma = (\theta\omega)(\omega^{-1}\delta)$.

3.4 Assume that σ is a unifier of s and t. Then by definition $\sigma = \theta\omega$ for some substitution ω. Moreover, $\sigma = \theta\theta\omega$ since θ is idempotent. Thus, $\sigma = \theta\sigma$.

Next, assume that $\sigma = \theta\sigma$. Since θ is an mgu it must follow that σ is a unifier.

3.5 $\{X/b\}$ is produced twice and $\{X/a\}$ once.

3.6 For instance, the program and goal:

$$\begin{array}{l} \leftarrow p. \\ p \leftarrow p, q. \end{array}$$

Prolog's computation rule produces an infinite tree whereas a computation rule which always selects the rightmost subgoal yields a finitely failed tree.

3.7 Infinitely many. But there are only two answers, $X = b$ and $X = a$.

4.4 Hint: Each clause of the form:

$$p(t_1, \ldots, t_m) \leftarrow B$$

in P gives rise to a formula of the form:

$$p(X_1, \ldots, X_m) \leftrightarrow \ldots \vee \exists \ldots (X_1 = t_1, \ldots, X_m = t_m, B) \vee \ldots$$

in $comp(P)$. Use truth-preserving rewritings of this formula to obtain the program clause.

4.7 Only P_1 and P_3.

4.8 $comp(P)$ consists of:

$$\begin{array}{l} p(X_1) \leftrightarrow X_1 = a, \neg q(b) \\ q(X_1) \leftrightarrow \Box \end{array}$$

and some equalities including $a = a$ and $b = b$.

4.14 The well-founded model is $\{r, \neg s\}$.

5.1 Without cut there are seven answers. Replacing $true(1)$ by cut eliminates the answers $X = e, Y = c$ and $X = e, Y = d$. Replacing $true(2)$ by cut eliminates in addition $X = b, Y = c$ and $X = b, Y = d$.

5.3 The goal without negation gives the answer $X = a$ while the other goal succeeds without binding X.

5.4 For example:

$$var(X) \leftarrow not(not(X = a)), not(not(X = b)).$$

5.6 For example:

$$
\begin{array}{l}
between(X, Z, Z) \leftarrow X \leq Z. \\
between(X, Y, Z) \leftarrow X < Z, W \ is \ Z - 1, between(X, Y, W).
\end{array}
$$

5.7 For instance, since $(n+1)^2 = n^2 + 2 * n + 1$, $n \geq 0$:

$$
\begin{array}{l}
sqr(0, 0). \\
sqr(s(X), s(Z)) \leftarrow sqr(X, Y), times(s(s(0)), X, W), plus(Y, W, Z).
\end{array}
$$

5.8 For instance:

$$
\begin{array}{l}
gcd(X, 0, X) \leftarrow X > 0. \\
gcd(X, Y, Z) \leftarrow Y > 0, W \ is \ X \ mod \ Y, gcd(Y, W, Z).
\end{array}
$$

6.2 For instance:

$$grandchild(X, Z) \leftarrow parent(Y, X), parent(Z, Y).$$

$$sister(X, Y) \leftarrow female(X), parent(Z, X), parent(Z, Y), X \neq Y.$$

$$brother(X, Y) \leftarrow male(X), parent(Z, X), parent(Z, Y), X \neq Y.$$

etc.

6.3 Hint: (1) Colours should be assigned to countries. Hence, represent the countries by variables. (2) Describe the map in the goal by saying which countries should be assigned different colours.

6.4 For instance:

$$
\begin{array}{l}
and(1, 1, 1). \\
and(0, 1, 0). \\
and(1, 0, 0). \\
and(0, 0, 0). \\
\\
inv(1, 0). \\
inv(0, 1). \\
\\
circuit1(X, Y, Z) \leftarrow \\
\qquad and(X, Y, W), inv(W, Z). \\
circuit2(X, Y, Z, V, W) \leftarrow \\
\qquad and(X, Y, A), and(Z, V, B), \\
\qquad and(A, B, C), inv(C, W).
\end{array}
$$

6.5 For instance:

$$p(X, Y) \leftarrow husband(K, X), wife(K, Y).$$

$$
\begin{array}{l}
q(X) \leftarrow parent(X, Y). \\
q(X) \leftarrow income(X, Y), Y \geq 20000.
\end{array}
$$

6.6 For instance:

$$\pi_{X,Y}(Q(Y,X)) \cup \pi_{X,Y}(Q(X,Z) \bowtie R(Z,Y))$$

6.7 For instance:

$$compose(X,Z) \leftarrow r_1(X,Y), r_2(Y,Z).$$

6.9 Take the transitive closure of the *parent*/2-relation.

6.11 For instance:

$$
\begin{array}{l}
ingredients(tea, needs(water, needs(tea\_bag, nil))).\\
ingredients(boiled\_egg, needs(water, needs(egg, nil))).\\
\\
available(water).\\
available(tea\_bag).\\
\\
can\_cook(X) \leftarrow\\
\qquad ingredients(X, Ingr), all\_available(Ingr).\\
\\
all\_available(nil).\\
all\_available(needs(X,Y)) \leftarrow\\
\qquad available(X), all\_available(Y).\\
\\
needs\_ingredient(X,Y) \leftarrow\\
\qquad ingredients(X, Ingr), among(Y, Ingr).\\
\\
among(X, needs(X,Y)).\\
among(X, needs(Y,Z)) \leftarrow\\
\qquad among(X,Z).
\end{array}
$$

7.1 Alternative list notation:

$$
\begin{array}{cc}
.(a, .(b, [\,])) & .(a, .(b, .(c, [\,])))\\
.(a, b) & .(a, .(b, [\,]))\\
.(a, .(.(b, .(c, [\,])), .(d, [\,]))) & .([\,], [\,])\\
.(a, .(b, X)) & .(a, .(b, .(c, [\,])))
\end{array}
$$

7.4 For instance:

$$
\begin{array}{l}
length([\,], 0).\\
length([X|Y], N) \leftarrow length(Y, M), N\ is\ M + 1.
\end{array}
$$

7.5 For instance:

$$lshift([X \mid YZ], YZX) \leftarrow append(YZ, [X], YZX).$$

7.9 For instance:

$$\begin{array}{l} sublist(X, Y) \leftarrow prefix(X, Y) \\ sublist(X, [Y|Z]) \leftarrow sublist(X, Z). \end{array}$$

7.12 For instance:

$$\begin{array}{l} msort([\,], [\,]). \\ msort([X], [X]). \\ msort(X, Y) \leftarrow \\ \qquad split(X, Split1, Split2), \\ \qquad msort(Split1, Sorted1), \\ \qquad msort(Split2, Sorted2), \\ \qquad merge(Sorted1, Sorted2, Y). \\ \\ split([X], [], [X]). \\ split([X, Y|Z], [X|V], [Y|W]) \leftarrow \\ \qquad split(Z, V, W). \\ \\ merge([\,], [\,], [\,]). \\ merge([X|A], [Y|B], [X|C]) \leftarrow \\ \qquad X < Y, merge(A, [Y|B], C). \\ merge([X|A], [Y|B], [Y|C]) \leftarrow \\ \qquad X \geq Y, merge([X|A], B, C). \end{array}$$

7.13 For instance:

$$\begin{array}{l} edge(1, 2, b). \\ edge(2, 2, a). \\ edge(2, 3, a). \\ edge(3, 2, b). \\ \\ final(3). \\ \\ accept(State, [\,]) \leftarrow \\ \qquad final(State). \\ accept(State, [X|Y]) \leftarrow \\ \qquad edge(State, NewState, X), accept(NewState, Y). \end{array}$$

7.17 For instance:

$$\begin{array}{l} palindrome(X) \leftarrow diff\_palin(X - [\,]). \\ \\ diff\_palin(X - X). \\ diff\_palin([X|Y] - Y). \\ diff\_palin([X|Y] - Z) \leftarrow diff\_palin(Y - [X|Z]). \end{array}$$

7.20 Hint: Represent the empty binary tree by the constant *empty* and the non-empty tree by $node(X, Left, Right)$ where X is the label and *Left* and *Right* the two subtrees of the node.

8.3 The following program provides a starting point (the program finds all refutations but it does not terminate):

$$\begin{array}{l}
prove(Goal) \leftarrow \\
\qquad int(Depth), dfid(Goal, Depth, 0). \\
\\
dfid(true, Depth, Depth). \\
dfid((X, Y), Depth, NewDepth) \leftarrow \\
\qquad dfid(X, Depth, TmpDepth), \\
\qquad dfid(Y, TmpDepth, NewDepth). \\
dfid(X, s(Depth), NewDepth) \leftarrow \\
\qquad clause(X, Y), \\
\qquad dfid(Y, Depth, NewDepth). \\
\\
int(s(0)). \\
int(s(X)) \leftarrow \\
\qquad int(X).
\end{array}$$

8.4 Hint: For instance, the fourth rule may be defined as follows:

$$d(X + Y, Dx + Dy) \leftarrow d(X, Dx), d(Y, Dy).$$

10.3 Definite clause grammar:

$$\begin{array}{lcl}
bleat & \rightarrow & [b], aaa. \\
aaa & \rightarrow & [a]. \\
aaa & \rightarrow & [a], aaa.
\end{array}$$

Prolog program:

$$\begin{array}{l}
bleat(X_0, X_2) \leftarrow connects(X_0, b, X_1), aaa(X_1, X_2). \\
aaa(X_0, X_1) \leftarrow connects(X_0, a, X_1). \\
aaa(X_0, X_2) \leftarrow connects(X_0, a, X_1), aaa(X_1, X_2). \\
connects([X|Y], X, Y).
\end{array}$$

A refutation is obtained, for instance, by giving the goal $\leftarrow bleat([b, a, a], [\,])$.

10.4 The DCG describes a language consisting only of the empty string. However, at the same time it defines the "concatenation"-relation among lists. That is, the nonterminal $x([a, b], [c, d], X)$ not only derives the empty string but also binds X to $[a, b, c, d]$.

12.1 The definition of *append*/3 and *member*/2 is left to the reader:

$$\begin{array}{l}
eq(T1, T2) \leftarrow \\
\qquad nodes(T1, N1), nodes(T2, N2), equal(N1?, N2?). \\
\\
nodes(empty, [\,]). \\
nodes(tree(X, T1, T2), [X|N]) \leftarrow \\
\qquad nodes(T1, N1), nodes(T2, N2), append(N1?, N2?, N). \\
\\
equal(X, Y) \leftarrow \\
\qquad subset(X, Y), subset(Y, X). \\
\\
subset([\,], X). \\
subset([X|Y], Z) \leftarrow \\
\qquad member(X, Z), subset(Y?, Z).
\end{array}$$

12.2 HINT: write a program which transposes the second matrix and then computes all inner products.

13.3 HINT: The overall structure of the proof is as follows:

$$\dfrac{\dfrac{E \vdash app(c(X,Y),Z) \doteq c(X,app(Y,Z))}{E \vdash \_ \doteq \_} \qquad \dfrac{E \vdash a \doteq a \qquad \dfrac{E \vdash app(nil,X) \doteq X}{E \vdash \_ \doteq \_}}{E \vdash \_ \doteq \_}}{E \vdash \_ \doteq \_}$$

14.3 The following program with real-valued or rational constraints can be used to answer e.g. the goal $\leftarrow jugs([M, 1-M, 0], Res, N)$.

$$\begin{array}{l}
jugs([A,B,C],[A,B,C],N) \leftarrow \\
\qquad N \doteq 0, A + B + C \doteq 1. \\
jugs([A,B,C], Res, N) \leftarrow \\
\qquad N > 0, \\
\qquad jugs([0.6 * A + 0.2 * B, 0.7 * B + 0.4 * A, C + 0.1 * B], Res, N - 1).
\end{array}$$

15.1 The transformed program looks as follows:

$$\begin{array}{l}
expr(X, Z) \leftarrow \\
\qquad call\_expr(X, Z), expr(X, [+|Y]), expr(Y, Z). \\
expr([id|Y], Y) \leftarrow \\
\qquad call\_expr([id|Y], Y). \\
call\_expr(X, [+|Y]) \leftarrow \\
\qquad call\_expr(X, Z). \\
call\_expr(Y, Z) \leftarrow \\
\qquad call\_expr(X, Z), expr(X, [+|Y]).
\end{array}$$

Adding $call\_expr([id, +, id], X)$ to the program yields the semi-naive iteration:

$$\begin{array}{lcl}
\Delta x_0 & = & \{call\_expr([id,+,id],A)\} \\
\Delta x_1 & = & \{expr([id,+,id],[+,id]),call\_expr([id,+,id],[+|A])\} \\
\Delta x_2 & = & \{call\_expr([id],A),call\_expr([id],[+|A])\} \\
\Delta x_3 & = & \{expr([id],[\,])\} \\
\Delta x_4 & = & \{expr([id,+,id],[\,])\}
\end{array}$$

Bibliography

Abiteboul, S., Hull, R., and Vianu, V. (1995). *Foundations of Databases.* Addison-Wesley.

Abramson, H. (1984). Definite Clause Translation Grammars. In *Proc. 1984 Symp. on Logic Programming,* Atlantic City, pages 233–241.

Aiba, A., Sakai, K., Sato, Y., Hawley, D., and Hasegawa, R. (1988). Constraint Logic Programming Language CAL. In *Proc. of Int'l Conf. on Fifth Generation Computer Systems 88,* Tokyo, pages 263–276.

Aït-Kaci, H. and Podelski, A. (1993). Towards a Meaning of LIFE. *J. of Logic Programming*, 16(3–4):195–234.

Apt, K. (1990). Introduction to Logic Programming. In van Leeuwen, J., editor, *Handbook of Theoretical Computer Science: Formal Models and Semantics*, volume B, chapter 10, pages 493–574. Elsevier.

Apt, K., Blair, H., and Walker, A. (1988). Towards a Theory of Declarative Knowledge. In Minker, J., editor, *Foundations of Deductive Databases and Logic Programming*, pages 89–148. Morgan Kaufmann, Los Altos.

Apt, K. and Bol, R. (1994). Logic Programming and Negation: A Survey. *J. of Logic Programming*, 19/20:9–71.

Apt, K. and Doets, K. (1994). A New Definition of SLDNF-resolution. *J. of Logic Programming*, 18(2):177–190.

Apt, K. and Pellegrini, A. (1992). Why the Occur-check is Not a Problem. In *Proc. of PLILP'92*, Lecture Notes in Computer Science 631, pages 1–23. Springer-Verlag.

Apt, K. and van Emden, M. (1982). Contributions to the Theory of Logic Programming. *J. of ACM*, 29(3):841–862.

Bacha, H. (1987). Meta-Level Programming: A Compiled Approach. In *Proc. of Fourth Int'l Conf. on Logic Programming,* Melbourne, pages 394–410. MIT Press.

Bancilhon, F., Maier, D., Sagiv, Y., and Ullman, J. (1986). Magic Sets and Other Strange Ways to Implement Logic Programs. In *Proc. Fifth ACM Symp. on Principles of Database Systems*, pages 1–15.

Bancilhon, F. and Ramakrishnan, R. (1988). An Amateur's Introduction to Recursive Query Processing Strategies. In Stonebraker, M., editor, *Readings in Database Systems*, pages 507–555. Morgan Kaufmann.

Barbuti, R., Bellia, M., Levi, G., and Martelli, M. (1986). LEAF: A Language which Integrates Logic, Equations and Functions. In DeGroot, D. and Lindstrom, G., editors, *Logic Programming, Functions, Relations and Equations*, pages 201–238. Prentice-Hall.

Beer, J. (1988). The Occur-Check Problem Revisited. *J. of Logic Programming*, 5(3):243–262.

Beeri, C. and Ramakrishnan, R. (1987). On the Power of Magic. In *Proc of 6th Symposium on Principles of Database Systems*, pages 269–283.

Bellia, M. and Levi, G. (1985). The Relation Between Logic and Functional Languages: A Survey. *J. of Logic Programming*, 3(3):217–236.

Birkhoff, G. (1935). On the Structure of Abstract Algebras. In *Proc. Cambridge Phil. Soc. 31*, pages 433–454.

Bjørner, D., Ershov, A., and Jones, N., editors (1988). *Partial Evaluation and Mixed Computation.* North Holland.

Bol, R. and Degerstedt, L. (1993a). Tabulated Resolution for Well Founded Semantics. In *Proc. of the Int'l Logic Programming Symposium,* Vancouver, pages 199–219. MIT Press.

Bol, R. and Degerstedt, L. (1993b). The Underlying Search for Magic Templates and Tabulation. In *Proc. of Int'l Conf. on Logic Programming,* Budapest, pages 793–811. MIT Press.

Boolos, G. and Jeffrey, R. (1980). *Computability and Logic.* Cambridge University Press.

Bossi, A., Gabbrielli, M., Levi, G., and Martelli, M. (1994). The *S*-semantics Approach: Theory and Applications. *J. of Logic Programming*, 19/20:149–197.

Bowen, K. (1985). Meta-Level Programming and Knowledge Representation. *New Generation Computing*, 3(1):359–383.

Bowen, K. and Kowalski, R. (1982). Amalgamating Language and Metalanguage in Logic Programming. In Clark, K. and Tärnlund, S.-Å., editors, *Logic Programming*, pages 153–172. Academic Press.

Bowen, K. and Weinberg, T. (1985). A Meta-Level Extension of Prolog. In *Proc. 1985 Symp. on Logic Programming,* Boston, pages 48–53.

Bratko, I. (1990). *Prolog Programming for Artificial Intelligence*. Addison-Wesley, 2nd edition.

Bry, F. (1990). Query Evaluation in Recursive Databases: Bottom-up and Top-down Reconciled. *IEEE Transactions on Knowledge and Data Engineering*, 5:289–312.

Cavedon, L. and Lloyd, J. (1989). A Completeness Theorem for SLDNF Resolution. *J. of Logic Programming*, 7(3):177–191.

Chadha, R. and Plaisted, D. (1994). Correctness of Unification without Occur Check in Prolog. *J. of Logic Programming*, 18(2):99–122.

Chan, D. (1988). Constructive Negation based on the Completed Database. In *Proc. of Fifth Int'l Conf./Symp. on Logic Programming,* Seattle, pages 111–125. MIT Press.

Chandra, A. and Harel, D. (1985). Horn Clause Queries and Generalizations. *J. of Logic Programming*, 2(1):1–16.

Chang, C. and Lee, R. (1973). *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, New York.

Chen, W. and Warren, D. S. (1993). Query Evaluation under the Well-founded Semantics. In *Proc. of SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 168–179, Washington DC.

Clark, K. (1978). Negation as Failure. In Gallaire, H. and Minker, J., editors, *Logic and Databases*, pages 293–322. Plenum Press, New York.

Clark, K. (1979). Predicate Logic as a Computational Formalism. Report DOC 79/59, Dept. of Computing, Imperial College.

Clark, K. and Gregory, S. (1981). A Relational Language for Parallel Programming. Research Report DOC 81/16, Department of Computing, Imperial College.

Clark, K. and Gregory, S. (1986). PARLOG: Parallel Programming in Logic. *ACM TOPLAS*, 8(1):1–49.

Clark, K. and McCabe, F. (1982). Prolog: A Language for Implementing Expert Systems. In Hayes, J., Michie, D., and Pao, Y.-H., editors, *Machine Intelligence 10*, pages 455–470. Ellis Horwood.

Clark, K., McCabe, F., and Gregory, S. (1982). IC-Prolog Language Features. In Clark, K. and Tärnlund, S.-Å., editors, *Logic Programming*, pages 253–266. Academic Press.

Clark, K. and Tärnlund, S.-Å. (1977). A First Order Theory of Data and Programs. In *Information Processing '77*, pages 939–944. North-Holland.

Clocksin, W. and Mellish, C. (1994). *Programming in Prolog.* Springer-Verlag, 4th edition.

Codd, E. F. (1970). A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6):377–387.

Cohen, J. and Hickey, T. (1987). Parsing and Compiling using Prolog. *ACM TOPLAS*, 9(2):125–163.

Colmerauer, A. (1978). Metamorphosis Grammars. In Bolc, L., editor, *Natural Language Communication with Computers*, Lecture Notes in Computer Science 63, pages 133–189. Springer-Verlag.

Colmerauer, A. (1982). Prolog and Infinite Trees. In Clark, K. and Tärnlund, S.-Å., editors, *Logic Programming.* Academic Press.

Colmerauer, A. (1984). Equations and Inequations on Finite and Infinite Trees. In *Proc. of Int'l Conf. on Fifth Generation Computer Systems 84,* Tokyo, pages 85–102. North-Holland.

Colmerauer, A. (1990). An Introduction to Prolog III. *Communications of the ACM*, 33(7):69–90.

Colmerauer, A. et al. (1973). Un Système de Communication Homme-Machine en Francais. Technical report, Technical Report, Group d'Intelligence Artificielle, Marseille.

Dahl, V. (1994). Natural Language Processing and Logic Programming. *J. of Logic Programming*, 19/20:681–714.

Dahl, V. and Abramson, H. (1984). On Gapping Grammars. In *Proc. of Second Int'l Conf. on Logic Programming,* Uppsala, pages 77–88.

Dahl, V. and Abramson, H. (1989). *Logic Grammars.* Springer-Verlag.

DeGroot, D. and Lindstrom, G., editors (1986). *Logic Programming, Functions, Relations and Equations.* Prentice-Hall.

Deransart, P., Ferrand, G., and Téguia, M. (1991). NSTO Programs (Not Subject To Occur-check). In *Proc. of 1991 Int'l Logic Programming Symposium,* San Diego, pages 533–547. MIT Press.

Deransart, P. and Małuszyński, J. (1993). *A Grammatical View of Logic Programming.* MIT Press.

Dershowitz, N. and Plaisted, D. (1988). Equational Programming. In Hayes, J. E., Michie, D., and Richards, J., editors, *Machine Intelligence 11*, pages 21–56. Oxford University Press.

Dijkstra, E. W. (1976). *A Discipline of Programming.* Prentice-Hall.

Dincbas, M., Van Hentenryck, P., Simonis, H., Aggoun, A., Graf, T., and Berthier, F. (1988). The Constraint Logic Programming Language CHIP. In *Intl. Conf. on Fifth Generation Computer Systems*, volume 2, pages 693–702.

Doets, K. (1994). *From Logic to Logic Programming.* MIT Press.

Drabent, W. (1995a). Completeness of SLDNF-resolution for Non-floundering Queries. Submitted for publication.

Drabent, W. (1995b). What is Failure? An Approach to Constructive Negation. *Acta Informatica*, 32(1):27–59.

Eder, E. (1985). Properties of Substitutions and Unifications. *J. Symbolic Computation*, 1:31–46.

Ershov, A. P. et al., editors (1988). *Selected Papers from the Workshop on Partial Evaluation and Mixed Computation.* Special issue of New Generation Computing, 6(2-3).

F. Pereira, F. and Warren, D. H. D. (1980). Definite Clause Grammars for Language Analysis—A Survey of the Formalism and a Comparison with Augmented Transision Networks. *Artificial Intelligence*, 13:231–278.

Fitting, M. (1985). A Kripke-Kleene Semantics for Logic Programs. *J. of Logic Programming*, 2(4):295–312.

Gallaire, H., Minker, J., and Nicolas, J.-M. (1984). Logic and Databases: A Deductive Approach. *Computing Surveys*, 16(2):153–185.

Gallier, J. and Raatz, S. (1986). SLD-Resolution Methods for Horn Clauses with Equality Based on *E*-Unification. In *Proc. 1986 Symp. on Logic Programming,* Salt Lake City, pages 168–179.

Galton, A. (1990). *Logic for Information Technology.* John Wiley & Sons.

Gazdar, G. and Mellish, C. (1989). *Natural Language Processing in Prolog.* Addison-Wesley.

Gelfond, M. and Lifschitz, V. (1988). The Stable Model Semantics for Logic Programming. In *Proc. of Fifth Int'l Conf./Symp. on Logic Programming,* Seattle, pages 1070–1080. MIT Press.

Gill, A. (1976). *Applied Algebra for the Computer Sciences.* Prentice-Hall.

Gregory, S. (1987). *Parallel Logic Programming in PARLOG.* Addison-Wesley.

Grimaldi, R. (1994). *Discrete and Combinatorial Mathematics.* Addison-Wesley.

Hansson, Å. and Tärnlund, S.-Å. (1981). Program Transformation by Data Structure Mapping. In Clark, K. and Tärnlund, S.-Å., editors, *Logic Programming*, pages 117–122. Academic Press.

Hanus, M. (1992). Improving Control of Logic Programs by Using Functional Logic Languages. In *PLILP'92*, Lecture Notes in Computer Science 631, pages 1–23. Springer-Verlag.

Hanus, M. (1994). The Integration of Functions into Logic Programming: From Theory to Practice. *J. of Logic Programming*, 19/20:583–628.

Haridi, S. and Brand, P. (1988). ANDORRA Prolog — An Integration of Prolog and Committed Choice Languages. In *Proc. of Int'l Conf. on Fifth Generation Computer Systems 88,* Tokyo, pages 745–754.

Heintze, N., Jaffar, J., Michaylov, S., Stuckey, P., and Yap, R. (1987a). The *CLP*($\Re$) Programmers Manual (version 2.0). Technical report, Dept. of Computer Science, Monash University.

Heintze, N., Michaylov, S., and Stuckey, P. (1987b). *CLP*($\Re$) and Some Electrical Engineering Problems. In *Proc. of Fourth Int'l Conf. on Logic Programming,* Melbourne, pages 675–703. MIT Press.

Herbrand, J. (1967). Investigations in Proof Theory. In van Heijenoort, J., editor, *From Frege to Gödel: A Source Book in Mathematical Logic, 1879–1931*, pages 525–581. Harvard University Press.

Hill, P. and Lloyd, J. (1988a). Analysis of Meta-Programs. Report CS-88-08, Dept. of Computer Science, University of Bristol.

Hill, P. and Lloyd, J. (1988b). Meta-Programming for Dynamic Knowledge Bases. Report CS-88-18, Dept. of Computer Science, University of Bristol.

Hill, P. and Lloyd, J. (1994). *The Gödel Programming Language.* MIT Press.

Hill, R. (1974). LUSH-resolution and its Completeness. DCL Memo 78, Dept. of Artificial Intelligence, University of Edinburgh.

Hoare, C. A. R. (1985). *Communicating Sequential Processes.* Prentice-Hall.

Hopcroft, J. and Ullman, J. (1979). *Introduction to Automata Theory, Language, and Computation.* Addison Wesley.

Huet, G. (1975). A Unification Algorithm for Typed λ-Calculas. *Theoretical Computer Science*, 1:27–57.

Huet, G. and Oppen, D. (1980). Equations and Rewrite Rules: A Survey. In Book, R., editor, *Formal Language Theory: Perspectives and Open Problems*, pages 349–405. Academic Press.

Hullot, J. M. (1980). Canonical Forms and Unification. In *Proc. of 5th CADE,* Les Arcs, France.

Imbert, J.-L., Cohen, J., and Weeger, M.-D. (1993). An Algorithm for Linear Constraint Solving: Its Incorporation in a Prolog Meta-interpreter for CLP. *J. of Logic Programming*, 16(3–4):195–234.

ISO (1995). Information Technology—Programming Language—Prolog—Part 1: General core. ISO/IEC DIS 13211-1:1995(E).

Jaffar, J. and Lassez, J.-L. (1987). Constraint Logic Programming. In *Conf. Record of 14th Annual ACM Symp. on POPL.*

Jaffar, J., Lassez, J.-L., and Lloyd, J. (1983). Completeness of the Negation as Failure Rule. In *Proc. of IJCAI-83*, pages 500–506, Karlsruhe.

Jaffar, J., Lassez, J.-L., and Maher, M. (1984). A Theory of Complete Logic Programs with Equality. *J. of Logic Programming*, 1(3):211–223.

Jaffar, J., Lassez, J.-L., and Maher, M. (1986). Logic Programming Language Scheme. In DeGroot, D. and Lindstrom, G., editors, *Logic Programming, Functions, Relations and Equations*, pages 441–467. Prentice-Hall.

Jaffar, J. and Maher, M. (1994). Constraint Logic Programming: A Survey. *J. of Logic Programming*, 19/20:503–581.

Jaffar, J. and Michaylov, S. (1987). Methodology and Implementation of a CLP System. In *Proc. of Fourth Int'l Conf. on Logic Programming,* Melbourne, pages 196–218. MIT Press.

Janson, S. (1994). *AKL—A Multiparadigm Programming Language.* Phd thesis, Uppsala Univ, Computing Science Dept.

Kakas, A., Kowalski, R., and Toni, F. (1992). Abductive Logic Programming. *J. of Logic and Computation*, 2.

Knight, K. (1989). Unification: A Multidisciplinary Survey. *ACM Computing Surveys*, 21(1):93–124.

Komorowski, H. J. (1981). *A Specification of an Abstract Prolog Machine and its Application to Partial Evaluation.* PhD thesis, Linköping University.

Komorowski, H. J. (1982). QLOG — The Programming Environment for Prolog in Lisp. In Clark, K. and Tärnlund, S.-Å., editors, *Logic Programming*, pages 315–324. Academic Press.

Kowalski, R. (1974). Predicate Logic as a Programming Language. In *Information Processing '74*, pages 569–574. North-Holland.

Kowalski, R. (1979a). Algorithm = Logic + Control. *Communications of the ACM*, 22(7):424–436.

Kowalski, R. (1979b). *Logic For Problem Solving.* Elsevier, North-Holland, New York.

Kowalski, R. and Kuehner, D. (1972). Linear Resolution with Selection Function. *Artificial Intelligence*, 2:227–260.

Kunen, K. (1987). Negation in Logic Programming. *J. of Logic Programming*, 4(4):289–308.

Kunen, K. (1989). Signed Data Dependencies in Logic Programming. *J. of Logic Programming*, 7(3):231–245.

Lassez, J.-L., Maher, M., and Marriott, K. (1988). Unification Revisited. In Minker, J., editor, *Foundations of Deductive Databases and Logic Programming*, chapter 15, pages 587–626. Morgan Kaufmann.

Lassez, K., McAloon, K., and Yap, R. (1987). Constraint Logic Programming and Option Trading. *IEEE Expert*, Fall:42–50.

Leler, W. (1988). *Constraint Programming Languages.* Addison Wesley.

Lloyd, J., Sonenberg, E. A., and Topor, R. (1987). Integrity Constraint Checking in Stratified Databases. *J. of Logic Programming*, 4(4):331–344.

Lloyd, J. and Topor, R. (1984). Making Prolog More Expressive. *J. of Logic Programming*, 1(3):225–240.

Lloyd, J. and Topor, R. (1985). A Basis for Deductive Database Systems. *J. of Logic Programming*, 2(2):93–110.

Lloyd, J. and Topor, R. (1986). A Basis for Deductive Database Systems II. *J. of Logic Programming*, 3(1):55–68.

Lloyd, J. W. (1987). *Foundations of Logic Programming.* Springer-Verlag, second edition.

Maher, M. (1987). Logic Semantics for a Class of Committed-choice Programs. In *Proc. of Fourth Int'l Conf. on Logic Programming,* Melbourne, pages 858–876. MIT Press.

Małuszyński, J., Bonnier, S., Boye, J., Kluźniak, F., Kågedal, A., and Nilsson, U. (1993). Logic Programs with External Procedures. In Apt, K., de Bakker, J., and Rutten, J., editors, *Current Trends in Logic Programming Languages, Integration with Functions, Constraints and Objects.* MIT Press.

Małuszyński, J. and Näslund, T. (1989). Fail Substitutions for Negation as Failure. In *Proc. of North American Conf. on Logic Programming,* Cleveland, pages 461–476. MIT Press.

Marriott, K. and Søndergaard, H. (1988). Prolog Program Transformation by Introduction of Difference-Lists. Technical Report 88/14, Department of Computer Science, The University of Melbourne.

Marriott, K. and Søndergaard, H. (1989). On Prolog and the Occur Check Problem. *Sigplan Notices*, 24(5):76–82.

Martelli, A. and Montanari, U. (1982). An Efficient Unification Algorithm. *ACM TOPLAS*, 4(2):258–282.

Matsumoto, Y., Tanaka, H., Hirakawa, H., Miyoshi, H., and Yasukawa, H. (1983). BUP: A Bottom-Up Parser Embedded in Prolog. *New Generation Computing*, 1(2):145–158.

Mellish, C. and Hardy, S. (1984). Integrating Prolog in the Poplog Environment. In Campbell, J., editor, *Implementations of Prolog*, pages 147–162. Ellis Horwood.

Mendelson, E. (1987). *Introduction to Mathematical Logic.* Wadsworth & Brooks, 3rd edition.

Minker, J. (1988). Perspectives in Deductive Databases. *J. of Logic Programming*, 5(1):33–60.

Moreno-Navarro, J. and Rodriguez-Artalejo, M. (1992). Logic Programming with Functions and Predicates: The Language BABEL. *J. of Logic Programming*, 12(3):191–223.

Nadathur, G. and Miller, D. (1995). Higher-order Logic Programming. In Gabbay, D., Hogger, C., and Robinson, A., editors, *Handbook of Logic in Artificial Intelligence and Logic Programming.* Oxford Univ. Press. To appear.

Naish, L. (1985). Automating Control for Logic Programs. *J. of Logic Programming*, 2(3):167–184.

Naish, L. (1986). *Negation and Control in Prolog.* Lecture Notes in Computer Science 225. Springer-Verlag.

Nilsson, U. (1986). AID: An Alternative Implementation of DCGs. *New Generation Computing*, 4(4):383–399.

Nilsson, U. and Małuszyński, J. (1990). *Logic, Programming and Prolog.* John Wiley & Sons, 1st edition.

O'Keefe, R. (1985). On the Treatment of Cuts in Prolog Source-Level Tools. In *Proc. 1985 Symp. on Logic Programming,* Boston, pages 68–72.

O'Keefe, R. (1990). *The Craft of Prolog.* MIT Press.

Paterson, M. and Wegman, M. (1978). Linear Unification. *J. Computer and System Sciences.*, 16(2):158–167.

Pereira, F. and Shieber, S. (1987). *Prolog and Natural-Language Analysis.* CSLI.

Pereira, F. and Warren, D. H. D. (1983). Parsing as Deduction. In *Proc. 21st Annual Meeting of the Assoc. for Computational Linguistics*, pages 137–144.

Pereira, L., Pereira, F., and Warren, D. H. D. (1979). User's Guide to DECsystem-10 Prolog. DAI. Occasional paper no. 15, Dept. of Artificial Intelligence, University of Edinburgh.

Plaisted, D. (1984). The Occur-Check Problem in Prolog. In *Proc. 1984 Symp. on Logic Programming,* Atlantic City, pages 272–280.

Prawitz, D. (1960). An Improved Proof Procedure. *Theoria*, 26:102–139.

Przymusinski, T. (1988a). On the Declarative Semantics of Logic Programs with Negation. In Minker, J., editor, *Foundations of Deductive Databases and Logic Programming*, pages 193–216. Morgan Kaufmann, Los Altos.

Przymusinski, T. (1988b). Perfect Model Semantics. In *Proc. of Fifth Int'l Conf./Symp. on Logic Programming,* Seattle, pages 1081–1096. MIT Press.

Przymusinski, T. (1989). Every Logic Program has a Natural Stratification and an Iterated Fixed Point Model. In *Proc. of the 8th Symposium on Principles of Database Systems*, pages 11–21.

Ramakrishnan, R. (1988). Magic Templates: A Spellbinding Approach to Logic Programming. In *Proc. of Fifth Int'l Conf./Symp. on Logic Programming,* Seattle, pages 140–159. MIT Press.

Ramakrishnan, R., editor (1995). *Applications of Logic Databases.* Kluwer Academic Publishers.

Reiter, R. (1978). On Closed World Data Bases. In Gallaire, H. and Minker, J., editors, *Logic and Databases*, pages 55–76. Plenum Press, New York.

Reiter, R. (1984). Towards a Logical Reconstruction of Relational Database Theory. In Brodie, M. et al., editors, *On Conceptual Modelling: Perspectives from Artificial Intelligence, Databases and Programming Languages*, pages 191–233. Springer.

Robinson, J. A. (1965). A Machine-Oriented Logic Based on the Resolution Principle. *J. of ACM*, 12:23–41.

Robinson, J. A. (1979). *Logic: Form and Function.* Edinburgh University Press.

Robinson, J. A. and Sibert, E. (1982). LOGLISP: Motivation, Design and Implementation. In Clark, K. and Tärnlund, S.-Å., editors, *Logic Programming*, pages 299–314. Academic Press.

Rogers, Jr., H. (1967). *Theory of Recursive Functions and Effective Computability.* McGraw-Hill.

Ross, K. (1992). A Procedural Semantics for Well-founded Negation in Logic Programs. *J. of Logic Programming*, 13(1):1–22.

Roussel, P. (1975). Prolog: Manuel de Référence et d'Utilisation. Technical report, Group d'Intelligence Artificielle, Marseille.

Saraswat, V. A. (1993). *Concurrent Constraint Programming.* MIT Press.

Schulte, C., Smolka, G., and Würtz, J. (1994). Encapsulated Search and Constraint Programming in Oz. In *Second Workshop on Principles and Practice of Constraint Programming*, Lecture Notes in Computer Science 874, pages 134–150. Springer-Verlag.

Scott, D. (1976). Data Types as Lattices. *SIAM J. Comput.*, 5(3):522–587.

Shapiro, E. (1983a). A Subset of Concurrent Prolog and Its Interpreter. Technical Report TR–003, ICOT.

Shapiro, E. (1983b). Logic Programs with Uncertainties: A Tool for Implementing Rule-based Systems. In *Proc. 8th Int'l Joint Conf. on Artificial Intelligence*, pages 529–532, Karlsruhe.

Shapiro, E. (1986). Concurrent Prolog: A Progress Report. *IEEE Computer*, August:44–58.

Shapiro, E., editor (1988). *Concurrent Prolog: Collected Papers.* MIT Press.

Shapiro, E. (1989). The Family of Concurrent Logic Programming Languages. *Computing Surveys*, 21(3):413–510.

Shepherdson, J. (1988). Negation in Logic Programming. In Minker, J., editor, *Foundations of Deductive Databases and Logic Programming*, pages 19–88. Morgan Kaufmann, Los Altos.

Shoenfield, J. (1967). *Mathematical Logic.* Addison-Wesley.

Siekmann, J. (1984). Universal Unification. In Shostak, R. E., editor, *Proc. of 7th CADE*, pages 1–42.

Siekmann, J. and Wrightson, G., editors (1983a). *Automation of Reasoning I.* Springer-Verlag.

Siekmann, J. and Wrightson, G., editors (1983b). *Automation of Reasoning II.* Springer-Verlag.

Slagle, J. R. (1974). Automated Theorem-Proving for Theories with Simplifiers, Commutativity and Associativity. *J. of ACM*, 28(3):622–642.

Snyder, W. and Gallier, J. (1990). Higher Order-Unification Revisited: Complete Sets of Transformations. In Kirchner, C., editor, *Unification.* Academic Press.

Stärk, R. (1990). A Direct Proof for the Completeness of SLD-resolution. In *CSL'89*, Lecture Notes in Computer Science 440, pages 382–383. Springer-Verlag.

Stärk, R. (1992). *The Proof Theory of Logic Programs with Negation.* Phd thesis, Univ of Bern.

Stärk, R. (1993). Input/Output Dependencies of Normal Logic Programs. *J. of Logic and Computation.* To appear.

Steele, G. L. (1980). *The Definition and Implementation of a Computer Programming Language based on Constraints.* PhD thesis, MIT AI–TR 595, M.I.T.

Sterling, L. and Beer, R. (1989). Metainterpreters for Expert System Construction. *J. of Logic Programming*, 6(1):163–178.

Sterling, L. and Lakhotia, A. (1988). Composing Prolog Meta-Interpreters. In *Proc. of Fifth Int'l Conf./Symp. on Logic Programming,* Seattle, pages 386–403. MIT Press.

Sterling, L. and Shapiro, E. (1994). *The Art of Prolog.* MIT Press, 2nd edition.

Stroetmann, K. (1993). A Completeness Result for SLDNF-resolution. *J. of Logic Programming*, 15(4):337–355.

Subrahmanyam, P. A. and You, J.-H. (1986). FUNLOG: A Computational Model Integrating Logic Programming and Functional Programming. In DeGroot, D. and Lindstrom, G., editors, *Logic Programming, Functions, Relations and Equations*, pages 157–198. Prentice-Hall.

Tamaki, H. and Sato, T. (1986). OLD Resolution with Tabulation. In Shapiro, E., editor, *Proc. of Third Int'l Conf. on Logic Programming,* London, Lecture Notes in Computer Science 225, pages 84–98. Springer-Verlag.

Tarski, A. (1955). A Lattice Theoretical Fixpoint Theorem and Its Applications. *Pacific J. Math*, 5:285–309.

Thom, J. and Zobel, J. (1987). NU-Prolog Reference Manual. Technical Report 86/10, Department of Computer Science, University of Melbourne. Revised May 1987.

Ueda, K. (1985). Guarded Horn Clauses. Technical Report TR–103, ICOT.

Ullman, J. D. (1985). Implementation of Logical Query Languages for Databases. *ACM Trans. Database Systems*, 10(3):289–321.

Ullman, J. D. (1988). *Principles of Database and Knowledge-base Systems*, volume I. Computer Science Press.

Ullman, J. D. (1989). *Principles of Database and Knowledge-base Systems*, volume II. Computer Science Press.

van Dalen, D. (1983). *Logic and Structure*. Springer-Verlag, second edition.

van Emden, M. and Kowalski, R. (1976). The Semantics of Predicate Logic as a Programming Language. *J. of ACM*, 23(4):733–742.

Van Gelder, A. (1988). Negation as Failure Using Tight Derivation for General Logic Programs. In Minker, J., editor, *Foundations of Deductive Databases and Logic Programming*, pages 149–176. Morgan Kaufmann, Los Altos.

Van Gelder, A., Ross, K., and Schlipf, J. (1991). The Well-Founded Semantics for General Logic Programs. *J. of the ACM*, 38(3):620–650.

Van Hentenryck, P. (1989). *Constraint Satisfaction in Logic Programming.* MIT Press.

Vieille, L. (1989). Recursive Query Processing: The Power of Logic. *Theoretical Comp. Sci.*, 69(1):1–53.

Walinsky, C. (1989). CLP(Σ^*): Constraint Logic Programming with Regular Sets. In *Proc. of Sixth Int'l Conf. on Logic Programming,* Lisbon, pages 181–198. MIT Press.

Warren, D. S. (1984). Database Updates in Pure Prolog. In *Proc. of Int'l Conf. on Fifth Generation Computer Systems 84,* Tokyo, pages 244–253. North-Holland.

Warren, D. S. (1992). Memoing for Logic Programs. *CACM*, 35(3):93–111.

Zhang, J. and Grant, P. W. (1988). An Automatic Difference-list Transformation Algorithm for Prolog. In *Proc. of ECAI'88*, pages 320–325.

Index