λProlog:

An Introduction to the Language and its Logic

Draft: not for general distribution or quotation (See Preface)
Comments welcome.
27 May 1998 (Penn State)

© **Dale Miller**
Department of Computer Science and Engineering
The Pennsylvania State University
220 Pond Laboratory
University Park, PA 16802-6106 USA
Office: 814-865-9505
`dale@cse.psu.edu`
`http://www.cse.psu.edu/~dale`

Contents

List of Figures

Preface for this draft

This draft is not intended for general circulation.

Included here are chapters that describe most features of λProlog. The examples given to illustrate these features are rather short. I intend to add several chapters each dealing with extended examples, in areas such as automated deduction, natural language processing, and meta-programming of functional programs.

Various details of λProlog as reported here will change between now and the next release of this monograph. In particular, certain details about the naming of modules and the use of module names, about built-in predicates and functions, and about the use of polymorphic types will almost certainly change. The essentially logical aspects of λProlog, however, have been set for many years now and what the monograph describes about them should remain unchanged.

Comments and corrections on any of this material will be greatly appreciated.

The URL `http://www.cse.psu.edu/~dale/lProlog` contains some WWW pages for λProlog.

Acknowledgments: I would like to thank Iliano Cervesato, Giorgio Delzanno, Joern Dinkla, H. Krishnapriyan, Gary Leavens, Jim Lipton, Tong Mei, Gopalan Nadathur, Olivier Ridoux, and Jenny Simon for useful comments on earlier drafts of this book.

Chapter 1

Introduction

λProlog is a logic programming language that supports higher-order programming, polymorphic typing, modular programming, abstract data types, and allows λ-abstractions in data structures. Most of these programming features can be seen as simple operational interpretations of the intuitionistic theory of *higher-order hereditary Harrop formula*, the logical foundation for λProlog.

1.1 The Logical Structure of λProlog

Figure 1.1 displays seven logical systems and the containment relationship between them. Four of these logics are abbreviations.

fohc	first-order Horn clauses
fohh	first-order hereditary Harrop formulas, extended to $fohh^+$
hohc	higher-order Horn clauses
hohh	higher-order hereditary Harrop formulas, extended to $hohh^+$

It is the largest of these languages, namely $hohh^+$, that constitutes the logical foundation of λProlog. As this figure reveals, *hohh* is an extension to *fohc* (the logical foundation of Prolog) along two dimensions. Both of these extensions provide for different notions of abstractions. The extension from first-order to higher-order (the horizontal dimension) allows for higher-order variables and quantification: this extension provides for higher-order programming and λ-abstractions in terms. The extension from Horn clauses to hereditary Harrop formulas (the vertical dimension) allows for the nesting of implications and universal quantifiers: this extension allows for modular programming, abstract data types, and hypothetical reasoning. The systems named $fohh^+$ and $hohh^+$ are extensions of *fohh* and *hohh*, respectively, that allow for more flexible quantification over predicates within queries. The logic L_λ is also displayed between *fohh* and *hohh*. This logic is a natural extension of *fohh* in which λ-abstractions are treated in a natural and primitive fashion.

1.2 Organization of this monograph

This monograph is organized into the following parts.

$$
\begin{tikzcd}
\mathit{fohh}^{+} \arrow[rr] & & \mathit{hohh}^{+} \\
\mathit{fohh} \arrow[u] \arrow[r] & L_{\lambda} \arrow[r] & \mathit{hohh} \arrow[u] \\
\mathit{fohc} \arrow[u] \arrow[rr] & & \mathit{hohc} \arrow[u]
\end{tikzcd}
$$

Figure 1.1: A map of the sublanguages within λProlog

1. Chapters 2, 3, and 4 describe how first-order Horn clauses are represented in λProlog. Their presentation here differs significantly from the presentation given by Prolog. In particular, constants and variables are given polymorphic types and programs are organized into modules that are qualified by signatures. In order to allow for richer logical expressions and term structures, the syntax of clauses and terms is also enriched.

2. Chapter 5 presents *fohh* and the extension to modular programming that they capture.

3. Chapters 6 and 7 present the simply typed λ-calculus and the L_λ extension of *fohh* that permits direct reasoning about such λ-terms. L_λ permits only a limited form of β-conversion and is implemented with a simple extension to first-order unification.

4. In various chapters (to be included here later), several extended examples of the use of λProlog in various domains are given.

1.3 Prerequisites

This monograph assumes some familiarity with programming with Prolog as well as an informal understanding of deduction within logic. The text [SS86], for example, provides more than an adequate background. In the course of this monograph, we shall have occasion to speak about the first-order and higher-order versions of classical and intuitionistic logics as well as the λ-calculus and sequential proof systems. We shall not assume that the reader is familiar with any of these topics. Our descriptions of these topics will, however, be partial: we shall deal only with those aspects of these formal systems that support the operational reading of λProlog specifications and programs. Familiarity with the programming language

Standard ML [MTH90] would be helpful although is not required: the type system of λProlog is related to that of ML and both languages permit higher-order programming, modules, abstract data-types, and polymorphic types.

1.4 Goals of this monograph

Informally describe λProlog. A large part of λProlog is based on a well studied, half-century old logic and, as a result, the character and semantics of that portion of the language is well understood. One goal of this monograph is to describe the operational interpretation of this logical core of λProlog.

The remaining part of λProlog is not based on logical principles, but rather has been influenced by several factors, such as other attempts to turn a declarative formalism into a useful programming language (for example, ML and Prolog), by available implementation techniques, by experience with existing implementations, and by insights form an analysis of the growing set of examples of λProlog programs. Thus features such as input/output, control of proof-search, polymorphic typing, and programming-in-the-large, which are not addressed directly by the logical core of λProlog, namely the intuitionistic theory of *hohh*, have had divergent developments in different implementations. A major goal of this monograph is to present a coherent and useful interpretation of these language features.

For various reasons, it seems clear that λProlog is not ready for a formal definition in the style of those given for Standard ML [MTH90] and Gödel [HL94]. What is contained here is an informal description of the language.

Provide a description of various subsets of λProlog. The largeness of λProlog is both an advantage and a disadvantage. Since it incorporates many aspects of computation with logic, λProlog is a useful tool for learning about these aspects and how they interact. Largeness will, however, almost certainly make effective analysis and compilation difficult: for that, subsets of the language will almost certainly need to be explored. Thus, we expose the logical foundation of λProlog by presenting various subsets of it: many of these subsets are interesting logic programming languages in their own right.

Provide a significant and self-contained set of examples. Although many examples of using λProlog have been published in various articles and dissertations, there is no one place where the full scope of the current language can be seen. λProlog is a big language containing many features and new programming idioms. Most existing sets of examples have been designed to focus on certain selected features. A reader of these early papers may be left with the feeling that the language is fragmented and resulted from gluing various languages together. The larger set of examples contained here should help show how the many features of λProlog work together and are aspects of a single logic.

Provide a common documentation for new implementations. Implementors of interpreters and compilers for λProlog will hopefully be able to use this monograph as a partial introduction to their individual implementations. This document, however, cannot be viewed as a description of any particular implementation.

Introduce the logic behind λProlog. Although λProlog is based on higher-order intuitionistic logic, this monograph does not provide a general analysis of this logic. Aspects of its operational semantics will be described, although its proof theory and declarative semantics will only briefly be described. References to the relevant literature on higher-order logic, intuitionistic logic, and the proof theoretic analysis of logic programming will be provided so that the interested reader can study these topics, central to a non-operational reading of λProlog, in greater detail.

1.5 The brief history of λProlog

In 1984 D. Miller and G. Nadathur worked at designing a higher-order extension of Horn clauses and at understanding their properties, possible applications, and the design of a logic programming language based on such clauses. The first paper describing this early work on λProlog was [MN85]. The theory and applications of higher-order Horn clauses was developed further in [MN86a, MN86b, Nad87, NM90]. Since the traditional techniques in the literature of logic programming appeared to be over-specialized for first-order, classical logic or were lacking in explanatory powers, the theoretical part of this early work relied, to a large extent, on the sequent calculus. This choice of formalism was fortunate since it helped isolate some intrinsic and high-level characteristics of logic programming. In particular, computation in logic programming can be usefully identified with the search for proofs, in particular, with *goal-directed search* of cut-free sequent proofs. In a series of papers [Mil86, Mil87a, MNS87, MNPS91] this characterization was used to develop and justify the intuitionistic theory of *hereditary Harrop formulas* as a suitable basis for logic programming. It is the higher-order version of hereditary Harrop formulas that is now the basis of λProlog as it is described here. The notion of goal directed search has been used to design extensions to hereditary Harrop formulas based on linear logic [Gir87]. For example, J. Hodas and D. Miller have designed *Lolli* [HM91, HM94, Hod94] to extend hereditary Harrop formulas with linear implication, and D. Miller has designed *Forum* [Mil94, Mil96] as an extension of Lolli to incorporate all of linear logic. These extensions are not incorporated into λProlog.

Along with the development of λProlog's design and logical foundation, numerous implementations have been written and planned. Miller and Nadathur collaborated on the Prolog implementations, LP2.6 and LP2.7, which have been available since 1987: these systems did not implement the full logic (in particular, implications in the body of clauses were not supported). The full language was first implemented in eLP, a Common Lisp implementation by C. Elliott and F. Pfenning, which has been available since 1989. Y. Bekkers, P. Brisset, and O. Ridoux implemented a compiler, called Prolog/MALI, of the full logic in 1989. The Common Lisp implementation of eLP has been rewritten by Pfenning using Standard ML: A. Felty and E. Gunter worked on extending that implementation, and during the 1995/96 academic year, P. Wickline took that system and rebuilt it extensively into a system named Terzo. The first release of that system as in the spring of 1996. Nadathur and several of his colleagues have designed an abstract machine for λProlog [KNW93, NJK95, KNW94] and he is currently building a compiler for the full language.

Chapter 2

Kinds, types, and signatures

Before programs, queries, and data can be used with λProlog, they must be properly qualified by giving them kind, type, or infix declarations. Such declarations are grouped together into signatures.

2.1 Three styles of declarations

Consider the following specification of the concatenation of two lists in λProlog syntax (described more fully in Chapter 3):

```
append nil K K.
append (X::L) K (X::M) :- append L K M.
```

This specification contains the four variables `X`, `L`, `K`, and `M`, one logical constant, namely `:-` for the converse of implication, and three non-logical constants, namely `nil`, `::`, and `append`, denoting the empty list, the list constructor, and the concatenation relation, respectively. Two of these constants, namely `::` and `:-`, are also used as infix symbols. This specification is not meaningful unless infix declarations and type declarations are given for these constants. In this case, the necessary declarations will be given as the following *signature*.

```
type   :-       o -> o -> o.
type   ::       A -> list A -> list A.
type   nil      list A.
type   append   list A -> list A -> list A -> o.
infixr ::    5.
infixl :-    1.
```

The various items in this signature must be provided before the above specification for `append` is considered complete. All logical constants and system predicates (predicates for doing special operations such as input and output) have their declarations built into the runtime system of λProlog so the programmer need not supply them. All other constants must have their declarations made explicitly by the programmer. Notice, however, that the above declarations have introduced two new constants not in the orginial specication, namely `o` and `list`. These type constants will need declarations, called *kind declarations*, which are described in the next section. The capital letter `A` also appears in some of the type declarations above. This token is an example of a *type variable*, which is described more in Subsection 2.3.

Some implementations of λProlog supply some form of type inference that simplifies the specification of type information. Here we assume no particular type inference mechanism is available for constants, and we shall declare types for them. Variables, such as `X`, `L`, `K`, `M` in the specification above are also typed, but since they have limited scope, their types will be inferred from the context in which they appear.

2.2 Kind declarations

Kind declarations are used to introduce *type constructors*. A *kind declaration* starts with the keyword `kind` and ends with a *fullstop*, a period followed by white space.. Following the `kind` keyword is a list of one or more tokens (separated by commas) followed with the *kind expression* that is to be associated with the given tokens. The tokens in this list, all of which must start with a lowercase letter, will be declared to have the given kind. The syntax of a kind expressions is given using the following grammar:

$$\langle\text{kind exp}\rangle ::= \texttt{type} \mid \texttt{type -> } \langle\text{kind exp}\rangle.$$

For example, the following are kind expressions. (Here, `->` is an infix symbol that associates to the right.)

```
type
type -> type
type -> type -> type
type -> type -> type -> type
```

The symbols that are declared by such kind declarations are called *type constructors*.

Examples of kind declarations are

```
kind o          type.
kind int        type.
kind real       type.
kind string     type.
```

These several lines can be replaced by the one line

```
kind o, int, real, string        type.
```

These type constructors denote the types for formulas, integers, real numbers, and strings and are available in implementations of λProlog. The following are also kind declarations.

```
kind list       type -> type.
kind pr         type -> type -> type.
```

Here, `list` takes a type and returns another type. For example, `(list int)` is the type of a list of integers and `(list (list string))` is the type of a list of lists of strings. Similarly, `pr` takes two types and returns a type: for example, `(pr string int)` is the type of a pair with its first component a string and second component an integer. Programmers are free to introduce new primitive types and type constructors as desired.

Notice that kind expressions are particularly simple: in particular, it is not possible to nest `->` to the left. Richer kind expressions could be considered for λProlog, but enhancing them would likely cause enhancements to type and term expressions and produce a language much different than the one we are considering. Thus, kind expressions only indicate the arity of a type constructor and, hence, could be replaced with non-negative integers: some of the above declarations might have been written as `int/0`, `list/1`, and `pair/2`. The syntax adopted here, however, is more suggestive and is analogous to that used to specify the type declarations described below.

Type constuctors of arity 0 are also called *primitive types*.

2.3 Type expressions

Type expressions are constructed from type constructors, *type variables*, and the *function type constructor*. Tokens with an initial upper case letter denote type variables: these are used to provide for *polymorphic typing* and *ad hoc typing* (see Section 3.10.2). We shall generally need only a few such variables, and these will be written as `A`, `B`, `C`, and `D`. The function type constructor, written as $\rightarrow$ in mathematical notation and as `->` in concrete syntax, is an infix constructor for types and it associates to the right. The expression `a -> b -> c` is parsed as `a -> (b -> c)` and this type expression is different from the type `(a -> b) -> c`. In contrast to kind expressions, there is no restriction on the use of the function type constructor: it is possible to nest `->` to the left or right of another `->`. The following is a grammar for type expressions.

$$
\begin{array}{lll}
\langle\text{type exp}\rangle & ::= & \langle\text{type variable}\rangle \mid \\
 & & (\langle\text{type exp}\rangle \texttt{ -> } \langle\text{type exp}\rangle) \mid \\
 & & (\langle\text{type constructor of arity } n\rangle\ \langle\text{type exp}\rangle_1 \ldots \langle\text{type exp}\rangle_n)\ (n \geq 0)
\end{array}
$$

Type variables are not allowed to have types as arguments: that is, they are treated as if they have a kind declaration of `type` (an arity of 0).

Given the kind declarations in the previous subsection, the following are all legal type expressions.

```
int -> real -> string
int -> int -> o
o -> int -> o
(int -> int) -> real
list A -> (A -> B) -> list B -> o
list (list A) -> list A -> o
(A -> B) -> list (A -> B)
((A -> B) -> A) -> A
```

We assume that the function type constructor `->` binds with the lowest priority: thus, the expression `list A -> B` is to be read as `(list A) -> B`.

Every type expression τ can be written in the form

$$\tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow \tau_0 \qquad (n \geq 0)$$

where τ_0 does not have a top-level function type constructor. In this case, we say that τ_0 is the *target type* of τ and the types $\tau_1, \ldots, \tau_n$ are the *argument types* of τ. A type expression is a *functional type* if it is of the form $\tau \rightarrow \sigma$; otherwise it is *non-functional type*.

The *order of a type expression* is defined by recursion on the structure of the expression: the order of a non-functional type expressions is 0 and the order of a functional type is one greater than the maximum of the order of its argument types. Thus, if $\mathrm{ord}(\tau)$ denotes the order of type expression τ then the following two equations define ord.

$$\begin{aligned} \mathrm{ord}(\tau) &= 0 \quad \text{provided } \tau \text{ is non-functional} \\ \mathrm{ord}(\tau_1 \to \tau_2) &= \max(\mathrm{ord}(\tau_1) + 1, \mathrm{ord}(\tau_2)) \end{aligned}$$

Order counts the number of times the function type constructor is nested to the left. Below are some examples of the order of types.

0	`int, A, (list (int -> int)), (pair int A)`
1	`int -> int, string -> list (pair string int) -> o`
	`int -> string -> o, int -> string -> A`
2	`(string -> string) -> string, int -> (int -> o) -> o`
	`(int -> int) -> (int -> o) -> int`

The order of a type expression may not be invariant under type variable substitution: the order of a type containing type variables may increase under type variable substitution. For example, while the order of the type `A -> list A -> o` is one, substituting `A` with a type expression of order n results in a type expression of order $n + 1$.

Notice that if the definition of order is applied to kind expressions, all kind expressions would have order either 0 or 1.

2.4 Type declarations

A *type declaration* starts with the keyword `type` and ends with a fullstop. Following the `type` keyword is a list of one or more tokens (separated by commas) followed with the type expression that is to be associated with the given tokens. The tokens in this list, all of which must start with a lowercase letter, will be declared to have the given type. Thus the syntax of a type declaration is similar to that of a kind declaration.

Consider the following kind and type declarations for pairs and lists.

```
kind list       type -> type.
type ::         A -> list A -> list A.
type nil        list A.
kind pair       type -> type -> type.
type pr         A -> B -> pair A B.
```

The first line declares that `list` is a type constructor of one argument. The two constructors for lists are given by the next two type declarations. The non-empty list constructor is written as the symbol `::` and the empty list is given by `nil`. The last two lines declare `pair` to be a type constructor for pairs and `pr` as the constructor for pairs.

Type variables in type declaratiosn are intended to be quantified universally with a quantifier around the type declaration. Thus, the type declaration for `nil` above asserts that "for every type `A`, the constant `nil` is of type `(list A)`." Thus, `nil` is considered to have an infinite number of different typings.

2.5 Infix declarations

The keywords `infixl` and `infixr` are used to declare a token as a constant that is to be parsed as an infix symbol that associates to either the left or right, respectively. The syntax of infix declarations is similar to that for kind and type declarations. They begin with either the `infixr` or `infixl` keyword, which are followed by a list of tokens, and these are followed by a single integer of value between 0 and 9, inclusively. This integer is called an *infix priority* and is used to disambiguate the parsing of numerous infix symbols given in a series: the larger the priority the tighter the symbol binds. (The possibility to use 10 numbers for priorities is generally seen to be sufficient: various other programming languages use a similar range). For example, the declaration for `::` is given as

```
infixr  ::    5.
```

A token declared to be infix must also be declared with a type of the form $\tau_1 \to \tau_2 \to \tau_0$, where τ_0 is some type. For example, let `compose` have the following declarations:

```
type   compose  (A -> B) -> (B -> C) -> A -> C.
infixr compose  6.
```

(Here, `compose` has the type of function composition.) Then the expression `(f compose g) x` should parse correctly (provided that `f`, `g`, and `x` have the appropriate types) and yields the expression that would be displayed as `(compose f g x)` if the infix declaration were not enforced. (The syntax of terms will be described in Section 3.1 and further extended in Chapter 6.)

Since constants can be given both type and infix declarations, the interaction between these declarations needs to be examined. Consider the following set of kind and type declarations.

```
kind number type.
type a,b,c  number.
type plus   number -> number -> number.
type less   number -> number -> o.
```

If `less` is to be declared to be infix, then either `infixr` or `infixl` declarations can be given, since it is not possible to have `less` be nested in a properly typed expression. Also, assume that both `less` and `plus` are infix and that we wish expressions of the form "`a less b plus c`" to parse as `a less (b plus c)`. While this is the only bracketing that will type check (that is, `(a less b) plus c` is type incorrect), the infix declarations must be given so that the priority of `plus` is higher than that for `less`. For example, the following infix declarations will cause the string above to be parsed as intended.

```
infixr plus 6.
infixr less 5.
```

When λProlog code is parsed, infix declarations are used in a phase that occurs prior to the use of type declarations.

2.6 Signatures

A *signature* is an ordered list of kind, type, and infix declarations for which the following restrictions apply.

1. If a token is given a kind declaration in this list, that declaration must appear prior to any use of it in a type declaration.

2. If a token is given an infix declaration, that token must also be given either a kind or a type declaration.

3. A token cannot be given two kind declarations, two type declarations, two infix declarations, or a kind and a type declaration. If a token is given two declarations, they must be either a type and infix declaration or a kind and infix declaration.

Two signatures are equal if they give the same set of tokens exactly the same set of declarations. For this comparison, we identify two type expressions if they differ only up to renaming of type variables. As we shall see in Chapter 4, signatures are contained within modules and are used to qualify modules much as kinds qualify types and types qualify terms. Signatures also play an important role in the runtime environment of a computation (see, for example, Section 3.3).

In summary, every item in a signature is declared with (at least) one of the following keywords: `infixl`, `infixr`, `type`, and `kind`. The keyword `type` is also allowed in kind expressions, and the arrow `->` is allowed in both kind and type expressions.

In subsequent chapters, the captial Greek letter Σ is used to range over signatures.

2.7 First-order types and signatures

While λProlog is a higher-order language, it has sublanguages that are first-order (see the map in Section 1.1). Chapter 3 through Chapter 5 will, in fact, only deal with these sublanguages. The first-order sublanguages are based on the notion of first-order signature, which is, in turn, based on the notion of first-order type. The latter is not simply a type that is of order 0 or 1, since that would allow types such as `o -> o` (the type of a predicate over formulas) and `list (int -> int)` (the type of a list of functions): such types are seldom considered to be first-order although their order is 0 or 1. Thus, a *first-order type* is defined as a type of order 0 or 1 such that the argument and target types do not contain occurrences of `->` and if the type contains an occurrence of `o`, that occurrence is the target type. A *first-order signature* is a signature such that all the types given in its declarations are first-order types.

Let symbol c be given a type τ in a first-order signature. If the target type of τ is `o`, then c is a *first-order predicate* symbol; otherwise, c is a *first-order function* symbol. If c is a first-order function symbol that is not functional, then we can also say that c is a *first-order individual* symbol.

These three classes can be described more directly as follows. Let τ_i range over first-order individual types, let τ_p range over first-order predicate types, and τ_f range over first-order function types. These three syntactic variables can be related using the following two grammar rules.

$$\tau_f ::= \tau_i \mid \tau_i \to \tau_f \qquad\qquad \tau_p ::= o \mid \tau_i \to \tau_p$$

Notice that in these two classes of types, there is never an occurrence of $\to$ that appears to the left of an $\to$: nesting of $\to$ occurs only to the right. Higher-order languages will allow $\to$ to be nested to the left of another $\to$.

The types of predicates are given using the function type constructor: that is, a predicate can be seen as a function that maps its arguments into formulas. This treatment of predicates is similar to the use of characteristic functions to denote sets. For more on the nature of types in λProlog, see Subsection 3.10.

The following is a first-order signature.

```
kind i          type.
type a, b       i.
type f          i -> i.
type g          i -> i -> i.
type r          o.
type p          i -> o.
type q          i -> i -> o.
```

Here, the type `i` is introduced, as well as two first-order individuals, `a` and `b`, two first-order functions `f` and `g` of arities 1 and 2, respectively, and three first-order predicates `r`, `p`, and `q` of arities 0, 1, and 2, respectively. (The symbols `a` and `b` can also be considered first-order function symbols.) The following signature, which contains type variables in its type expressions, is also first-order.

```
type append     list A -> list A -> list A -> o.
type member     A -> list A -> o.
```

Notice that argument types may be type variables.

Chapter 3

First-order Horn clauses

While λProlog is based on a higher-order logic, it is natural and desirable to start a description of λProlog by considering a first-order logic that lies within λProlog. After presenting that sublogic, we describe first-order Horn clauses (*fohc*) also in this chapter and first-order hereditary Harrop formulas (*fohh*) in the next chapter.

3.1 First-order terms

λProlog comes with certain built-in data structures, such as those for integers, real numbers, and strings (see Section 4.4). In order to build other kinds of data structures, such as lists and trees, terms are used. First-order terms are particularly simple structures and can be used for representing a wide variety of data structures.

The term structure of λProlog is actually richer than that of first-order terms. In particular, the language allows λ-terms and these generalize first-order terms by allowing bound variables within data structures. The topic of λ-terms will be left, however, until Chapter 6. One consequence of the use of λ-terms is that λProlog makes use of *curried syntax*: expressions of functional type are applied to one argument and, if the resulting expression is of functional type, can be applied to a second argument, and so on. A functional expression of type `a -> b -> c` can be applied to an argument of type `a` yielding a result of type `b -> c`. This latter expression can now be applied to an argument of type `b` to yield an expression of type `c`. An uncurried syntax convention, which is used in Prolog, for example, applies the functional or predicate expression to both arguments simultaneously. For example, in Prolog, if `f` is a function of two arguments, a term that involves `f` must have `f` applied to two arguments, as in the expression `f(t,s)`. In λProlog, it is possible to form the expressions `(f t)` and `((f t) s)`: since application associates to the left, this last expression can be written more simply as `(f t s)`.

The following signature can be used to represent binary trees in which leaves are labelled with integers.

```
kind btree           type.
type root            int -> btree.
type bt              btree -> btree -> btree.
```

Here, `btree` is a type constructor, and the symbols `root` and `bt` are used to build terms of type `btree`. The following are terms of type `(bt int)`.

```
root 5
bt (root 4) (root 5)
bt (bt (root 4) (root 5)) (root 3)
```

It is also possible to form term `(bt (root 4))` of type `btree -> btree`. The flexiblity that such "partial application" allows will become apparent starting in Chapter 6. Since this latter expression is not considered a part of first-order logic, we shall not be concerned with it now.

Lists can be encoded as terms by introducing the following declarations. Consider the following signature regarding lists.

```
kind   list          type -> type.
type   nil           list A.
type   '::'          A -> list A -> list A.
infixr '::'          5.
```

Here, `list` is a type constructor: that is, it can be used to take a type, for example, `int` (for integers) or `string` (for strings), and produce another type, for example, `(list int)` (for lists of integers) or `(list string)` (for list of strings). The symbol `nil` will be used to denote the empty list while the list constructor `::` (pronounced "cons") is the constructor that places an element on the front of a list. The following are examples of terms of type `(list int)`.

```
nil
1 :: nil
1 :: 4 :: 9 :: 25 :: nil
```

Notice that since `::` is declared to be an infix symbol that associates to the right, the last list can be fully parenthesized as

```
(1 :: (4 :: (9 :: (25 :: nil))))
```

The following are examples of lists of lists of integers.

```
nil
(1 :: nil) :: nil
(1 :: nil) :: (1 :: 2 :: nil) :: nil
```

Since lists are frequently used structures, implementations of λProlog generally contain these declarations already built into them.

We now provide a more formal definition of typed terms by introducing the following definitions.

In order to capture the instantiation of type variables within type expressions, we first introduce the following binary relation on type expressions. If τ is the result of substituting some variables in σ with type expressions, then we write $\tau \triangleleft \sigma$. Notice that $\triangleleft$ is reflective and transitive, and if both $\tau \triangleleft \sigma$ and $\sigma \triangleleft \tau$ hold, then σ and τ are equal up to changes in the names of type variables. It is possible that $\tau \triangleleft \sigma$ holds for two types, and that σ is a first-order type and τ is not. To avoid this possibility, we denote by $\triangleleft_f$ the binary relation that is the restriction of $\triangleleft$ relation to first-order type expressions.

The *typing judgment* $\Sigma; \Gamma \Vdash_f t : \tau$ is used to judge that t is a first-order term of type τ with respect to signatures Σ and Γ. The signature Σ is used to hold the declarations for

$$\frac{c{:}\,\sigma \in \Sigma_0 \quad \tau \lhd_f \sigma}{\Sigma; \Gamma \Vdash_f c{:}\,\tau} \qquad \frac{c{:}\,\sigma \in \Sigma \quad \tau \lhd_f \sigma}{\Sigma; \Gamma \Vdash_f c{:}\,\tau} \qquad \frac{c{:}\,\tau \in \Gamma}{\Sigma; \Gamma \Vdash_f c{:}\,\tau}$$

$$\frac{\Sigma; \Gamma \Vdash_f g{:}\,\tau_1 \to \tau_2 \quad \Sigma; \Gamma \Vdash_f t{:}\,\tau_1}{\Sigma; \Gamma \Vdash_f (g\ t){:}\,\tau_2}$$

Figure 3.1: Rules for defining typed first-order terms.

constants while the signature $\Gamma$is used to hold the types of bound variables (which we will encounter in the next section). This relation is defined using the rules in Figure 3.1. The subscript on both$\Vdash_f$ and $\lhd_f$ indicates that they are restricted to first-order terms and types. Later we will give versions of these relations that are not so restricted. Notice that the difference between Σ and Γ in these rules is that the types variables in types from Σ can be instantiated while type variables in types from Γ can not be instantiated. When using this judgment, we shall alway assume that no symbol is given a declaration in both of Σ and Γ. Also notice the reference to the signature Σ_0 in the first rule in Figure 3.1: this is the "ambient" signature in which all builtin constants of the λProlog interpreter have been placed. This signature will be revealed as we proceed. For now, we shall assume that it contains declarations for all integers, reals, and string.

Let τ be a first-order type of order 0. The expression t is a *first-order* Σ*-term of type* τ if $\Sigma;\ \Vdash_f t{:}\,\tau$ is provable (here, Γ is empty so we elide it). For example, let Σ be a signature containing the three declarations above for binary trees. To show that `(bt (root 4) (root 5))` is a first-order Σ-term of type `btree`, we can proceed as follows. First notice that $\Sigma;\ \Vdash_f$ `4`: `int` holds since 4 is given type `int` in Σ_0 (since Σ_0 contains all builtin constants) and $\lhd_f$ is reflexive. Similarly, we know $\Sigma;\ \Vdash_f$ `5`: `int`. Next, $\Sigma;\ \Vdash_f$ `bt`: `int -> btree -> btree` since `bt` is given type `int -> btree -> btree` in Σ and $\lhd_f$ is reflexive. Given these, we can then conclude that $\Sigma;\ \Vdash_f$ `bt (root 4)`: `btree -> btree` and, finally, $\Sigma;\ \Vdash_f$ `bt (root 4) (root 5)`: `btree`.

Notice that if Σ was assumed to have the declarations for lists above, the establishing that `(1 :: nil)` would require showing that the type `(list int)` is related by $\lhd_f$ to the type `(list A)`, which is the type declared for `nil`. (A similar step is required for the `::` constructor.)

Typing is used in a wide variety of situations in logic and programming languages. See [**?**, Pfe92] for some additional references to uses of types and typing judgments.

3.2 First-order formulas

The logic underlying λProlog is based on six logical connectives, which, when written using a mathematical notion, are $\top$ (truth), $\wedge$ (conjunction), $\vee$ (disjunction), $\supset$ (implication), $\exists_\tau$ (existential quantification over type τ), and $\forall_\tau$ (universal quantification over type τ). In the concrete syntax of λProlog, these are written as follows. The constant $\top$ is written as `true`. The two infix symbols `,` and `&` both denote conjunction: while they denote the same connective and can be used interchangeably, a convention we adopt later (Section 3.5) suggests using them in different situations. The infix symbol `=>` denotes implication while the infix symbol `:-` is the converse of implication or "implied-by." Since they are both converses of each other, we need only one of these connectives; but as for conjunction, we adopt later a convention that suggests using them in different situations. The infix semicolon `;` is used to denote disjunction.

In λProlog, both universal and existential quantification range over explicit domains, which are specified by a type: $\forall_\tau x$ and $\exists_\tau x$ denote universal and existential quantification over type τ. The concrete syntax for these quantifiers is written as `pi x\` and `sigma x\`, respectively, where the concrete syntax of the bound variable x is written as `x`. Notice that in the concrete syntax the type of `x` is not written: the syntax of λProlog is such that it is possible for this type to be inferred from context.

In Section 3.1, we defined terms by introducing the judgment $\Sigma;\Gamma \Vdash_f t\colon\tau$. To define formulas, we could introduce a new judgment, but if we use type information appropriately (and extend this typing judgment to handle the bound variables of quantified expressions), we can reuse the judgment for terms for defining formulas. In particular, if we reserve the type `o` for formulas (as was suggested in Section 2.2), then we can use the judgment $\Sigma;\Gamma \Vdash_f t\colon o$ to define t as a first-order formula. In particular, λProlog uses the signature in Figure 3.2 to declare the types of the logical constants. We shall assume that the ambient signature Σ_0 introduced in the previous section contains these declarations. The signature in Figure 3.2 is not a first-order signature (see Section 2.7). In particular the types of `pi` and `sigma` are of order 2 and the other symbols have occurrences of `o` outside of their target types.

Since predicate symbols have `o` as their target type (see Section 2.7), they can be used to build *atomic formulas*: these are formulas of the form $p\ t_1 \cdots t_n$, where $n \geq 0$, p is a first-order predicate of type $\tau_1 \to \cdots \to \tau_n \to o$, and for each $i = 1, \ldots, n$, t_i is a term of type τ_i. With the following declaration of three predicates

```
type   memb           A -> list A -> o.
type   append, join   list A -> list A -> list A -> o.
```

and the declarations for logical connectives in Figure 3.2, the following can all be judged to be first-order formulas.

```
append (1 :: nil) (2 :: nil) (1 :: 2 :: nil)
join (2 :: 5 :: nil) (1 :: 4 :: 9 :: 25 :: nil) (2 :: 4 :: nil)
memb 1 (2 :: nil) => memb 2 (2 :: nil)
(append nil nil nil ; memb 1 (2 :: nil)), join nil nil (3 :: nil)
```

The first two of these formulas are atomic formulas.

Quantified expressions introduce the notion of *bound variable*: this is represented in the concrete syntax using a backslash. (In Chapter 6 we shall see that the backslash is actually λ-abstraction.) When reading an expression containing such a backslash symbol, the body of the quantified expression is the expression as far to the right as is possible (given the presence of other parentheses and the end of the expression). For example, the expression,

```
pi x\ pi z\ join x z x => sigma y\ join x y z, pi x\ join y z x
```

will be parsed as the following expression.

```
(pi x\ (pi z\ (join x z x => (sigma y\ (join x y z, pi x\ (join y z x))))))
```

(All the bound variables in this expression will have the type `(list A)`.) Such a convention is useful when a series of binders is necessary. For example, the formula

```
kind   o                             type.

type   true                          o.
type   '&', ',', ':-', ';', '=>'     o -> o -> o.
type   pi, sigma                     (A -> o) -> o.

infixl ':-'                          0.
infixl ';'                           1.
infixr ','                           2.
infixr '&'                           3.
infixr '=>'                          4.
```

Figure 3.2: Declarations for the logical constants.

```
pi x\ pi y\ pi z\ join x y z
```

denotes the universal closure of the atomic formula `join x y z`. The following are some more examples of formulas containing quantification.

```
pi x\ pi k\ memb x (x :: k)
pi X\ pi L\ pi K \ pi M\ append (X::L) K (X::M) :- append L K M
sigma X\ pi y\ sigma h\ append X y h
```

The lexical notion of *bound variable scope* used in λProlog is the usual one taken from logic or form most programming languages. The scope of bound variable is the expression that follows the backslash. An important aspect of this scoping is that the name of the bound variable is not as important as its pattern of binding. In particular, the names of bound variables can be changed systematically without affecting the meaning of a quantified formula. For example, all the following formulas are related to each other by renaming of bound variables.

```
pi x\ (p x) => sigma y\ (q x y, pi x\ (q y x))
pi x\ (p x) => sigma U\ (q x U, pi x\ (q U x))
pi z\ (p z) => sigma y\ (q z y, pi x\ (q y x))
pi z\ (p z) => sigma y\ (q z y, pi v\ (q y v))
```

We say that these formulas are related by *α-conversion* (described in more detail in Chapter 6).

Given the types used to declare logical constants, *first-order formulas* can be defined using the type judgement $\Sigma;\Gamma \Vdash_f t\colon\tau$ if we add the additional rules displayed in Figure 3.3 to those in Figure 3.1. Notice that the Γ signature now plays the role of holding the type of bound variables. When Σ be is signature, we shall say that B is a *Σ-formula* if $\Sigma; \Vdash_f B\colon o$ (here, the Γ-signature is empty). Similary, we shall say that t is a *Σ-term of type* τ if $\Sigma; \Vdash_f t\colon\tau$. Clearly, a Σ-term of type `o` is also a Σ-formula.

3.3 Programs, goals, and proof search

In order to specify computation in λProlog, a programmer will first specify a signature, say Σ, that contains kind, type, and infix declarations for the *non-logical constants* used for

$$\frac{\Sigma;\Gamma,x\colon\tau \Vdash_f B\colon o}{\Sigma;\Gamma \Vdash_f \forall_\tau x\ B\colon o} \qquad \frac{\Sigma;\Gamma,x\colon\tau \Vdash_f B\colon o}{\Sigma;\Gamma \Vdash_f \exists_\tau x\ B\colon o}$$

provide that x is not declared as a type or kind in Σ_0, Σ, or Γ.

$$\frac{\Sigma;\Gamma \Vdash_f B\colon \tau}{\Sigma;\Gamma \Vdash_f C\colon \tau}$$

provided B and C differ only in the names of bound variables.

Figure 3.3: Additional rules for dealing with quantifiers.

specifying a computation. These constants are then used to build a formulas. Formulas are used in two roles. A *logic program* is a collection of formulas that specifies, at least partially, the meaning of the constants in Σ. We can view formulas in logic programs as givens that describe the meaning of the non-logical constants they use. When constructing proofs, we reason *from* these formulas. A *query* or *goal* is a formula that serves as a question to ask of a logic program: we reason *to* goals and use them to explore consequences of the specifications given by programs.

Computation is then the process of attempting to prove that a given goal follows from a given logic program. If this attempt is unsuccessful, the result of the computation is simply an indication that there was such a failure. If the attempt is successful, the resulting proof could be returned: however, since proofs in this setting are essentially traces of entire computations, an extraction from this proof is returned instead. This extract is a substition for certain of the variables found in the goal formula. This *answer substitution* is describe more below.

Since computation is the search for a proof, we need to examine this notion more. To this end, we can view an idealized interpreter for λProlog as having three components: a signature Σ, a set of Σ-formulas $\mathcal{P}$ denoting a program, and a Σ-formula G denoting the goal we wish to prove from $\mathcal{P}$. We use the *sequent* notation $\Sigma;\mathcal{P} \longrightarrow G$ to denote the *state* of this idealized interpreter: read this notation for now as simply a triple containing the key elements of the interpreter's state. If the program component is empty, we write simply $\Sigma;\longrightarrow G$. We shall also consider just the pair $\langle\sigma,\mathcal{P}\rangle$ and refer to this as a *signature-program pair*: these supply the context for goals. For more detailed information about organizing logical proofs using sequents, see [Gen69, Gal86, Mil90].

If the interpreter has state $\Sigma;\mathcal{P} \longrightarrow G$, then how should a proof that G follows from $\mathcal{P}$ be attempted? In general, logical derivation is complex and intricate, and to navigate to a successful proof can demand cleverness and invention. A common technique in proving mathematical theorems, for example, is the invention of a sequence of lemmas that breaks a proof into small pieces. While attempting to automate the selection of lemmas is certainly an interest problem, it would seem to fall outside the domain of programming language executing, even in a high-level programming language like λProlog. Given results in the theory of proofs, particularly the result known as *cut-elimination* [Gen69], it is always possible, in principle, to search for proofs without looking for lemma: instead, only the formulas in the current state of the interpreter (in $\mathcal{P}$ and G) need to be examined and manipuated. Focusing on proofs that do not involve any use of lemma means that we are not really thinking of provability in the mathematical sense (since proofs of non-trivial mathematical theorems involve numerous uses of lemmas) but in a simple computational sense: λProlog will not attempt to proof theorems in point-set topology but will attempt more modest goals such as sorting and merging lists using logic. This focus affords a great deal of simplication in the search for proof, but even then, there is still a great deal of richness attempting proofs. We now argue that we should make additional restrictions on the kind of the search we will allow for proofs.

The principal restriction that we shall make in restricting the search for proofs is that that search should be *goal-directed*: that is, if the goal formula has a logical connective as its toplevel symbol, then the proof should be attempted by reducing that logical connective in a specific fashion. The particular rule depends on the logical connective. (The notion of goal-directed search can be formalized in the sequent calculus using the technical notion of *uniform proofs*[MNPS91].) In particular, we shall have the following reduction strategies for sequents.

AND Reduce $\Sigma; \mathcal{P} \longrightarrow B_1 \wedge B_2$ to the two sequents $\Sigma; \mathcal{P} \longrightarrow G_1$ and $\Sigma; \mathcal{P} \longrightarrow B_2$. Proofs of both sequents must now be attempted.

OR Reduce $\Sigma; \mathcal{P} \longrightarrow B_1 \vee B_2$ to either $\Sigma; \mathcal{P} \longrightarrow B_1$ or $\Sigma; \mathcal{P} \longrightarrow B_2$. A proof on only one of these needs will be sufficient.

INSTAN Reduce $\Sigma; \mathcal{P} \longrightarrow \exists_\tau x.B$ to $\Sigma; \mathcal{P} \longrightarrow B[t/x]$, for some Σ-term t of type τ.

AUGMENT Reduce $\Sigma; \mathcal{P} \longrightarrow B_1 \supset B_2$ to $\Sigma; \mathcal{P} \cup \{B_1\} \longrightarrow B_2$.

GENERIC Reduce $\Sigma; \mathcal{P} \longrightarrow \forall_\tau x.B$ to $\{c : \tau\} \cup \Sigma; \mathcal{P} \longrightarrow B[c/x]$, where c is a token that is not in the current signature Σ. We shall often refer to c as a "new constant".

TRUE The sequent $\Sigma; \mathcal{P} \longrightarrow \top$ is provable immediately and does not need to be reduced further.

These reduction rules are goal-directed and do not consider either the signature or the logic program. Thus logical connectives get reflected into the search for proofs in a fixed fashion that cannot be modified by a program. For example, the connectives $\wedge$ and $\vee$ are always mapped into AND and OR search steps. Logic programs are responsible for determining the meaning of only the non-logical constants that are used to build atomic formulas.

If these reduction rules are reversed, the result can be seen as inference rules of logic. For example, if the sequent $\Sigma; \mathcal{P} \longrightarrow B[t/x]$ can be proved for some Σ-term t of type τ, then we have a justification for the sequent $\Sigma; \mathcal{P} \longrightarrow \exists_\tau x.B$. Figure 3.4 displays the inference rules corresponding to these reduction rules. These inference rules are also called *right-introduction rules* since below the horizonal line there is an occurrence of a logical connective on the right of the sequent arrow that does not occur above the line.

Since we are attempting to justify the design of a *logic* programming language, it is natural to ask to what extent are these inference rules related to logic. There are also choices to be made regarding which logic we should be considering. For now, we shall assume that the reader is familar what is sometimes called *classical logic*: that is, the logic used in informal mathematical arguments (also in the treatment of elementary syllogism and in boolean circuits).

$$\frac{}{\Sigma; \mathcal{P} \longrightarrow \top}\ \top R \qquad \frac{\Sigma; \mathcal{P} \longrightarrow B_1 \qquad \Sigma; \mathcal{P} \longrightarrow B_2}{\Sigma; \mathcal{P} \longrightarrow B_1 \wedge B_2}\ \wedge R$$

$$\frac{\Sigma; \mathcal{P} \longrightarrow B_1}{\Sigma; \mathcal{P} \longrightarrow B_1 \vee B_2}\ \vee R \qquad \frac{\Sigma; \mathcal{P} \longrightarrow B_2}{\Sigma; \mathcal{P} \longrightarrow B_1 \vee B_2}\ \vee R$$

$$\frac{\Sigma; \mathcal{P} \cup \{B_1\} \longrightarrow B_2}{\Sigma; \mathcal{P} \longrightarrow B_1 \supset B_2}\ \supset R$$

$$\frac{\Sigma; \mathcal{P} \longrightarrow B[t/x] \qquad \Sigma; \Vdash_f t : \tau}{\Sigma; \mathcal{P} \longrightarrow \exists_\tau x.B}\ \exists R \qquad \frac{\{c : \tau\} \cup \Sigma; \mathcal{P} \longrightarrow B[c/x]}{\Sigma; \mathcal{P} \longrightarrow \forall_\tau x.B}\ \forall R$$

Figure 3.4: Right-introduction rules. The rule for universal quantification has the proviso that c is not declared in Σ.

It is easy to see that each of these inference rules is sound: that is, if the premise sequents (the sequents above the horizontal line in the inference rule) are true (in, say classical logic), then the original sequent is true. Soundness can be established without knowing the exact nature of signatures and programs.

The converse property, that of logical completeness of these rules, can be phrased as follows: if a sequent with a non-atomic goal is true, are the sequents that it reduces to also true? Achieving completeness is more involved and, in fact, dominates the design of the logic programming languages we considered here. To see that these reductions are not generally complete, consider the following examples. Here, let signature Σ contain the declarations $\{p : o, q : o, r : i \rightarrow o, a : i, b : i\}$ (and the kind declaration for i).

1. The OR rule reduces the sequent $\Sigma; p \vee q \longrightarrow q \vee p$ to either $\Sigma; p \vee q \longrightarrow q$ or $\Sigma; p \vee q \longrightarrow p$. Neither of these sequents is true while the original sequent is true.

2. The OR rule reduces the sequent $\Sigma; \longrightarrow p \vee (p \supset q)$ to either $\Sigma; \longrightarrow p$ or $\Sigma; \longrightarrow p \supset q$. The first sequent is not provable and the second sequent would reduce to $\Sigma; p \longrightarrow q$, which is also not provable. It is easy to see, however, that $p \vee (p \supset q)$ is a classical logic tautology: if p is true, then the disjunction $p \vee (p \supset q)$ is true and if p is false, then $p \supset q$ is true and again the disjunction is true.

3. The INSTAN rule reduces the sequent

$$\Sigma; (ra \wedge rb) \supset q \longrightarrow \exists_i x (r\ x \supset q)$$

to the sequent $\Sigma; (r\ a \wedge r\ b) \supset q \longrightarrow r\ t \supset q$, where t is a Σ-term of type i. But there is not such term which makes this sequent provable. For example, if we used a for t, we would have the sequent

$$\Sigma; (ra \wedge rb) \supset q, r\ a \longrightarrow q$$

and this is no longer represents a true statement. To see that the original sequent is true classically, we know that $r\ a$ is either true or false. If it is false, then $\exists_i x (r\ x \supset q)$ is true (by picking a for x). If $r\ a$ is true, then $(ra \wedge rb) \supset q$ is equivalence to $rb \supset q$, so once again we have shown $\exists_i x (r\ x \supset q)$.

To achieve completeness for our logic programming languages, we shall need to either restrict the formulas allowed to be programs and goals (so as to avoid these counterexamples) or change our logic to be different from classical logic. In fact, we shall need to take both of these steps to formally justify the completeness of the logical basis of λProlog. As the first example illustrates, it seems likely that we will need to avoid having disjunctions in our programs. In fact, the formulas we eventually allow in programs will also be called *definite* formulas since they do not contain the indefinite information supplied by disjunctions. The last two examples illustrate that this step will not be enough since classical logic itself has built into it a kind of indefinite assumption, called the *excluded middle*: for every formula B, classical logic makes the formula $B \vee \neg B$ true. Thus, we shall also need to move to *intuitionistic logic*, a logic weaker than classical logic where the excluded middle does not hold.

For the rest of this chapter, however, we shall consider a restriction of programs and goals for which both classical and intuitionistic logics are complete. This restriction will be based on *first-order Horn clauses* (*fohc*), the logical foundations of the Prolog language. In Chapter 5 we shall extend the syntax of programs and goals to obtain a more expressive language: the extended language will no longer be complete for classical logic. We shall then rely on intuitionistic logic to supply us our logical foundations.

The discussion about proof search in this section is quite idealized in at least two senses. First, the reduction rule INSTAN requires that a term be used to instantiate a existentially quantified goal. Exactly what this term should be can be hard to determine, and finding just such a term is generally considered the result of an entire computation. Second, the eventual interpreter we describe for λProlog will be incomplete in practice. The λProlog interpreter uses a rather simple strategy for finding proofs: although goal-directed search will be complete with respect to intuitionistic logic for the logic underlying λProlog, its interpreter uses a simple and inflexible search procedure. For now our interests have been the design of a logic that can be used to provide a foundation for a logic programming language. There might be many implementations of that design: in Section 3.9, we present more information about how proof search is implemented in λProlog.

3.4 The syntax of first-order Horn clauses

There are several, roughly equivalent ways to describe first-order Horn clauses (*fohc* for short). We present three here. In making these definitons, we make use of three syntactic variables: A denotes atomic formulas, G denotes goal formulas, and D denotes program formulas (also called definite formulas). Programs formulas are also called *clauses*.

A common definition of Horn clauses (see, for example, [AvE82]) is given using the following grammar.

$$\begin{aligned} G &::= \quad A \mid G \wedge G \\ D &::= \quad A \mid G \supset A \mid \forall_\tau x\ D. \end{aligned} \tag{3.1}$$

(Here and in the rest of this chapter, we assume that the type τ is of order 0.) That is, goal formulas are conjunctions of atomic formulas and program clauses are of the form

$$\forall_{\tau_1} x_1 \ldots \forall_{\tau_1} x_m [A_1 \wedge \ldots \wedge A_n \supset A_0]$$

for $m, n \geq 0$. (If $m = 0$ then we do not write any universal quantifiers, and if $n = 0$ then we do not write the implication.)

A richer formulation is given by the following definition.

$$\begin{aligned} G &::= \top \mid A \mid G \wedge G \mid G \vee G \mid \exists_\tau x\, G \\ D &::= A \mid G \supset D \mid D \wedge D \mid \forall_\tau x\, D. \end{aligned} \tag{3.2}$$

Here, the connectives $\top$, $\vee$, and $\exists$ are permitted in goals and $\wedge$ and $\forall$ can be mixed in definite formulas. Also, the "head" of a definite clause does not need to be immediately present at the top-level of a clause: it might be buried to the right of implications and conjunctions and under universal quantifiers.

A compact presentation of Horn clauses can be given simply as

$$\begin{aligned} G &::= A \\ D &::= A \mid A \supset D \mid \forall_\tau x\, D. \end{aligned} \tag{3.3}$$

Notice that in this definition, definite clauses are composed only of implications and universal quantifiers where the nesting of implications and universal quantifiers is allowed only in the conclusion of an implication and not in a premise.

It is the D-formulas that are considered Horn clauses. A Horn clause program is then a finite set of *closed* D-formulas. The symbol $\mathcal{P}$ will often be used as a syntactic variable to denote programs.

These three ways of defining program clauses give rise to programming languages of the same expressive power: that is, if a program clause in one definitions is classically equivalent (also intuitionistically equivalence) to a conjunction of program clauses in another definition. This is easily shown by using suitable applications of the following classical equivalences.

$$\begin{aligned} \forall x(B_1 \wedge B_2) &\equiv (\forall x\, B_1) \wedge (\forall x\, B_2) \\ B_1 \supset (B_2 \supset B_3) &\equiv (B_1 \wedge B_2) \supset B_3 \\ B_1 \wedge (B_2 \vee B_3) &\equiv (B_1 \wedge B_2) \vee (B_1 \wedge B_3) \\ B_1 \vee (B_2 \wedge B_3) &\equiv (B_1 \vee B_2) \wedge (B_1 \vee B_3) \\ (B_1 \vee B_2) \supset B_3 &\equiv (B_1 \supset B_3) \wedge (B_2 \supset B_3) \\ B_1 \supset (B_2 \wedge B_3) &\equiv (B_1 \supset B_2) \wedge (B_1 \supset B_3) \\ B_1 \supset (\forall x\, B_2) &\equiv \forall x\, (B_1 \supset B_2) \\ (\exists x\, B_2) \supset B_1 &\equiv \forall x\, (B_2 \supset B_1) \end{aligned}$$

In the last two equivalences, x is not free in B_1. Since we shall have occasions to use all eight of these equivalences again when we will be concerned with intuitionistic logic, it is important to note that all of these equivalences hold in that logic also.

If we take the size of a program to be the number of occurrences of logical connectives it contains, equivalent programs using (3.2) are generally smaller than those using (3.1) or (3.3). For example, a Horn clause of the form $G \supset (D_1 \wedge D_2)$ is logically equivalent to the formula $(G \supset D_1) \wedge (G \supset D_2)$, but since G is duplicated, this second formula could be much larger than the first. Thus, removing conjunctions from the right of an implication in this way can cause the size of the formula to grow exponentially. For another example, consider the following propositional Horn clause given using (3.2):

$$((p \vee r) \wedge (q \vee t)) \supset s.$$

If some of the equivalences above are used (most notably the distributivity of conjunction over disjunction), this formula is equivalent to the conjunction of the following clauses given using (3.1):

$$(p \wedge q) \supset s \qquad (r \wedge q) \supset s \qquad (p \wedge t) \supset s \qquad (r \wedge t) \supset s$$

The conjunction of these formulas can be much larger than the original formula since it contains two occurrences each of p, r, q, and t, and each of these could be replaced with large formulas. This doubling of subformulas can lead to a exponential growth in the size of formulas. If new predicate constants are allowed, disjunctions can be replaced by new propositional constants. For example, if the propositional constants pr and qt are introduced to denote the disjunctions $p \vee r$ and $q \vee t$, then the original clause can be written as

$$p \supset pr \qquad r \supset pr \qquad q \supset qt \qquad t \supset qt \qquad (pr \wedge qt) \supset s.$$

Replacing disjunctions in this fashion can cause at most a linear (in the number of disjunctions) growth in the size of formulas. Since we have introduced new constants, the original formula is not logically equivalent to the resulting formula. The following statement about their relationship, however, can be made. Let D_1 denote the original Horn clause and Let Σ_1 be a signature containing $\{p : o, q : o, r : o, s : o, t : o\}$ and not containing declarations for pr and qt: Let Σ_2 be the result of adding $\{pr : o, qt : o\}$ to Σ_1. Then the sequent $\Sigma_1; \mathcal{P} \cup \{D_1\} \longrightarrow G$ is provable if and only $\Sigma_2; \mathcal{P} \cup \{D_2\} \longrightarrow G$ is provable. If we are only concerned with provability, this latter transformation on such clauses is acceptable. We shall return to this transformation again after we introduce higher-order Horn clauses.

We shall usually assume that first-order Horn clauses are based on the richest of these three definitions, namely (3.2).

3.5 Polarity and clausal order

In order to describe the structure of logical formulas, we use the notions of *positive subformula occurrences* and *negative subformula occurrences*.

If a subformula C of B occurs to the left of an even number of occurrences of implications in a B, then C is a *positive* subformula occurrence of B. On the other hand, if a subformula C occurs to the left of an odd number of occurrences of implication in a formula B, then C is a *negative* subformula occurrence of B. More formally:

- B is a positive subformula occurrence of B.
- If C is a positive subformula occurrence of B then C is a positive subformula occurrence in $B \wedge B'$, $B' \wedge B'$, $B \vee B'$, $B \vee B'$, $B' \wedge B$, $B' \supset B$, $\forall_\tau x.B$, and $\exists_\tau x.B$; C is also a negative subformula occurrence in $B \supset B'$.
- If C is a negative subformula occurrence of B then C is a negative subformula occurrence in $B \wedge B'$, $B' \wedge B'$, $B \vee B'$, $B \vee B'$, $B' \wedge B$, $B' \supset B$, $\forall_\tau x.B$, and $\exists_\tau x.B$; C is also a positive subformula occurrence in $B \supset B'$.

In all of the various subsystems of λProlog, the following invariance will always hold: positive subformulas of G-formulas are G-formulas and negative subformulas of G-formulas are D-formulas. Dualy, positive subformulas of D-formulas are D-formulas and negative subformulas of D-formulas are G-formulas.

Using any of the definitions for Horn clauses, we see also that a G-formula has no negative subformulas.

Analogous to types, formulas can be given orders. We define *clausal order* using the following recursion on first-order formulas.

$$
\begin{array}{rcl}
\text{clausal}(A) & = & 0 \quad \text{provided } A \text{ is atomic or } \top \\
\text{clausal}(B_1 \wedge B_2) & = & \max(\text{clausal}(B_1), \text{clausal}(B_2)) \\
\text{clausal}(B_1 \vee B_2) & = & \max(\text{clausal}(B_1), \text{clausal}(B_2)) \\
\text{clausal}(B_1 \supset B_2) & = & \max(\text{clausal}(B_1) + 1, \text{clausal}(B_2)) \\
\text{clausal}(\forall x.B) & = & \text{clausal}(B) \\
\text{clausal}(\exists x.B) & = & \text{clausal}(B)
\end{array}
$$

Notice that in all three variations, first-order Horn clauses have clausal order of either 0 or 1, and goal formulas have clausal order 0.

There are several different notions of order in the literature. Here we use order to simply count the number of occurrences of function type constructor to the left of a function type constructor, or the number of occurrences of an implication to the left of an implication. For more about these other senses of order, see Section 6.5.1.

3.6 Proof search with first-order Horn clauses

Following the idealized model of computing introduced in Section 3.3, programming with *fohc* means that we have constructed a first-order signature Σ and a program $\mathcal{P}$ (a finite set of first-order Horn clauses) and that we wish to know if a certain goal formula G follows from that program. (Of course, the formula G and all the formulas in the set $\mathcal{P}$ are Σ-formulas.) In other words, we wish to search for a proof of the sequent $\Sigma; \mathcal{P} \longrightarrow G$. In this setting, goal-directed search is complete for classical logic. To give a more complete picture of proof search within *fohc*, we need to describe how proof search deals with atomic goals.

For examples, if the goal G is the atomic formula A and the program $\mathcal{P}$ contains the formula A, then we clearly have a proof and computation (search) finishes immediately with a success. If $\mathcal{P}$ contains instead a clause of the form $G' \supset A$ then we know that if we can prove G' from $\mathcal{P}$, then we have again found a proof for A: since $G' \supset A$ and G' follow from $\mathcal{P}$, then so to does A. In this case, we have reduced the problem of proving $\Sigma; \mathcal{P} \longrightarrow A$ to proving $\Sigma; \mathcal{P} \longrightarrow G'$. Using a program clause in this manner to reduce the problem of proving an atomic formula is generally called *backchaining*.

To describe backchaining we use the additional inference rules found in Figure 3.5. To indicate that the interpreter is attempting to prove the atomic goal A by backchaining on the program clause D, we use the expression $\Sigma; \mathcal{P} \xrightarrow{D} A$. The first of the rules in Figure 3.5 specifies that when reducing the problem of finding a proof of $\Sigma; \mathcal{P} \longrightarrow A$, we need first to pick a member D of $\mathcal{P}$ and then attempt to backchain on it. The second rule in this figure states the obvious: if the formula that we are using for backchaining is the formula we are attempting to prove, then we are finished. If the formula fir backchaining is a conjunction, then reduce this attempt to one using one of the conjuncts. If the backchain forumula is universally quantified, then pick a Σ-term and continue backchaining with that instance of the formula. Finally, if the the backchain formula is an implication, say $G \supset D$, then we need to need to do two things: we must prove G and continue using D to do backchaining.

$$\frac{\Sigma;\mathcal{P} \xrightarrow{D} A}{\Sigma;\mathcal{P} \longrightarrow A}\ \mathit{decide} \qquad \frac{}{\Sigma;\mathcal{P} \xrightarrow{A} A}\ \mathit{initial} \qquad \frac{\Sigma;\mathcal{P} \xrightarrow{D_1} A}{\Sigma;\mathcal{P} \xrightarrow{D_1\wedge D_2} A}\ \wedge L \qquad \frac{\Sigma;\mathcal{P} \xrightarrow{D_2} A}{\Sigma;\mathcal{P} \xrightarrow{D_1\wedge D_2} A}\ \wedge L$$

$$\frac{\Sigma;\mathcal{P} \xrightarrow{D} A \quad \Sigma;\mathcal{P} \longrightarrow G}{\Sigma;\mathcal{P} \xrightarrow{G\supset D} A}\ \supset L \qquad \frac{\Sigma;\mathcal{P} \xrightarrow{D[t/x]} A \qquad \Sigma;\Vdash_f t\colon\tau}{\Sigma;\mathcal{P} \xrightarrow{\forall_\tau x.D} A}\ \forall L$$

Figure 3.5: Rules for backchaining. In the first rule, D is a member of $\mathcal{P}$.

If we restrict to using Horn clauses given by the first definition (3.1), that is, where program clauses are of the form

$$\forall_{\tau_1} x_1 \ldots \forall_{\tau_1} x_m [A_1 \wedge \ldots \wedge A_n \supset A_0]$$

where $m, n \geq 0$, then it is possible to simplify the inference rules for backchaining into one rule. In particular, if D is a formula of the above form, then the inference rule

$$\frac{\Sigma;\mathcal{P} \longrightarrow A_1\theta \qquad \cdots \qquad \Sigma;\mathcal{P} \longrightarrow A_n\theta}{\Sigma;\mathcal{P} \xrightarrow{D} A}$$

suffices to describe backchaining. This rule has the proviso that θ is a substitution such that for all $i = 1, \ldots, m$ θ maps the variable x_i to the Σ-terms t_i of type τ_i, and that A is equal to $A_0\theta$. The soundness of this rule can be argued as follows: Assume that $A_1\theta, \ldots, A_n\theta$ follow from $\mathcal{P}$ (that is, that the premises of the inference rule are provable). Scine $\mathcal{P}$ contains the clause $\forall_{\tau_1} x_1 \ldots \forall_{\tau_1} x_m [A_1 \wedge \ldots \wedge A_n \supset A_0]$ then the instance $[A_1\theta \wedge \ldots \wedge A_n\theta \supset A_0\theta]$ also follow from $\mathcal{P}$. Finally, using modus ponens, we know that $A_0\theta$ follows from $\mathcal{P}$.

The inference rules in Figure 3.5 are instances of *left-introduction rules* since below the horizonal line there is an occurrence of a logical connective on the left of the sequent arrow that does not occur above the line: note that a formula on top of the sequent arrow is actually a distinquished formula that is taken from the left of the sequent arrow.

Combining the right and left introduction rules in Figure 3.4 and 3.5 now yields a complete proof system for *fohc* with respect to classical logic. A proof of this fact can be found in [NM90]. When read bottom-up, these inference rules provide a complete set of reduction steps for finding a proof.

Consider a proof of the sequent $\Sigma;\mathcal{P} \longrightarrow G$ using these inference rules. It is easy to see that every sequent that appears in such a proof will either be of the form $\Sigma;\mathcal{P} \longrightarrow G'$ or $\Sigma;\mathcal{P} \xrightarrow{D} A$, for some A, G, and D. Notice that all of these sequents contain the same signature and program. Thus, during the search for such a proof, all goals will be attempted from the same signature and program and all the selection of program clauses will be from the same program and all instances of such clauses will be from the same signature. This observation has important consequences for the large scale organization of code. If we initiate a computation (that is, a search for a proof), all the program clauses that will ever be needed to complete the proof must be present in the initial sequent. Similarly, every constant, and hence the constructors for every data structure that might every need to be built, need to be present in the initial sequent. Thus, proof search using *fohc* provides no mechanisms for hiding code or data constructors. Signatures and programs are global, flat structures that do not change over the lifetime of a computation. Thus it will not be possible in *fohc* to have auxillary programs available only when they are needed and it will not be possible to build data structures (terms) that only certain code will be allowed to access. If any code or term constructors are ever needed they must be available from the start on equal footing with all other code and data constructors. This lack abstraction will be one of the motivations for going beyond *fohc*, as we shall do starting in Chapter 5.

3.7 Concrete syntax for program clauses

Program clauses often have numerous universal quantifiers at their outermost level. To make such clauses simpler to enter and display, λProlog uses the following conventions.

- A token in a program clause that is not explicitly quantified or otherwise reserved is assumed to be either a constant or implicitly universally quantified with maximum scope: if its initial letter is uppercase, it is assumed to be universally quantified; otherwise, it is assumed to be constant. Of course, if it is assumed to be a constant, its type should have been explicitly given. If it is a variable, the λProlog interpreter will attempt to determine its type.
- The underscore _ can also be used as an *anonymous variable*. All occurrences of an underscore denote different variables and their occurrences in program clauses are assumed to be universally quantified with outermost scope.
- The names of bound variables can be tokens with an initial letter that is either lower or upper case.

To illustrate these conventions, consider the following signature.

```
kind node type.
type adj  node -> node -> o.
type path node -> node -> o.
```

Given this convention, the clause

```
path X Y :- adj X Z, path Z Y.
```

is intended to mean the clause

```
pi X\ pi Y\ pi Z\ path X Y :- adj X Z, path Z Y.
```

Notice that program clauses are terminated by a fullstop. Using equivalences mentioned in Section 3.4 and the above conventions, this clause is equivalent to all of the following clauses.

```
pi X\(pi y\(pi z\(path X y :- adj X z, path z y))).
path X Y :- sigma Z\(adj X Z, path Z Y).
path X Y :- sigma z\(adj X z & path z Y).
path X Y :- path Z Y :- adj X Z.
adj X Z => path Z Y => path X Y.
```

```
adj X Z => pi Y\(path Z Y => path X Y).
(path Z Y => path X Y) :- adj X Z.
(pi Y\(path Z Y => path X Y)) :- adj X Z.
pi x\(pi y\(path x y :- sigma z\(adj x z, path z y))).
```

All occurrences of free and bound variables in these formulas can be inferred to have type `node`.

As was mentioned in Section 3.2, λProlog has two symbols for conjunction, namely `&` and the comma, and has one symbol for implies, namely `=>`, and one for implied-by `:-`. In this text, we shall exploit this redundancy by using following the following conventions: when forming the conjunction of two definite formulas, we use `&` and when forming the conjunction of two goal formulas, we use the comma; when a definite formula is an implication, we use `:-` and when a goal formula is an implication we use `=>`. (Goals related to Horn clauses do not allow implications: this possibility does not happen until Chapter 5.) Notice that several of the formulas above do not satisfy this convention: while they are still legal expressions, following this convention often makes reading examples simplier.

For some additional examples of Horn clauses, consider the following signature

```
kind bool           type.
type neg            bool -> bool.
type and, or, imp   bool -> bool -> bool.
type ident          bool -> bool -> o.
```

and following clauses.

```
ident (neg B)   (neg D)   :- ident B D.
ident (and B C) (and D E) :- ident B D, ident C E.
ident (or  B C) (or  D E) :- ident B D, ident C E.
ident (imp B C) (imp D E) :- ident B D, ident C E.
```

Notice that several of these clauses have common parts. Using Horn clauses formulated with definition (3.2), some of this redunancy can be factored as follows.

```
ident (neg B)   (neg D)   :- ident B D.
ident (and B C) (and D E) &
ident (or  B C) (or  D E) &
ident (imp B C) (imp D E) :- ident B D, ident C E.
```

Here, the binding of `&` is tighter than it is for `:-`. It is even possible to compress this presentation of clauses further to obtain the following logically equivalent clause.

```
ident (neg B)    (neg D)   &
(ident (and B C) (and D E) &
 ident (or  B C) (or  D E) &
 ident (imp B C) (imp D E) :- ident C E) :- ident B D.
```

Of course, compression like this may not always improve readability of the resulting code.

3.8 Read-prove-print loop

An implementation of λProlog will presumably provide programmers with various facilities for tracing and debugging computations, reading and saving program code, etc. It is beyond the scope of this text to attempt to describe many of these functions. Here, we focus on just the *read-prove-print loop*, a facility for interacting with proof search (as outlined in Section 3.3).

The search for a proof can only start once a sequent $\Sigma; \mathcal{P} \longrightarrow G$ is specified. Before entering the read-prove-print loop, we shall assume that the left-hand side of the sequent has been specified. The signature Σ is be composed of the ambient signature Σ_0 (Section 3.1) and declarations supplied by a programmer. The program $\mathcal{P}$ is composed of the ambient program $\mathcal{P}_0$, a set of clauses that define predicates that are always available in λProlog, plus program clauses provided by a programmer. The exact way that a programmer specifies signatures and program clauses uses the module system of λProlog and is described in Chapter 4. For our purposes now, we shall assume that the programmer has correctly described to the λProlog interpreter what modules, and hence what signatures and program clauses, should be used in our initial sequent. For example, assume that the following declarations and clauses are added to the ambient signature and program to yield Σ and $\mathcal{P}$.

```
type  append    list A -> list A -> list A -> o.

append nil L L.
append (X::L) K (X::M) :- append L K M.
```

Now that the signature Σ and program $\mathcal{P}$ has been determined, we enter a loop that requests a goal G, attempts to prove the sequent $\Sigma; \mathcal{P} \longrightarrow G$, prints out information regarding that search, and then repeats.

3.8.1 The read phase

The read phase will prompt the user of the interpreter for a goal using the `?-` prompt. The text of the goal ends with a fullstop (a period followed by white space). In Section 3.7, a convention was adopted so that not all universal quantifiers around a program clause needed to actually be written: tokens that are not explicitly quantified and which start with an initial capital letter are taken as implicitly universally quantified with outermost scope. When entering goal formulas, we allow a dual convention: when a goal is entered in the read phase, tokens that are not explicitly quantified and which start with an initial capital letter are taken as implicitly existentially quantified with outermost scope. For example, the follow three expressions will all be interpreted as the same goal formula.

```
?- sigma X\ sigma Y\ append X Y (1::nil).
?- sigma Y\ append X Y (1::nil).
?-append X Y (1::nil).
```

Similar also to program clauses, the underscore `_` can also be used as an *anonymous variable*. All occurrences of an underscore denote different variables and their occurrences in goal formulas are assumed to be existentially quantified with outermost scope.

3.8.2 The prove phase

This phase generally comprises the major effort of the interpreter: attempting to find a proof of a sequent by attempting to use the inference rules displayed in Figures 3.4 and 3.5. This phase is quite complicated and we have not provided many of the details needed to actually understand what happens during this phase. Some details will be presented in the next section (Section 3.9). For now, imagine that there is a simple search engine that attempts to build a proof. Given the nature of the rules in Figures 3.4 and 3.5, certain aspects of this phase are easy to describe. If the goal is a logical connective, then a right rule (Figure 3.4) is attempted: for example, if the goal is a conjunction, two separate proofs much be found, one for each conjunct. If the goal is atomic, then a backchain rule (Figure 3.5) is attempted.

Proof search can have three possible outcomes: the search might reveal that there are no proofs of the given sequent, it might find a proof, and it might not terminate. The third possibility exists necessarily because the problem of determining provability in logic, even in as small a subset of logic as first-order Horn clauses, is undecidable. However, the search strategy employed by λProlog is rather simple and incomplete: it will often not terminate even for subsets of *fohc* that are theoretically decidable.

3.8.3 The print phase

If a proof has been found or if it is known that no proof can be found, then we need to report this to the user who supplied the goal. In the event that no proof is found, the interpreter will simply print a `no`. For example,

```
?- append (1::nil) (2::nil) (3::nil).

no
?-
```

We can interpret this response as saying that it is not the case that the result of appending the list `(1::nil)` to the list `(2::nil)` is the list `(3::nil)`. If a proof is found, the interpreter can simply respond with `solved`. For example,

```
?- append (1::nil) (2::nil) (1::2::nil).

solved
?-
```

We can imagine, however, getting more information from a successful proof than the fact that it was successful.

If a proof is found, that proof could be returned at the result of the proof search. Such proofs are, however, essentially complete traces of computations and as such contain far too much information to be useful. A similar situation exists with Turing Machines: when such a machine halts, its entire computation trace could be seen as the result of its computation. Since the entire trace is generally considered too much information, a convention is used instead: when the machine halts, the result of a computation is the string that is left on the tape. A similar convention is used here as well. When a proof is discovered, only terms used to instantiate the implicitly existential quantifiers (via the $\exists$ right introduction rule) at the root of the proof are reported as the result of a computation. Terms used to instantiate quantifiers assocated to anonymous (underscore) variables are not printed.

Consider the following dialog with a λProlog interpreter.

```
?- sigma X\ append (1::nil) (2::nil) X.

solved
?- append (1::nil) (2::nil) X.

X = (1::2::nil)

?- append (1::nil) (2::nil) _.

solved
?-
```

In the first query, there were no implicitly bound variables (`X` there is explicitly bound) while in the second query `X` is implicitly bound. In the third, an anonymous variable is used. These three formulas denote the same goal formula: the only difference is that the second and third have a variable that is bound implicitly. Thus, while these goals have proofs, a term is printed only for the second of these two goals. Similarly, consider the following queries.

```
?- sigma X\ sigma Y\ append X Y (1::nil).

solved
?- sigma Y\ append X Y (1::nil).

X = nil

?- append X Y (1::nil).

X = nil
Y = (1::nil)

?-
```

All three queries result in the same sequent being attempted while all three have different implicitly bound variables.

The listing of which terms are substituted for which implicitly quantified variable is also called an *answer substitution*, and we often think of an answer substitution as the result of proof search.

3.8.4 Multiple proofs

There might, of course, be many different proofs of a sequent and these different proofs might have associated with them different answer substitutions. When an answer substitution is presented, as in the dialogs above, the interpreter will pause for input from the user. The user can then enter either a carriage return (as in the example dialogs above) to signify that no additional proofs are desired, or the user can enter a semicolon `;` to request a search for another proof. The following dialog illustrates this use of semicolon. In this case, additional proofs provided additional answer substitutions.

```
?- append L K (1::2::nil).
L = nil
K = 1::2::nil;

L = 1::nil
K = 2::nil;

L = 1::2::nil
K = nil;
no
?-
```

A total of three proofs have been found and the corresponding answer substitutions have been printed. The final `no` is indicates that no additional proofs was found in response to the third use of the semicolon.

3.9 Operational aspects of proof search

Since computation in λProlog is based on proof search, the actual steps that are taken in searching for a proof must be understandable and predicable to a programmer of λProlog, at least to a some level of detail. In this section, we supply some of these details. Unlike automatic theorem provers where rich and sophisticated methods are often used to search for proofs, λProlog employs a simple and rigid search strategy. Using a simple search strategy has several implications. First, proof search will be incomplete: there will be many sequents that have proofs that will not be found by λProlog. An implementation will loop indefinitely where another more sophisticated search strategy would find a proof. Since the search strategy is known and fixed, however, it will often be possible to write programs in such ways as to avoid such incomplete behaviors. Second, since a search strategy is the vehicle that carries a logic program into an actual series of computation steps, a simple strategy means that a programmer can predict to a large degree the computational researchs (time and space, for example) that a logical specifications will consume. This latter aspect of the interpreter is, of course, particularly important.

When attempting to prove a sequent using the right- and left-introduction rules in Figures 3.4 and 3.5 there are several decisions that need to be made.

When the goal is a disjunction or the formula we are backchaining over is conjunction, there are two rules that can be applied. The interpreter will always attempt to use the rule that involves the left subformula before the right subformula.

When the goal is a conjunction or the formula we are backchaining over is an implication, then there are two subproofs produced. The order in which the premises are attempted is from left-to-right in the order that the premises are listed in the inference rules in Figures 3.4 and 3.5. Notice that this means that the operational reading of the clause $G_2 \supset G_1 \supset A$ is the same as that for the (logically equivalent clause) $G_1 \wedge G_2 \supset A$: that is, there is a twist in the way that goal formulas are written.. This is easy to understand if we examine the following two proof fragments.

$$\dfrac{\dfrac{\overline{\Sigma; \mathcal{P} \xrightarrow{A} A} \quad \Sigma; \mathcal{P} \longrightarrow G_1}{\Sigma; \mathcal{P} \xrightarrow{G_1 \supset A} A} \qquad \Sigma; \mathcal{P} \longrightarrow G_2}{\Sigma; \mathcal{P} \xrightarrow{G_2 \supset G_1 \supset A} A} \qquad \dfrac{\overline{\Sigma; \mathcal{P} \xrightarrow{A} A} \quad \dfrac{\Sigma; \mathcal{P} \longrightarrow G_1 \quad \Sigma; \mathcal{P} \longrightarrow G_2}{\Sigma; \mathcal{P} \longrightarrow G_1 \wedge G_2}}{\Sigma; \mathcal{P} \xrightarrow{G_1 \wedge G_2 \supset A} A}$$

If the first clause was written using the reverse implication, namely as $A \subset G_1 \subset G_2$, then it would be operationally equivalent to $A \subset G_1 \wedge G_2$: no twist is present.

When selecting the formula D from $\mathcal{P}$ in the decide rule (Figure 3.5), the order of selection is important. For this reason, we must actually think of the program $\mathcal{P}$ as being a list instead of a set: the order and multiplicity of formulas in $\mathcal{P}$ is important in the way proofs are attempted. The search strategy will also attempt to select formulas from the list in the order in which they occur in left-to-right fashion. This list order will derive from the order of that clauses are presented in text file versions of programs (see Chapter 4).

Finally, the remaining search strategy issue involves determining what term t to instantiate the existential quantifier in ∃R (Figure 3.4) or in the universal quantifier in ∀L (Figure 3.5). Each of these inference rules are conjunctive in the sense that they have two premises, one of which is $\Sigma; \Vdash_f t : \tau$. If there are a finite number of terms of type τ, then replacing t with each member of that type might be a suitable way to attempt these rules. Of course, there will generally be an infinite number of Σ-terms of type τ, so a more general mechanism for handling these inference rules is needed. The more general mechanism that is used is based on notions of *logic variables* and *unification*.

3.9.1 Logic variables

** To be written.

$$\dfrac{\Sigma; \Vdash_f X : \tau \qquad \Sigma; \mathcal{P} \longrightarrow B[X/x]}{\Sigma; \mathcal{P} \longrightarrow \exists_\tau x.B}\ \exists R \qquad \dfrac{\Sigma; \mathcal{P} \xrightarrow{D[X/x]} A \qquad \Sigma; \Vdash_f X : \tau}{\Sigma; \mathcal{P} \xrightarrow{\forall_\tau x.D} A}\ \forall L$$

3.9.2 Unification

** To be written.

3.10 The Uses and Roles for Types

λProlog is a strongly typed programming language in the style of the Standard ML programming language [MTH90].

3.10.1 Types denote syntactic expressions

In λProlog, types are used to denote various classes of *syntactic expressions* and two elements of a type are equal if they are the same syntactic expression. (When we introduce λ-terms, this notion of "same syntactic expression" will grow to include λ-conversion.) For example, the expressions `2 + 3`, `3 + 2`, and `5` all have type `int`, but they are different elements of that type since they are different expressions. This is in contrast to the standard use of types in functional programming: there the three expressions `2 + 3`, `3 + 2`, and `5` all have type `int` and are also equal in that type since they have the same expression. Thus, in λProlog, the goal formula fails while in ML the expression `(2 + 3 = 5)` evaluates to `true`. Similarly, the type `o` denotes the set of formulas and not some set of truth values. As we shall see later, treating members of types as expression makes it possible for λProlog to compute on expressions of functional type in ways that are not possible in functional programming: testing equality at function type means checking equality of the "code" and not the (possibly infinite) graph of the function (that is, the "value" of the function). Computation in functional programming is the rewriting of an expression until its value is uncovered. In λProlog, there is no rewriting phase: expressions of a given type are the intended members of that type. Computation in λProlog is not based on rewriting by on the search for proofs.

Of course, it is useful for λProlog to know that the expressions `2 + 3`, `3 + 2`, and `5` all denote the same mathematical value (the number 5) and it is for this reason that λProlog is equipped with a simple evaluator (described in Section 4.4) that is able to carry the expression `2 + 3` to `5`. While such evaluation and rewriting can be accommodated in logic programming, they are not part of the logical foundation of λProlog.

3.10.2 Polymorphic typing

Type expressions can have type variables and this allows for a degree of *polymorphic typing*. One way to understand type variables in a type declaration is to think of the keyword `type` as a predicate relating a token and a type (ignoring the problem of how we might type the predicate `type`). Thus, the declaration for `nil` in Section 3.1 can be viewed as the formula

$$\forall \mathtt{A}\ (\mathtt{type\ nil\ (list\ A)})$$

instead of the clause

$$\mathtt{type\ nil}\ (\forall \mathtt{A}\ (\mathtt{list\ A})).$$

The first interpretation of a polymorphic type declaration is closest to that used here. Thus, `nil` has many types, although they are all substitution instances of a common expression. Similar, a predicate, like `append` can be declared with a type containing a type variable; for example,

```
type   append   list A -> list A -> list A -> o.
```

Thus, `append` can be applied to three lists but the type of the elements of those three lists is not fixed: hence, `append` is an example of a predicate that can be applied to "many structures", hence, the name *polymorphic typing*.

Consider a type expression of order 1 of the form $\tau_1 \rightarrow \cdots \tau_n \rightarrow \tau_0$. A type variable that is free in τ_0 is called a *transparent type variable* for that expression. If all type variables in a type are transparent, that expression is *determinate* . Knowing the target type of such a determinate type allows the argument types to be determined uniquely. For example, the types in the declarations

```
type ::    A -> list A -> list A.
type nil   list A.
type pr     A -> B -> pair A B.
```

are all determinate while the type for `cons` in

```
kind lst    type.
type null   lst.
type cons    A -> lst -> lst.
```

is not transparent. Here, a term of type `lst` represent lists of possibly heterogeneous types. (Type declarations such as the one for `lst` are not available in, say, Standard ML.)

3.10.3 Type checking and type inference

Type checking is a process that determines if a given term or formula can be built correctly using the typed constants that have been declared. This process is essentially one of determining that the typing judgments described in Figures 3.1 and 3.3 can be established. It is a common observation that type checking is useful for detecting statically many errors that are only caught at runtime in languages without strong typing disciplines. A program that successfully passes the type checking phase is likely to have all the arguments to predicates present and in the right order and to have the correct spelling for constant symbols. Terms that cannot be given a type are not admitted in either program clauses or queries. For example, a parser should report that the term `1::(2::nil)::nil` cannot be typed and it should refuse to either interpret or compile a program or query in which it is embedded.

λProlog is a strongly typed language in the sense that all constants and variables must be given a type. In this text, we assume that the programmer must supply the type of all constants that are not built into λProlog. A λProlog parser is responsible for supplying types to all (explicitly or implicitly) bound variables: the types for these are easily determined from context since they can have exactly one type at all their occurrences within a given scope, even if that type contains type variables. For example, in the formula

```
append (1::nil) (2::nil) X, append ("abc"::nil) ("efg"::nil) Y
```

the constant `append` (given the types declared earlier) can appear at two different types, namely, `list int -> list int -> list int -> o` and `list string -> list string -> list string -> o`. The variables `X` and `Y` are inferred to have the types `list int` and `list string`, respectively. If the variable `Y` was, however, replaced with `X` then this expression would not be typeable: it is not possible for two occurrences of the variable `X` to have two different types. Thus λProlog does more than just check types, it also infers types for such variables.

It is possible to supply the type of any occurrence of a subterm within a formula or term using a colon. For example,

```
append X X Y:(list int)
append X:(list int) X Y
append:(list int -> list A -> list int -> o) X X Y
(append X):(list int -> list int -> o) X Y
```

all yield the same typing for the constant and two variables in these expressions. This is in contrast to the types given the simple expression `(append X X Y)`: here, `X` and `Y` are both given the type `(list A)`. When a type is explicitly provided using a colon, the actual type attributed to that occurrence is some instance of that type. There is seldom a need to use a colon in this fashion: it is used mostly to specify ad hoc polymorphism, as is illustrated in Section 3.10.4.

```
kind lst           type.
type null          lst.
type cons          A -> lst -> lst.
type separate      lst -> list int -> list real -> o.

separate (cons X:int  L) (X::K) M :- separate L K M.
separate (cons X:real L) K (X::M) :- separate L K M.
separate null nil nil.
```

Figure 3.6: Heterogeneous lists.

```
kind numb          type.
type inj_int       int  -> numb.
type inj_real      real -> numb.
type separate      list numb -> list int -> list real -> o.

separate ((inj_int  X)::L) (X::K) M :- separate L K M.
separate ((inj_real X)::L) K (X::M) :- separate L K M.
separate nil nil nil.
```

Figure 3.7: Lists contains only integers and reals.

Type inference could be extended to the inference of types for constants as well. Although we shall not consider such inference here, such type inference was implemented in an early versions of λProlog[**?**, EP89] and is discussed in [NP92]. Other references for similar typing can be found in [Han89, **?**].

3.10.4 Runtime behavior of types

Types play a role in the runtime behavior of programs (another difference with types in Standard ML). Consider, for example, the declarations and clauses in Figure 3.6 that declare constructors for building heterogeneous lists. As we mentioned before, the type of `cons` is not determinate. The predicate `separate` can be used to separate a heterogeneous list containing only integers and reals into two homogeneous lists, one containing only integers and one containing only reals. Here, explicit typing using a colon is required: the type attributed to the first element of the list in the first argument of `separate` (the variable `X`) determines which of the first two clauses of `separate`'s definition should be selected to process that first element. In the search for a proof, type information is needed to determine which clause of `separate` is used.

Figure 3.7 contains another specification of the `separate` predicate with a related functionality. Here, the runtime determination of which clauses to select for processing the first element of the list in the first argument of `separate` is provided by examining terms and not types: if the first item of the list in the first argument has the top-level function symbol `inj_int` (injection into `int`) then the first clause is selected; otherwise, if the top-level function symbol `inj_real` then the second clause is selected.

For an example of using these two implementations, if the goal

```
separate (cons 1.0 (cons 2 (cons 3.0 null))) L K
```

is attempted using the specification in Figure 3.6 and the goal

```
separate ((inj_real 1.0)::(inj_int  2)::(inj_real 3.0)::nil) L K
```

is attempted using the specification in Figure 3.7, both queries will bind `K` to `(2::nil)` and `L` to `(1.0::3.0::nil)`.

There are at least two reasons for preferring implementations that exclusively use data structures for which all constructors are declared to have determinate types. These reasons argue, for example, that the second implementation of `separate` above is to be preferred over the first.

Better static analysis is possible. When data constructors have determinate types, the type of subexpressions place constraints on the type of larger expressions. For example, in the homogeneous list structure, if one element of the list is determined to be of a particular type, all elements of that list must also be of that type. In these cases, the static properties of type checking provide much more information about a program during the type checking phase. In the example involving `separate` predicates, for example, we know that the second `separate` specification will work only for lists of numbers (that is, lists of intergers and reals) whereas the first specification allows for any kind of items to appear in lists.

Type information is often not needed during proof search. In many cases when only determinate types are used for data structures, it is possible to determine that types are, in fact, not needed during the execution of a logic program. Such a fact means that execution of such logic programs can possibly be made more efficient. Of course, type information is sometimes moved from types into terms, as is the case with the `inj_int` and `inj_real` constants in Figure 3.7.

One model for understanding the nature of polymorphic types in λProlog is to think of constants and variables as being record structures that contain at least their name and their type. When checking the equality of two constants or variables, not only do their names have to be equal, their types must be equal. In an implementation using unification, term unification will then cause type unification and the instantiation of type variables. One optimization of this is to store in that record not the type but the non-transparent type variables in that type [**?**].

Given the flexibility of λProlog's type system, it is possible to trivialize the typing system. For example, it is possible to give constructors such as `cons` and `null` the type `A`, that is, assert that both of these constants have every possible type. It would then be possible to build arbitrary term structures from these constants, such as `(null (cons cons) cons)`, most of which have little to do with list structures. Of course, the use of such type declarations is not helpful to either programmer or program analysis.

3.11 Prolog and λProlog

While Prolog and the fragment of λProlog described so far are implementations of first-order Horn clause, there are many differences between these languages.

For example, there are syntactic differences. For example, Prolog uses `[]` and `[X|L]` to denote `nil` and `(X::L)`. Lists in Prolog can also be written using syntax such as `[1,2,3]`:

in λProlog this would be written as as `(1::2::3::nil)` (the LP2.7 and Prolog/Mali implementations of λProlog had Prolog-style list syntax). Also, formulas and term syntax is written in a *curried*-style, similar to that used by other languages built on higher-order types, such as the ML programming language. The symbol `append` denotes a predicate that takes three arguments and then becomes an atomic formula. The use of curried syntax makes it possible for these arguments to be applied one-by-one to yield meaningful structure: thus, `(append (1::nil))` denotes a predicate that must take 2 arguments before it becomes an atomic formula. In the higher-order setting described in later chapters, this style of syntax will prove natural and useful. As a result of these two differences, the comma, which serves three roles within Prolog formulas — list separator, argument separator, and conjunction — is used only for conjunction in λProlog.

More serious differences arise because of the typing discipline of λProlog. Some Prolog programs will not be allowed in λProlog because they are not type correct for λProlog. There are also many builtin features to Prolog that are not incorporated into λProlog. Some Prolog builtins, such as `=..` would not easily fit into a typed language and others, such as `var`, `assert`, and `retract`, would need to be significantly modified in order to be incorporated into λProlog (see, for example, [CM84] and [SS86] for a description of these predicates).

Chapter 4

Modules and Built-in Values

This chapter describes two aspects of how a λProlog implementation interacts with the larger world of a computer system in which it is situated. Many of the components of such systems — the operating system, file system, and the machine's built-in values — are not embraced by the logic underlying λProlog. Since the exact dependence of an interpreter on its environment depended to a large extent on that environment, what are described here are only those features that should be expected from any implementation of λProlog.

4.1 A desiderata for modular programming

Many modern programming languages are based on declarative, formal systems. In their early stages of development, such languages generally focus on programming-in-the-small. As they mature, problems with programming-in-the-large became more important and at that point, a second language is often imposed over the initial language. To address the problem of building large programs, parsing and compiler directives, such as `use`, `import`, `include`, and `local`, are added. This imposed language generally has little connection with the original declarative foundation of the initial language: it was born out of the necessity to build large programs and its function was expediency. The meaning of the resulting hybrid language is often complex and the declarative purity of the underlying language seldom extends to the full language; the hybrid language might even interfer with the behaviour of the core language.

When designing a module system for logic programming, we should ask more than that it separates code elements and can be efficiently implemented. For example, it is desirable that it satisfies several high-level principles, such as those listed below.

- Constructs for programming-in-the-large should not complicate the meaning of the underlying, declarative core language.
- Modules should support transitions from high-level program specification to lower-level program implementations.
- Modular programming should work smoothly with higher-order programming. For example, a particular challenge in Prolog is getting the `call/1` predicate to interact correctly with modules.

- Rich forms of abstraction, hiding, and parametrization should be possible.
- Modules should allow a rich calculus of transformations. These should include partial evaluation, fold/unfold, and even compilation.
- Important aspects of a module's meaning should be available and verified without examining the module in detail. Notions of interfaces often support this property.
- The additional syntax for programming-in-the-large should be readable, natural, and support separate compilation and re-usability.
- There should be a non-trivial notion of the equivalence of modules that would guarantee that a module can be replaced by an equivalent module with little to no impact on the behavior of a larger program. This property is sometimes called *representation independence*.

The module system of λProlog addresses all of these principles with varying degrees of success.

One approach to developing a principled modular programming language is to reduce programming-in-the-large to programming-in-the-small: in the logic programming setting, this could mean that modular programming can be explained completely in terms of the logical connectives of the underlying language. Thus, a collection of modules would be mapped to a (possibly large) collection of (possibly large) formulas. Furthermore, we would like the combinators for building modules to correspond closely to logical connectives.

4.2 Concrete syntax of modules

The text files containing λProlog code contains one or more modules. As text, modules can contain three parts: a one-line header, a preamble, and a collection of declarations and clauses. Taken together, the declarations comprise a signatures, and, therefore, they may contain the four keywords `kind`, `type`, `infixl`, and `infixr`. The concrete syntax of modules will introduce a total of 5 new keywords. Of these we introduce only two in this chapter; the other three are introduced in Chapter 5.

Along with these, modules may also contain comments. There are two ways to enter comments in a module. The percent sign `%` can be used as it is used in Prolog; that is, if a `%` is found, then all characters from that character to the end of the line are ignored. The symbols `%(` and `)%` can also be used to enclose comments.

The first line of the text for a module, the module's *header*, is of the form

```
module moduleName.
```

The argument of the `module` keyword is the name of the module being defined, in this case, the token `moduleName`. A module starts with the keyword `module` and ends with the end of the file or when another module begins.

The *preamble* to a module contains just two kinds of entries, namely those that declare that other modules can be either *accumulated* or *imported*, using the `accumulate` and `import` keywords. The `import` keyword will be described in Chapter 5. The `accumulate` keyword is used to incorporate other modules as if those other modules were actually typed at the beginning of the current module. If this keyword appears in a module, it must follow immediately after the `module` keyword. The line containing this keyword has the form

```
accumulate mod1, mod2, mod3.
```

That is, `accumulate` is followed by one or more module names, separated by whitespace. A module's preamble can be empty; that is, it may not contain either the `accumulate` or `import` keywords.

The remaining lines of a module are combinations of signature declarations, introduced by the `kind`, `type`, `infixl`, and `infixr` keywords. Lines that start with no keywords are program clauses. Signature declarations are intended to have global scope within a module, and, for readability, it is possible to interleave signature declarations and programming clauses. The only restriction on such interleaving is that if a token is given a kind, type, or infix declaration, those declarations must appear prior to its first occurrence in a program clause. It is always possible to move signature declarations to the top of a module, immediately following the preamble.

Figure 4.1 lists one text file containing the specification of the two modules `smlists` and `smpairs`. The preamble for `smlists` is empty and the preample for `smpairs` contains just one line used to accumulate `smlists`. Notice also that in `smlists` all declarations are at the top of the module and in `smpairs` they are mingled with the clauses.

4.3 Static semantics of modules

The meaning of modules is divided into two parts. The *static semantics* of a module describes what collection of constants and program clauses the module denotes. The *dynamic semantics* of a collection of modules describes how the constants and clauses in those modules are used during the search for proofs of goal formulas. The static semantics of modules is presented in this section: the remaining sections of this Chapter deal with their dynamic semantics.

The process of converting the concrete syntax of modules into a signature-program pairs is called *module elaboration*.

An important difference in the treatment of signatures and program clauses is that tokens within a signature are associated with a kind or type in a *functional* fashion. For example, a constant is given at most one type. Program clauses have no such restriction. Thus, when two signatures are brought together, it is important that all tokens given declarations in both signatures must, in fact, have the same declarations. In the case of bringing together two lists of program clauses, no such check is required: a given predicate can have clauses for various modules.

Two signatures can be *merged* if the following conditions are true.

- If a token has a kind declaration in both signatures, then those declarations must be identical.

- If a token has a type declaration in both signatures, then those declarations must be equal. Here, two type expressions are considered equal if they are differ only up to alphabetic changes to the names of type variables.

- If a token has an infix declaration in one signature and some declaration in the other signature, then that token must have exactly the same infix declarations in both signatures.

```
% This file contains two modules.
module smlists.

kind list     type -> type.

type id                list A -> list A -> o.
type memb, member      A -> list A -> o.
type append            list A -> list A -> list A -> o.
type member_and_rest   A -> list A -> list A -> o.

id nil nil.
id (X::L) (X::K) :- id L K.

memb X (X::L).
memb X (Y::L) :- memb X L.

member X (X::L) :- !
member X (Y::L) :- member X L.

append nil K K.
append (X::L) K (X::M) :- append L K M.

member_and_rest X (X::L) L.
member_and_rest X (Y::K) (Y::L) :- member_and_rest X K L.

module smpairs.

%( The module smlists is accumulated here because
   we use the memb and member predicates below.    )%
accumulate smlists.

kind pair    type -> type -> type.
type pr      A -> B -> pair A B.

type assoc, assod   A -> B -> list (pair A B) -> o.

assoc X Y L :- memb (pr X Y) L.
assod X Y L :- member (pr X Y) L.

type domain         list (pair A B) -> list A -> o.

domain nil nil.
domain ((pr X Y)::Alist) (X::L) :- domain Alist L.

type range          list (pair A B) -> list B -> o.

range nil nil.
range ((pr X Y)::Alist) (Y::L)  :- range Alist L.
```

Figure 4.1: A text file containing two modules.

```
module mod1.
kind item      type.
type p,q       item -> o.
p X :- q X.

module mod2.
accumulate mod1.
type a         item.
q a.

module mod3.
kind item      type.
type p,q       item -> o.
type a         item.
p X :- q X.
q a.
```

Figure 4.2: The modules `mod2` and `mod3` elaborate to the same signature-program pair.

Thus, when merging two modules, it is not possible for a token to have a kind or type declaration in both modules but an infix declaraion in only one signature. It is also not possible for a type to be given a polymorphic type in one module and an instance of that polymorphic type in another module. The result of merging two signatures is the signature that results from providing declarations for all tokens that appear in both signatures.

If a module does not contain any `accumulate` keyword, then the signature and list of program clauses associated with that module are exactly those items enumerated within the module, listed in the same order as they appear in the module's text file. If the module contains the `accumulate` keyword, all of the named modules must have been elaborated previously so that their associated signature-program pairs are known. Let $\langle\Sigma_1, \mathcal{P}_1\rangle, \ldots, \langle\Sigma_n, \mathcal{P}_n\rangle$ be the signature-program pairs associated with all the modules provided by the `accumulate` keyword and let Σ_0 be the signature that is declared in the module. It is an error if the signatures $\Sigma_0, \ldots, \Sigma_n$ cannot be pairwise merged. The signature associated with the module is then the result of merging all the signatures $\Sigma_0, \ldots, \Sigma_n$. The list of clauses associated with the module is the result of appending the lists $\mathcal{P}_1, \ldots, \mathcal{P}_n, \mathcal{P}_0$, in that order: here, $\mathcal{P}_0$ is the list of clauses explicitly written in the module.

Figure 4.2 shows a text file containing three simple modules. After elaboration, modules `mod2` and `mod3` would yield indistinguishable signature-program pairs.

When elaborating a module, the initial context is needed to determine the types of logical and built-in types and constants. Also, when a module is used in a computation, it will be merged with a context that contains the initial context. Thus, it is also an error of module elaboration if the given module declares symbols in such a way that they cannot later be merged with the initial context.

4.4 Built-ins Values

Several types are built into the λProlog interpreter. The types for formulas and for lists is described in Chapter 3. There are a few additional builtin types listed here and describe more in the remainder of this section.

Numbers. Both integers and reals and elementary functions on them. Both of these are mapped directly to the underlying representation of numbers provided by a computer processor.

String. Sequences of characters, including the empty sequence, with elementary string manipulation functions. There is no separate datatype for characters.

Streams. When a file is opened for input or output, a logical handle for that stream is produced. Reading and writing can be directed on these channels.

Exceptions. When a error or exception situation is encountered, it should be possible to abort and redirect a computation. Exceptions are a nonlogical feature of λProlog that will allow us to do this kind of control. Generally we limit their use to interacting with users, file processing, and errors from arithmetical operations.

4.4.1 Integers and Reals

The syntax and semantics numbers is similiar to that adopted by Standard ML [MTH90]. Thus, negative integers and real numbers use `~` for the negative sign — `~5` is "minus 5" and `~5.4` is "minus 5.4". One major difference with Standard ML is that *ad hoc* polymorphism is advoided with the built-in operations.

The intended meaning for all of these constants in Figure 4.3 should clear from their name and type, except possibily for following: `~` and `r~` denote the unary minus signs for integer and real numbers; `div` is the quotient of x and y rounded down; and `quot` is the quotient rounded towards zero.

4.4.2 Strings

Strings are delimited using double quotations. The exact syntax of strings, including escape sequences and formating characters, is taken from the Standard ML definition. Figure 4.4 contains the built-in predicates and functions available in λProlog.

The circumflex `^` denotes string concatenation, `size` computes the length of a string, `chr` returns a singleton string whose ascii value is its argument, `ord` returns the ascii value of the first character in its argument, and `substring`, when applied to a string and the interger arguments i and j returns the string starting at position i in the string (indexed starts at 0), and ends at position $i+j$. The four displayed predicates compare strings using lexicographic ordering.

4.4.3 Files and Streams

The builtin constants and predicates for streams and files are presented in Figure 4.5. Files can be open for input and output. To open them, a string is used to name them. In general, this name is either the name of a file in the current directory or it is a full pathname. When a file is open, it is associated to a channel and it on such channels that input or output is

```
kind   int, real                       type.

type   ~                               int -> int.
type   +, -, *, div, quot, mod         int -> int -> int.
type   int_to_string                   int -> string.
type   <, >, =<, >=                    int -> int -> o.

infixr <, >, =<, >=                    4.
infixl +, -                            6.
infixl *, div, mod                     7.

type   r+, r-, r*, r/                   real -> real -> real.
type   r~, sqrt, sin, cos, arctan, ln  real -> real.
type   floor, ceiling, truncate         real -> int.
type   int_to_real                      int -> real.
type   real_to_string                   real -> string.
type   r<, r>, r>=, r=<                 real -> real -> o.

infix  r<, r>, r>=, r=<                 4.
infixl r+, r-                           6.
infixl r*, r/                           7.
```

Figure 4.3: Built-in functions and predicates for integers and real numbers.

```
kind    string              type.

type   ^                    string -> string -> string.
type   size, ord            string -> int.
type   chr                  int -> string.
type   substring            string -> int -> int -> string.
type   s>, s<, s>=, s=<     string -> string -> o.

infixl ^                    6.
infix  s>, s<, s>=, s=<     4.
```

Figure 4.4: Built-in functions and predicates for strings.

```
kind in_stream, out_stream   type.

type std_in                  in_stream.
type std_out, std_err        out_stream.
type open_string             string -> in_stream -> o.
type input                   in_stream  -> int -> string -> o.
type output                  out_stream -> string -> o.
type input_line, lookahead   in_stream  -> string -> o.
type eof                     in_stream  -> o.
type flush                   out_stream -> o.

type open_in                 string -> in_stream  -> o.
type open_out, open_append   string -> out_stream -> o.
type close_in                in_stream  -> o.
type close_out               out_stream -> o.
```

Figure 4.5: Built-in functions and predicates for files and streams.

directed. The `open_in` predicate opens a file for input and associates with it an `in_stream`. This `in_stream` can be closed using the `close_in` predicate. There are two predicates for opening a file for output. The `open_out` and `open_append` both open the file and associate it with an `out_stream`. In both cases, if the file does not already exist, it will be created. If the file does exist, the `open_out` predicate will cause its contexts to be overwritten while with the `open_append` predicate, its contents will kept and new output will be appended to its end.

It is also possible to convert a string into an `in_stream` using the `open_string` predicate. Doing input from a stream assocated to a string allows the contents of the string to be read as if it were a file.

There is one builtin `in_stream`, named `std_in`, which is generally attached to the keyboard, and two builtin `out_stream`s, named `std_out` and `std_err`, which are also generally attached to the keyboard: the first is for outputing to the user of the λProlog system and the second for printing error messages. These names and their semantics are derived from Unix conventions.

The `output` predicated is used to output a string on an `out_stream`. On systems that used buffered output, the predicate `flush` can be used to flush out any buffered output; on other systems, this predicate has no affect.

There are two ways to read a string from an `in_stream`: the goal `input InStream N String` will read at most `N` characters from the `in_stream InStream`, blocking until there are `N` characters available or until an end-of-file is reached: the resulting string will be unified with `String`. The goal `input_line InStream String` will blocks until a newline or end-of-file has been read from `InStream` and then unifies with `String` the string composed of all characters up to and including that first newline or end-of-file. The goal `lookahead InStream String` unifies with `String` the next character from `InStream` without removing it from the stream or the empty string if at the end-of-file has been encountered. The goal `eof InStream` can be used to check if `InStream` is at an end-of-file.

```
kind exn                              type.
type exception                        exn -> o.
type handle                           exn -> o -> o -> o.

type divide_by_zero, overflow         exn.
type index_out_of_bounds, not_chr     exn.
type file_not_found                   string -> exn.
type file_unreadable                  string -> exn.
```

Figure 4.6: Built-in constants and predicates for exceptions.

4.4.4 Exceptions

In order to handle a certain class of errors and exceptional situations, we shall consider version of λProlog that contains a mechanism for raising and handling exceptions. This feature is not a logically motivated one and its exact operational semantics is probably quite difficult to understand with respect to the rest of the language. None-the-less, we shall make some use of the predicates and constants in Figure 4.6. The type `exn` is used for naming exceptions. The goal `exception E` will cause the current computational context to be aborted and to have the exception "thrown" to some handler "above" it. Exception handlers are installed using a goal of the form `handle F G H`. The operational semantics of this is intended to be the follow: attempt to prove the goal `G`. If its execution does not raise an exception, then the entire `handle` goal should behave as `G`, whether it succeeds or fail. If, however, the execution of `G` results in an exception being raised, say `E`, then check if `E` and `F` unify. If they do, then call the goal `F`; otherwise, the exception `E` will be raised again for other possible handers. If no handler is available, the top-level interpreter loop should be entered with a suitable message being displayed.

In Figure 4.6, various builtin exceptions are listed. These exceptions are raised under the following situations.

- `divide_by_zero`: when division by zero is attempted using either integer or real division.
- `overflow`: when an overflow is detected in either integer or real arithmetic.
- `index_out_of_bounds`: when `ord` is given an empty string or `substring` is asked to compute on a part of a string that is not present.
- `not_chr`: when `chr` is given an integer that is not in the range between 0 and 255, inclusively.
- `file_not_found`: when `open_in` is given the name of a file that cannot be found. The argument of the exception is the name given for the file.
- `file_unreadable`: when the file that `open_in` is attempting to open cannot be opened, because, for example, file protections restrict access. The argument of the exception is the name given for the file.

The program in Figure 4.7 contains an illustration of exceptions.

4.4.5 Evaluation of built-in functions

In the description of built-in operations, the read may have noticed that some were presented as functions and some as predicates. For example, the length of a string is computed using a function `size` of type `string -> int` instead as with a predicate of type `string -> int -> o`, which is the usual vehicle in which computation is done in logic programming. While the choice between these two presentations of `size` is largely *ad hoc*, the choosing to the basic operations on integers, such as addition, as functions instead of predicates is largely forced on us by programming convention. This convention and the desire to make direct use of machine-supplied integers, reals, and strings, for example, leads us to to add the following non-logical predicate to λProlog.

```
type  is   A -> A -> o.
infix is   4.
```

The intention of a goal of the form `X is Exp`, where `Exp` is some expression denoting a value and `X` is some value or variable, is that `Exp` will be computed in a *functional* fashion and its resulting value with be unified with `X`.

In order to more carefully describe the meaning of the is predicates, we need to make distinctions between the various kinds of terms that can exist in a runtime system implementing λProlog. A term can be *open*, that is, it can contain occurrences of logic variables. A term that contains no logic variables will be called *closed*. The set of closed terms of type `int`, for example, can further be classified as those that are *genuine integers*, that is, one of the numbers

```
..., ~2, ~1, 0, 1, 2, ...
```

and those that are *integer expressions*, that is, a term built out of genuine integers and the functions symbols in Figure 4.3. These include, for example, the following terms

```
4 * 5         ~4 + 5 * ~1         4 * 5 mod 3.
```

Of course, these integer expressions all denote genuine integers, namely `20`, `~9`, and `2`. To compute the correspondence between integer expressions and genuine integers, a predicate external to the logical foundation of λProlog is used: this predicate is the `is` predicate.

The goal `X is Exp` has the following semantics: if `Exp` is an open term, this goal produces a runtime error. Otherwise, `Exp` is a closed expression, the value of which is computed and then unified with `X`. Thus, an attempt to prove the goal `(N is 2 * 7, M is N + N)` will succeed and bind `N` to 14 and `M` to 28; the goal `(15 is ~3 * ~5)` will succeed; the goal `(14 is 3 * 5)` will fail; and the goal `(P is Q + 1)` will fail with an error reporting that the second argument of `is` contains an unbound variable.

The `is` predicate will work as an evaluator at types `int`, `real`, and `string`: that is, the polymorphic typing given to it is really a kind of *ad hoc* polymorphism.

4.4.6 A simple program using various builtin operations

The module `readints` in Figure 4.7 illustrates a simple program that will read integers from one file and print into another file the result of adding up those integers.

```
module readints.

type string_to_int, digit      string -> int -> o.
type aux                       string -> int -> int -> o.
type bad_int_exp               exn.

digit Str N :-
  N is (ord Str) - 48, (0 =< N, N =< 9, !; exception bad_int_exp).

string_to_int "" _   :- exception bad_int_exp.
string_to_int Str N  :- Size is (size Str), aux Str N Size.

aux Str N 1    :- !, digit Str N.
aux Str N Size :-
  Size' is Size - 1, Digit is (substring Str Size' 1),
  digit Digit D, Str' is  (substring Str 0 Size'),
  aux Str' N' Size', N is 10 * N' + D.

type add_numbers      string -> o.
type read_and_acc     in_stream -> int -> int -> o.

add_numbers File :-
  InFile is File ^ ".list", OutFile is File ^ ".report",
  open_in InFile InC, read_and_acc InC 0 Total, close_in Inc,
  open_out OutFile OutC,
  output OutC "The sum of the numbers in the file ",
  output OutC InFile, output OutC " is ",
  String is (int_to_string Total),
  output OutC String, output OutC "./n", close_out OutC.

read_and_acc Chan Acc Result :-
  input_line Chan String,
  (String = "", !, Acc = Result;
   string_to_int String Num, Acc' is Acc + Num,
   read_and_acc Chan Acc' Result).
```

Figure 4.7: A simple program for reading in a file of integers and writting out their sum.

Chapter 5

First-order hereditary Harrop formulas

The first extension to the logic of first-order Horn clauses we consider is to allow implications and universal quantification in goals and the body of clauses. The resulting formulas, the first-order hereditary Harrop formulas (*fohh*, for short) are formulas with clause order greater than 1.

5.1 Three presentations of *fohh*

In this section we present the *first-order hereditary Harrop formulas* or *fohh*, for short. These formulas extend Horn clauses by allowing implications and universal quantifiers in goals (and, thus, in the body of program clauses). Parallel to the three presentations of *fohc* in Section 3.4, there are the following three presentations of goals and program clauses for *fohh*. The first presentation is similar to that of definition 3.1 in Section 3.4.

$$\begin{aligned} G &::= \quad A \mid G \wedge G \mid D \supset G \mid \forall_\tau x.G \\ D &::= \quad A \mid G \supset A \mid \forall x.D \end{aligned} \tag{5.1}$$

Notice now that the definitions of G- and D-formulas are mutually recursive, that a negative (positive) subformula of a G-formula is a D-formula (G-formula), and that a negative (positive) subformula of a D-formula is a G-formula (D-formula). A richer formulation is given by the following definition.

$$\begin{aligned} G &::= \quad \top \mid A \mid G \wedge G \mid G \vee G \mid \exists x.G \mid D \supset G \mid \forall x.G \\ D &::= \quad A \mid G \supset D \mid D \wedge D \mid \forall x.D \end{aligned} \tag{5.2}$$

It will be this set of richer D-formulas that we shall consider the proper definition of first-order hereditary Harrop formulas.

A simple presentation of a class of definite formulas similar to the one above is given by the definition

$$D ::= \quad A \mid D \supset D \mid D \wedge D \mid \forall x.D \tag{5.3}$$

Any first-order formula that does not contain occurrences of disjunction and existential quantification is an example of both a D-formula and G-formula in the sense of definitions 5.2 and 5.3. The formula $(p \vee q) \supset (p \vee q)$ is neither a D-formula and G-formula in any of the definitions above.

Classical logic does not support a goal-directed search interpretation of logical connectives for any interesting uses of $\supset$ and $\forall$ in goal formulas. For example, the formula $p \vee (p \supset q)$ is a classical tautology but it is not provable using the search operations given above: p is not provable and q does not follow from p. Similarly, if the current program $\mathcal{P}$ contains the single formula $(p\ a \wedge p\ b) \supset q$ then the formula $\exists x.(p\ x \supset q)$ is a classical conclusion but it cannot be found using the search reductions described above.

The three presentations of *fohh* given above are not related using intuitionistic equivalence. First notice that the definite formulas of definition 5.2 strictly contain the definite formulas of definitions 5.1 and 5.3. In particular, the formula

$$(p \supset (q \vee r)) \supset s$$

is a legal definite clause using Definition 5.2, but it is not logically equivalent to a formula or conjunction of formulas using either 5.1 or 5.3. While it is the case that the displayed formula above does imply the conjunction

$$((p \supset q) \supset s) \wedge ((p \supset r) \supset s),$$

the converse is not true (although the converse is a classical logic entailment). As program clauses, however, these two formulas can be used interchangeably since they can be used to prove exactly the same goal formulas.

The existential quantifiers allowed in goals in Definition 5.2 cannot always be eliminated as was possible with *fohc*. In *fohc*, an existential quantifier in a goal can be given a larger scope until it can be converted to a universal quantifier surrounding a Horn clause. There are two ways that an existential quantifier in a goal can be "stuck" within a goal. First, it is possible for it to be to the right of an implication, as in the goal formula $D \supset \exists x\ G$. Even if x is not free in D, this formula is not intuitionistically equivalent to $\exists x(D \supset G)$. It is also possible for an existential quantifier to be inside the scope of a universal quantifier. For example, consider the program clauses

```
p X :- pi y\(sigma Z\(q X y Z)).
```

The existential quantifier for `Z` cannot be removed by simple logical equivalences of first-order logic. It is possible, however, to introduce a new predicate constant (similar to that done for disjunctions in Section 3.4) to obtain a program that proves the same goals (not involving the new predicate constants). In particular, the two clauses

```
p X :- pi y\(r X y).
r X Y :- q X Y Z.
```

can be used instead of the above clause. In Chapter 7 we will introduce an extension to *fohh* for which it will be possible to use a higher-order variable to move an existential quantifier from inside to outside the scope of a universal quantifier.

In the rest of this chapter, programs will be assumed to be finite sets of D-formulas as given by 5.2 above. Similarly, goals will be given by G-formulas using the recursive definition. Notice that in this case, a pair $\langle \Sigma, \mathcal{P} \rangle$ is such that Σ is first-order and $\mathcal{P}$ is of arbitrary clausal order.

5.2 Substitution and quantification

As we have seen, the program clauses and goals for first-order Horn clauses can be simplified so that neither of these classes of formulas require explicit quantification. As a result, the notion of substitution into goals and clauses of *fohc* is a particularly simple operation. To compute the result of substituting the term t for the free occurrences of x in a Horn clause, say D, where D has no explicit quantification, simply replace all occurrences of x in D with t.

In the richer logic programming language *fohh*, substitutions must occasionally be applied to a formula containing bound variables. The simple replacement operation that works for *fohc* will not provide a logically meaningful operation. For example, consider the formula

```
p X :- pi y\(q X y).
```

and consider substituting the term `(f a)` for `X`. The result is simply

```
p (f a) :- pi y\(q (f a) y).
```

Clearly, if the first clause is true, so is the second clause. On the other hand, consider substituting the term `(f y)` for `X`. The result of simply replacing instances of `X` with `(f y)` would be the formula

```
p (f y) :- pi y\(q (f y) y).
```

Notice that the token `y` occurs both as a constant and as a bound variable. In fact, where there was only one occurrence of `y` bound in the original formula, there are two occurrences bound in this latter formula. In fact, this formula is not a logical consequence of the original clause. The reason for this failure is that a *variable capture* has occurred. Proper substitution must avoid such captures.

Let x be an occurrence of a free variable in a formula B, and let t be some term of the same type as x. The term *t is free for x in B* if every free occurrence of x in B is not in the scope of a quantifier that binds a variable free in t. The operation of replacing x with t in B, written as $B[t/x]$, is sound substitution if t is free for x in B. If t is not free for x in B, it is always possible to change bound variables names in B to obtain a formula B' for which t is free for x in B'. Of course, there are many choices for B', but they only differ up to bound variable names.

In the example above, the result of substituting `(f y)` for `X` is a formula such as

```
p (f y) :- pi z\(q (f y) z).
```

where many other tokens could have been used instead of `z`. We shall always assume that substitution is performed in this logically sound fashion.

5.3 The core of a logic programming language

Given the distinctions we have made between the program clauses and the goal formulas of a given logic programming language it is interesting to identify that class of formulas that can be in both classes. The *core* of a logic programming language is the intersection of its goal formulas and its program clauses. For example, using the definitions of logic programming based on first-order Horn clauses given in Section 3.4, the core of *fohc* is either the set of atomic formulas (using definitions (3.1) or (3.3)) or the set of conjunctions of atomic formulas (using definition (3.2)).

The core of *fohh* is, however, much richer. Using either definition (5.2) or (5.3), the core is the set of formulas built from atomic formulas using $\wedge$, $\supset$, and $\forall$: only $\vee$ and $\exists$ are excluded. The core of *fohh* coincides with the definition of program clauses given by (5.3). Notice, however, that first-order Horn clauses defined using either (3.1) or (3.3) are contained within the core of *fohh*.

Formulas in the core of logic programming language can be both proved and used as program clause. Since the core of *fohh* contains a rich set of formulas, it will sometimes be possible to use *fohh* to reason about programs directly, a theme that we shall turn to frequently in this chapter.

5.4 Proving implicational goals

As we mentioned above, the goal $D \supset G$ follows from program $\mathcal{P}$ if G follows from the augmented program $\mathcal{P} \cup \{D\}$. After the success or failure of G, the increment D must be removed from the program. Thus the current program part of the current context is actually a stack: it is possible using *fohh* for the current context to change during the search for a proof of a goal. For example, proving the query

$$(D_0 \supset ((D_1 \supset G_1) \wedge (D_2 \supset G_2))) \wedge G_3$$

from the program $\mathcal{P}$ will require that G_1 be solved with the current program being equal to $\{D_1, D_0\} \cup \mathcal{P}$, that G_2 be solved with the current program being equal to $\{D_2, D_0\} \cup \mathcal{P}$, and that G_3 be solved with the current program being simply $\mathcal{P}$.

In the depth-first interpretation of program clauses that is used in λProlog interpreters, the order in which clauses appear is important: different orders of the same clauses can produce different results. Thus, when new clauses are added to the current context, it is important to be explicit about where these new clauses are with respect to those in the current context. There seems to be only two natural choices: clauses should be appended either to the front or rear of the list of clauses in the current program. λProlog uses the rule that clauses are appended to the front: that is, the most recently added clauses are the first to be used in backchaining. To illustrate this, assume that the current context contains just the following.

```
type  p   int -> o.

p 1.
```

Then the following queries should yield the following answer substitutions in the order that they appear.

```
?- p 2 => p 3 => p X.
X == 3;
X == 2;
X == 1
yes

?- (p 2 & p 3) => p X.
```

```
type  q, r, s, t, u   o.

s :- r, q.
t :- q, u.
q :- r.
```

Figure 5.1: Some propositional Horn clauses.

```
X == 2;
X == 3;
X == 1
yes

?-
```

5.4.1 Inferences among propositional clauses

Consider a module that contains just the type declarations and propositional Horn clauses displayed in Figure 5.1. While these clauses do not have any atomic consequences, other clauses can be proved from them. For example, the two Horn Clauses

```
t :- r, u.
s :- r.
```

are both provable from the three clauses in Figure 5.1. In particular, the query

```
?- s :- r.
```

(which could also be written as `r => s`) would be replaced by the query `s` but where the additional clause `r` is added to the current program. At this point, traditional Horn clause reasoning would provide a proof of `s`. As a result, the implication `s :- r` follows from the original set of clauses. In a similar fashion, the clause

```
(r => u) => (r => t)
```

is provable from the clauses in Figure 5.1: in this case, the query `t` is attempted from the original clauses plus the two propositional Horn clauses `r` and `r => u`.

Of course, depth-first search is incomplete and can easily fail to prove obvious inferences. For example, not all instances of $G \supset G$ are provable by λProlog's depth-first interpreter. For example, the query

```
?- (p :- (p => p)) => (p :- (p => p)).
```

does not terminate, although it is clearly a logically correct deduction.

5.4.2 Natural deduction for propositional logic

λProlog provides a convenient setting for specifying and implementing various proof systems. Since λProlog is itself a logic with a proof system, it will be convenient at times to make a distinction between the meta-level logic of λProlog and a given, application dependent

$$\frac{A \quad A \supset B}{B} \qquad \frac{\begin{array}{c}(A)\\ \vdots \\ \mathrm{B}\end{array}}{A \supset B} \qquad \frac{A \wedge B}{A} \qquad \frac{A \wedge B}{B} \qquad \frac{A \quad B}{A \wedge B}$$

$$\frac{A \vee B \quad \begin{array}{c}(A)\\ \vdots \\ \mathrm{C}\end{array} \quad \begin{array}{c}(B)\\ \vdots \\ \mathrm{C}\end{array}}{C} \qquad \frac{A}{A \vee B} \qquad \frac{B}{A \vee B}$$

Figure 5.2: Natural deduction rules for a simple propositional logic.

object-level logic. For example, consider a propositional logic built from the propositional letters p, q, r, and s and logical connectives for conjunction, disjunction, and implication. A natural deduction style proof system for this object-level logic is given in Figure 5.2 and a direct encoding of this proof system is given in Figure 5.3. In this encoding, we have used the λProlog predicate `pv` to indicate the proposition that its argument, which encodes an object-level propositional formula, has an object-level proof. Several of the inference rules in Figure 5.2 have the form

$$\frac{C \quad D}{B}$$

for some formulas B, C, and D. Such an inference rule translates directly into a first-order Horn clause

```
pv B :- pv C, pv D.
```

Thus the horizontal bar corresponds to the `:-`. Other rules have more complex premises since they include a hypothetical, written as

$$\begin{array}{c}(\mathrm{A})\\ \vdots \\ \mathrm{B}.\end{array}$$

Here, the vertical dots correspond naturally to `=>`: the rule for proving a implication $A \supset B$ states that if whenever A is provable, B is provable, then $A \supset B$ is provable. Similarly, the *rule of cases*, that is, the rule for how to prove a formula C from the disjunction $A \vee B$, specifies that C is provable if the disjunction $A \vee B$ is provable and two hypothetical cases hold: if A is provable then C is provable and if B is provable then C is provable.

It should be clear that the λProlog code in Figure 5.3 correctly specifies the object-level syntax and the inference rules in Figure 5.2. While the code in Figure 5.3 provides a simple specification of a proof system, a depth-first interpretation of this code (as is supplied by the λProlog interpreter) is almost worthless. For example, consider an attempt to prove the query

```
?- pv (imp p p).
```

If the second clause in Figure 5.3 is used first, a proof could be discovered quickly. However, the simple-minded, depth-first search that is part of a λProlog interpreter will not be able

```
kind bool            type.
type p, q, r, s      bool.
type and, or, imp    bool -> bool -> bool.
type pv              bool -> o.

pv B :- pv A, pv (imp A B).
pv (imp A B) :- pv A => pv B.
pv A :- pv (and A B).
pv B :- pv (and A B).
pv (and A B) :- pv A, pv B.
pv C :- pv (or A B), (pv A => pv C), (pv B => pv C).
pv (or A B) :- pv A.
pv (or A B) :- pv B.
```

Figure 5.3: A specification of natural deduction rules for $\wedge$, $\vee$, and $\supset$.

to find this proof. The first clause in Figure 5.3 will first be used to find a proof of this goal. Backchaining on this rule, however, generates the query

```
?- pv A, pv (imp A (imp p p)).
```

where `A` is a new variable denoting some object-level formula yet to be determined. When the first of these goals is attempted, the first clause would again be used and the resulting backchaining will generate goals that are not closer to establishing a proof. In fact, the λProlog interpreter will neither succeed nor fail but will loop on this simple query.

Re-organizing the clauses in Figure 5.3 can help in guiding the interpreter on some simple queries, but, in general, a depth-first interpretation of these clause is of little practical use. We shall return to this example of implementing inference rules. With a combination of additional information about the structure of object-level proofs (normal form proofs) and a more explicit management of hypotheses, we will be able to construct a complete theorem prover for this object-logic on top of a depth-first interpreter for the meta-logic. The point of this example is to illustrate that the notion of hypothetical reasoning familiar to natural forms of reasoning can be directly encoded using implications in goal formulas.

5.4.3 Minimal versus intuitionistic negation

A common way to define a logical connective for negation is to first introduce a symbol for falsehood, say $\perp$, and then to define the negation of B as the implication $B \supset \perp$.

Thus, within λProlog a weak form of negation can be accommodated by picking an propositional symbol, say `incon`, to denote inconsistency, and to use that with implication to define a negation. Since `incon` is a non-logical constant, the rule "from false, anything can be proved" (*ex falso guod libet*) is not available for this proxy of falsehood. The negation arising from this simple approach to `incon` is called *minimal logic negation*: if the *ex falso guod libet* rule is used to interpret `incon`, the result is the intuitionistic logic notion of negation.

Although the λProlog interpreter does not support intuitionistic negation directly, an interpreter can be designed to do so: before such an interpreter can fail to prove a query, the interpreter must make certain that `incon` is not provable. If `incon` is provable then any goal is provable. Such a check for inconsistency might be made whenever the program is augmented.

Clearly, the negation of a goal formula is a definite clause; the negation of a definite clause is a goal formula; and the negation of a core formula of *fohh* is again a core formula of *fohh*.

Minimal logic is weak but does satisfy some laws generally connected with negation. For example, the following implications (half of the contrapositive rule and half of the double negation rule) are provable queries in λProlog (assuming that `p` and `q` are constants of type `o`)

```
(p => q) => ((q => incon) => (p => incon)).
p => ((p => incon) => incon).
```

The converse of each of these implications is not provable within minimal logic negation (nor within intuitionistic negation: they are provable, however, in classical logic). Clearly the query

```
?- p; (p => incon).
```

which encodes a particular instance of the classical logic law of the excluded middle, is not provable in the logic of *fohh*. The doubly negated version of this formula, written as the goal formula,

```
?- ((p; (p => incon)) => incon) => incon.
```

is provable. A proof of this is interesting to see and is displayed as a list of some of the queries that arise from attempting this goal (given an initially empty program). Formulas to the left of `?-` are the formulas that are augmented to the current program context (the interpreter does not display such formulas in this way).

```
   (p; (p => incon)) => incon ?- incon.
   (p; (p => incon)) => incon ?- p; (p => incon).
   (p; (p => incon)) => incon ?- p => incon.
p, (p; (p => incon)) => incon ?- incon.
p, (p; (p => incon)) => incon ?- p; (p => incon).
p, (p; (p => incon)) => incon ?- p.
```

This last query immediately succeeds since the goal `p` is also in the current program.

This use of double negation will reappear in Subsections 5.4.6 and 5.4.7.

5.4.4 Hypothetical reasoning

Implications in goals can be used to formulate hypothetical reasoning. For example, consider the modules in Figure 5.4. Here, `db` is a simple looping program that reads a command from the keyboard and performs that command. (As with many interative programs, this one uses several nonlogical features of λProlog.) Most of the commands will call the `db` programs when they finish. Using also the small example database in Figure 5.5, the following queries are possible (assuming that the modules `hyp_db`, `comp_sci_int`, and `comp_sci_ext` are in the current context).

```
?- db.
Command? query (enrolled dana 101).
yes
Command? query (graduates dana).
no
Command? whatif (enrolled dana 301).
Command? query (graduates dana).
yes
Command? query (cs_major dana).
yes
Command? quit.
Command? query (enrolled kim 101).
no
Command? query (graduates kim).
no
Command? whatif (enrolled kim 301).
Command? query (graduates kim).
yes
Command? query (cs_major kim).
no
Command? quit.
Command? quit.
?-
```

This example illustrates how it is possible to use the current context to implement a small database. Here the operations of adding a fact, inferring facts, and doing hypothetical reasoning are successfully specified in the logic of *fohh*.

A simple notion of database constraint can be given elementary support using *fohh*. Consider the clauses in Figure 5.6. The minimal logic negation of a goal formula is used as a constraint. If `incon` is provable, the database evolved into an inconsistent database. In this example, this is only possible if a person has enrolled in both 210 and 250.

The additional database command `consis` reports whether or not the current database is inconsistent. To give the meaning of the `check` command, assume that it is used only when the database is consistent. The command `(check Entry)` prints "`yes`" if `Entry` is present in the database. If `Entry` is not present, then there are two possibilities: either every extension of the current database in which `Entry` is present is inconsistent or some such extension is consistent. In the first case, the `check` command prints "`no`" and in the second case this command prints "`no, but it could be true`." Consider the following uses of this code (assuming that the module `hyp_neg_db` is in the current context).

```
?- db.
Command? whatif (enrolled kim 301).
Command? query (graduates kim).
yes
Command? query (cs_major kim).
no
Command? whatif (enrolled kim 250).
Command? query (cs_major kim).
no
```

```
module db_sig.

kind entry                type.
type fact                 entry -> o.
```

```
module hyp_db.
accumulate db_sig.

kind command                type.
type db                     o.
type do                     command -> o.
type enter, query, whatif   entry -> command.
type quit                   command.

db :- print "Command?", read Command, do Command.
db :- print "Try again.", nl, db.

do (enter Fact) :- fact Fact => db.
do (query Q) :- (fact Q, !, print "Yes", nl; print "No", nl), db.
do (whatif Conjecture) :- (fact Conjecture => db), db.
do quit.
```

Figure 5.4: The two modules db_sig and hyp_db.

```
module comp_sci_ext.
accumulate db_sig.

kind person               type.
type enrolled             person -> int -> entry.

type kim, dana            person.

fact (enrolled kim  102).
fact (enrolled dana 101).
fact (enrolled kim  210).
fact (enrolled dana 250).

module comp_sci_int.
accumulate comp_sci_ext.
type cs_major, graduates       person -> entry.

fact (cs_major X) :-
  (fact (enrolled X 101); fact (enrolled X 102)),
  fact (enrolled X 250), fact (enrolled X 301).
fact (graduates X) :-
  (fact (enrolled X 101); fact (enrolled X 102)),
  (fact (enrolled X 210); fact (enrolled X 250)),
  fact (enrolled X 301).
```

Figure 5.5: The extensional (top) and intensional (bottom) parts of a sample database.

```
module hyp_neg_db.
accumulate hyp_db.

type check            entry -> command.
type quit, consis     command.
type incon            o.

do consis :- incon, print "no", nl, !; print "yes".
do (check Entry) :-
   (fact Entry, print "yes", nl, !;
    fact Entry => incon, print "no", nl, !;
    print "no, but it could be true", nl),
   db.

incon :- fact (enrolled X 210), fact (enrolled X 250).
```

Figure 5.6: Some clauses to help in using database constraints.

```
Command? consis.
no
Command? quit.
Command? quit.
?-
```

A little reflection on the structure of this database shows that there is no consistent way for Kim to simultaneously graduate and be a CS major.

5.4.5 Quantifiers in goals and logic variables in programs

Program clauses are, by definition, closed formulas: if quantification is missing when clauses are presented, outer-most universal quantifiers are assumed. An implementation technique used by λProlog interpreters as well as most logic programming interpreters is that of using a combination of free variables (called *logic variables*) and unification to delay the determination of instances for some quantifiers. Thus, although a query and program may start a computation as closed formulas, an interpreter may consider open versions of both of these. While in Prolog, logic variables only find their way into goal formulas and not into the program, in λProlog, logic variables may appear in both. λProlog and Prolog also differ in that program and goals in λProlog occasionally must have quantifiers in them, where as in Prolog, there is no syntax for expressing even the implicit quantification that is present there. The following example illustrates both of these aspects of λProlog.

Consider the two modules in Figure 5.7. They both present tail-recursive implementations of the list reversal predicate. In each case, these implementations make use of an auxiliary predicate `rev` and the code specifying the meaning `rev` is assumed only for the scope of its use. For example, assuming that the `reverse1` module is in the current context, the query

```
?- reverse (1::2::nil) P.
```

reduces to the query

```
module reverse1.
accumulate lists.

type reverse    list A -> list A -> o.
type rev        list A -> list A -> list A -> o.

reverse L K :-
   (pi L\(rev nil L L) &
    pi X\(pi L\(pi K\(pi M\(rev (X::L) K M :- rev L K (X::M))))))
         => rev L K nil.

module reverse2.
accumulate lists.

type reverse, rev    list A -> list A -> o.

reverse L K :-
   (rev nil K &
    pi X\(pi L\(pi K\(rev (X::L) K :- rev L (X::K)))))
         => rev L nil.
```

Figure 5.7: Two implementations of the list reversal predicate.

```
?- rev (1::2::nil) P nil.
```

where the following two clauses for `rev`

```
rev nil L L.
rev (X::L) K M :- rev L K (X::M).
```

are also added to the current program for the duration of the reversing process. Thus the code for the auxiliary predicate `rev` is available only for a specific part of the computation. Furthermore, the universal quantification explicitly written into the `reverse` clause cannot be dropped: if it were, the assumed scope of the quantification for the variables in embedded clauses for `rev` would be on the outside of the `reverse` clause and thus it would not work.

Given this style of programming, there is another way that `reverse` can be written. One way to reverse a list, say `(a::b::c::nil)`, is to start with the program

```
rv nil (a::b::c::nil).
pi X\(pi N\(pi M\(rv (X::N) M :- rv N (X::M)))).
```

from which the goal `(rv (c::b::a::nil) nil)` is provable. Obviously, if we replace `(a::b::c::nil)` with any list `L`, we can prove the atomic goal `(rv K nil)` if and only if `L` and `K` are reverses of each other. While this is a natural approach to specifying `reverse` (more natural it seems than the first one described above), it is not possible to code it directly in Horn clauses since it describes the `reverse` predicate as relating a list contained in a program and one contained in a goal. It is easy to write this relation in *fohh*. Consider the second specification of `reverse` in the `reverse2` module. Here, the query

```
?- reverse (1::2::nil) P.
```

reduces to the (closed!) query

```
?- rev (1::2::nil) nil.
```

The variable into which the answer substitution for this computation will be placed is actually within the program, since the program was augmented with the clauses

```
rev nil K.
rev (X::L) K :- rev L (X::K).
```

Here, the variable `K` in the first clause is free in that clause. Our convention for displaying clauses using implicit quantification breaks down in displaying such clauses. The above `rev` goal is reduced using the second `rev` clause to `(rev (2::nil) (1::nil))` and then to `(rev nil (2::1::nil))`. This final goal then succeeds by binding the variable `P` to the list `(2::1::nil)`. Since `P` was originally free in the initial query, this binding for `P` is reported as the result of reversing the list `(1::2::nil)`.

5.4.6 Partial control of clause selection

Using a technique similar to that of forming double negations in minimal logic, we can partially specify the selection of clauses in the search for proofs. Given that controlling deduction is difficult to do from within logic, any logical technique for achieving some aspects of control is of interest. Since we shall use these techniques only within this chapter, our interest here is mostly with illustrating embedded implications.

In this section, fix Σ to be a first-order signature and let $q : o$ be a propositional constant not in Σ. Let B be a Σ-formula. The $\Sigma \cup \{q : o\}$-formula $(B \supset q) \supset q$, expressing a double negation of B using q as the marker for inconsistency, will be denoted as simply B^q. If $\mathcal{B}$ is a set of Σ-formulas, then $\mathcal{B}^q$ is defined as $\{B^q \mid B \in \mathcal{B}\}$.

Let $n \geq 1$, let $\mathcal{P} \cup \{D_1, \ldots, D_n\}$ be a finite set of *fohh* program clauses, and let G be a *fohh* goal, all of which are Σ-formulas. Furthermore, let $q_1, \ldots, q_n$ be propositional constants not contained in Σ and let Σ' be $\Sigma \cup \{q_1 : o, \ldots, q_n : o\}$. We shall argue that G^{q_i} follows from $\langle \Sigma', \mathcal{P} \cup \{D_1^{q_1}, \ldots, D_n^{q_n}\} \rangle$ if and only if G follows from $\langle \Sigma, \mathcal{P} \cup \{D_i\} \rangle$. Thus, the propositional constant q_i can be used to mark the clause D_i in such a way that the goal G^{q_i} will be able to access only that marked clause. To see why this is the case, first assume that G follows from $\langle \Sigma, \mathcal{P} \cup \{D_i\} \rangle$. Then it is easy to build a proof that G^{q_i} follows from $\langle \Sigma', \mathcal{P} \cup \{D_1^{q_1}, \ldots, D_n^{q_n}\} \rangle$ using backchaining and the AUGMENT rule. Conversely, consider a goal-directed proof of G^{q_i} from $\langle \Sigma', \mathcal{P} \cup \{D_1^{q_1}, \ldots, D_n^{q_n}\} \rangle$, and for the sake of the following argument, assume that this proof is the shortest one possible. This proof must end in an AUGMENT rule, and so contains a subproof showing that q_i follows from $\langle \Sigma', \mathcal{P} \cup \{D_1^{q_1}, \ldots, D_n^{q_n}\}, G \supset q_i\} \rangle$. The last rule in this subproof is a backchaining over either $G \supset q_i$ or over $D_i^{q_i}$. In the first case, our proof has a subproof of G from $\langle \Sigma', \mathcal{P} \cup \{D_1^{q_1}, \ldots, D_n^{q_n}, G \supset q_i\} \rangle$. Since G and $\mathcal{P}$ do not contain occurrences of $q_1, \ldots, q_n$, G is provable from that context if and only if G is provable from $\langle \Sigma, \mathcal{P} \rangle$. In the second choice, we have a subproof of q_i from $\langle \Sigma', \mathcal{P} \cup \{D_1^{q_1}, \ldots, D_n^{q_n}, G \supset q_i, D_i\} \rangle$. Again, this proof could have been proved by backchain over either $G \supset q_i$ or $D_i^{q_i}$. The first case leads to a subproof of G from $\langle \Sigma', \mathcal{P} \cup \{D_1^{q_1}, \ldots, D_n^{q_n}, D_i\} \rangle$ and, by the observation made above, this is possible if and only if G is provable from $\langle \Sigma, \mathcal{P} \rangle$. In the second case, that of backchaining over $D_i^{q_i}$,

```
kind i type.

type a     i.
type f,g   i -> i.
type p     i -> o.

type q1,q2 o.

p a.
((pi X\ p (f X) :- p X) => q1) => q1.
((pi X\ p (g X) :- p X) => q2) => q2.
```

Figure 5.8: Using the double negation construction for specifying clause selection.

we again have a proof of q_i from $\langle \Sigma', \mathcal{P} \cup \{D_1^{q_1}, \ldots, D_n^{q_n}, G \supset q_i, D_i\}\rangle$. This case, however, is not possible since we assumed that we had started with the shortest possible proof.

For example, assume that we need to use the clauses

```
p a.
p (f X) :- p X.
p (g X) :- p X.
```

in a controlled fashion. In particular, we may want to sometimes use the first and second clause only, or sometimes the first and third, or sometimes all three clauses. The clauses in Figure 5.8 provide for a way to realize this control. The goal `(p U => q1) => q1` will only access the first two clauses; the goal `(p U => q2) => q2` will only access the first and third clauses, and the more complex goal

```
(p U => q1) => (p U => q2) => (q1 ; q2).
```

will access all three clause.

In general, if we allow the formulas $D_1, \ldots, D_n$ to contain certain occurrences of the symbols $q_1, \ldots, q_n$, more subtle clause selection can be managed. For example, consider the module in Figure 5.9. The `adj` predicate describes a small graph containing cycles. If we simply used the two clauses

```
path U V :- adj U V.
path U W :- adj U V, path V W.
```

for specifying the path relation in this graph, a depth-first interpreter would generally not terminate in computing the `path` relationship since the cycle in the graph would lead to a looping computation. The above technique can be used to partially ameliorate this problem by specifying a kind of stratified path predicate. The stratification, which is only specified for paths of length 3 or less, is achieved by labeling path clauses using special propositional constants. In particular, if `t` and `s` are particular nodes in this graph, the goal `(path t s => q3) => q3` is provable if and only if there is a path of length 3 from `t` to `s`.

While this technique of labeling and selecting clauses is of interest, its use is reminiscent of those unfortunate programming languages where statements are numbered and control is specified with `goto` statements. In the next subsection, we employ this labeling technique in a less drastic way.

```
module path.

kind node  type.

type a,b,c,d,e   node.

adj a b.
adj b a.
adj b c.
adj c a.
adj b d.

type q0,q1,q2,q3  o.

((pi U\(path U U)) => q0) => q0.
(pi U\(pi V\(pi W\(path U W :- (adj U V, ((path V W) => q0) => q0
                                 )))) => q1) => q1.
(pi U\(pi V\(pi W\(path U W :- (adj U V, ((path V W) => q1) => q1
                                 )))) => q2) => q2.
(pi U\(pi V\(pi W\(path U W :- (adj U V, ((path V W) => q2) => q2
                                 )))) => q3) => q3.
```

Figure 5.9: Cascading control in the computation of a path in a cyclic graph.

5.4.7 Bottom-up interpretation and memoization

Given the dynamically changing nature of the current program when embedded implications are used, it should be possible to store goal formulas in programs once they have been proved. This is at least true when the goal formula is in the core of *fohh*. Such a memoization technique, however, does not generally work well in practice. Consider, for example, the two goal formulas

$$A \wedge G \qquad \text{and} \qquad A \wedge (A \supset G).$$

Although these two are intuitionistically equivalent, they have different behaviors in a depth-first theorem prover. For example, assume that the goal G may generate A as a subgoal many times. In the first goal above, those subproofs would be recomputed each time A is generated. In the second goal, however, A is stored away in the program and can be used to produce an immediate proof of A. While the second goal can therefore have shorter proofs, all of the (long) proofs of the first goal still exist to prove the second goal: the additional assumption A need not be used. Thus the second goal has many more proofs and only some are shorter than for the first goal. Memoization will be successful if the interpreter can be directed to find only these shorter proofs. Below we show one technique that can sometimes be used to do just this. This technique again relies on the use of "double-negation."

Let Σ be a first-order signature and assume that first-order Horn clauses satisfy Definition 3.1 in Section 3.4. Furthermore, assume that q is some token not in Σ. Let $\mathcal{P}$ be a finite set of first-order Horn clauses over Σ. Define

$$\mathcal{P}_q = \{\forall\bar{x}(G \supset A^q) \mid \forall\bar{x}(G \supset A) \in \mathcal{P}\}.$$

Notice that the formula $\forall\bar{x}(G \supset A^q)$ is intuitionistically equivalent to $\forall\bar{x}[(G \wedge (A \supset q)) \supset q]$. These are all $\Sigma \cup \{q : o\}$-formulas of clausal order of 2: no formula in $\mathcal{P}_q$ is a Horn clause but they are all hereditary Harrop formulas.

Let $\mathcal{P}$ be some finite set of *fohh* formulas and let $\mathcal{Q}$ be a set of *fohc* formulas, both over the signature Σ. Consider proving q from the context $\Sigma \cup \{q : o\}; \mathcal{P} \cup \mathcal{Q}_q \cup \{H \supset q\}$. The first step in such a proof is a backchaining over some clause in $\mathcal{Q}_q$ or over the clause $H \supset q$. In the first case, assume that the backchaining rule used the formula $\forall\bar{x}[G \supset A^q]$, where $\forall\bar{x}[G \supset A]$ is a Σ-formula in $\mathcal{Q}$. Thus, there is some closed substitution θ so that there are proofs of θG from $\mathcal{P} \cup \mathcal{Q}_q \cup \{H \supset q\}$ and of q from $\mathcal{P} \cup \mathcal{Q}_q \cup \{H \supset q, \theta A\}$. Since q does not appear in θG, it must be the case that θG follows, in fact, from $\mathcal{P}$. Thus, the context for the subproof for q is the same as for the full proof except that a logical consequent of $\mathcal{P}$, namely θA, has been added to its context. If this subproof was proved by backchaining on a clause from $\mathcal{Q}_q$, say $\forall\bar{x}[(G' \wedge (A' \supset q)) \supset q]$, then there would be some closed substitution θ' for which $\theta' G'$ is provable from $\mathcal{P} \cup \{\theta' A'\}$ and q is provable from $\mathcal{P} \cup \mathcal{Q}_q \cup \{H \supset q, \theta A, \theta' A'\}$.

Eventually, a proof must backchain over the clause $H \supset q$. Thus, we would have a subproof of H from $\mathcal{P} \cup \mathcal{Q}_q \cup \mathcal{A}$ where $\mathcal{A}$ is a set of closed atomic Σ-formulas that are consequences of $\mathcal{P} \cup \mathcal{Q}$. Since H does not contain q, it must be true that H follows from $\mathcal{P} \cup \mathcal{A}$ and therefore from $\mathcal{P} \cup \mathcal{Q}$. Thus, we have achieved a kind of bottom-up style proof (even using an depth-first interpreter).

Given the above discussion, the following is not difficult to prove: Let $\mathcal{P}$ be a set of *fohh* formulas and let $\mathcal{Q}$ be a set of *fohc* formulas, both over the signature Σ. Let q be a token not in Σ. Then for any Σ-formula goal G, $\Sigma; \mathcal{P} \cup \mathcal{Q} \vdash_I G$ if and only if $\Sigma \cup \{q : o\}; \mathcal{P} \cup \mathcal{Q}_q \vdash_I G^q$.

Consider applying these ideas to the computation of Fibonacci numbers as specified in Figure 5.10. (To stay entirely within logic, a version of the non-negative integers are represented here using terms: the built-in integers require using predicates that are not fully explained by first-order logic (see Section 4.4.1).) As is well known, this specification of the Fibonacci sequence is computationly expensive: to compute the n^{th} Fibonacci number requires an exponential number of recursive calls to `fib`. If we set $\mathcal{P}$ to the three clauses specifying `fib`, then $\mathcal{P}_q$ would contain the clauses in Figure 5.11. Using the discussion above, it is easy to see that the query

```
?- (fib N M => q) => q.
```

is provable from the code in Figure 5.11 if and only if `fib N M` is provable from the code in Figure 5.10. While there are some proofs of `(fib N M => q) => q` that are linear in the size of `N`, there are also a great number of large proofs. The program in Figure 5.11 is certainly a poor one to be used in a depth-first interpreter. This particular example, however, can easily be improved by "driving" the interpreter to select particular clauses on which to backchain. This can be done by elaborating the simple propositional letter `q` into `qq N`, where `N` serves as an index that increases as a proof is built. Given this elaborated program, there is only one proof of the query

```
?- (fib N M => qq (s N)) => qq z.
```

It is also interesting to note that if `H` is a goal that contains possibly many calls to `fib` with their first argument in the range from 0 to `N`, then the query

```
?- (H => qq (s N)) => qq z.
```

will first generate a table of Fibonacci numbers and then evaluate `H` against that table.

```
kind nat     type.
type z       nat.
type s       nat -> nat.
type plus    nat -> nat -> nat -> o.

plus z N N.
plus (s N) M (s P) :- plus N M P.

type fib     nat -> nat -> o.

fib z z.
fib (s z) (s z).
fib (s (s N)) V :- fib N V1, fib (s N) V2, plus V1 V2 V.
```

Figure 5.10: A simple formulation of non-negative integer addition and a naive computation of Fibonacci numbers.

```
type fib     nat -> nat -> o.
type q       o.

q :- fib z z => q.
q :- fib (s z) (s z) => q.
q :- fib N V1, fib (s N) V2, plus V1 V2 V, (fib (s (s N)) V => q).
```

Figure 5.11: A bottom-up, undirected computation of Fibonacci numbers.

```
type qq     nat -> o.

qq z         :- fib z z => qq (s z).
qq (s z)     :- fib (s z) (s z) => qq (s (s z)).
qq (s (s N)) :- fib N V1, fib (s N) V2, plus V1 V2 V,
                fib (s (s N)) V => qq (s (s (s N))).
```

Figure 5.12: A bottom-up, directed computation of Fibonacci numbers.

```
module sterileJar.
kind bug, jar          type.

type j                 jar.
type sterile, heated   jar -> o.
type dead, bug         bug -> o.
type in                bug -> jar -> o.

sterile J :- pi x\(bug x => in x J => dead x).
dead B    :- heated J, in B J, bug B.
heated j.
```

Figure 5.13: The file `jar.mod`.

5.5 Universal goals

λProlog allows universal quantifiers to also appear in goals. There are at least two different ways to interpret the goal $\forall x.G(x)$.

The *extensional* interpretation of this quantifier might be the first to come to mind. Using this intepretation, a universal quantifier is true if it is true for all instances of that quantifier. To be a bit more precise, assume that the type of the bound variable x above is τ and assume that our intended notion of τ is a particular set of objects. For example, `int` denotes the set of integers and `list string` denotes the set of lists of strings. Then this quantified expression is true if all instances of the goal $G(x)$ are true when x ranges over all the elements of the domain denoted by τ.

There are at least three reasons why such an interpretation of universal quantification is not used in λProlog. First, given a suitable definition of model, provability in logic is generally tied to truth in *all* models and not truth in one model. Second, there can generally be an infinite number of members of a type, in which case this check is computationally intractable. (In some database applications, for example, types might be restricted in such a way as to be finitary, so this kind of check is feasible.) Third, a type may not, in fact, be *closed* or we might wish to draw conclusions without necessarily knowing all elements of a type. As we shall see in this chapter and in Chapter 7, there are a number of applications where a type should be considered open.

λProlog uses an *intensional* interpretation of universal quantification: $\forall_\tau x.G(x)$ follows from the context $\Sigma; \mathcal{P}$ if $G[c/x]$ follows the context $\Sigma \cup \{c : \tau\}; \mathcal{P}$ for some token c that does not occur in Σ. That is, $\forall_\tau x.G(x)$ follows if it follows *generically*. We have already introduced this interpretation in Section 3.3. The new constant used in this computation will be called a *scoped constant*. In the sequent calculus, these are called *eigenvariables*. Notice that there is a strong similarity in the interpretation of implicational goals and universal quantifiers: implicational goals require the current program to be augmented for a certain scope and universal quantifiers require the current signature to be augmented for a certain scope.

For a simple example of using universal quantifiers in goals, consider the following problem. Assume that a jar is sterile if every bug (germ) in it is dead, that a bug in a heated jar is dead, and that a given jar has been heated. What reasoning is necessary to establish that the given jar is sterile? The intensional interpretation of the quantification will work here. Let $\langle \Sigma, \mathcal{P} \rangle$ be the signature-program pair associated with the the module in Figure 5.13. (Notice that there are no bugs provided by the `sterileJar` module: that is, there are no Σ-terms of type `bug`.) Consider proving the goal `sterile j` from the context $\Sigma; \mathcal{P}$. Backchaining on the first clause in $\mathcal{P}$ yields the goal

```
?- pi x\(bug x => in x j => dead x).
```

Given the intensional interpretation of universal quantification, we proceed by selecting a constant, say `g`, that does not occur in Σ. We now attempt to prove the goal

```
?- bug g => in g j => dead g.
```

This goal succeeds if the goal `dead g` follows from the program $\mathcal{P}$ augmented with `bug g` and `in g j`. It is easy to see that this in fact follows by simple backchaining steps. After this goal succeeds, the two clauses `bug g` and `in g j` are removed from the current program: the constant `g` is similarly removed (discharged).

In order to use unification to implement an interpreter for *fohh*, unification must account for the introduction of scoped constants. For example, there is no substitution for `X` such that the goal

```
?-  pi y\(p (f y) => p X).
```

succeeds from the empty program. If we naively simplify this goal as described above, we would first generate a scoped constant, say `c`, and then try to prove `(p X)` from `(p (f c))`. But this reduced problem is satisfied with the substitution of `(f c)` for `X`. Notice, however, that the result of applying this substitution to the goal above, namely

```
?-  pi y\(p (f y) => p (f c)).
```

does not yield a provable goal. This unsoundness arises from the fact that when `c` was selected, the future instantiations of `X` must be restricted to be terms that cannot contain the constant `c`. This restriction, which blocks the only route to a proof of the above goal, is central to many uses of universals in goals made by λProlog programs.

In general, whenever a new constant is used to instantiate a universal goal, all free variables, in the goal and the program, must be restricted so that the substitution terms that will eventually instantiate them will not contain that new constant. Free variables generated by subsequent backchaining steps, however, may be instantiated with terms containing this new constant. For example, consider proving the goal

```
?- pi X\(p X) => (p Y) => pi Z\(p Z).
```

where the current program contains no clauses for `p`. The only free variable in this goal is `Y`. After adding the clause `pi X\(p X)` and `(p Y)` to the current program, the goal `pi Z\(p Z)` is attempted by first introducing the new constant `c` and attempting to prove `(p c)`. The first clause to be checked for backchain is `(p Y)`. When `c` was introduced, however, `Y` was constrainted so that it could not be instantiated with a term containing `c`. Thus, the clause `(p Y)` cannot be used to prove this goal. The second clause to be attempted for backchaining is `pi X\(p X)`: first a new logic variable, say `U`, is introduced, and `(p U)` is unified with `(p c)`. Since `U` was introduced after `c`, there is no constraint on `U` being instantiated with a term containing `c`. Thus, this goal has one proof.

For some types τ, such as `int` and `(list A)`, there is a third method for proving the universal quantifier $\forall_\tau x.G$: namely, *induction*. λProlog does not have induction built-in, even for types such as `int`. Notice, however, that induction does make use of the intensional interpretation of universal quantification. For example, to prove that a given formula, say $C(n)$, is true for all non-negative integers n, the implication $\forall m(C(m) \supset C(m+1))$ must be proved: this latter quantification is treated using the intensional interpretation. (The base case $C(0)$, of course, must also be proved.)

For example, assume that the `lists` module with the usual two clauses for `append` is in the current program space. While the query

```
?- pi L\(append nil L L).
```

is provable (this is actually one of the clauses defining `append`), it is not the case that the query

```
?- pi L\(append L nil L).
```

is provable. The generic reading of this quantifier is not strong enough to prove this query. It is possible to use induction in a straightforward fashion to prove this query. In particular, the following query, which states the base case and inductive case, is provable.

```
?- append nil nil nil,
   pi L\(append L nil L => pi X\(append (X::L) nil (X::L))).
```

Generally, proofs using induction will not be so simple for λProlog's depth-first interpreter to complete.

5.5.1 Inferences among clauses

We generally think of λProlog as proving a relationship between programs and goals. However, if we restrict ourselves to the core of *fohh*, there is syntactically no difference between these objects. Thus, it is natural, for example, to ask if one specification of a predicate can prove another specification.

For example, consider the three different specifications of the Fibonacci relation given in Figures 5.10, 5.11, and 5.12. In Figure 5.14 the initial, immediate specification of this relation is given, while the other two specifications are placed within the body of the clauses specifying the `check1` and `check2` predicates. It is a (surprisingly) simple λProlog computation to show that `check1` is provable, that is, that the first specification implies the second. In particular, let F_1 denote the three clauses in the first specification and let F_2 be the three clauses in the second specification. Then the computation of `check1` demonstrates that $F_1 \vdash_I F_2$. Let Q denote the program clause for `qq` in Figure 5.14. Then the fact that `check2` is provable from this module demonstrates that $F_1, Q \vdash_I F_3$ where F_3 is the set of clauses in the third specification of the Fibonacci relation.

These entailments of λProlog can be used to supply part of the arguments that F_2 and F_3 specify the same relationship that is specified by F_1. Assume that $F_2 \vdash_I (fib\ n\ m \supset q) \supset q$. Since $\vdash_I$ is transitive, $F_1 \vdash_I (fib\ n\ m \supset q) \supset q$, which is possible if and only if $F_1, (fib\ n\ m \supset q) \vdash_I q$. However, the latter is provable if and only if $F_1 \vdash_I fib\ n\ m$. Thus, if F_2 can prove $(fib\ n\ m)^q$, then m is the n^{th} Fibonacci number. Similarly, assume that

$$F_3 \vdash_I (fib\ n\ m \supset qq\ (s\ n)) \supset qq\ z.$$

By transitivity again, $F_1, Q \vdash_I (fib\ n\ m \supset qq\ (s\ n)) \supset qq\ z$, which is possible if and only if $F_1, Q, (fib\ n\ m \supset qq\ (s\ n)) \vdash_I qq\ z$. However, the latter is provable if and only if $F_1 \vdash_I fib\ n\ m$ has a proof.

The converses of these results, that is, if m is the n^{th} Fibonacci number then

$$F_2 \vdash_I (fib\ n\ m \supset q) \supset q \qquad \text{and} \qquad F_3 \vdash_I (fib\ n\ m \supset qq\ (s\ n)) \supset qq\ z$$

is not addressed here. Instead of using transitivity of provability, induction on the structure of goal-directed proofs is needed.

5.5.2 Hiding constants

Let *fohh*$^+$ be the extension to *fohh* that allows universal quantifiers in goal formulas to bind variables of order 1 types as well as at order 0. Furthermore, these types can be either of the form τ_f or τ_p (see Subsection 2.7). The logical significance of this extension to *fohh* is provided by the more general higher-order logic presented in Chapter 6.

One use of universal quantification in goals at predicate types is to introduce a predicate symbol that can be hidden, that is, given local scope. A standard way to write the list reversing program, `reverse`, in Prolog (see also Section 5.4.5) is to first write a tail recursive auxiliary function `rev` of three arguments. Although this second predicate is intended to be used only locally in the definition of `reverse`, there is no way in simple Horn clause logic for the scope of `rev` to be localized to just the definition of `reverse`. Making use of the universal quantification of predicates and of implications in goals, we can write a version of `reverse` where `rev` is given local scope. Consider the program in Figure 5.15. First, notice that the variables `L` and `K` are bound with different scopes in this clause: in particular, `L` is bound with the outermost scope around this clause and twice on the internal clauses for the local predicate `rev`. Three differently named variables could have been used to denote these quantified variables.

In attempting to prove the goal `(reverse (1::2::3::nil) K)` from the clause in this file, an interpreter would first generate a new predicate symbol, say `c`, then add the Horn clauses

```
pi L\(c nil L L).
pi X\(pi L\(pi K\(pi M\(c (X::L) K M :- c L K (X::M))))).
```

to the current program, and then try to prove `(c (1::2::3::nil) K nil)`. After the answer substitution `K == (1::2::3::nil)` is discovered, both `c` and the new clauses pertaining to `c` would be discharged.

Using *fohh*$^+$, the second style of specifying reverse given in Section 5.4.5 can also be written so that the auxiliary predicate and its Horn clause specification are given local scope. See Figure 5.16.

5.5.3 Abstract data types

Support for abstract data types is present in *fohh*$^+$ since it is possible to hide the constructors of a given data structure as well as hide various procedures that help in processing their internal structure.

For example, the bodies of each of the clauses in Figure 5.17 introduce local constants and local definitions for the three predicates `empty`, `enter`, and `remove`, and then evaluate

```
module fibimp.

kind nat     type.
type z       nat.
type s       nat -> nat.
type plus    nat -> nat -> nat -> o.

plus z N N.
plus (s N) M (s P) :- plus N M P.

type fib     nat -> nat -> o.

fib z z.
fib (s z) (s z).
fib (s (s N)) V :- fib N V1, fib (s N) V2, plus V1 V2 V.

type q,check1     o.

check1 :-
  (q :- fib z z => q),
  (q :- fib (s z) (s z) => q),
  pi N\(pi V1\(pi V2\(pi V\(
   (q :- fib N V1, fib (s N) V2, plus V1 V2 V,
         (fib (s (s N)) V => q)))))).

type check2    o.
type qq        nat -> o.

qq X :- qq (s X).

check2 :-
  (qq z     :- fib z z => qq (s z)),
  (qq (s z) :- fib (s z) (s z) => qq (s (s z))),
  pi N\(pi V1\(pi V2\(pi V\(
    qq (s (s N)) :- fib N V1, fib (s N) V2, plus V1 V2 V,
                    fib (s (s N)) V => qq (s (s (s N))))))).
```

Figure 5.14: The naive specification of the Fibonacci relation can prove two other specifications.

```
type reverse   list A -> list A -> o.

reverse L K :-  pi rev\(
  (pi L\(rev nil L L) &
   pi X\(pi L\(pi K\(pi M\(rev (X::L) K M :- rev L K (X::M))))))
   => rev L K nil).
```

Figure 5.15: An implementation of `reverse`.

```
type reverse   list A -> list A -> o.

reverse L K :- pi rv\(
  (                   rv nil K &
   pi X\(pi N\(pi M\( rv (X::N) M :- rv N (X::M)))))
   => rv L nil).
```

Figure 5.16: Another implementation of `reverse`.

a given goal. The first and third clauses implement a stack; the second implements a queue. The data constructors for the stack and queue (given collectively the type `bag`) are hidden by universal quantifiers around the bodies of these clauses. An attempt to carry these local constants outside of their scope results in a failure. For example, we have the following queries, assuming that the module `stackqueue` is in the current context.

```
?- test1 A B.
A == 2
B == 1.
?- test2 A B.
A == 1
B == 2.
?- test3 B.
no
?-
```

The last failure results from an attempt to carry the constructors `emp` and `stk` out of their quantified scope.

These examples of specifying abstract data types are cumbersome. In Section 5.6 we present a better syntax for this style of programming.

For two related but simpler examples, consider how to specify goals that fail in all program contexts or that succeed only once in all program contexts. A predicate, say `fail`, will fail if there are no clauses defining it. In a dynamic setting where implications allow new clauses to be added, there is no guarantee that clauses defining `fail` are not added during some computation. The goal `pi p\p`, however, will fail in all programming contexts: when the interpreter encounters this goal, it must select a new null-ary predicate, say `c`, and then attempt to prove `c`, an attempt that must fail since `c` is new. Similarly, the goal `pi p\(p => p)` will succeed exactly once in all programming contexts: again the interpreter will need to select a new null-ary predicate, say c, then assume `c` and then attempt to prove

```
module stackqueue.
kind bag type.

type empty          bag -> o.
type enter, remove  int -> bag -> bag -> o.
type test1, test2   int -> int -> o.
type test3          bag -> o.

test1 A B :- pi emp\(pi stk\(
    empty emp                          =>
    pi S\(pi X\( enter X S (stk X S)))  =>
    pi S\(pi X\( remove X (stk X S) S)) =>
      sigma S1\(sigma S2\(sigma S3\(sigma S4\(sigma S5\(
        empty S1, enter 1 S1 S2, enter 2 S2 S3,
        remove A S3 S4, remove B S4 S5 ))))))).

test2 A B :- pi qu\(pi
    pi L\( empty (qu L L))                                 =>
    pi X\(pi L\(pi K\( enter X (qu L (X::K)) (qu L K))))  =>
    pi X\(pi L\(pi K\( remove X (qu (X::L) K) (qu L K)))) =>
      sigma S1\(sigma S2\(sigma S3\(sigma S4\(sigma S5\(
        empty S1, enter 1 S1 S2, enter 2 S2 S3,
        remove A S3 S4, remove B S4 S5 )))))).

test3 V :- pi emp\(pi stk\(
    empty emp                          =>
    pi S\(pi X\( enter X S (stk X S)))  =>
    pi S\(pi X\( remove X (stk X S) S)) =>
        sigma U\(empty U, enter 1 U V))).
```

Figure 5.17: Implementations for stacks and queues.

`c`, which will, of course, have exactly one proof in all programming contexts. Similarly, the goal `pi p\((p :- p) => p)` will cause the interpreter to loop.

5.6 Additional module directives

The additional expressiveness of *fohh* and *fohh*$^+$ over that of *fohc* allows two more module directives, `import` and `local`, to be defined in terms of embedded implications and universal quantifiers.

5.6.1 Declaring a local scope to constants

In Sections 5.5.2 and 5.5.3 we frequently used a universal quantifier over an implication, $\forall x(D \supset G)$ where x is not free in G, to introduce a scoped constant (instantiating x) and new clauses describing the meaning of those constants (the appropriate instance of D). This formula, however, is intuitionistically equivalent to $(\exists x.D) \supset G$. This suggests that existential quantification over program clauses can be used to establish local scoping of constants. Unfortunately, formulas of the form $\exists x.D$ are not program clauses in either *fohh* or *fohh*$^+$. We can resolve this lack by introducing a new class of formulas, for existentially quantified program clauses:

$$E ::= \quad D \mid \exists x.E \mid E \wedge E.$$

We shall now allow modules to denote E-formulas. When such a formula, say $\exists \bar{x}.D$, is merged with the current context, new scoped constants will be introduced to instantiate the variables in the list $\bar{x}$ and the current program will be expanded with the appropriate instance of D. Such an interpretation of existential quantification is justified by the equivalence

$$(\exists \bar{x}.D) \supset G \equiv \forall \bar{x}(D \supset G).$$

The `local` declaration will exactly match this form of existential quantification over program clauses.

The `stack` module in Figure 5.18 uses the `local` directive to declare that the scope of the two constructors for the `bag` type, `emp` and `stk`, are local to this module. The syntax for the `local` declaration is the same as for the type declaration, except that the keyword `local` replaces the keyword `type`. (It is an error for the same token to be given a `type` and `local` declaration within the same module.) The signature attached to this module is exactly

```
kind bag             type -> type.
type empty           bag A -> o.
type enter, remove   A -> bag A -> bag A -> o.
```

The hidden constants are not part of this signature. Because the intended meaning of `local` is an existential quantifier and since these can be accounted for as universal quantification over goals, the fact that a constant does not leave a locally-declared scope is enforced by unification, as illustrated in Subsection 5.5.3.

The `local` keyword should appear prior to the first occurrence within the module of a formula containing the constant it declares. The intended scope of `local` is, however, global and can alway be written in the preamble of a module.

```
module stack.
kind bag           type -> type.

local emp          bag A.
local stk          A -> bag A -> bag A.

type empty         bag A -> o.
type enter, remove  A -> bag A -> bag A -> o.

empty emp.
enter X S (stk X S).
remove X (stk X S) S.
```

Figure 5.18: A module implementing stacks as an abstract data type.

```
module queue.
kind bag type -> type.

type empty         bag A -> o.
type enter, remove  A -> bag A -> bag A -> o.

local qu   list A -> list A -> bag A.

empty (qu L L).
enter X (qu L (X::K)) (qu L K).
remove X (qu (X::L) K) (qu L K).
```

Figure 5.19: An abstract data type for a queue data structure.

The `queue` module in Figure 5.19 also makes use of the `local` directives. In Section 5.5.3 universal quantification over a propositional variable was used to describe goals that alway succeed, fail, or diverge. These goals are named in Figure 5.20: there, `local` instead of universal quantification is used to provide the scope for the propositional variable.

The match between existential quantification over program clauses and the `local` declaration is not exact because of the presence of polymorphic typing. In particular, a quantified variable can be used at only one type within its scope while a locally declared constant of polymorphic type can be used at many different instances of its type.

5.6.2 Importing modules

The `import` directive is similar to the `accumulate` directive in syntax but its meaning is more complex. If a module `mod1` contains the line

```
import mod2, mod3.
```

then the modules `mod2` and `mod3` are made available (via implications) during the search for proofs of the body of clauses listed in `mod1`. Thus, if the formulas E_2 and E_3 are associated with `mod2` and `mod3`, then a clause $G \supset A$ listed in `mod1` is elaborated to the

```
module succfail.
local    p  o.
type     fail, succeed, twice, repeat, diverge   o.

fail     :- p.
succeed  :- p => p.
twice    :- p => p => p.
repeat   :- (p => p) => p => p.
diverge  :- (p => p) => p.
```

Figure 5.20: Some simple definition of successes and failure.

```
module revmod.
import lists.

type  reverse, nrev  list A -> list A -> o.
local rev            list A -> list A -> list A -> o.

reverse L K :-  rev L K nil.

rev nil L L.
rev (X::L) K M :- rev L K (X::M).

nrev nil nil.
nrev (X::L) K :- nrev L M, append L (X::nil) K.
```

Figure 5.21: Hiding an auxiliary predicate using `local`.

clause $((E_2 \wedge E_3) \supset G) \supset A$, or rather to the program clause that is equivalent to this formula once the existential quantifiers surrounding E_2 and E_3 (if any) are changed to universal quantification of the goal formula G (as described above). The signature of the module `mod1` is the result of merging the signatures for `mod2` and `mod3` with the declarations explicitly declared in `mod1`.

Consider the module in Figure 5.21 that specifies the list reversal in two different ways. Notice that the `lists` module is imported. Thus, the signature of the module `revmod` is that of `lists` with the type declarations for `reverse` and `nrev` added. While all of the constants in the `lists` module are part of `revmod`'s signature, the clauses in `lists` are not made available via the `revmod` module.

To illustrate the nature of the importing further, consider the modules in Figure 5.22. Here, `modA` simply declares that `p` is provable. Module `modB` imports `modA` and also contains the clause `p :- q`. Consider the question: is `p` provable from `modB`? Given the definition of importing, proving `p` leads to the attempt to prove `q` with `modA` added to the current context. But even with `modA` available, there is no way to prove `q`, so the attempt to prove `p` from `modB` fails. This illustrates the fact that importing is different from accumulating: if the `accumulate` keyword was used instead in `modB`, `p` would be provable from `modB`. If `modB` is elaborated into a formula, that formula would be `p :- (p => q)`,

```
module modA.
type p o.
p.

module modB.
import modA.
type q o.
p :- q.
```

Figure 5.22: The `modA` clause `p` is available only locally in `modB`.

or equivalently, `(p => q) => p`. Asking the query `p` from this formula yields the formula `((p => q) => p) => p`, which is the well-known formula called *Peirce's formula.* It is an example of a formula containing only implications that is classically and not intuitionistically provable. This supplies yet another example of that fact that if classical logic were used as the proof system for *fohh*, the natural notions of scoping described here would not be supported.

5.6.3 Declaring a local scope to type constructors

The `localkind` keyword can be used to provide a local scope to type constructors in a fashion similar to the the `local` keyword. The `localkind` keyword, however, does not correspond directly to an existential quantifier since the underlying logic used here does not have explicit quantification at the level of types. Instead, the `localkind` keyword simply declares that a certain type constructor should not appear in the signature of a module. It is an error, however, if some constant in a module's signature has a type that contains a type constructor given this limited scope.

Chapter 6

Simply typed λ-terms and formulas

When there are notions of scoping at the level of logical connectives, as there are in *fohh* and *fohh*$^+$, notions of scoping within terms are forced upon us. We argue such a connection between logic-level and term-level scoping in Section 7.1 when we identify λ-abstractions as a way to overcome a kind of computational incompleteness within *fohh*. Besides this "completion" role within *fohh*, λ-terms represent a valuable primitive concept for the manipulation of the syntax of programs and logics. Since λProlog has λ-terms built into it, an important area of application for it is a wide range of meta-programming tasks. In this Chapter we review some elementary facts about a logic that contains λ-abstractions.

6.1 Syntax for λ-terms and formulas

The simply typed λ-terms are built from typed versions of λ-abstraction and application. Application, denoted by juxtaposition, has already been used extensively in first-order terms and formulas. Abstraction is denoted by an infix backslash placed between the bound variable and the body of the abstraction. When reading an expression containing such a backslash symbol, the body of the abstraction goes as far to the right as is possible (given the presence of other parentheses and the end of the expression). For example, the λ-terms $\lambda x(f(g\lambda y(hxy))x)$, $\lambda f\lambda x(f(f(fx)))$, $\lambda x\lambda y\ x$ are written in λProlog as

```
x\f (g y\(h x y)) x
f\x\f (f (f x))
x\y\x
```

Notice that \ binds more tightly than does application and that it is right associative.

Typing for λ-terms is a modification of the typing judgment $\Sigma;\Gamma \Vdash_f t\colon\tau$ for first-order terms given in Figures 3.1 and 3.3. The rules in Figure 6.1 are used to define $\Sigma;\Gamma \Vdash t\colon\tau$: notice that the only difference between these rules and those in Figures 3.1 and 3.3 are that the two rules for typing universal and existential quantification are replaced with a rule for typing λ-abstraction instead. A consequence of this change is that the type τ in the judgment $\Sigma;\Gamma \Vdash t\colon\tau$ many no longer be a primitive type.

$$\frac{c\colon\sigma \in \Sigma_0 \quad \tau \lhd \sigma}{\Sigma;\Gamma \Vdash c\colon\tau} \qquad \frac{c\colon\sigma \in \Sigma \quad \tau \lhd \sigma}{\Sigma;\Gamma \Vdash c\colon\tau} \qquad \frac{c\colon\tau \in \Gamma}{\Sigma;\Gamma \Vdash c\colon\tau}$$

$$\frac{\Sigma;\Gamma \Vdash g\colon\tau_1 \to \tau_2 \quad \Sigma;\Gamma \Vdash t\colon\tau_1}{\Sigma;\Gamma \Vdash (g\ t)\colon\tau_2}$$

$$\frac{\Sigma;\Gamma, x\colon\tau \Vdash t\colon\sigma}{\Sigma;\Gamma \Vdash \lambda x\ t\colon\tau \to \sigma}$$

provide that x is not declared as a type or kind in Σ_0, Σ, or Γ.

$$\frac{\Sigma;\Gamma \Vdash B\colon\tau}{\Sigma;\Gamma \Vdash C\colon\tau}$$

provided B and C differ only in the names of bound variables.

Figure 6.1: Rules for typing λ-terms.

The simply typed λ-terms are used in λProlog to replace the separate notions of term and formula: in particular, a formula will be identified as a λ-term of type o. To accommodate universal and existential quantification, the global context Σ_0 will be assumed to have the constants constants $\forall$ and $\exists$ are both given the type $(\tau \to o) \to o$ (where τ is a type variable). In λProlog syntax, these would be declared with the typing declaration

```
type  pi, sigma     (A -> o) -> o.
```

The use of `pi` for $\forall$ and `sigma` for $\exists$ is taken directly from an early paper of Church [Chu40]. For an example of typing a quantified formula, assume that Σ is a signature such that the judgment $\Sigma;\Gamma, x : \sigma \Vdash B\colon o$ is provable (assume that σ is some type expression). Thus, B is a formula that may contain the variable x free. Using the typing rule for λ-abstractions we have $\Sigma;\Gamma \Vdash \lambda x\ B\colon\sigma \to o$. Since $\forall : (\tau \to o) \to o$ is contained in Σ_0, we can derive the judgments $\Sigma;\Gamma \Vdash \forall\colon(\sigma \to o) \to o$ and, finally, $\Sigma;\Gamma \Vdash \forall\lambda x\ B\colon o$. We shall adopt the convention that we shall write $\forall x\ B$ for $\forall\lambda x\ B$ and $\exists x\ B$ for $\exists\lambda x\ B$. For another example using λProlog syntax, consider the formula

```
pi y\ append (1::2::nil) y X.
```

the logical constant `pi` has type `(list int -> o) -> o` (in this context) and it is applied to the abstraction

```
y\ append (1::2::nil) y X.
```

which has type `list int -> o`.

λProlog contains only one term-level binding operation: the one for λ. All other binders, such as what we have seen for universal and existential quantification, as well as any that a programmer wishes to have within various data structures, must all be reduced to λ-abstraction. For a wide variety of binding operations, such a reduction is generally easy and accomplished essentially the way it was for quantification.

Given the typing discipline we have chosen, it is possible for logical connectives to appear within the scope of non-logical symbols. Higher-order programming (see Chapter 8) will require just such a possibility. For example, assume that we have the constant

```
type forevery   (A -> o) -> list A -> o.
```

This constant takes two arguments to become an atomic formula: the first argument is a predicate over one argument of type `A` and the second argument is a list of items of type `A`. An example of an atomic formula using this constant is

```
forevery (x\ x > 5, x < 9) (3::10::6::8::nil).
```

Notice that this atomic formula contains a subterm, `x > 5, x < 9`, that is itself a formula. The conjunction in this expression is a logical connective that occurs in the scope of the nonlogical symbol `forevery`.

6.2 Equality and λ-conversion

The intended interpretation of λ-abstraction and juxtaposition are the operations of function definition and function application. These intentions are partially formalized by the rules of *λ-conversion*. First extend the notions of substitution and free-for given in Section 5.2 to terms: we write $s[t/x]$ to denote the operation of replacing all free occurrences of a variable x in the term s by a term t of the same type as x. In performing this operation of replacement, there is the danger that the free variables of t become bound inadvertently. As in Section 5.2, we say that t is *free for x in s* if the free occurrences of x are not in the scope of an abstraction in s that binds a free variable of t. Since quantification is accounted for by λ-binding, these definitions extend those given in Section 5.2.

The rules of *α-conversion*, *β-conversion* and *η-conversion* are then, respectively, the following operations on terms:

- Replacing a subterm $\lambda x.s$ by $\lambda y.s[y/x]$, provided y is free for x in s and y is not free in s, is called *α-conversion*.

- Replacing a subterm $(\lambda x.s)t$ by $s[t/x]$, provided t is free for x in s, is called *β-reduction*. The converse operation is call *β-expansion*. These two operations are both called *β-conversion*.

- Replacing a subterm $\lambda x.(sx)$ by s, provided x is not free in s, is called *η-reduction*. The converse operation is call *η-expansion*. These two operations are both called *η-conversion*.

The rules above, collectively referred to as the λ-conversion rules, are used to define the following relations on terms. A term t *λ-converts* to s if there is a sequence of applications of α-conversion, β-conversion, and η-conversion that transforms t into s. This relation is clearly an equivalence and congruence. λProlog implements λ-conversion as its notion of equality. Thus, the following terms are equal within λProlog:

```
x\y\ f (g x) y
X\Y\ f (g X) Y
x\ f (g x)
x\y\ f ((u\v\v) (2 + 3) (g x)) y
```

The first two terms are related by α-conversions. The third term is the result of doing an η-reduction on the first. Finally, doing a β-reduction on the fourth term yields

```
x\y\ f ((v\v) (g x)) y
```

and doing a second β-conversion yields the first term. It is impossible for λProlog to tell these four terms apart. For this reason also, it is impossible for λProlog to determine the name of a bound variable since such a determination would change under α-conversion.

A term t is in *β-normal form* if it does not contain a β-redex (that is, a subterm of the form $(\lambda x.t_1)t_2$), and in *λ-normal form* if, in addition, it does not contain an η-redex (that is, a subterm of the form $\lambda x.(t_0 x)$ with x not occurring free in t_0). Sometimes, λ-normal is also called *$\beta\eta$-normal*.

For convenience, we extend $s[t/x]$ to the case where t is not necessarily free for x in s, by first picking a term, say s', that is α-convertible to s and for which t is free for x in s' and then set $s[t/x]$ to the result of substituting t for x in s'. Although the result is dependent on the actual formula s' picked, all results will, themselves, be α-convertible. In this sense, the extended definition of $s[t/x]$ is well defined.

6.3 The meta-theory of λ-conversion

If t is a λ-normal form and s is λ-convertible to t, then t is said to be a λ-normal form of s. It is well known that every simply typed λ-term has a λ-normal form and that this normal form is unique up to α-conversion (see [Bar84, HS86]). This normal form can be computed by repeatedly replacing subterms of the form $(\lambda x.s)t$ by $s[t/x]$ and subterms of the form $\lambda x.(sx)$ with s (provided x is not free in s). We denote the λ-normal form of s by $\lambda norm(s)$. Thus it is possible to determine if two λ-terms of the same type are equal (modulo λ-conversion) by first computing their λ-normal forms and then checking to see if these are equal up to α-conversion.

The computation of λ-normal forms is a rich operation, largely because of the presence of β-conversion. When $(\lambda x.s)t$ is replaced by $s[t/x]$, there may be many or no occurrences of x in s. If there are many, then $s[t/x]$ may contain many copies of the term t. Also, while both t and s may be λ-normal, the term $s[t/x]$ may not be in λ-normal form. Given this complexity, it is not surprising that λ-normalization can be used as a computing device itself. In particular, consider the simple signature

```
kind i  type.
```

The only closed, simply typed, λ-normal terms of second-order type `(i -> i) -> i -> i` are terms λ-convertible to one of the following terms

```
f\x\x     f\x\ f x     f\x\ f (f x)    f\x\ f (f (f x)) ... .
```

These terms are called the *Church numerals* and can be used to denote the non-negative integers: the series above represents the numbers 0, 1, 2, and 3.

It is an easy matter to compute the successor, addition, and multiplication functions for non-negative integers using this encoding of integers. The λ-term

```
n\f\x\ f (n f x)
```

can be used to compute successor. For example, the successor of the Church numeral for 3 is the λ-normal form of

```
(n\f\x\ f (n f x)) (f\x\ f (f (f x)))
```

which is the Church numeral 4, namely the term `f\x\ f (f (f (f x)))`. Similarly, addition and multiplication can be implemented using the two λ-terms `n\m\f\x\ n f (m f x)` and `n\m\f\x\(n (m f) x)`, respectively. Since a λProlog interpreter computes the λ-normal form of expressions prior to printing them out, we can compute the multiplication of 2 with 2, using Church numerals, by simply using the query

```
?- N = ((n\m\f\x\ n (m f) x)(f\x\ f (f x)) (f\x\ f (f x))).
N == f\x\ f (f (f (f x))).

yes
?-
```

The result is, of course, the Church numeral for 4. The functions over non-negative integers that can be computed in this fashion are limited to essentially polynomials. The presence of simple types restrict the computational power of such computations greatly.

Define the *size of a λ-term* to be the number of occurrences of application within the term. The previous example can be used to show that the size of a λ-term can be made polynomially larger by passing to its λ-normal form. For a more dramatic example of the increase in size, consider the series of λ-terms of which the following are the first four:

```
(g\e\e)             (e\f\ e (e f)) (f\x\ f (f x)).
(g\e\ g e)          (e\f\ e (e f)) (f\x\ f (f x)).
(g\e\ g (g e))      (e\f\ e (e f)) (f\x\ f (f x)).
(g\e\ g (g (g e)))  (e\f\ e (e f)) (f\x\ f (f x)).
```

Here the type of the bound variables are

```
x : i
f : i -> i
e : (i -> i) -> i -> i
g : ((i -> i) -> i -> i) -> (i -> i) -> i -> i.
```

The subterms that start with `g\e\` are a version of Church numeral but with the type `i` replaced with the type `(i -> i) -> i -> i`: for example, the term `g\e\(g (g e))` has the fourth-order type

```
(((i -> i) -> i -> i) -> (i -> i) -> i -> i) ->
 ((i -> i) -> i -> i) -> (i -> i) -> i -> i .
```

The n^{th} term of this series has a size $n+6$. The normal form of the first term in this series is `f\x\(f (f x))`, encoding the numeral 2, while the normal form for the second term is the encoding of the numeral 4. The third λ-term normalizes to

```
f\x\ f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f x)))))))))))))))
```

which encodes the numeral 16. The fourth λ-term normalizes to the encoding of the numeral 256. It is easy to show that the n^{th} term of this series has a λ-normal form that is of size

$$\left.2^{2^{2^{\cdot^{\cdot^{2}}}}}\right\}n$$

For small values of n, this increase in the size of terms is dramatic. Statman has further analyzed the computational costs of λ-conversion within the simply typed λ-calculus in [Sta79].

In most of the λProlog programs we shall consider, such blow-ups in the size of λ-normal terms is the exception. There are several reasons for this.

- When computing with structures such as integers, we make use of built-in integers instead of those constructed as above. Thus, computations on integers are efficient and familiar. Similarly, it is possible to specify other structures, such as binary trees, using simply typed λ-terms to construct them. We shall not do so, however, since we shall prefer the usual practice of introducing non-logical constants to represent such structures.

- In most examples, it is rare to need λ-abstraction within a term to bind a variable of order 1 or more. If all bound variables are of order 0 (primitive types), then β-conversion has the following property: if in the expression $(\lambda x.s)t$ both s and t are λ-normal, then the β-redex, $s[t/x]$ is in λ-normal form. In other words, although t may be duplicated into several different positions, no new β-redexes are created.

- A large number of λProlog programs actually belong to the sublanguage L_λ, which is described in Chapter 7. In this setting, all occurrences of β-conversion are instances of a simpler form of β-conversion called β_0-conversion: replace a subterm of the form $(\lambda x.s)x$ with s. Notice that in such a case, passage to the λ-normal form actually produces smaller terms.

As a further example of computing with λ-terms, consider the problem of representing and computing with the two element domain of Booleans. We can represent true with `x\y\x` and false with `x\y\y` both at type `i -> i -> i`. The conjunction and disjunction of booleans can then both be modelled as λ-terms that take the representation of two booleans and returns the representation of either their conjunction or disjunction. These two boolean-valued functions can be represented, respectively, as

```
r\s\x\y\ r (s x y) y          r\s\x\y\ r y (s x y)
```

each having type `(i -> i -> i) -> (i -> i -> i) -> i -> i -> i`. The conditional for type `i`, that is the function of type `(i -> i -> i) -> i -> i -> i` that returns the second argument if the first argument represents true and returns the third argument if the first argument represents false, can be represented simply as the the λ-term `r\x\y\ r x y`.

The declarative underpinning of functional programming can be viewed as a setting where programs are identified with λ-terms and computation is a process for computing λ-normal forms. Thus λProlog can be used to evaluate some functional programs, as some of the examples above illustrate. λProlog was never intended as a language that combines logic and functional programming, which is fortunate since it does a poor job of realizing such a combination. For example, while λProlog contains λ-terms, these terms cannot express general recursion: as we mentioned, all terms in λProlog have normal forms so it would be impossible to represent non-terminating programs using such terms. Of course, it is the role of the logic to provide for general computation. The intended role of λ-terms in computations will be discussed in the next chapter.

For an even more direct example of the weakness of the term language for capturing functions, assume that the given a signature contains the declarations

```
kind i             type.
type a, b, c, d    i.
```

there is no λ-term `F` of type `i -> i` such that `(F a)` reduces to `c` and `(F b)` reduces to `d`. There clearly is a functional program that would do such an operation, name one that can be written in Scheme syntax as

```
(lambda (x) (if (equal x a) c d)).
```

Thus, the λ-terms of λProlog are not adequate to express even simple functional programs.

6.4 Typing constants and variables

The process of converting a string representation of a program clause or a goal (say, as is found in a text file) into an actual logical formula is a complex process. In this section, we shall focus on how it is determined that a given token is a constant or a variable and what its type should be.

The following sequence of checks are used to determine which tokens within a program clause or goal are constants, free variables, or bound variables.

1. Is the given token in the scope of a binding occurrence for that token? If so, that token is a variable bound by the closest occurrence of a binding for it.

2. Does the token have an initial uppercase letter? If so, then this token is a free variable.

3. Otherwise, this token must be a constant declared in the relevant signature.

The types of constants and variables must also be determined. A signature declares the types of constants and context within a term determines the types of free and bound variables. It is also possible to attribute a typing to a subexpression of a formula by using a colon (see Subsection 3.10.3).

To describe how λProlog determines which tokens are variables and constants and at which types, we shall use the λProlog program in Figure 6.2. Here, we must be careful: to what extent can a polymorphic λProlog program be used to *define* how to resolve syntactic issues about λProlog? First, the program in Figure 6.2 is essentially a *fohc* program except for several (important) uses of cut. Second, while polymorphic typing is used in this example, it could be dropped entirely. In fact, the program here is essentially a Prolog program: typing in this example makes it easier to read. Thus, we are essentially using a Prolog program to describe syntactic aspects of λProlog.

The type `lptype` is used to encode types of λProlog: the constructor `arr` denotes the function type arrow and the constants `o` and `int` denote the type of the same name: remember that types and constants can have common names (Section 2.4). For example, the λProlog type `int -> int -> o` would be represented by the term `(arr int (arr int o))` of type `lptype`.

The type `preterm` is used for the result of parsing a string representation of a formula: the parser lexicalized the input and identified three special constructions: applications, abstraction (via \), and type attribution (via `:`). Tokens and these three constructions are denoted by the four constructors for `preterm`: `tok`, `ap`, `ab`, and `colon`, respectively.

The type `lpterm` is used to record the result of elaborating a `preterm` with the determination of which tokens denote constants and variables and at what type. This is done by converting each `preterm` of the form `tok String` into an `lpterm` of the form `v String Type` or `c String Type`: the first of which denotes a variable and the second, a constant. The value `Type` is then the inferred type for that variable or constant.

```
module lp_typing.
import pairs.

kind lptype        type.
type arr           lptype -> lptype -> lptype.
type o, int        lptype.

kind preterm       type.
type tok           string  -> preterm.
type ap, ab        preterm -> preterm -> preterm.
type colon         preterm -> lptype  -> preterm.

kind lpterm        type.
type c, v          string -> lptype -> lpterm.
type app, abs      lpterm -> lpterm -> lpterm.

type con_type      string  -> lptype -> o.
type ty_infer      list (pair string lptype) ->
                   list (pair string lptype) ->
                   list (pair string lptype) ->
                   preterm -> lptype -> lpterm -> o.

con_type ":-"     (arr o (arr o o)).
con_type "=>"     (arr o (arr o o)).
con_type ","      (arr o (arr o o)).
con_type ";"      (arr o (arr o o)).
con_type "pi"     (arr (arr A o) o).
con_type "sigma"  (arr (arr A o) o).

ty_infer Vs In Out (colon M Ty) Ty P :- ty_infer Vs In Out M Ty P.
ty_infer Vs In Out (ap M N) Ty (app P Q) :-
   ty_infer Vs In Mid M (arr Ty1 Ty) P, ty_infer Vs Mid Out N Ty1 Q.
ty_infer Vs In Out (ab B M) (arr Ty1 Ty) (abs (v Str Ty1) P) :-
    B = (tok Str), ty_infer ((pr Str Ty1)::Vs) In Out M Ty P.
ty_infer Vs In In (tok Str) Ty (v Str Ty) :- assoc Str Ty Vs, !.
ty_infer Vs In In (tok Str) Ty (v Str Ty) :- assoc Str Ty In, !.
ty_infer Vs In ((pr Str Ty)::In) (tok Str) Ty (v Str Ty) :-
   uppercase Str, !.
ty_infer Vs In In (tok Str) Ty (c Str Ty) :- con_type Str Ty.
```

Figure 6.2: A model of some simple type inference.

The binary predicate `con_type` is used to declare the types of all constants in which a given term is to be processed. Finally, the predicate `ty_infer` is used to elaborate a `preterm` into an `lpterm`. The first argument of `ty_infer` is used to accumulate the tokens that are to be matched to bound variables. The second and third arguments are used to accumulate the typing judgments for the free variables encountered. To illustrate `ty_infer`, consider the encoding of various constants that are contained in Figure 6.3. The predicate, `uppercase`, over strings is assumed to be defined elsewhere: it succeeds if and only if its argument is a string with an initial uppercase letter.

Consider attempting to prove the goal

```
?- test 3 Out Tm.
```

from the signature-program pair associated with the `lists_signature` pair in Figure 6.3. Here, `Out` will be bound to the association list that attributes types to the free variables of example 3, and `Tm` will be bound to the result of elaborating example 3. The binding for `Out` will be the list

```
(pr "Y" (list A)) :: nil
```

and `Tm` will be bound to the term in Figure 6.4. Consider also attempting to prove the query

```
?- ty_infer nil nil Out (ab (tok "x")(ap (tok "x") (tok "x"))) Tm.
```

This query attempts to give a typing for the expression written `x\(x x)`. This term has no typing and the above query will fail. Failure is caused by the so-called *occurs check* within unification: this check involves noticing that it is possible for a variable, say, x to unify with the term (assumed not to be the variable x, if and only if x does not occur free in t. Since the occurs check can be costly, many Prolog systems do not have the occurs check implemented. Thus we need to modify our claim above: the program in Figure 6.2 should be thought of as a Prolog program in which occurs check is employed.)

6.5 Higher-order logics

Many philosophical and mathematical concepts are naturally expressed in logic using quantification over functions and predicates. Leibniz's principle of equality, for example, states that two objects are to be taken as equal if they share the same properties; that is, $a = b$ can be defined using quantification of predicates as $\forall P[P(a) \equiv P(b)]$. Unfortunately, naive mixing of such quantification and logical connectives gives rise to inconsistent systems, containing, for example, Russell's paradox.

One approach for avoiding such a paradox is to adopt first-order logic and then to describe theories on top of it that capture various mathematical and philosophical structures. For example, Leibniz's principle can be captured in multisorted first-order logic by letting *app* be a first-order predicate symbol of arity two that denotes the application of a property to an individual. Intuitively, $app(P, x)$ means that the property P satisfies x or that the extension of the property P contains x. The first argument of *app* would range over the sort of properties while the second argument would range over the sort of individuals. Leibniz's property can then be expressed as the first-order expression $\forall P[app(P, a) \equiv app(P, b)]$, where appropriate axioms for describing *app* are specified. Set-theory is another first-order language that encodes such higher-order concepts using membership $\in$ as the (sort-less) converse of *app*.

```
module lists_signature.
import lp_typing.

% kind int      type.
% kind list     type -> type.
% type nil      list A.
% type ::       A -> list A -> list A.
% type append   list A -> list A -> list A -> o.

type int     lptype.
type list    lptype -> lptype.
con_type "nil"    (list A).
con_type "::"     (arr A (arr (list A) (list A))).
con_type "append" (arr (list A) (arr (list A) (arr (list A) o))).

type example         int -> preterm -> o.

example 1    %  append nil L L.
 (ap (ap (ap (tok "append") (tok "nil")) (tok "L")) (tok "L")).

example 2    %  append (X::L) K (X::M) :- append L K M.
  (ap (ap (tok ":-")
          (ap (ap (ap (tok "append") (ap (ap (tok "::") (tok "X"))
                         (tok "L"))) (tok "K"))
  (ap (ap (tok "::") (tok "X")) (tok "M"))))
      (ap (ap (ap (tok "append") (tok "L")) (tok "K")) (tok "M"))).

example 3    % sigma x\(append nil x x, append x x Y)).
(ap (tok "sigma") (ab (tok "x") (ap (ap (tok ",")
  (ap (ap (ap (tok "append") (tok "nil")) (tok "x")) (tok "x")))
   (ap (ap (ap (tok "append") (tok "x")) (tok "x")) (tok "Y"))))).

example 4    % sigma x\(append nil (x:i) x, append x x x)).
(ap (tok "sigma") (ab (tok "x") (ap (ap (tok ",")
   (ap (ap (ap (tok "append") (tok "nil"))
 (colon (tok "x") (list i))) (tok "x")))
           (ap (ap (ap (tok "append") (tok "x")) (tok "x"))
               (tok "x"))))).

type test   int -> list (pair string lptype) -> lpterm -> o.
test N Out Tm :- example N Pre, ty_infer nil nil Out Pre o Tm.
```

Figure 6.3: Example expressions for use in type inference of λProlog.

```
(app (c "sigma" (arr (arr (list A) o) o))
 (abs (v "x" (list A))
   (app  (app (c "," (arr o (arr o o)))
         (app (app
   (app (c "append" (arr (list A) (arr (list A) (arr (list A) o))))
(c "nil" (list A)))
   (v "x" (list A)))
  (v "x" (list A))))
    (app (app
  (app (c "append" (arr (list A) (arr (list A) (arr (list A) o))))
   (v "x" (list A))) (v "x" (list A))) (v "Y" (list A)))))).
```

Figure 6.4: The elaboration of a `preterm` into an `lpterm`.

Higher-order logics are another approach to avoiding the paradoxes. In such logics, quantification over functions and predicates is directly available and is not achieved via an encoding. Types, however, are generally introduced to restrict the nature of quantification. There are several typing schemes that have been employed. One approach types first-order individuals with ι, sets of individuals with $\langle\iota\rangle$, sets of pairs of individuals with $\langle\iota\iota\rangle$, sets of sets of individuals with $\langle\langle\iota\rangle\rangle$, etc. Such a typing scheme does not provide types for function symbols: often higher-order logics have been used to formalize mathematics and in that setting, functions can be represented by their graphs, *i.e.* certain kinds of sets of ordered pairs.

6.5.1 Several senses to "higher-order logic"

The term "higher-order logic" is often used in different senses in the literature of computational logic, a fact that leads to some confusions. It is possible to identify at least three different readings for this term.

1. Philosophers of mathematics usually divide logic into first-order logic and second-order logic. The latter is a formal basis for all of mathematics and, as a consequence of Gödel's first incompleteness theorem [Goe65], cannot be recursively axiomatized. Thus, higher-order logic in this sense is basically a model theoretic study [Sha85].

2. To a proof theorist, all logics correspond to formal systems that are recursively presented and a higher-order logic is no different. The main distinction between a higher-order and a first-order logic is the presence in the former of predicate variables and comprehension, *i.e.*, the ability to form abstractions over formula expressions. Cut-elimination proofs for higher-order logics differ qualitatively from those for first-order logic in that they need techniques such as Girard's "candidats de réductibilité," whereas proofs in first-order logics can generally be done by induction [GTL89]. Semantic arguments can be employed in this setting, but general models (including non-standard models) in the sense of Henkin [Hen50] must be considered.

3. To many working in automated deduction, higher-order logic refers to any computational logic that contains typed λ-terms and/or variables of some higher-order type, although not necessarily of predicate type. Occasionally, such a logic may incorporate

the rules of λ-conversion, and then unification of expressions would have to be carried out relative to these rules.

Clearly, it is not sensible to base a programming language on a higher-order logic in the first sense. λProlog is higher-order in the second and third senses. Notice that these two senses are distinct. That is, a logic can be higher-order in the second sense but not in the third: there have been proposals for adding forms of predicate quantification to computational logics that do not use λ-terms and in which the equality of expressions continues to be based on the identity relation (see, for example, [Wad91]). Conversely, a logic that is higher-order in the third sense may well not permit a quantification over predicates and, thus, may not be higher-order in the second sense. An example of this kind is the specification logic that is used by the Isabelle proof system [Pau90].

6.5.2 The Simple Theory of Types

Church in [Chu40] introduced a typing system containing function types and a special type for booleans used to represent formulas and predicates. The logic introduced in [Chu40], called the *Simple Theory of Types* also contains simply typed λ-term and λ-conversion.

The logical foundation of λProlog is based on the Simple Theory of Types, partly because it is an elegant and succinct logic for incorporating higher-order features and partially because it has a well developed meta-theory (an overview of which is given below). The syntax of λProlog terms and formulas is directly inspired by the syntax of term and formulas in [Chu40]. The three main differences between the logic employed in λProlog and the Simple Theory of Types are (i) the choice of logical connectives to take as primitive, (ii) reliance on intuitionistic logic in λProlog instead of classical logic, and (iii) the structure of types: the Simple Theory of Types contains neither type constructors nor type variables. These have been added to λProlog to provide for flexible polymorphic typing of some programs. The use of `o` as the type of formulas and the use of `pi` and `sigma` as quantifiers comes directly from [Chu40].

6.5.3 Semantics for the Simple Theory of Types

There are many ways to interpret higher-order logic with category theory providing one of the richest possibilities [LS86]. Here we outline an early approach used by Henkin [Hen50]. Higher-order logic can be interpreted over a pair $\langle\{\mathcal{D}_\sigma\}_\sigma, \mathcal{J}\rangle$, where σ ranges over all types. The set $\mathcal{D}_\sigma$ is the collection of all semantic values of type σ and $\mathcal{J}$ maps (logical and non-logical) constants to particular objects in their respectively typed domain. Thus, $\mathcal{D}_\iota$ is the set of all first-order individuals, $\mathcal{D}_o$ is the set $\{\mathit{true}, \mathit{false}\}$, $\mathcal{D}_{\iota\to o}$ is the set of characteristic functions of subsets of $\mathcal{D}_\iota$, etc. The mapping $\mathcal{J}$ must send the logical constants to their intended meanings; for example, $\mathcal{J}(\wedge)$ is the curried function that returns *true* when its arguments are both *true*, and *false* otherwise. A *standard model* is one in which the set $\mathcal{D}_{\sigma\to\tau}$ is the set of *all* functions from $\mathcal{D}_\sigma$ to $\mathcal{D}_\tau$. Such models are completely determined by supplying only $\mathcal{D}_\iota$ and $\mathcal{J}$. If $\mathcal{D}_\iota$ is denumerably infinite, then $\mathcal{D}_{\iota\to o}$ is uncountable: standard models can be large. In fact, if $\mathcal{D}_\iota$ is infinite, it is possible to build a model of Peano's axioms for the non-negative integers. As a corollary of Gödel's incompleteness theorem, the set of true formulas in such a standard model is not recursively axiomatizable; that is, there is no theorem proving procedure that could (even theoretically) uncover all true formulas.

A key property of a higher-logic logics is whether or not it is *extensional*, that is, whether or not the formula

$$\forall_\tau x(pz \equiv qz) \supset p = q$$

holds for all predicates p and q. This formula holds if whenever two predicates have the same truth values on the same arguments implies that they are equal predicates. The logic underlying λProlog is not extensional: we wish to limit equality to be more syntactic. Equality should be solvable by unification and not by checking the equivalence of two predicates on all of their arguments. Another form of extensionality, namely,

$$\forall_\tau x(fz = gz) \supset f = g$$

does hold for the logic underlying λProlog. In the context of unification, this requires the addition of the η-conversion rule.

It is possible to interpret terms and formulas over domains other than the standard one. Henkin [Hen50] developed a notion of *general model* that included non-standard as well as standard models. In the general setting, it is possible for $\mathcal{D}_{\sigma\to\tau}$ to be a proper subset of the set of all functions from $\mathcal{D}_\sigma$ to $\mathcal{D}_\tau$ as long as there are enough functions to properly interpret all expressions of the language of type $\sigma \to \tau$. Henkin's completeness result is then: a higher-order formula is valid in all general models if and only if it has a proof (possibly involving the axiom of extensionality). Thus, considered from the point of view of general models, higher-order logic with extensionality can be given a completeness result and this avoids the negative result due to Gödel's incompleteness results. The cost of this completeness result is giving up the desire to model only the standard model. Since the standard model is uncountable and includes functions and predicates that are not computable, such a cost is acceptable in many areas of computer science.

The focused semantics above is on classical logic versions of the Simple Theory of Types. Intuitionistic variations of such models, based on Kripke models [MM91, Mil92a] or topoi [LS86], might be employed to study the more general versions of higher-order hereditary Harrop formulas presented in Chapter 9.

6.5.4 Proof theory for the Simple Theory of Types

In [Chu40], a series of axioms were presented to describe the Simple Theory of Types. The first six axioms (which do not include extensionality) describe a logic that extends first-order logic by permitting quantification at all types and by replacing first-order terms by simply typed λ-terms modulo β and η-conversion. Many standard proof-theoretic results – such as cut-elimination [Tak67] and [Gir86], unification [Hue75], resolution [And71], and Skolemization and Herbrand's Theorem [Mil87b] – have been formulated for this fragment. Using these results as a foundation, it is possible to write theorem provers for this fragment of higher-order logic [ACMP84]. The presence of predicate quantification, however, makes theorem proving particularly challenging. In first-order logic, the result of substituting into an expression does not change its logical structure. In the higher-order setting, however, universal instantiation may increase the number of logical connectives and quantifiers in a formulas. For example, if P, in the expression $[\ldots \wedge (Pc) \wedge \ldots]$, is substituted with $\lambda x[\exists w(Axw \supset Bww)]$ then the resulting expression (after doing λ-conversion) would be $[\ldots \wedge [\exists w(Acw \supset Bww)] \wedge \ldots]$, which has one new occurrence each of a quantifier and logical connective. Theorem provers in first-order logic need to only consider substitutions that are generated by the unification of atomic formulas. Since logical connectives within substitutions are possible in higher-order logic, as this example shows, atomic formula unification does not suggest enough substitution terms.

The textbook [And86] and the handbook article [Lei94] are good sources for getting more information on higher-order logic.

Chapter 7

Computing with λ-terms

In this chapter we describe a fragment of λProlog in which λ-terms first appear within terms. The logic underlying this fragment, called L_λ, extends *fohh* with an elementary treatment of simply typed λ-terms. L_λ appears to be the weakest extension to *fohh* that contains a principled and flexible treatment of λ-terms. Many of the interesting meta-programming features of λProlog appear first within this fragment. Also, unification within L_λ is decidable (in fact, linear) and when two terms are unifiable, they have a most general unifier. When we consider a richer setting for λ-terms in Chapter 8, unification will become a radically harder problem involving numerous complications to the operational behavior of programs. Fortunately, a large number of λProlog programs actually fall within L_λ, and those programs that do not lie in this weaker language can be translated simply into L_λ programs.

7.1 Discharging a constant from a term

Let Σ be the signature for the `smlists` module given in Figure 4.1. Consider the problem of finding a substitution term over Σ for the variable `X` so that the query

```
?- pi y\ append (1::2::nil) y X.
```

can be proved. The first step in building a proof of this goal is to introduce a constant, say `k` of type `list int`, that is not declared in Σ and then attempting a proof of

```
append (1::2::nil) k X.
```

After backchaining thrice on the definition of `append`, we find that this goal is provable if and only if `X` is instantiated with `(1::2::k)`. This is not possible, however, since `X` can be instantiated with terms over Σ, but `k` was picked not to be in Σ. Such a failure here is quite sensible since the value of `X` should be independent of the choice of the constant used to instantiate `pi y\`. It might be desirable, however, to have this computation succeed if this particular choice of constant could be abstracted away. That is, an interesting value is computed here but it cannot be used since it is not well defined. Admitting λ-abstraction into this logic provides a representation of such a value.

Now consider proving the goal

```
?- pi y\ append (1::2::nil) y (H y).
```

where `H` is a functional variable that may be instantiated with a λ-term whose constants are again from the set Σ. Here, `H` is a variable of type `list int -> list int`. Assume that `pi y\` is again instantiated with the constant `k`. This time, `(H k)` must equal `(1::2::k)`. There are two simply-typed λ-terms (up to λ-conversion) that when substituted for `H` into `(H k)` and then λ-normalized yields `(1::2::k)`, namely, the terms `(w\ 1::2::k)` and `(w\ 1::2::w)`. Since `H` cannot contain `k` free, only the second of these possible substitutions will succeed in being a legal solution for this goal. In a sense, the λ-term `(w\ 1::2::w)` is the result of *discharging* the constant `k` from the term `(1::2::k)`. Notice, however, that discharging a first-order constant from a first-order term is now a λ-term: it can be used to instantiate a function variable.

The higher-order variable `H` in this example is restricted in such a way that when it is involved in a solvable unification problem, there is a single, most general unifier for it. We shall define L_λ so that this is the only kind of non-first-order unification problem that can occur. All such uses of a higher-order variable in unification will be associated with discharging a constant from a term.

It is interesting to note that the query

```
?- pi y\ append y (1::2::nil) (H y)
```

is also not provable in L_λ (nor in full λProlog). A description of a function that satisfies this query requires recursion over list structures and such recursion is not available *within the terms* of L_λ. Consider how the function that satisfies this query can be described in Scheme.

```
(define (H y)
   (if (null y) (cons 1 (cons 2 nil))
                (cons (car y) (H (cdr y)))))
```

While this definition can be considered a λ-term, that λ-term contains both recursion and a conditional, neither of these are built into the theory of λ-terms within λProlog. (See related comments at the end of Section 6.3.) List induction could be employed to prove the universally quantified query displayed above, but that is beyond the scope of λProlog's interpreter.

7.2 The syntax for L_λ

Let Σ be a signature and let B be a Σ-formula. A bound variable occurrence in B is *essentially universal* if it is bound by a positive occurrence of a universal quantifier, by an negative occurrence of an existential quantifier, or by a (term-level) λ-abstraction; otherwise, it is *essentially existential*; that is, it is either bound by a negative universal quantifier or a positive existential quantifier. (See Section 3.5 for the definition of postive and negative occurrences of subformulas.) Within the context of logic programming, it is essentially existential bound variables that can be instantiated with general terms (via logic variables and unification) while essentially universal bound variables can be instantiated with only scoped constants. The "polarity" of bound variable occurrences defined here is with respect to goal formulas: to get the proper classification of bound variables within a program clause, dualize the above definition.

The logic programming language L_λ is the result of extending *fohh* by allowing all quantifiers to be over any non-predicate type while also imposing the following restriction on variables:

> for every subterm in B of the form $(x\ y_1 \ldots y_n)$ $(n \geq 0)$ where x is essentially existentially quantified in B, it must be the case that $y_1, \ldots, y_n$ is a list of distinct variables that are essentially universally quantified within the scope of the binding for x.

This restriction ensures that if x is ever instantiated by some term, say t, then the only β-redexes that appear after that substitution are of the form $(t\ y_1 \ldots y_n)$ where the variables $y_1, \ldots, y_n$ are not free in t. Using α and η-conversions, we can assume that t is of the form $\lambda y_1 \ldots \lambda y_n.t'$. Thus, β-reduction simply reduces $(\lambda y_1 \ldots \lambda y_n.t')y_1 \ldots y_n$ to t'. Let β_0-conversion be that subcase of β-conversion that relates redexes of the form $(\lambda x.s)x$ with s.

All goals and program clauses in *fohh* are also goals and program clauses in L_λ. If the constant p has type $i \to o$ and f has type $i \to i$ then the formula

$$\forall_{i\to i} x \forall_i y (p\ (x\ y) \supset p\ (f\ y))$$

is an example of a goal in L_λ but not a program clause. As a program clause of L_λ, it has a subterm occurrence $(x\ y)$ where both x and y are essentially existential, and this is ruled out by L_λ restriction on variables.

Notice that the core of L_λ is the same as the core of *fohh*. Later we shall present the logic *hohh* that contains L_λ in its core.

7.3 Simplifying quantifier alternation with raising

It was mentioned in Section 5.1 that existential quantification cannot be removed from program clauses using the second presentation of *fohh* without the introduction of extra predicates and clauses. Given the presence of higher-order variables, there is a logical transformation that allows existential quantifiers in the body of clauses to be moved to a larger scope. Consider the two goals

$$\forall y \exists x.G \qquad \text{and} \qquad \exists h \forall y.G[hy/x]$$

where the bound variables x, y, h have types τ, σ, and $\sigma \to \tau$, respectively. It is easy to show that one of these goals can be proved if and only if the other goal can be proved. Assume that the first goal is proved by substituting the new constant c for y and the term t for x. Here, t may contain occurrences of c. Then the second goal is proved by substituting $\lambda c.t$ for h and c for y. Notice that c is not free in $\lambda c.t$. For the converse, assume that the second goal is proved by substituting the term s for h and the new constant c for y. In this case, c cannot be free in s. The first goal is then proved by substituting c for y and substituting the term $(s\ c)$ for x.

In the two goals above, the second is said to be the result of *raising* the first goal at $\exists x$. Raising can increase the scope of an existential quantifier. Consider again the following program clause taken from Section 5.1.

```
p X :- pi y\ sigma Z\ q X y Z.
```

By raising on the quantifier for `Z` in the body of this clause, we get the clause

```
p X :- sigma H\ pi y\ q X y (H y).
```

which is equivalent to the clause

```
p X :- pi y\ q X y (H y).
```

This clause and the original, although not logically equivalent, do, in fact, prove the same goal formulas.

As this example shows, raising can often be used to simplify quantifier alternation. This technique is, in a sense, dual to the familiar technique of *Skolemization* that is often used in automated reasoning systems. To illustrate this difference, first recall how Skolemization is used. If $\forall x \exists y.D$ is an assumption, the result of Skolemizing this pair of quantifiers is the formula $\forall x.D[fx/y]$, where f is a new function constant, called a *skolem constant*. (Assume that x, y, f have types σ, τ, and $\sigma \to \tau$, respectively.) Since goals are the dual of assumptions, the Skolemization of a goal formula is the result of switching around the quantifiers. That is, the goal $\exists x \forall y.G$ is Skolemized to form $\exists x.G[fx/y]$, where again f is a new function constant. Since we are able to quantify at higher-types, it is possible to phrase this last goal and the restriction on f as simply the goal $\forall f \exists x.G[fx/y]$. It is possible to show that the correctness of Skolemization must be qualified, and its proof is more complex than that used for raising. Skolemization is dual to raising in two senses. First, Skolemization moves an existential quantifier to a larger scope while raising moves a universal quantifier to a larger scope. Second, Skolemization causes an introduction of a new constant of type $\sigma \to \tau$ instead of type σ, while raising causes an introduction of a new free variable of type $\sigma \to \tau$ instead of type σ. Skolemization is not used further here: see [Mil92b] for more about both raising and Skolemization.

7.4 Specifying an object-logic

Much of the formal and technical detail of a complete specification of L_λ [Mil91] is caused by needing to keep track of bound variable names and scope. Since all these details are formally incorporated into L_λ, programs written using L_λ should be relieved of much of the need to deal with such details. The following examples attempt to illustrate this point.

Three meta-programs — substitution, Horn clause interpretation, and the computation of prenex normal forms — are presented in this section and all compute with the same first-order, object-logic which is specified in the module `ot_logic.mod` in Figure 7.1. This object-logic contains two primitive types, one for typing object-level terms (the type `term`) and one for typing object-level formulas (the type `form`). It also contains logical constants for truth, conjunction, disjunction, implication, and universal and existential quantification, which are denoted by the non-logical, meta-level constants `truth`, `and`, `or`, `imp`, `all`, and `some`. The object-logic contains just five non-logical constants: an individual constant, `a`, a function symbol of one argument, `f`, and another of two arguments, `g`, and a predicate symbol of one argument, `p`, and another of two arguments `q`. Terms over this signature of type `form` denote object-logic formulas and those of type `term` denote object-logic terms. The predicate `atom` is provable of a meta-level term if and only if that term denotes an object-level atomic formula, that is, a formula that is not a top-level logical connective. The predicate `quant_free` is true if its argument denotes a quantifier-free object-level formula.

The module in Figure 7.2 contains the definition of several λProlog (meta-level) predicates that determine various classes of object-level formulas. Most of the clauses in that module are Horn clauses except for the few that deal with quantification at the object-level.

```
module ot_logic.
kind  term, form      type.

type truth, false    form.
type neg             form -> form.
type or, and, imp    form -> form -> form.
type all, some       (term -> form) -> form.

type atom, quant_free      form -> o.

quant_free truth.
quant_free false.
quant_free A         :- atom A.
quant_free (neg B)   :- quant_free B.
quant_free (and B C) &
quant_free (or  B C) &
quant_free (imp B C) :- quant_free B, quant_free C.


module ot_constants.
accumulate ot_logic.

type  a    term.
type  f    term -> term.
type  g    term -> term -> term.
type  p    term -> form.
type  q    term -> term -> form.

atom (p X).
atom (q X Y).
```

Figure 7.1: Specification of a first-order object-logic.

```
module formula_classes.
accumulate ot_logic.

type fohcG, fohcD, fohhG, fohhD   form -> o.

fohcG truth.
fohcG A         :- atom A.
fohcG (and B C) &
fohcG (or  B C) :- fohcG B, fohcG C.
fohcG (some B)  :- pi x\ fohcG (B x).

fohcD A           :- atom A.
fohcD (imp G D)   :- fohcG G,  fohcD D.
fohcD (and D1 D2) :- fohcD D1, fohcD D2.
fohcD (all D)     :- pi x\ fohcD (D x).

fohhG truth.
fohhG A         :- atom A.
fohhG (and B C) &
fohhG (or  B C) :- fohhG B, fohhG C.
fohhG (imp D G) :- fohhD D, fohhG G.
fohhG (some B)  &
fohhG (all B)   :- pi x\ fohhG (B x).

fohhD A           :- atom A.
fohhD (and D1 D2) :- fohhD D1, fohhD D2.
fohhD (imp G D)   :- fohhG G,  fohhD D.
fohhD (all D)     :- pi x\ fohhD (D x).
```

Figure 7.2: Specifying various syntactic classes of object-level formulas.

The meaning for most of the predicates specified in that module should be clear. The predicates `fohcG` and `fohcD` can be used to determine if an object-level formula is either a goal or program clause within *fohc* (using Definition 3.2 of Section 3.4). Similarly, the predicates `fohhG` and `fohhD` can be used to determine if an object-level formula is either a goal or program clause within *fohh* (using Definition 5.2 of Section 5.1).

To illustrate how object-level quantification is handled within L_λ, consider using the `fohcD` predicate to verify that the meta-level term

```
all u\ all v\ imp (and (q v a) (q a u)) (p u).
```

denotes an object-level Horn clause. To prove the query

```
?- fohcD (all u\ all v\ imp (and (q v a) (q a u)) (p u)).
```

the last of the `fohcD` clauses causes the goal

```
pi x\ fohcD ((u\ all v\ imp (and (q v a) (q a u)) (p u)) x).
```

to be attempted. Here, the higher-order, meta-level variable `D` of type `term -> form` is bound to the abstraction

```
u\ all v\ imp (and (q v a) (q a u)) (p u).
```

By doing a β_0 reduction on the above goal (done automatically by a λProlog interpreter) the above goal reduces to

```
pi x\ fohcD (all v\ imp (and (q v a) (q a x)) (p x)).
```

Notice now that the variable `u`, which denoted an object-level universal quantification, is now replaced with the meta-level, universally quantified variable `x`. Proceeding to establish this goal, λProlog will now pick a new constant, say `d`, and attempt to prove the following instance of this goal

```
fohcD (all v\ imp (and (q v a) (q a d)) (p d))).
```

In a similar manner, this goal will reduce to

```
fohcD (imp (and (q e a) (q a d)) (p d))).
```

where the new constant `e` is added to the current signature. Processing of this goal now follows standard first-order Horn clause reasoning as long as this extended signature is used.

Consider unifying the following two λ-terms

```
u\ all v\ imp (q a v) (p u)            x\B.
```

Here, the only free variable is `B`, which is of type `form`. Notice that in the term on the left, the outer-most bound variable `u` occurs within the body of the abstraction. On the other hand, every instantiation of `B` in `x\B` yields a term for which the outer-most bound variable is a vacuous abstraction. This results from the fact that substitutions are not allowed to capture free variables. For example, if `B` is substituted with the term `(p x)`, the result would be a term equal to `(x1\ p x)`: that is, the outer-most variable would need to be renamed prior to instantiating `B`. Thus all instances of `x\B` are vacuous abstractions, and as a result, the above two terms cannot be unified. On the other hand, the two terms

```
module rem_vacuous.
accumulate ot_logic.

type vacuous    form -> o.
type rem_vac    form -> form -> o.

vacuous (all x\B).
vacuous (some x\B).

rem_vac A A :- atom A.
rem_vac (and B1 B2) (and C1 C2) &
rem_vac (or  B1 B2) (or  C1 C2) &
rem_vac (imp B1 B2) (imp C1 C2) :- rem_vac B1 C1, rem_vac B2 C2.
rem_vac (some x\B) C & rem_vac (all x\B) C :- !, rem_vac B  C.
rem_vac (some B) (some C) &
rem_vac (all B)  (all C)  :- pi x\ rem_vac (B x) (C x).
```

Figure 7.3: Removing vacuous quantifiers from formulas.

```
u\ all v\ imp (q a v) (p v)          x\B.
```

can be unified by substituting the term `(all v\ imp (q a v) (p v))` for `B`.

Using these observations about vacuous abstractions, we can write programs that can recognize vacuous abstractions in an object-logic. For example, the predicate `vacuous` specified in Figure 7.3 succeeds with a closed argument of type `form` if and only if its argument is an object-level, vacuously quantified formula. Similarly, the procedure `rem_vac` can remove vacuous quantifiers from an object-level formula: for example, the goal

```
?- rem_vac (some x\ some y\ all x\imp (p x) (p x)) B.
```

will succeed once in λProlog with the answer substitution for `B` equal to

```
(all x\ imp (p x) (p x)).
```

If the cuts were removed from this program, this query has 4 solutions, in which each of the vacuous object-level quantifiers may or may not be removed.

For another example of manipulating object-level formulas, consider the computation of two kinds of normal form presentations of formulas in classical logic: negation normal form and prenex normal form. A formula is in *negation normal form* if it contains no occurrences of implications and every occurrence of a negation has only an atom in its scope. A formula is in *prenex normal form* if no quantifier occurrence is in the scope of any propositional connectives: that is, quantifiers occur at the outer-most level only. It is a theorem of first-order classical logic (not of intuitionistic logic) that a formula is equivalent to one in negation normal form and to one in prenex normal form and to one in both negation normal and prenex normal forms.

The computation of a formula in negation normal form that is equivalent to a given formula can be done using the following classical equivalences.

$$B_1 \supset B_2 \quad \equiv \quad \neg B_1 \vee B_2$$

```
module nnf.
accumulate ot_logic.

type nnf   form -> form -> o.

nnf A A & nnf (neg A) (neg A) :- atom A.
nnf (neg (neg B)) D           :- nnf B D.
nnf (neg (and B C)) (or D E)  &
nnf (neg (or B C)) (and D E)  :- nnf (neg B) D, nnf (neg C) E.
nnf (imp B C) (or D E)        :- nnf (neg B) D, nnf C E.
nnf (or B C)  (or D E)        &
nnf (and B C) (and D E)       :- nnf B D, nnf C E.

nnf (neg (all B)) (some D) &
nnf (neg (some B)) (all D) :- pi x\ nnf (neg (B x)) (D x).
nnf (all B)  (all D)       &
nnf (some B) (some D)      :- pi x\ nnf (B x) (D x).
```

Figure 7.4: Relating a formula to its negation normal form.

$$\begin{array}{rcl} \neg\neg B & \equiv & B \\ \neg(B_1 \wedge B_2) & \equiv & \neg B_1 \vee \neg B_2 \\ \neg(B_1 \vee B_2) & \equiv & \neg B_1 \wedge \neg B_2 \\ \neg\forall x\ B & \equiv & \exists x\ \neg B \\ \neg\exists x\ B & \equiv & \forall x\ \neg B \end{array}$$

These equivalences can be used directly to specify the `nnf` relationship of Figure 7.4. Since no two heads of the clauses for `nnf` overlap, it is easy to see that the relation `nnf` specifies a partial function. A simple argument by induction shows that the function represented is total.

Finally, consider computing prenex normal forms of formulas that are in negation normal form (this restriction is only to shorten the specification of this relation). This computation makes use of the following equivalences of classical logic.

$$\begin{array}{rcl} (\forall x\ B_1) \wedge (\forall x\ B_2) & \equiv & \forall x (B_1 \wedge B_2) \\ (\exists x\ B_1) \vee (\exists x\ B_2) & \equiv & \exists x (B_1 \vee B_2) \\ B_1 \wedge (\forall x\ B_2) & \equiv & \forall x\ (B_1 \wedge B_2) \\ (\forall x\ B_2) \wedge B_1 & \equiv & \forall x\ (B_2 \wedge B_1) \\ B_1 \wedge (\exists x\ B_2) & \equiv & \exists x\ (B_1 \wedge B_2) \\ (\exists x\ B_2) \wedge B_1 & \equiv & \exists x\ (B_2 \wedge B_1) \\ B_1 \vee (\forall x\ B_2) & \equiv & \forall x\ (B_1 \vee B_2) \\ (\forall x\ B_2) \vee B_1 & \equiv & \forall x\ (B_2 \vee B_1) \\ B_1 \vee (\exists x\ B_2) & \equiv & \exists x\ (B_1 \vee B_2) \\ (\exists x\ B_2) \vee B_1 & \equiv & \exists x\ (B_2 \vee B_1) \end{array}$$

Incorporation of the forward equivalences as rewriting rules is not as straightforward as it was for computing negation normal forms. A suitable specification is given in Figure 7.5. Notice that an auxiliary and local predicate is used to merge two formulas that are already in prenex normal form. Given a specification of `prenex`, the unique prenex normal form of the formula

```
imp (all x\ and (p x) (and (all y\ q x y) (p (f x)))) (p a)
```

is the formula

```
some x\ some y\ imp (and (p x) (and (q x y) (p (f x)))) (p a).
```

In general, the predicate `prenex` is not functional: that is, a single formula can have multiple prenex normal forms to which it is equivalent using the above equivalences. For example, the λProlog query

```
?- prenex (and (all x\ q x x) (all z\ all y\ q z y)) P
```

will generate the following five answer substitutions for `P` (in this order).

```
all z\ all y\ and (q z z) (q z y)
all x\ all z\ all y\ and (q x x) (q z y)
all z\ all x\ and (q x x) (q z x)
all z\ all x\ all y\ and (q x x) (q z y)
all z\ all y\ all x\ and (q x x) (q z y)
```

7.5 Implementing object-level substitution

The preceding section contained examples of manipulating first-order formulas as data values. One manipulation that was not demonstrated in that section was doing object-level substitution. For example, we have not yet seen how to instantiate an object-level universal quantifier with an object-level term. We address such instantiation in this section.

Equality and substitution for object-level terms and formulas can be specified by using the `copy`-clauses contained in Figure 7.6. These clauses can be derived directly from the object-level signature using the following function. Let $[\![t, s : \tau]\!]^{\pm}$ be a formula defined by recursion on the structure of the type τ, which is assumed to be built only from the base types `term` and `form`, with the following clauses:

$$
\begin{aligned}
[\![t, s : \mathtt{term}]\!]^{+} = [\![t, s : \mathtt{term}]\!]^{-} &= \quad \mathtt{copyterm}\ t\ s \\
[\![t, s : \mathtt{form}]\!]^{+} = [\![t, s : \mathtt{form}]\!]^{-} &= \quad \mathtt{copyform}\ t\ s \\
[\![t, s : \tau \to \sigma]\!]^{+} &= \quad \forall x([\![x, x : \tau]\!]^{-} \supset [\![t\ x, s\ x : \sigma]\!]^{+}) \\
[\![t, s : \tau \to \sigma]\!]^{-} &= \quad \forall x \forall y([\![x, y : \tau]\!]^{+} \supset [\![t\ x, s\ y : \sigma]\!]^{-})
\end{aligned}
$$

The `copy`-clauses displayed in Figure 7.6 are essentially those clauses that are equal to $[\![c, c : \tau]\!]^{-}$ where the signature for representing the object-logic contains $c : \tau$.

These clauses are given the rather operational name `copy` since they are used to specify both equality and substitution for the object-level logic. That is, `(copyterm t s)` is provable from these clauses if and only if `t` and `s` are the same term. Similarly `(copyform t s)`

```
module prenex.
import ot_logic.

type  prenex       form -> form -> o.
local merge        form -> form -> o.

prenex A A &
prenex (neg A)  (neg A)  :- atom A.
prenex (and B C) D  :- prenex B U, prenex C V, merge (and U V) D.
prenex (or  B C) D  :- prenex B U, prenex C V, merge (or  U V) D.
prenex (all B)  (all D)  &
prenex (some B) (some D) :- pi x\ prenex (B x) (D x).

merge (and (all B) (all C)) (all D) :- pi x\ merge (and (B x) (C x)) (D x).
merge (and (all B) C)  (all D)      &
merge (and (some B) C) (some D)     :- pi x\ merge (and (B x) C) (D x).
merge (and B (all C))  (all D)      &
merge (and B (some C)) (some D)     :- pi x\ merge (and B (C x)) (D x).

merge (or (some B) (some C)) (some D) :- pi x\ merge (or (B x) (C x)) (D x).
merge (or (some B) C) (some D) &
merge (or (all B) C)  (all D)  :- pi x\ merge (or (B x) C) (D x).
merge (or B (some C)) (some D) &
merge (or B (all C))  (all D)  :- pi x\ merge (or B (C x)) (D x).

merge B B :- quant_free B.
```

Figure 7.5: Specification of the prenex normal relation for formulas in negation normal form.

```
module ot_subst.
accumulate ot_constants.

type   copyterm    term -> term -> o.
type   copyform    form -> form -> o.

copyterm a a.
copyterm (f X) (f U)         :- copyterm X U.
copyterm (g X Y) (g U V)     :- copyterm X U, copyterm Y V.

copyform truth truth.
copyform false false.
copyform (neg B)   (neg D)   :- copyform B D.
copyform (and B C) (and D E) &
copyform (or  B C) (or  D E) &
copyform (imp B C) (imp D E) :- copyform B D, copyform C E.
copyform (all B)   (all D)   &
copyform (some B)  (some D)  :- pi y\ copyterm y y => copyform (B y) (D y).

copyform (p X)   (p U)       :- copyterm X U.
copyform (q X Y) (q U V)     :- copyterm X U, copyterm Y V.

type subst       (term -> form) -> term -> form -> o.
subst M T N :- pi x\ copyterm x T => copyform (M x) N.

type uni_instan  form -> term -> form -> o.
uni_instan (all B) T C :- subst B T C.
```

Figure 7.6: Specifying object-level substitution.

is provable from these clauses if and only if `t` and `s` are the same formula. If equality had been all that we wanted to specify, there is, of course, a more direct and less explicit specification we could have used, namely,

```
copyterm T T.
copyform T T.
```

These clauses can, however, be used to also specify object-level substitution. Consider adding a new constant, say `c` of type `term`, to the current signature. If this constant were to be considered a proper constant of our object-level logic, we should add the clause `(copyterm c c)` to the current program. Instead, add the clause `(copyterm c (f a))`. Given this extended set of `copy`-clauses, `(copyterm t s)` is provable if and only if `s` is the result of replacing every occurrence of `c` in `t` with `(f a)`: in other words, we have just specified object-level substitution. The `subst` predicate formalizes this substitution relationship. Here, the first argument of `subst` is an abstraction over formulas, the second argument is some term, and the third argument is the result of substituting the second argument into the first abstraction. Notice the type of `subst`: if `B` has type `term -> form`, then the expression `(subst B)` has the type `term -> form -> o`; that is, `subst` carries the functional type to the corresponding predicate type denoting that function.

The clause for `uni_instan` in Figure 7.6 specifies how to instantiate an object-level universal quantifier with a given term. It is also possible to substitute for several bound variables simultaneously. For example, consider the following two clauses.

```
instan (all A) B :- pi x\ copyterm x T => instan (A x) B.
instan B C       :- copyform B C.
```

The predicate `instan` relates an object-level formula to the result of instantiating 0 or more of its outermost universal quantifiers.

Consider the computation initiated by the query

```
?- subst (x\ all y\ q x y) (f a) S.
```

when the module `ot_subst` is in the current context. This query reduces to the query

```
pi x\ copyterm x (f a) => copyform (all y\ q x y) S.
```

Assume that the new constant introduced is `c` and that the clause `(copyterm c (f a))` is added to the current program. The next query to be attempted is

```
copyform (all y\ q c y) S.
```

The `copy`-clause for `all` is the only clause that can be used to reduce this goal. Backchaining using it yields the goal

```
pi y\ copyterm y y => copyform (q c y) (S1 y).
```

Here, `S1` is a new logic variable and `S` is bound to `(all S1)`. This goal then reduces to

```
copyform (q c d) (S1 d).
```

At this point, the original context has been extended with the two constants `c` and `d` of type `term` and with the two clauses

```
copyterm c (f a).
copyterm d d.
```

Notice that `d`, which is now playing the role of the name of a bound variable, is specified in exactly the same way as other constants of type `term` in the object-level. The specification for `c` is, in a sense, "impure": that clause specifies that `c` does not copy to `c` but some other term, in this case `(f a)`. Now backchaining on the `copy`-clause for `q` yields the goal

```
copyterm c S2, copyterm d S3.
```

where `(S1 d) = (q S2 S3)`. These two goals are uniquely solvable and bind `S2` to `(f a)` and bind `S3` to `d`. Thus, `S1` must be bound to `(d\ q (f a) d)` and `S` must be bound to `(all d\ q (f a) d)`.

The `copy`-clauses presented here are important for at least the following four reasons.

First, they are our first example of an important example of using *signature dependent* clauses. One approach to specify meta-level manipulations of object-level logics makes frequent use of clauses that are parameterized via the object-level's signature.

Second, object-level substitution is a particularly important operation and our specification of substitution can be generalized to signatures involving arbitrary kinds and type orders. We return to this in a later chapter.

Third, this example illustrates the central method available in λProlog for processing λ-abstraction in terms: to "descend" through an abstraction, a universally quantified goal is used to generate a new constant and that new constant is used to name the bound variable. A β_0 redex is used to insert this new constant into the body of the λ-abstraction. Since this new constant is now temporarily part of the object-logic, clauses that were determined from the signature of the object-logic may need to be extended to account for this new constant. Thus, the definition of the `copyterm` predicate needs to be extended using an implication in a goal when `copyform` recurses through an object-level quantifier: the new constant that names the object-level bound variable must be treated as another constant of the object-level.

Finally, $\beta\eta$-unification ("higher-order" unification) can be accounted for making use of `copy`-clauses. For now, we only indicate this connection by presenting a simple example. Consider the problem of existentially generalizing (at the object-level) `a` from the formula `(q a a)`. There are four possible such generalizations, namely,

```
(some x\ q x x)    (some x\ q x a)
(some x\ q a x)    (some x\ q a a).
```

Computing these generalizations can be done by asking the question: for what terms `B` of type `term -> form` is the formula `(B a)` equal, modulo $\beta\eta$-conversion, to the term `(q a a)`? This is an example of $\beta\eta$-unification, which in this setting simplifies down to the question: If we substitute `a` into what abstraction `B` do we get the term `(q a a)`? Since our specification of substitution above is completely declarative, we should be able to get the required four answers by asking the query

```
?- subst B a (q a a).
```

This query will reduce to the query

```
pi x\ copyterm x a => copyform (B x) (q a a).
```

After the introduction of a new constant, say `c` of type `term`, and the addition of the clause `(copyterm c a)` to the current program, the next query attempted is

```
copyform (B c) (q a a).
```

There is exactly one way to prove this goal, and that is to backchain over the `copy`-clause for `q`. This yields the conjunctive goal

```
copyterm (B1 c) a, copyterm (B2 c) a.
```

Here, `B1` and `B2` are new free variables such that

```
(B c) = (q (B1 c) (B2 c)).
```

Each of these conjunctive goals has two solutions. The first can be proved by backchaining on the clause `(copyterm a a)` that is part of the original set of `copy`-clauses. In this case, `B1` is instantiated to the λ-term `c\a`. This goal can also be proved by backchaining over the added clause `(copyterm c a)` in which case `B2` is instantiated to the λ-term `c\c`. Combining these two sets of independent choices together yields the following four possible instantiations for `B`:

```
(x\ q x x)          (x\ q x a)          (x\ q a x)          (x\ q a a).
```

It is possible to specify object-level substitution without using the `copy`-clauses directly. The module in Figure 7.7 provides such a specification, which is based loosely on the equational presentation of λ-conversion found in [And71, And86]. The clauses in that module are also signature dependent. The specification using the `copy`-clauses, however, seems more natural and flexible.

The above description of the behavior of the meta-level universal quantifier and implication sounds particularly operational and it may appear to have little to do with a logical or declarative reading of these connectives. In fact, the declarative interpretation for these logical constants given by intuitionistic logic matches exactly this operational reading. To illustrate, it is indeed trivial to prove that the universal instantiation of a Horn clause is another Horn clause.

7.6 Interpreters for object-level *fohc* and *fohh*

In order to specify an interpreter for object-level *fohc* and *fohh*, it is necessary to make use of object-level substitution. Such an interpreter for *fohc* is presented in Figure 7.8: it assumes that syntax of program clauses and goals follow the richest definition of *fohc* (definition 3.2 given in Section 3.4). The interpreter for *fohh* in Figure 7.9 results from adding two clauses to the specification for the *fohc* interpreter. This interpreter also assumes the richest definition of *fohh* (definition 5.2 given in Section 5.1).

Let `D` and `G` be a *fohc* program clause and goal, respectively, in the object-level. Then the query

```
?- interp D G
```

is provable if and only if the formula denoted by `G` follows intuitionistically from the formula denoted by `D`. In fact, the operational behavior of the λProlog interpreter on this goal

```
module subst_only.
import ot_constants.

type subst      (term -> form) -> term -> form -> o.
type substterm  (term -> term) -> term -> term -> o.

subst (x\ truth) T truth.
subst (x\ false) T false.
subst (x\ neg (B x))       T (neg D)   :- subst B T D.
subst (x\ and (B x) (C x)) T (and D E) &
subst (x\ or  (B x) (C x)) T (or  D E) &
subst (x\ imp (B x) (C x)) T (imp D E) :- subst B T D, subst C T E.
subst (x\ all  (B x))      T (all D)   &
subst (x\ some (B x))      T (some D)  :-
   pi y\ subst x\y T y => subst (x\ B x y) T (D y).

subst (x\ p (X x))       T (p U)    :- subst X T U.
subst (x\ q (X x) (Y x)) T (q U V)  :- subst X T U, subst Y T V.

substterm (x\ x) T T.
substterm (x\ a) T a.
substterm (x\ f (F x)) T (f S) :- substterm F T S.
substterm (x\ g (F x) (G x)) T (g S1 S2) :-
  substterm F T S1, substterm F T S2.
```

Figure 7.7: Specification of substitution without using the auxiliary `copy`-clauses.

```
module fohc_interp.
import  ot_logic, ot_subst.

type interp      form -> form -> o.
type backchain   form -> form -> form -> o.

interp D truth.
interp D (and G1 G2) :- interp D G1, interp D G2.
interp D (or  G1 G2) :- interp D G1; interp D G2.
interp D (some G)    :- subst G X H, interp D H.
interp D A           :- atom A, backchain D D A.

backchain D A A.
backchain D (and D1 D2) A  :- backchain D D1 A; backchain D D2 A.
backchain D (imp G D1) A   :- backchain D D1 A, interp D G.
backchain D (all D1) A     :- subst D1 X D2, backchain D D2 A.
```

Figure 7.8: An interpreter for object-level *fohc*.

```
module fohh_interp.
accumulate fohc_interp.

interp D (imp D1 G)  :- interp (and D1 D) G.
interp D (all G)     :- pi x\ copyterm x x => interp D (G x).
```

Figure 7.9: An interpreter for object-level *fohh*.

matches the operational behavior of the λProlog interpreter if given meta-level versions of these object-level formulas.

Notice that in these two interpreters, object-level substitution is used twice: once to instantiate an existentially quantified goal formula and once to instantiate a universally quantified program clause. In each case, the instantiation is made with a logic variable which will be interpreted later via the meta-level unification that is implicit in the first clause specifying `backchain`.

Object-level substitution can be delayed to the point where object-level unification is done, as is shown by the modules in Figure 7.10 and 7.11. Here, instead of doing a substitution of a logic variable for a bound variable when attempting an existential goal or backchaining over a universal program clause, both a new logic variable `X` and a scoped constant (naming the meta-level universal quantified variable `x`) are introduced and associated via the hypothetical assumption `(copyterm x X)`. (This will then be another example of having logic variables stored in program clauses.) When using the first of the `backchain` clauses, the goal `(copyform A1 A, copyform A2 A)` will be called. In the operational setting, the only free variable in this goal is `A`: `A1` and `A2` will be instantiated with closed terms that may contain scoped constants that have been associated with logic variables in the current program.

For an example of using this interpreter, consider an interpreter for the following *fohc*

```
module fohc_interp_exp.
import  ot_logic, ot_subst.

type interp      form -> form -> o.
type backchain   form -> form -> form -> o.

interp D truth.
interp D (and G1 G2) :- interp D G1, interp D G2.
interp D (or  G1 G2) :- interp D G1; interp D G2.
interp D (some G)    :- pi x\ copyterm x X => interp D (G x).
interp D A           :- atom A, backchain D D A.

backchain D A1 A2          :- copyform A1 A, copyform A2 A.
backchain D (and D1 D2) A :- backchain D D1 A; backchain D D2 A.
backchain D (imp G D1)  A :- backchain D D1 A, interp D G.
backchain D (all D1) A    :- pi x\ copyterm x X => backchain D (D1 x) A.
```

Figure 7.10: An interpreter for object-level *fohc*.

```
module fohh_interp_exp.
accumulate fohc_interp_exp.

interp D (imp D1 G)  :- interp (and D1 D) G.
interp D (all G)     :- pi x\ copyterm x x => interp D (G x).
```

Figure 7.11: An interpreter for object-level *fohh*.

program.

```
adj a b.
adj b c.
path X Y :- adj X Y.
path X Z :- adj X Y, path Y Z.
```

To be used by the interpreter in Figure 7.8 or Figure 7.10, we must encode these clauses into the object-logic by taking the following steps.

1. Declare all the constants in this program to have the correct types over `term` and `form`. These declarations extended those in the module `ot_constants` in Figure 7.1.

2. Add the appropriate clauses extending the definition of the `atom` predicate. There is one clause for each object-level predicate. These clauses extend those clauses in Figure 7.1.

3. Add the appropriate clauses to extend the definition of `copyterm` and `copyform`. These clauses extend those clauses in Figure 7.6.

4. Encode the object-level Horn clauses as a conjunction at the object-level (of type `form`).

Figure 7.12 contains all of these declarations and clauses. Given that module `ot_path.mod` is part of the current context, the following query yields the following answer substitutions.

```
?- prog Cs, interp Cs (path a X).
X == b;
X == c;
no

?-
```

There are several observations to be made about this interpreter. First, the object-level clauses are not polymorphically typed. For example, in Section 3.7 we presented both `adj` and `path` as being binary predicates over the type `node`. Object-level types are not part of this specification of the object-level code, although there is no particular difficulty in incorporating them.

These interpreters for object-level *fohc* and *fohh* involve no non-logical features and the specifications that needed to be made — those of `atom`, `copyterm`, and `copyform` — are completely natural and of logical significance themselves. Furthermore, the implementation of object-level substitution is particularly simple and natural, in contrast to the complexities involved in implementing substitution in more conventional programming languages.

In Subsection 8.3.3 we shall present another approach to the specification of object-level substitution.

7.7 Unification in L_λ

Unification in L_λ is an extension to unification over first-order terms in which the complexities of λ-bound variables are incorporated. We shall only present L_λ-unification by example. Consider the following equations.

```
module ot_path.
accumulate ot_logic.

type a, b, c    term.
type adj, path  term -> term -> form.

atom (adj X Y).
atom (path X Y).

copyterm a a.
copyterm b b.
copyterm c c.
copyform (adj  X Y) (adj  U V) &
copyform (path X Y) (path U V) :- copyterm X U, copyterm Y V.

type prog       form -> o.

prog (and (adj a b)
     (and (adj b c)
     (and (all x\ all y\ imp (adj x y) (path x y))
          (all x\ all y\ all z\ imp (and (adj x y) (path y z))
                                    (path x z))))).
```

Figure 7.12: Object-level specification of a small *fohc* program.

```
    (x\y\ g (U x y) (V y)) = (v\w\ X w)
    (all x\ some y\ q x y) = (all x\ some y\ B y x)
(all x\ imp (B x) (q x x)) = (all y\ imp (p y) (C y))
   (all x\ some y\ q x y)) = (all x\ some y\ B x)
    (all x\ imp B (q x x)) = (all y\ imp (p y) (C y))
```

The last two do not have unifiers while the first three have the following unifiers:

```
U == x\y\ U1 y         X == w\ g (U1 w) (V w)
B == y\x\ q x y
B == x\ p x            C == y\ q y y
```

We use the name L_λ-unification and not $\beta_0\eta$-unification since the latter does not capture the restrictions on terms that is an important part of L_λ. For example, the equation

```
(x\ F (G x) (H x)) = x\x.
```

(where `F` has type `i -> i -> i`) has the two unifiers

```
F == w1\w2\ w1    G == v\v    H == H
F == w1\w2\ w2    G == G      H == v\v.
```

If reductions are chosen in the correct order, only β_0 is required to solve this unification problem. Notice, however, that this problem has two, incomparable unifiers. Of course, this unification problem does not satisfy the L_λ-restrictions about arguments for the occurrences of logic variables.

7.8 Three notions of syntactic representation

In this chapter we have shown some examples of computing with λ-terms and have illustrated how programs involved with manipulating object-level logical expressions can benefit from seeing such object-level expressions as meta-level λ-terms. This observation can be expanded to a more general way of representing syntax within a programming language.

Consider writing programs in which the data objects to be computed are syntactic structures, such as programs, formulas, types, and proofs, involve notions of abstractions, scope, bound and free variables, substitution instances, and equality up to alphabetic changes of bound variables. Although the data types available in most computer programming languages are, of course, rich enough to represent all these kinds of structures, such data types do not have direct support for these common characteristics. Instead, "packages" need to be implemented to support such data structures. For example, although it is trivial to represent first-order formulas in Lisp, it is a more complex matter to write Lisp code to test for the equality of formulas up to alphabetic variation, to determine if a certain variable's occurrence is free or bound, and to correctly substitute a term into a formula (being careful not to capture bound variables). This situation is the same when structures like programs or (natural deduction) proofs are to be manipulated and if other programming languages, such as Pascal, Prolog, and ML, replace Lisp.

Generally, syntax is divided into *concrete* and *abstract* syntax. The first is the linear form of syntax that is readable and typable by a human. Figure 7.13 characterizes some properties of concrete syntax. The costs listed in that figure can be overcome by parsing concrete syntax into *parse trees*. Figure 7.14 characterizes some properties of parse trees.

Implementation	Strings, text (arrays or lists of characters)
Access	Parsers, editors
Advantages	1. Readable, publishable. 2. Simple computational models for implementation (arrays, iteration).
Costs	1. Contains too much information not important for many manipulations: white space, infix/prefix notation, keywords. 2. Important information is not represented explicitly: recursive structure, function–argument relationship, term–subterm relationship.

Figure 7.13: Characteristics of concrete syntax

Implementation	first-order terms, linked lists
Access	car/cdr/cons in Lisp, first-order unification in Prolog, or matching in ML.
Advantages	1. Recursive structure is immediate. 2. Recursion over syntax is easy to specify. 3. Term–subterm relationship is identified with tree-subtree relationship. 4. Algebra provides a model for many operations on syntax.
Costs	1. Requires higher-level language support: pointers, linked lists, garbage collection, structure sharing. 2. Notions of scope, abstraction, substitution, and free and bound variables occurrences are not supported.

Figure 7.14: Characteristics of parse trees

The costs concerning concrete syntax are now properly addressed, although at significant costs. For example, higher levels of support are required for the programming language and runtime system that encode parse trees. Parse trees, however, are so much more convenient and natural to compute with than strings that these additional costs are outweighed by the advantages to the programmer who must write programs to manipulate syntax. The term *abstract syntax* is often identified with parse trees: we shall reserve the former term for the more "abstract" form of syntax described below.

As Figure 7.14 shows, parse trees are not without their costs also. In particular, notions of abstraction within syntax are not supported directly. In particular, the following are unfortunate properties of parse trees for representing syntax containing bound variables.

- Bound variables are, like constants, treated as global objects.
- Concepts such as free and bound occurrences of variables are derivative notions, supported not by programming languages but by programs added on top of the data type

Implementation	α-equivalence classes of $\beta\eta$-normal λ-terms of simple types
Access	$\beta\eta$-unification or a restriction of it, such as in L_λ
Advantages	1. Bound variable names are inaccessible so many technical problems regarding them disappear. 2. Substitution is easy to support for every data structure containing abstracted variables. 3. Semantics is provided by proof theory, logical relations, and Kripke models.
Costs	1. Requires higher-level support: dynamic contexts, extensions to first-order unification, and a richer notion of equality. 2. Few modern programming languages currently support this representation: λProlog being probably the only one. Some specification languages, such as Isabelle and Elf, do support it as well.

Figure 7.15: Characteristics of abstract syntax.

for parse trees.

- Although alphabetic variants generally denote the same intended object, the correct choice of such variants is unfortunately very often important.
- Substitution is generally difficult to implement correctly.
- An implementation of substitution for one data structure, say first-order formulas, will not work for another, say functional programs.

Given that all of these notions are intimately related to logic, it is natural to believe that a logic programming language might be able to support all of these notions directly, instead of via some programmer supplied package. In fact, L_λ and, more generally, λProlog are examples of logic programming languages that do support such notions directly: the resulting representation of syntax we shall call *abstract syntax*. This approach to syntax is characterized in Figure 7.15. Thus, we have identified three levels of syntactic description: concrete syntax, parse trees, and finally abstract abstract. When using abstract syntax, details of object-level bound variables and about substitution are handled declaratively by λProlog.

Behind the concept of abstract syntax are two notions. First, λ-terms and their equational theory should be used uniformly to represent syntax containing bound variables. Already in [Chu40], Church was doing exactly this to encode the universal and existential quantifiers as well as the definite description operator. Following this approach, instantiation of quantifiers can be specified using β-reduction, much as will be done in Subsection 8.3.3.

The second important notion behind abstract syntact is that of defining operations for composing and decomposing syntax that respect α-equivalence classes. This appears to have first been done by Huet and Lang in [HL78] where the authors clearly discuss the advantages of using simply typed λ-terms and with doing matching modulo the equational rules for λ-conversion. Their approach, however, was limited since it only used matching (not unification more generally). Their approach was extended by Miller and Nadathur in [MN87b] (also see earlier papers [MN86a] and [MN86b]) by moving to a logic programming setting that contained $\beta\eta$-unification of simply typed λ-terms. In that paper the central ideas and advantages behind abstract syntax are discussed. Further examples were also contained in the paper [FM88] by Felty and Miller. In the context of theorem proving, Paulson also independently proposed similar ideas [Pau86].

In [PE88] Pfenning and Elliot extended the observations in [MN87b] by producing examples where the meta-language that incorporated λ-abstractions contained not just simple types but also product types. In that paper they coined the expression *higher-order abstract syntax*. Also at about this time, Harper, Honsell, and Plotkin in [HHP87] proposed representing logics in a dependent typed λ-calculus. While they did not deal with the issue of the computational treatment of syntax directly, this was addressed later by considering the unification of dependent typed λ-expressions by Elliott [Ell89] and Pym [Pym90].

The treatment of abstract syntax in the above mentioned papers had a couple of unfortunate aspects. First, the treatments involve unification with respect to the full $\beta\eta$-theory of the λ-calculus, and this general theory is computationally expensive. In the case, of [HL78], only second-order matching was used and this is NP-complete. In the later papers, full unification was used and this is an undecidable operation. Second, various different type systems were employed to illustrate abstract syntax: simple types, product types, and dependent types. However, if abstract syntax is essentially just about a treatment of bound variables in syntax, it should have a form that is independent from typing.

The introduction of L_λ in [Mil91] provided solutions to both of these problems regarding abstract syntax. First, L_λ provides a setting where the unification of λ-terms is computational cheap: it was shown by Qian [Qia93] that L_λ-unification can be done in linear time and space (as with first-order unification). Also, Nipkow showed that the exponential unification algorithm presented in [Mil91] can be effectively used [Nip93]. Second, it was also shown in [Mil91] that L_λ-unification can be described for *untyped* λ-terms: that is, typing may impose additional constraints on unification, but the L_λ-unification can be defined without types. Thus, it is possible then to define L_λ-like unification for various typed calculi [Pfe91].

Thus L_λ seems to be a natural setting for supporting abstract syntax. Since types are not necessary, it seems best to avoid the adjective "higher-order" with respect to abstract syntax, since orders are generally calculated from types. The version of L_λ that we have discussed in this chapter involved simple types.

Chapter 8

Higher-order Horn clauses

While first-order hereditary Harrop formulas and L_λ are much more expressive than first-order Horn clauses, neither of these classes of formulas directly support what is generally called *higher-order programming*. λProlog supports higher-order programming by allowing logical connectives to appear in terms and by allowing quantification of some occurrences of predicate variables. We describe this aspect of λProlog by first considering in this chapter a higher-order version of Horn clauses.

8.1 Higher-order programming examples

There have been several proposals for higher-order programming within logic programming and they do not generally agree. Thus it is good to consider some examples of programming that should clearly be considered higher-order. All of the following examples are captured simply and declaratively within *higher-order Horn clauses* (*hohc*).

1. Given a predicate of one argument and a list, check that every (some) element of that list satisfies that predicate.
2. Given a predicate of two arguments and two lists, check that corresponding elements of these two lists are related by the given predicate.
3. Given a predicate and two lists, check that all of the elements of the second list satisfy the given predicate and are members of the first list.
4. Given a predicate of two arguments, construct its transitive closure.
5. Given two binary predicates, construct their relational composition (natural join).

Figure 8.1 contains λProlog code that specifies some of these higher-order predicates. The intended meaning of these higher-order predicates is summarized as follows. If the goal `(mappred P L K)` is provable given the clauses in the module in Figure 8.1 then `L` and `K` are lists of equal length and corresponding members of these lists are `P`-related. If the goal `(forsome P L)` is provable then `L` is a list in which some member satisfies the predicate `P`. If the goal `(forevery P L)` is provable then `L` is a list all of whose members satisfy the predicate `P`. If goal `(trans R X Y)` is provable then `X` and `Y` are members of the transitive closure of the binary relation `R`. Finally, if the goal `(sublist P L K)` is provable then `L` is

```
module maps.

type mappred            (A -> B -> o) -> list A -> list B -> o.
type foreach, forsome   (A -> o) -> list A -> o.
type trans              (A -> A -> o) -> A -> A -> o.
type sublist            (A -> o) -> list A -> list A -> o.

mappred P nil nil.
mappred P (X::L) (Y::K) :- P X Y, mappred P L K.

foreach P nil.
foreach P (X::L) :- P X, foreach P L.

forsome P (X::L) :- P X; forsome P L.

trans R X Y :- R X Y.
trans R X Z :- R X Y, trans R Y Z.

sublist P (X::L) (X::K) :- P X, sublist P L K.
sublist P (X::L) K :- sublist P L K.
sublist P nil nil.
```

Figure 8.1: Various examples of higher-order programs.

a sublist of `K` in which all elements satisfy `P`. (The order of the clauses in the specification of `sublist` finds maximal solutions first.)

Given the small database of clauses in Figure 8.2, the following query and results are all provable in λProlog.

```
?- mappred age (ned::bob::sue::nil) L.
L == (23::23::24::nil).
yes
?- mappred age L (23::24::nil).
L == (bob::sue::nil);

L == (ned::sue::nil).
yes
?- sublist male (ned::bob::sue::nil) L.
L == ned::bob::nil;
L == ned::nil;
L == bob::nil;
L == nil;
no
?- forsome female (ned::bob::sue::nil).
yes
?- trans adj a d.
yes.
?-
```

These examples seem natural and should be present in any logic programming language that admits higher-order programming.

Given the availability of λ-terms, it should be possible to build more complex expressions denoting predicates and these should be allowable within higher-order programs. The following queries are also possible within λProlog.

```
?- mappred (x\y\ age x y) (ned::bob::sue::nil) L.
L == (23::23::24::nil).
yes

?- mappred (x\y\ age y x) (23::24::nil) K.
K == (bob::sue::nil);

K == (ned::sue::nil).
yes
?- forevery (x\ sigma y\ age x y) (ned::bob::sue::nil).
yes

?- forevery (x\ age x A) (ned::bob::sue::nil).
no
?-
```

The second last query succeeds because every person in the list has an age. The last query fails because it is not the case that every one in that list has the same age `A`. On the other hand, the query

```
?- forevery (x\ age x A) (ned::bob::nil).
A == 23.
yes
?-
```

will succeeds and returns the age that is common to both `ned` and `bob`.

The use of variables of higher-order type and of λ-terms here exceeds those uses in L_λ: the restriction on the application of function variables to distinct bound variables is not followed in these examples. In fact, function variables, and hence the λ-terms that instantiate them, can be applied to arbitrary terms (of the appropriate types, of course). Thus full β-conversion (instead of just β_0-conversion) may need to be used in computing with *hohc* (see Section 6.2).

The higher-order predicates in Figure 8.3 permit some natural computations on relations. For example, if `R` and `S` are two binary relations of the same type, then `(union R S)` is the union of their extensions and `(compose R S)` is their relational composition (natural join). Notice that since `union` and `compose` are not defined recursively, they can be expressed using λ-expression: the predicate denoted by `(union R S)` can be written instead as the expression

```
x\y\ R x y; S x y.
```

and the predicate denoted by `(compose R S)` can be written instead as the expression

```
x\z\ sigma Y\ R x Y, S Y z.
```

```
module misc.

kind person          type.
type bob, sue, ned   person.
type age             person -> int -> o.
type male, female    person -> o.

age bob 23.
age sue 24.
age ned 23.

male   bob.
female sue.
male   ned.

kind node  type.
type adj   node -> node -> o.

adj a b.
adj b c.
adj b d.
adj d c.
```

Figure 8.2: A simple database of miscellaneous facts.

If `P` is an `A`-indexed set of binary relations over `B`, that is, if it has type

`A -> B -> B -> o,`

then `(iter P)` iterates `P` to get a predicate of type

`list A -> B -> B -> o.`

Such higher-order predicates often make it possible to avoid explicit uses of some quantifiers. For example, `reverse` is defined in Figure 8.4 by repeatedly entering elements on to a stack (similar to the one in Figure 5.18) and then removing them again. The `pre_post` predicates is used to force the stack to be empty at the start and conclusion of the reversing operation. (Using the `local` and `localkind` directives, it is possible to give local scope to `stack`, `emp`, `stk`, `empty`, `enter`, and `remove`.

As a final example of a style of programming that should be available in a higher-order version of Horn clauses, consider *continuation passing* style programming in the logic programming setting. For example, in [Tar92], Horn clauses of the form

$$A_1 \wedge \ldots \wedge A_n \supset A_0 \qquad (n > 0)$$

are transformed to higher-order clauses of the form

$$(A_1\ (\ldots\ (A_n\ G)\ldots)) \supset (A_0\ G).$$

The simple clause A_0 is transformed to the clause

$$G \supset (A_0\ G).$$

```
module rels.

type union          (A -> B -> o) -> (A -> B -> o) -> A -> B -> o.
type compose        (A -> B -> o) -> (B -> C -> o) -> A -> C -> o.
type iter           (A -> B -> B -> o) -> list A -> B -> B -> o.
type pre_post       (A -> o) -> (A -> B -> o) -> (B -> o) -> o.

union R S X Y   :- R X Y; S X Y.

compose R S X Y :- R X Z, S Z Y.

iter P nil    X X.
iter P (Z::L) X Y :- P Z X M, iter P L M Y.

pre_post R S T :- R X, S X Y, T Y.
```

Figure 8.3: Some simple relational programs.

```
module reverse3.
import rels.

kind stack           type -> type.
type emp             stack A.
type stk             A -> stack A -> stack A.

type empty          stack A -> o.
type enter, remove   int -> stack A -> stack A -> o.

empty  emp.
enter  X S (stk X S).
remove X (stk X S) S.

reverse L K :-
  pre_post empty (compose (iter enter L) (iter remove K)) empty.
```

Figure 8.4: An implementation of list reverse.

Of course, G is a variable of type o, and all the predicates on that atoms $A_0, \ldots, A_n$ must be modified to have an additional argument of type o. The definition of *hohc* that we give now allows for such clauses as these.

8.2 Some design issues

All the above example programs illustrate natural candidates for higher-order programming and are, in fact, examples of λProlog programs. Before defining higher-order Horn clauses, we explore some aspects of predicate quantification that are not directly motivated by principles of higher-order programming but which must be considered if they are to be incorporated into a logic setting.

8.2.1 Flexible atoms as goals

Atomic formulas must have the form $(h\ t_1\ \ldots\ t_n)$, where h is either a variable or non-logical constant and $t_1, \ldots, t_n$ are terms. If h is a non-logical constant, this atom is a *rigid atom*; if h is a variable, it is a *flexible atom*. As we have seen in the examples so far, atomic formulas in the body of clauses can be flexible. Thus, flexible atoms can become complex goals via substitution. Consider proving the flexible atom `(P bob 23)` from the clauses in Figure 8.2. One answer substitution for this query is the substitution `(x\y\ age x y)` for `P` (this substitution term is equal to `age` by η-conversation). Many other substitutions, however, are also valid. For example, substituting

```
x\y\ age x 23, age ned y
```

will also lead to a provable goal. Furthermore, let `G` be any provable closed query. The substitution `x\y\G` for `P` is a legal answer substitution. Thus, substituting `(x\y\ age sue 24)` and `(x\y\ memb 4 (3::4::5::nil))` (if the basic list operations are in the current context) are both answer substitutions. Clearly, there are a large number of answer substitutions for this goal, most of which will not be interesting: it would be undesirable for an interpreter to systematically search for all of them.

There appears to be three different approaches to treating flexible goals in a λProlog interpreter.

1. Suspend such goals in the hope that processing other goals further instantiates the predicate variable at the head of the goal. In that case, resume processing that goal. In the event that all goals are reduced leaving only flexible goals, then instantiate the predicate variables in the head of all suspended goals as follows: if such a predicate variables has a type with $n \geq 0$ argument types, then substitute $\lambda x_1 \ldots \lambda x_n.\top$ for that predicate variable. That is, instantiate such predicate variables with the universally true relation of the correct type.

2. Do not suspend the flexible goal. Instead simply instantiate the predicate variable head with the substitution described above.

3. Produce an error message.

The first option can be shown to be complete [Nad87, NM90]: the other two are incomplete. An early implementation of λProlog [MN87a] chose not to reorder or suspend goals and used the second option. Experience with that implementation suggested that in the

```
module primrel.

kind i type.

type primrel, rel     (i -> i -> o) -> o.

primrel father.
primrel mother.
primrel wife.
primrel husband.
rel R :- primrel R.
rel (x\y\ sigma z\ R x z, S z y) :- primrel R, primrel S.

mother jane mary.
wife john jane.
```

Figure 8.5: An example of building predicates from other predicates.

vast majority of times that this option needed to be employed, the program being executed contained errors, which, when removed, no longer gave rise to this situation. More recent implementations have, thus, chosen the third option. The choice of implementation strategies used will not be important for us here since none of our examples will be sensitive to them.

The range of a predicate variable can, however, be usefully restricted by a program. For example, given the clauses in Figure 8.5, it is not sensible to ask for the predicates `R` over the current signature that are true when applied to `john` and `mary`. The programmer, however, can specify some collection of predicates that are considered relevant and then restrict the predicate variable head of the flexible goal to be in that collection. For example, the ill-posed query above can be qualified to be the query

```
?- rel R, R john mary.
```

This query is only solvable if `R` is substituted for by the term

```
x\y\ sigma Z\ wife x Z, mother Z y.
```

The second-order predicate `rel` specifies a domain of "relevant" predicates: this specification of relevance must be done by the programmer.

8.2.2 Flexible atoms as heads of clauses

All examples of clauses so far have allowed flexible atoms to appear in the body of clauses but not as the head of a clause. This is because λProlog does not permit the head of a clause to be a flexible atom. There are two major reasons for this design choice.

First, if a Horn clause has a flexible atom as its head, it is possible that all goal formulas can be proved from that clause. Consider, for example, the two clauses

```
P 5 :- q.
q.
```

Let Σ be the current signature and let G be a Σ-formula. To prove G, simply backchain over the first clause above, instantiating `P` with $\lambda x.G$. This then gives rise to the attempt to prove `q`. Since this is provable, we have a proof of G. Thus, if the current context contains these two clauses, all goals over the current signature can be proved: that is, the current context has become *inconsistent*. As further proof: the goal `(pi p\p)` described in Section 5.5.3 as denoting false can also be proved from these two clauses. When clauses can only have rigid atoms at their head then all programs are consistent: that is, they do not prove all goals.

Second, a clause can be seen as being part of a specification for a given predicate, in particular, the one that is the head of a clause. If that head is a variable, then such a clause could be considered as adding meaning to *every* predicate, a possibility that seems to be too liberal to allow [NM90].

For these two reasons, it seems best to define a higher-order version of Horn clauses so that flexible atoms are not the head of any clause. This restriction is made, however, largely because we wish to avoid the operational problems just mentioned. There are, however, meaningful uses to be made of predicates at the head of a clause. For example, Felty has provided an encoding of the *Calculus of Constructions* [CH88] into higher-order logic that makes use of flexible atoms in the heads of clauses [Fel93]. Also, Leibniz defined equality by stating that two terms are equal if they satisfy the same predicates. Thus, the equation $x + 0 = x$ could be specified using the two clauses

```
P (X + 0) :- P X.
P X :- P (X + 0).
```

Thus, any attempt to prove a goal containing an occurrence of a subterm of the form `X + 0` can be replaced by the attempt to prove the same goal but with that subterm replaced with `X`. This style of implemetation of equality reasoning is problematic for other reasons (see Section 8.6).

8.2.3 Logical connectives in terms

Besides addressing the issue of where predicate variables can appear within formulas, it is also important to decide to what extent logical connectives can appear within atomic formula. To address this question, notice that the predicate variables at the head of flexible atoms in goals can be instantiated via unification: somewhere two atomic formulas are unified and the resulting substitution will be applied to the flexible atom. Thus, any logical connectives that appear within atomic formulas can then appear in a goal. For example, using the code in Figure 8.1, the query

```
?- mappred (x\y\ p x y => q x y) (1::nil) (2::nil).
```

would give rise to the query

```
?- p 1 2 => q 2 1.
```

which is not a query within the usual Horn clause setting. Thus, if we are not careful in our definition, a computation that starts out within a Horn clause setting may not remain in that setting. While this example can be made sensible by referring to hereditary Harrop formulas, we would like to restrict the definition of *higher-order Horn clauses* in such a way that the computation process remains within the Horn clause framework. If the only logical constants that appear in atomic formulas are the same as those that can appear in goals (namely, $\top$, $\vee$, $\wedge$, and $\exists$) then it is possible to arrange the search for proofs in higher-order Horn clauses so that the only instances of such clauses that need to be considered are again higher-order Horn clauses.

8.2.4 Definition of *hohc*

Given the above design decisions, we now define *hohc* so as to meet them. Let Σ be a signature (of any order) and let $\mathcal{H}_1$ be the set of all λ-normal Σ-terms in which the only logical constants these terms may contain are $\top, \wedge, \vee$, and $\exists$. Note that $\mathcal{H}_1$ contains terms of all types, including $\mathtt{o}$. Let the syntactic variable A now denote an atomic formula in $\mathcal{H}_1$. The syntactic variable A_r is used to denote rigid atomic formulas in $\mathcal{H}_1$.

It is possible to supply three definitions of the program clauses and goals for *hohc* that parallel the three definitions for *fohc* given in Section 3.4. Below we supply only one definition; the one that parallels definition 3.2. A *goal formula* of *hohc* is any formula in $\mathcal{H}_1$. Notice that goal formulas satisfy the clause

$$G ::= \top \mid A \mid G_1 \wedge G_2 \mid G_1 \vee G_2 \mid \exists_\tau x\, G.$$

This clause alone does not serve to define the entire structure of these goals since it does not reveal the structure of logical connectives that can appear inside atomic formulas. The program clauses of *hohc* are defined by:

$$D ::= A_r \mid G \supset D \mid D_1 \wedge D_2 \mid \forall_\tau x\, D.$$

The quantification in both G- and D-formulas may be over variables of any type and bound variables may be applied to any terms, as long as typing restrictions are correct: the restriction from L_λ is not enforced in *hohc*. The use of rigid atomic formulas in the definition of program clauses rules out the possibility of having flexible atoms as the head of a program clause.

Let *hohc* be formally defined as the logic programming language in which goals and definite formulas are defined as above and provability is identified with classical provability. As is the case in *fohc*, the logic used in this setting is weak enough that provability could also be taken to be intuitionistic: this weaker provability relation would still coincide with the operation behavior of *hohc*. Furthermore, in constructing proofs involving *hohc*, the only substitutions that need to be considered are contained in the set $\mathcal{H}_1$: hence, we refer to this set as the *Herbrand universe for hohc*.

The proof of this latter fact is not straightforward. Consider, for example, the following declarations and *hohc* program clause.

```
kind i    type.
type a,b  i.
type p    i -> o.

p a :- Q b.
```

There is a classical logic proof of the query `sigma x\p x)` from this clause in which the proof does not prove a particular instance of the existential quantifier: that is, the proof is not *goal directed*, as in the sense described in Section 3.3. Such a proof can arise by instantiating `Q` in this clause with the λ-term `(x\ neg (p b))`, where `neg` is used here to denote classical logic negation (this term is, of course, not in $\mathcal{H}_1$). This instance of that clause is classically equivalent to `p a; p b`: that is, the substitution instance of a Horn clause would not be a Horn clause but a disjunction. It is possible to prove `(sigma x\ p x)` from this disjunction although no instance of this existential quantifier are provable from the disjunction (provability could be either classical or intuitionistic).

Fortunately, it turns out that this apparent problem can be overcome. It was shown in [Nad87] (and later in [NM90]) that whenever there is any proof that a certain goal follows from a set of higher-order Horn clauses, there is a goal directed proof, and furthermore, all instances of Horn clauses in that proof will again be Horn clauses (substitution terms only need to be selected from $\mathcal{H}_1$). In the example above, there is a goal-directed proof of `(sigma x\ p x)` in which the variable `Q` is instantiated with the term `x\true`.

The core of *hohc*, in the sense of Section 5.3, is a disappointing small set, namely the set of closed atoms or conjunctions of atoms. This is the same as the core for *fohc* except that the atoms can now contain λ-abstractions and non-logical constants of higher-order type.

8.2.5 Comparison with functional programming

At this point it is worth comparing how the higher-order programming supported by *hohc* compares with that supported in functional programming languages like Scheme and ML. As the examples using `mappred` and `sublist`, for example, show, most of the functionality of higher-order programming in functional programming can naturally be transfered to the logic programming setting. If our goal had been to simply capture a suitable generalization of the functionality behind higher-order programming in functional programming then we should have restricted the structure of *hohc* greatly. For example, quantification over higher-order but non-predicate types would not be necessary. For more on such restrictions, see [Wad91]. While such a restriction has its interest and merits, it only provides logic programming with a concept (higher-order programming) that is already familiar. With the stronger definition above, *new* programming features emerge that did not appear in any existing programming language. These new features — the presence of λ-abstraction and functional quantification — have been at the center of much of the development of λProlog (historically speaking, *hohc* was the first extension of *fohc* that was studied in the context of λProlog).

There are at least two ways in which the notion of higher-order programming in *hohc* is stronger than the notion found in functional programming. First, although predicate variables in λProlog correspond naturally to function variables in functional programming, the general notion of function variables in λProlog corresponds to nothing in conventional functional programming languages. Since predicate variables are essentially just special cases of function variables and since the proof theory of quantification at higher-types is well understood, it was natural to design λProlog to include this more general notion. While the notion of function variable quantification does not correspond to anything in functional programming, it also supplies λProlog with computational expressiveness not found in functional programming. For example, the ability to directly manipulate λ-terms as in Chapter 7 is not available in functional programming languages.

Second, *hohc* allows two predicate expressions to be compared for equality: in an implementation, this would give rise to the unification of predicate expressions. For example, the following clause is a legal *hohc* clause.

```
type eq_pred (A -> o) -> (A -> o) -> o.
```

```
eq_pred R R.
```

A goal with `eq_pred` as its head would succeed if its two arguments are equal (unifiable) predicates. Such an operation is not possible in such higher-order functional programming languages as ML and Scheme. Note, however, that such a check on equality is based on the *intension* and not the *extension* of the predicates. For example, the query

```
?- eq_pred (x\ p x, q x) (x\ q x, p x).
```

will fail no matter the definitions of `p` and `q`: equality is decided only at the level of the λ-terms: the fact that these λ-terms may denote sets and that these sets may be equal is not considered. Functional programming does not allow for such checks between functions, chiefly because the intended semantics of functional programming languages is an extensional one and determining extensional equality is, in general, undecidable. Clearly, the semantics of *hohc* and λProlog is not extensional: the weaker intensional check of closed terms is decidable (reduces to $\beta\eta$-conversion).

8.3 Computations on λ-terms in *hohc*

We now explore to what extent *hohc* can compute with terms containing λ-abstractions. The logical structure of program clauses in *hohc* is weaker than for L_λ: the former has formulas limited to clausal order 0 or 1 whereas the latter has no limit on order. However, *hohc* has no restrictions on how quantifier variables can be applied. An implementation of *hohc* must, therefore, employ a much stronger form of unification to deal with the more general structure of terms. This unification, *$\beta\eta$-unification*, is illustrated with examples in this section.

8.3.1 Computing with functional expressions

The `mapfun` predicate in Figure 8.6 is a natural parallel to the `mappred` predicate . This predicate relates a term of functional type to two list of equal length if the elements of the second list are the result of applying that functional term to corresponding elements of the first list. Here, of course, function application is simply $\beta\eta$-conversion for simply typed λ-terms. Assume that the constants `a`, `b`, `c`, and `d` have type, say `i`, and that `g` has type `i -> i -> i` and consider the query

```
?- mapfun (x\ g a x) (a::b::nil) L.
```

The unique answer substitution for `L` is `((g a a)::(g a b)::nil)`. In order to compute this term an interpreter would form the terms `((x\ g a x) a)` and `((x\ g a x) b)` and λ-reduce them. Thus, computation in `mapfun` arises chiefly from β-reductions whereas computation in `mappred` arises from calling the goal that results from applying a predicate to arguments: the latter is therefore capable of calling arbitrarily complex subcomputations. Clearly, `mappred` is stronger than `mapfun`: the clause

```
mapfun F L K :- mappred (x\y\ y = F x) L K.
```

implements the latter using the former.

It is, of course, possible to ask an interpreter to construct a functional expression that will satisfy a `mapfun` goal. The query

```
module funprog.

type mapfun  (A -> B) -> list A -> list B -> o.

mapfun F nil nil.
mapfun F (X::L) ((F X)::K) :- mapfun F L K.

type reducefun (A -> B -> B) -> list A -> B -> B -> o.

reducefun F nil Z Z.
reducefun F (H::T) Z (F H R) :- reducefun F T Z R.
```

Figure 8.6: Two programs for manipulating function expressions.

```
?- mapfun F (a::b::nil) ((g a a)::(g a b)::nil).
```

is provable and has one answer substitution, namely, that which substitutes `(x\ g a x)` for `F`. An interpreter for λProlog would need to consider first unifying the pair of terms `(F a)` and `(g a a)`. This particular unification problems has four unifiers, namely, the substitutions where `F` is replaced with the following terms.

```
(x\ g x x)    (x\ g a x)    (x\ g x a)    (x\ g a a)
```

If the second substitution term is not selected first, the interpreter will need to backtrack until it does in fact select the second substitution term, which is the only substitution term that also unifies the pair `(F b)` and `(g a b)`. Notice that the following goal is not provable:

```
?- mapfun F (a::b::nil) (c::d::nil).
```

There is no simply typed λ-term which maps `a` to `c` and maps `b` to `d`. There are, clearly, an infinite number of functions that would map `a` to `c` and `b` to `d`, but none of them are expressible in this rarefied setting (see related discussion in Section 6.3).

Consider also the `reducefun` predicate in Figure 8.6. Below are two examples of queries using this predicate.

```
?- reducefun (x\y\ x + y) (3::4::8::nil) 6 R.
R == 3 + (4 + (8 + 6)) .
 yes

?- reducefun F (4::8::nil) 6 (1 + (4 + (1 + (8 + 6)))).
F == x\y\ 1 + (4 + (1 + (8 + 6))) ;
F == x\y\ 1 + (x + (1 + (8 + 6))) ;
F == x\y\ 1 + (x + y) ;
no

?-
```

The second query has three answer substitutions. It is possible to get only the third if the query is rewritten using a universal quantifiers as follows.

```
?- pi z\ reducefun F (4::8::nil) z (1 + (4 + (1 + (8 + z)))).
F == x\y\ 1 + (x + y) ;
 no

?-
```

Technically speaking, universally quantified goals are not part of *hohc*. We shall, however, be presenting a common extension of both *hohc* and L_λ in Chapter 9. Since λProlog is based on that common extension, the above query is appropriate in λProlog.

8.3.2 A functional version of difference lists

A common programming technique in Prolog is to use *logic variables* in data structures not to denote a partial description of that structure but as a site into which additional information can be inserted. One such data structures is called *difference lists* (see, for example, [SS86]). Difference lists can be represented simply in λProlog as terms of type `(pr (list A) (list A))` where the second list of the pair is a tail of the first list; for example,

```
(pr (a::b::c::L) L).
```

A difference list is intended to be a representation of the list which precedes the designated tail: in this example, the intended list is simply `(a::b::c::nil)`. One use of difference lists is to implement concatenation, using the predicate

```
type concat   pr (list A) (list A) ->
              pr (list A) (list A) ->
              pr (list A) (list A) -> o.

concat (pr L1 L2) (pr L2 L3) (pr L1 L3).
```

One problem with this approach to appending lists is that a given difference list can only be used in a `concat` once: after that, the logic variable representing the tail of a list is set.

It is possible, however, to use abstracted variables instead of logic variables for inserting information into a data structure [BR91]. Forexample, a *functional difference list* will be a list with its tail abstracted. For example, given the following declarations

```
kind i       type.
type a,b     i.
type f       i -> list i -> i.
```

the term `(l\ a::b::c::l)`, of type `list A -> list A`, denotes the difference list above. Clearly, not all terms of type `list A -> list A` are functional difference lists: for example

```
(l\ a::b::c::nil)       (l\ a::(f a l)::nil)
```

are both of this type but neither are abstractions over only their tail. The module in Figure 8.7 specifies some predicates over such functional different lists.

The `list_dlist` predicate can be used to convert between lists and difference lists. L_λ provides another implementation of `list_dlist`, namely replace the specification in Figure 8.7 of `list_dlist` with

```
module diff_lists.

type concat        (list A -> list A) ->
                   (list A -> list A) -> (list A -> list A) -> o.
type insert_front, insert_rear
              A -> (list A -> list A) -> (list A -> list A) -> o.
type list_dlist    list A -> (list A -> list A) -> o.
type palindrome    (list A -> list A) -> o.

concat F G (x\ F (G x)).

insert_front Y F (x\ Y::(F x)).

insert_rear  Y F (x\ F (Y::x)).

list_dlist nil x\x.
list_dlist (H::T) (x\ H::(D x)) :- list_dlist T D.

palindrome (x\x).
palindrome (x\ Y::x).
palindrome (x\ Y::(F (Y::x))) :- palindrome F.
```

Figure 8.7: A specification of functional difference lists.

```
list_dlist L D :- pi x\ append L x (D x).
```

assuming, of course, that the `lists` module is either imported or accumulated into the `diff_lists` module. Consider the following predicate of a type similar to that for `list_dlist`.

```
type convert_list    list A -> (list A -> list A) -> o.
convert_list (F nil) F.
```

This predicate does not faithfully relate lists to functional difference lists: it specifies a one-to-many relationship. That is, given a list as it first argument, there are various functional structures that can be in the second argument position, only one of which is the intended difference list. For example, consider the following two queries:

```
?- convert_list (a::b::nil) F.
F == x\ a::b::nil;
F == x\ a::b::x;
 no

?- convert_list (a::(f (b::nil))::c::nil) F.
F == x\ a::(f (b::nil))::c::nil;
F == x\ a::(f (b::x))::c::nil;
F == x\ a::(f (b::nil))::c::x;
F == x\ a::(f (b::x))::c::x;
 no

?-
```

In each case, only one of the various answers is, in fact, the intended difference list. Unification alone cannot be used to produce a functional different list from a list: recursion as in the specifications of `list_dlist` above is necessary.

For a final example of functional difference lists, consider the predicate that determines whether or not the list intended by a functional difference list is a palindrome, that is, a list that can be read the same forwards and backwards. Since it is possible to access the first and last element of a difference list in one unification step, this predicate has a particularly simple specification. Consider the following two queries to `palidrome`.

```
?- palindrome (x\ a::b::c::b::a::x).
 yes

?- palindrome (x\ a::b::a::(F x)).
F == x\ x ;
F == x\ b::a::x ;
F == x\ a::b::a::x ;
F == x\ Y::a::b::a::x ;
F == x\ Z::Z::a::b::a::x ;
F == x\ Y::Z::Y::a::b::a::x ;
F == x\ Y::Z::Z::Y::a::b::a::x ;
F == x\ Y::Z::U::Z::Y::a::b::a::x ;
F == x\ Y::Z::U::U::Z::Y::a::b::a::x .
 yes
?-
```

The second query is asked to compute the functional difference list, `F`, so that the result after concatenating with the difference list `(x\ a::b::a::x)` is a palindrome. There are an infinite number of answers to this query, several of which are displayed above.

8.3.3 Object-Level Substitution

Given the presence of β-conversion in the meta-theory of *hohc* (in contrast to just having β_0-conversion in L_λ) and the fact that β-conversion involves substitution, it is possible to recode the interpreter for *fohc* written in L_λ in Section 7.6 in a simpler fashion. In particular, object-level substitution can be specified simply using the atomic clause

```
type subst   (term -> form) -> term -> form -> o.
subst M T (M T).
```

Notice that substitution can now be defined without using `copy`-clauses. Given this approach to object-level substitution, the signature of object-level, non-logical constants is needed only for the specification of the `atom` predicate. The modules in Figure 8.8 actually drops even this reliance on the object-level signature. The module `ot_logic1` specifies the only aspect about the object-level logic that is needed to write an interpreter in *hohc*. The module `fohc_in_hohc` contains the *hohc* code for specifying the interpreter and the module `ot_path` represents the object-level `adj` and `path` clauses.

Since the signature dependent parts of the specification used in Chapter 7 are not required here, the specification of a *fohc* interpreter here is somewhat simpler. The interpreter here is open in the sense that we do not need to know anything about the programs that we are going to interpret except that they are formulas that are correctly restricted to the Horn clause definition (3.2) given in Section 3.4.

Since we have not implemented a predicate such as `atom`, the interpreter in Figure 8.8 does not have the behavior we would wish in all cases. For example, assume that we have two constants `p` and `q` of type `form` and that we ask the query

```
?- interp (and p q) (and p q).
```

This query will succeed twice, where as we would have liked it to succeed once. The reason for two success is that one arises from decomposing this query into the conjunctive query

```
?- interp (and p q) p, interp (and p q) q.
```

Each of these goals has one solution. However, there is another way to decompose this goal and that is to decompose it into

```
?- backchain (and p q) (and p q) (and p q).
```

which again has one solution. Generally speaking, we would prefer that `backchaining` be used only for atomic goals, but since we do not have a meta-level predicate for determining this property of object-level goals, this use of backchaining is not stopped. We can, of course, make use of cuts `!` immediately after the `:-` in all but the last of the `interp` clauses. This would also stop backchaining on non-atomic goals.

There appear to be three ways to introduce the `atom` predicate again to this interpreter. The first is the *positive* approach where we declare what object-level formulas are to be considered atoms. Such a specification is exactly what is given in Figure 7.8: this re-introduces a degree of signature-dependence again. There is, however, the negative approach: we can describe what formulas are not atomic and then negate that predicate. This will, of course, rely on features of λProlog that are outside the scope of the logics underlying λProlog. Such a negative specification can be made by adding the following lines to the module `ot_logic1` in Figure 8.8.

```
type atom, non_atom   form -> o.

non_atom truth     & non_atom (or _ _) & non_atom (and _ _) &
non_atom (imp _ _) & non_atom (all _)  & non_atom (some _).

atom A :- not (non_atom A).
```

A third approach would be to introduce an explicit constructor for atoms. This might be done as follows.

```
kind prop         type
type atm          prop -> form.
type adj, path    term -> term -> prop.
```

Then we place `atm` around all atomic formulas. For example, the object-level formula encoding the adjaceny and path clauses (see Figure 8.8) would then be written as

```
(and (atm (adj a b))
   (and (atm (adj b c))
     (and (all x\ all y\ imp (atm (adj x y)) (atm (path x y)))
          all x\ all y\ all z\ imp (and (atm (adj x y)) (atm (path y z)))
                                   (atm (path x z)))))).
```

```
module ot_logic1.

kind  term, form      type.

type truth           form.
type or, and, imp    form -> form -> form.
type all, some       (term -> form) -> form.

module fohc_in_hohc.
accumulate ot_logic1.

type interp      form -> form -> o.
type backchain   form -> form -> form -> o.

interp D truth.
interp D (and G1 G2) :- interp D G1, interp D G2.
interp D (or  G1 G2) :- interp D G1; interp D G2.
interp D (some G)    :- interp D (G X).
interp D A           :- backchain D D A.

backchain D A A.
backchain D (and D1 D2) A  :- backchain D D1 A; backchain D D2 A.
backchain D (all D1) A     :- backchain D (D1 X) A.
backchain D (imp G D1) A   :- backchain D D1 A, interp D G.

module ot_path.
accumulate ot_logic1.

type a, b, c    term.
type adj, path  term -> term -> form.

type prog       form -> o.

prog (and (adj a b)
      (and (adj b c)
       (and (all x\ all y\ imp (adj x y) (path x y))
            (all x\ all y\ all z\ imp (and (adj x y) (path y z))
                                      (path x z))))).
```

Figure 8.8: Three modules specifying an object-level encoding of *fohc* for an interpreter in written in *hohc*.

We would also need to modify the interpreter in that module as well, by giving `backchain` a different type and changing a couple clauses, as shown below.

```
type backchain    form -> form -> prop -> o.
interp D (atm A) :- backchain D D A.
backchain D (atm A) A.
```

It does not seem possible to provide a direct implementation of object-level *fohh* within *hohc*. In Figure 7.9 the object-level universal quantifier was implemented using a meta-level universal quantifiers. Since *hohc* lacks this meta-level quantifier, this approach is not available, and no other approach seems apparent. It is easy to provide for object-level implications in this *hohc* interpreter by simply using the same technique used in Figure 7.9: the first clause in Figure 7.9 can be used directly in this interpreter.

8.4 More on computing with λ-terms in *hohc*

Although *hohc* admits full β-conversion, it does not appear to be flexible enough to perform a large number of operations on λ-terms. For example, consider writing meta-level predicates for deciding whether or not an object-level formula is a goal or definite clause in *fohc*. Figure 7.2 contains the simple specification of these predicates in L_λ: Figure 8.9 contains a specification of these predicates in *hohc*. This later specification is unsatisfactory since its meaning relies on an extra constant which must be added to deal with quantification. (The predicate `atom` can be specified in the various ways described in the previous section.) Since the universal quantifier `pi` is not available in the goal formulas of *hohc*, the treatment of object-level quantifiers is different: to determine whether or not `(all D)` is a definite clause, this code checks whether or not `(D dummy)` is a definite clause. While this reduction is correct, it lacks the elegance of the implementation in Figure 7.2: while it was easy to show that the substitution instance of a first-order Horn clause is another first-order Horn clause using the L_λ implementation showing the similar properties here is less straightforward.

If we make `dummy` local to this module this specification improves somewhat: this new object-level term is introduced only for a certain computation and is not inserted into the object-logic as other constants. Technically, however, the use of `local` is not accounted for by *hohc* alone.

While the use of `dummy` succeeded in producing an *hohc* specification of `fohcG` and `fohcD`, such a technique does not work with more general computations on object-level formulas. For example, it seems impossible to provide a simple specification of the `nnf` predicate in Figure 7.4 in *hohc*. The lack of universal quantification in goals seriously restricts the ability of *hohc* to compute on λ-terms by recursion.

It is worth contrasting the two specifications of interpreters for object-level *fohc*: the one using L_λ (Figure 7.8) and the one using *hohc* (Figure 8.8). The one written in L_λ provides a specification of object-level substitution while the one in *hohc* uses $\beta\eta$-conversion to affect object-level substitution: hence, the latter interpreter is more succinct. Such achieve this succinctness, the λProlog interpreter (as an interperter for *hohc*) must implement the powerful operation of $\beta\eta$-conversion and the associated operation of $\beta\eta$-unification. For this task of implementing an interpreter for an object-level logic, it is an interesting question if the complexity related to providing $\beta\eta$-conversion is really needed. Clearly, we only need to handle declaratively object-level substitution for this problem, so the L_λ solution seems more appropriate for this kind of task. It is possible to show, however, that, from a

```
module fohc_check.
accumulate ot_logic1.

type  fohcG, fohcD   form -> o.
type  dummy          term.

fohcG truth.
fohcG (and B C) &
fohcG (or  B C) :- fohcG B, fohcG C.
fohcG (some B)  :- fohcG (B dummy).
fohcG A         :- atom A.

fohcD (imp G D)   :- fohcG G, fohcD D.
fohcD (and D1 D2) :- fohcD D1, fohcD D2.
fohcD (all D)     :- fohcD (D dummy).
fohcD A           :- atom A.
```

Figure 8.9: An *hohc*-based specification of the classes of goals and definite formulas for object-level *fohc*.

high-level point-of-view, an interpreter for *hohc*, when given this problem, will behavior in a "conservative" fashion: the unification problems it will attempt will be simple to compute and can be related to operations that the L_λ interpreter will do with the specification it is given. It is the case that a particular λProlog system may implement various parts of the logic of λProlog differently: for example, if λ-conversion is well supported, the L_λ implementation may be less efficient, where as, if the logical connectives, particularly implications and universal quantifiers in goals, are efficiently implemented, the L_λ specification may be more efficient.

8.5 Partial definition of some logical connectives

In Section 5.5.3 we presented a goal formula that succeeds once and another that always fails. In a certain sense, these were "definitions" of `true` and `false`. Using *hohc* it is possible to define, in a certain, weak sense, the logical constants $\top$, $\vee$, and $\exists$. The clauses in Figure 8.10 describe an implementation of these three constants. This are not complete definitions since these clauses only supply half of the meaning of logical connectives. In particular, they describe how to prove a formula using that connective but they do not describe how such formulas are used in a proof. For example, the rule of cases, which describes how a disjunctive assumption can be used in a proof, is not specified by this inference rule. In the language of the sequent calculus (Subsection 3.3), the clauses in Figure 8.10 specify the right-introduction rules but not the left-introduction rules for those connectives.

Let $\mathcal{P}$ and G be a *hohc* program and goal that have no occurrences of the constants `tt`, `or`, `exists`, and let $\mathcal{D}$ be the set of clauses in Figure 8.10. Also, let $\mathcal{P}'$ and G' be the result of replacing all occurrences of `true`, `;`, and `sigma` in $\mathcal{P}$ and G with `tt`, `or`, and `exists`, respectively. Then G is provable from $\mathcal{P}$ if and only if G' is provable from $\mathcal{P}' \cup \mathcal{D}$.

```
type tt      o.
type or      o -> o -> o.
type exists  (A -> o) -> o.

tt.
or P Q :- P.
or P Q :- Q.
exists B :- B T.
```

Figure 8.10: The "definition" of some logical constants.

Furthermore, cut-free proofs that G follows from $\mathcal{P}$ correspond directly to cut-free proofs that G' follows from $\mathcal{P}' \cup \mathcal{D}$: the right-introduction of $\top$, $\vee$, and $\exists$ in one proof corresponds the backchaining over one of the clauses in Figure 8.10 in the other proof.

8.6 Two examples of bad programs

There are abysses within declarative programming which programmers must learn about and avoid. In Prolog there is left-recursion. In λProlog there are additional pitfalls involving higher-order quantification and $\beta\eta$-unification.

One common pitfall is an attempt to use $\beta\eta$-unification to perform "extraction". For example, given the signature

```
kind i   type.
type a,b i.
type f   i -> i -> i.
```

consider the problem of taking a term of type `i` and forming the λ-abstraction that results from replacing all occurrences of `a` with the abstracted variables. For example, given the term `(f a (f a b))` the intended abstraction would be `(x\ f x (f x b))`. The specification

```
type extract_a   i -> (i -> i) -> o.
extract_a (F a) F.
```

will not, in general, compute the intended extraction. For example, given this specification, the query

```
?- extract_a (f a (f a b)) F.
F == x\ f x (f x b);
F == x\ f x (f a b);
F == x\ f a (f x b);
F == x\ f a (f a b);
no

?-
```

produces four solutions, only the first of which is the intended expression. Of course, if unifiers are produced in the order shown above, it would be possible to use cut to eliminate all but the first result.

```
extract_a (F a) F :- !.
```

Instead of using this non-logical aspect of λProlog, it is possible to be more explicit in the specification of this predicate. The following specification performs the intended extraction.

```
extract_a a x\x.
extract_a b x\b.
extract_a (f T S) (x\ f (U x) (V x)) :-
  extract_a T U, extract_a S V.
```

This specification is another example of a signature-dependent specification.

The specification of term rewriting is often attempted using a direct but naive use of higher-order quantification. Consider, for example, the following predicate.

```
type rewrite   int -> int -> o.

rewrite (0 + X) X.
rewrite (1 * X) X.
rewrite (X - X) 0.
rewrite (C X) (C Y) :- rewrite X Y.
```

The first three clauses specify certain rewriting steps. The last clause is a naive specification of the fact that two terms are equal if a subexpression is replaced by an equal subexpression. The following are provable instances of this predicate.

```
rewrite ((5 - 5) + 6) 6.
rewrite ((1 * 5) - 5) 0.
```

While these are all examples of the intended meaning of this predicate, this specification has far too many proofs to be interesting. For example, any provable goal has an infinite number of different proofs. Clearly, this specification gives rise to a poor implementation in λProlog.

Chapter 9

Higher-order hereditary Harrop formulas

Each of the languages L_λ and *hohc* contain features not contained in the other. The only remaining design issue for the logical foundation of λProlog is to come up with a single logic programming language in which both of these languages exist. For this purpose, we present the intuitionistic theory of *higher-order hereditary Harrop formulas*, or *hohh* for short, as a logic programming language that contains both *hohc* and L_λ (and, hence, *fohh*). After introducing *hohh*, we shall also strengthen it to $hohh^+$ much as we did when *fohh* was strengthened to $fohh^+$. With this chapter we shall have finished describing the various logic programming languages contained in Figure 1.1.

There are two logical features of *hohc* that are lacking from L_λ. First, there is no restriction on what essentially existential variables are applied to in *hohc*, whereas in L_λ they can be applied to at most distinct essentially universal variables. Second, it is possible to quantify over predicates and to place logical connectives within terms within *hohc*, while neither of these features were allowed in L_λ. We discuss each of these features in the next two sections.

9.1 Removing restrictions from L_λ

One interesting aspect of the L_λ logic programming language is that it could compute on the structure of λ-terms even though an implementation of it involved a unification algorithm that had properties familiar to those of first-order unification: namely, such unification is decidable and unary (most general unifiers exist when unifiers exist). If the restriction of the application of essentially existential variables if removed from L_λ, an implementation of the resulting language needs to invoke full $\beta\eta$-unification, which, as we have pointed out before, is undecidable and not unary. Giving up these features of L_λ may seem like a high price to pay, but there are at least three reasons to pursue such an extension to L_λ.

First, L_λ does not express object-level substitution directly: as we have seen in Section 7.5, it must be implemented. Although such an implementation using `copy`-clauses is simple and natural, it is sensible to consider a more direct treatment of something as simple and declarative as object-level substitution. Allowing richer forms of application for essentially existential variables requires β-redexes, not just β_0-redexes to be reduced. As we have seen in Section 8.3.3, β-reduction can achieve such a direct specification of object-level substitution. As a result, many calls to `subst`-goals can simply be replaced by meta-level applications. Consider, for example, the interpreter for object-level *fohc* in Figure 7.8, where `subst` was used to do object-level substitution, and the interpreter in Figure 8.8, where meta-level β-reduction achieved object-level substitution.

Second, it should be expected that any implementation of $\beta\eta$-unification will actually provide an effective implementation of L_λ-unification. Huet's procedure for $\beta\eta$-unification [Hue75], can be easily modified to do L_λ-unification: the chief modification is that flexible-flexible pairs in the L_λ-setting can be solved uniquely, where as Huet's procedure do not attempt to solve them. Thus, if one attempts to use an L_λ program in an interpreter designed to handle all of *hohh*, the interpreter should provide an effective implementation of the L_λ program in that it should decide all unification programs as well as pick only most general unifiers when a unifier exists. Those, one should pay the price of using the richer language only when one explicitly leaves the weaker language.

em The following statement is false. The following is in the core.

```
(pi x\ p (f x)) => p (abs F)
```

Third, the core of L_λ is unfortunately small, being the same as the core of *fohh*. Thus, L_λ does not introduce any new programs that can be reasoned about (as opposed to only being used in reasoning from). The logic of *hohh*, presented in this chapter, has a much larger core; in fact that core contains L_λ.

9.2 Adding predicate quantification to L_λ

When *fohc* was extended to *hohc* in Chapter 8, two design problems were addressed: what kind of logical connectives could appear in terms and whether or not the the head of a clause could be an essentially existential variable. The resolution of these design problems for *hohc* was to allow the logical connectives that were allowed within goals, namely $\top$, $\wedge$, $\vee$, and $\exists$, to appear within terms and to not allow the head of clauses to be essentially existential variables. When considering a similar extension to *fohh* and L_λ to *hohh*, we need to reconsider these design issues again.

9.2.1 Heads of clauses

The restriction that the head of a clause in *hohc* could not be an essentially existentially quantified variable was made for two reasons: allowing such a variable to be the head of a clause can lead to inconsistent programs and to program clauses that add meaning to all predicates in the language.

While these are still important considerations in the design of a higher-order version of hereditary Harrop formulas, if this ban on existentially quantified predicates is interpreted broadly, it would rule out an interesting and useful programming technique. Consider, for example, proving the goal

```
?- sigma P\ convert d P, P => g.
```

where `g` is some formula, `P` is a variable of type `o`, and `d` is some specification of a program clause. Assume also that the current context contains clauses that define the binary predicate `convert` that relates a specification of a clause to a program clause (a specific example of just such a computation is Section 9.4). Thus we have described a computation that computes a λProlog program from the description `d` and then executes the resulting program against the goal `g`. Such a computation is essentially a way to do Lisp's `eval` operation: it allows a term to be turned into a program. Notice, however, that the second occurrence of `P` is in the above example is in the position of a definitive clause: that is, the entire clause (not just its head) is an essentially existential variable. Imposing the constraint described for *hohc* on *hohh* would rule out this example and this ability to do evaluation-like tasks.

Of course, having the ability to do this style of evaluation can be problematic as well as powerful. In particular, it could be the case that the result of converting `d` in the above example is, in fact, not a proper *hohh* clauses (whatever this is eventually defined to be). Thus, after conversion is completed, an attempt to add an invalid formula to the current context would be made and this would cause a run-time error in a λProlog implementation. Avoiding such runtime errors is one of the motivations for defining *hohh* in Section 9.3 so that the head of clauses cannot be existentially quantified predicates. That is, the above example will not be allowed in the formal definition of *hohh*, but will be allowed in λProlog: when code departs from the formal definition of *hohh*, runtime errors might be generated.

9.2.2 Logical connectives in terms

Consider the proposal to permit all the connectives that can appear as the top-level connective of goal formulas in *fohh* to appear within atomic formulas of *hohh*. This choice is similar to that made for extending *fohc* to *hohc*. Thus consider allowing $\supset$ and $\forall$ to appear within atomic formulas, along with $\top$, $\wedge$, $\vee$, and $\exists$. The resulting language, unfortunately, does not constitute a logic programming language in the sense that there are provable formulas that do not have goal-directed proofs (Section 3.3). For example, consider the formula [MNPS91]

$$\exists Q[\forall p \forall q[r(p \supset q) \supset r(Qpq)] \wedge Q(t \vee s)(s \vee t)],$$

where r is a constant of type $o \to o$, s and t are constants of type o, Q is a variable of type $o \to o \to o$, and p and q are constants of type o. This formula has exactly one cut-free proof, obtained by using $\lambda x \lambda y(x \supset y)$ as the substitution term for the existentially quantified variable Q. A proof of this goal must reduce to proving the goal $(t \vee s) \supset (s \vee t)$. Since this formula does not have a goal-directed proof, there can be no uniform proof for the original sequent.

This example reveals the following problem with allowing, in particular, implications within terms. When implications are not allowed in terms, as in *hohc*, it is easy to determine the polarity of all logical connectives within *hohc*; namely, following a higher-order substitution, logical formulas within atomic formulas can possibility become goal formulas but they can never become definite clauses. Thus in *hohc*, a substitution instance of the goal $Q(t \vee s)(s \vee t)$ will place both of these disjunctions into positions consistent with the definition of goal formulas. Introducing implications into atomic formulas, however, makes it impossible, in general, to determine the polarity of those logical connectives within atomic formulas. In the above example, Q is instantiated with a formula containing an implication, so the resulting instance of $Q(t \vee s)(s \vee t)$, namely $(t \vee s) \supset (s \vee t)$, is no longer a formula for which goal-directed provability is complete. This example suggests that one way to maintain completeness of goal-directed proofs, implications must be eliminated from within terms. This is, indeed, the approach taken in the definition of *hohh* in the next section. A. Felty has shown that if disjunctions and existentials are not allowed within formulas and terms, then it is possible to allow implications within terms and still maintain the completeness of goal-directed provability [Fel93].

9.3 The definitions of *hohh* and *hohh*$^+$

Two possible problems in designing *hohh* have been discussed. In each case, we can define things so that the problems disappear but this also rules out some interesting and useful programming techniques. Thus we take the following approach: we define two classes of formulas, *hohh* and *hohh*$^+$, which are restricted and for which no run-time errors can occur. On the other hand, we allow λProlog to be based on a richer set of formulas than these, a set that contains the more interesting programs but which could cause run-time errors. The first languages provide a secure framework for the mixing of higher-order programming features and L_λ, but we allow λProlog programmers to specify richer and possibly insecure programs.

Let $\mathcal{H}_2$ be the set of λ-normal terms that do not contain occurrences of the logical constants $\supset$ and $\perp$. In other words, the only logical constants that terms in $\mathcal{H}_2$ may contain are $\top$, $\wedge$, $\vee$, $\forall$, and $\exists$. Let the syntactic variable A denote atomic formulas in $\mathcal{H}_2$ and let the syntactic variable A_r denote rigid atomic formulas in $\mathcal{H}_2$. Then the goal formulas and definite clause of *hohh* are defined by the following mutual recursion:

$$\begin{array}{rcl} G & := & \top \mid A \mid G_1 \wedge G_2 \mid G_1 \vee G_2 \mid \forall x\, G \mid \exists x\, G \mid D \supset G \\ D & := & A_r \mid G \supset A_r \mid \forall x\, D \mid D_1 \wedge D_2. \end{array}$$

Quantification here may be over variables of any type. The D-formulas will be called *higher-order hereditary Harrop formulas*. It is proved in [MNPS91] that goal-directed search in intuitionistic logic is complete in this setting.

This definition can be liberalized a bit to *hohh*$^+$, where richer quantification of predicates is allowed and where goal-directed proof search is still complete for intuitionistic logic. For this extension, remove the restriction on rigid atomic formulas in D, formulas, that is, use the following clauses instead

$$\begin{array}{rcl} G & := & \top \mid A \mid G_1 \wedge G_2 \mid G_1 \vee G_2 \mid \forall x\, G \mid \exists x\, G \mid D \supset G \\ D & := & A \mid G \supset A \mid \forall x\, D \mid D_1 \wedge D_2, \end{array}$$

but add the proviso that the head of any atom that appears in a D formula cannot be a variable that is essentially existentially quantified. Let *hohh*$^+$ be the result of using this definition of D-formulas instead. The difference between *hohh* and *hohh*$^+$ is that the head of a clause in *hohh* can only be a constant while in *hohh*$^+$ it can also be essentially universally quantified.

In both of these settings, quantifiers need only be instantiated using terms from the set $\mathcal{H}_2$: for this reason, we may think of $\mathcal{H}_2$ as the Herbrand Universe for higher-order hereditary Harrop formulas.

We shall allow λProlog programs to be build from a set of formulas larger than *hohh*$^+$, in that atomic formulas can contain implications and goal formulas and definite clauses can be defined using the same recursive clauses used for *hohh*$^+$, except that the proviso is not employed. We shall assume, however, that a λProlog interpreter for this language will refuse to continue the search for a proof if a formula is being moved into the current program space that does not have a top-level structure that conforms to the definition used for *hohh*$^+$: that is, if the formula has positive occurrence of a disjunction or existential quantifier or if it has a positive occurrence of an atomic formula whose head symbol is essentially existentially quantified.

Consider the example of the `convert` query given above. While this is not a legal goal in the sense of *hohh*$^+$, a λProlog interpreter should process it, attempting to find an instance of `P` that is the result of `converting d`. Once that instance is found, the interpreter will attempt to add it to the current program, and it is then that it's top-level structure will be examined. If it does not satisfy the criterion mentioned about various positive subformulas, the interpreter will produce a run-time error and stop searching for a proof. If, however, the `convert` predicate is written so that if it is proved, its second argument is always a valid definite clause, then no such runtime error will be caused.

The demands of a compiler on a programming language are often quite different from those of an interpreter. In particular, compilers often examine program code much more deeply in order to find ways to avoid various kinds of runtime checks. Thus, it seems sensible that compilers of λProlog might require programs to be taken from *hohh*$^+$, for example, since no runtime errors of this kind can occur.

9.4 Examples of *hohh*$^+$ programs

Since the design of *hohh*$^+$ is based on the intuitionistic theory of hereditary Harrop formulas, which supports modular programming, and on a higher-order theory of predicates and λ-terms, which supports higher-order programming, *hohh*$^+$ supports a mixing of such programming styles. Such mixing is quite natural and unproblematic given that this design has been carried out using logical notions. In contrast, most approaches to adding higher-order and modular programming features to Prolog are based on non-logical and *ad hoc* constructions. As a result, the mixing of these two styles of programming can cause serious semantic problems. Below we illustrate examples of how these two styles of programming can be mixed.

9.4.1 Mixing modular and higher-order programming

The program in Figure 9.1 computes part of the Fibonacci relation (see also Subsection 5.4.7), loads that part into the current context, and calls a goal that is parameterized by the name of the binary predicate that is used to store the Fibonacci relation. For example, using this code, the following query could be used to search for all numbers `N` between 0 and 100 inclusively such that the `N`th Fibonacci number is the square of `N`.

```
?- fib_memo 100 (fib\ fib N M, M is N * N).
```

(There are exactly three pairs of values for `N` and `M` that satisfies this predicate.) Using this style of programming, the necessary Fibonacci relation is computed iteratively (using the local predicate `loop`) prior to accessing parts of it.

Two simple *hohh*$^+$ programs are presented in the modules given in Figures 9.2 and 9.3. Both programs are similar in structure and attempt to mimic a notion of state encapsulation and both programs make use of *hohh*$^+$ along with the cut control operator.

In the first of these modules, a named switch that can be turned on and off is specified. The status of the switch is stored as in the current context using the atomic formula `(sw Name V)`, where `sw` is a predicate local to the module, `Name` is a string used to name

```
module fibmemo.

type fib_memo   int -> ((int -> int -> o) -> o) -> o.

local memo      int -> int -> o.
local loop      int -> int -> int -> o -> o.

fib_memo N G :- memo 0 0 => memo 1 1 => loop 0 1 N (G memo).

loop N1 N2 N2 G :- G.
loop N1 N2 M  G :- N2 < M, memo N1 F1, memo N2 F2, N3 is N2 + 1,
                   F is F1 + F2, memo N3 F => loop N2 N3 M G.
```

Figure 9.1: An implementation of a memo-ized version of the Fibonacci predicate.

```
module switch.

kind  onoff      type.
type  on, off    onoff.
local sw         string -> onoff -> o.

type setsw       string -> onoff -> o -> o.
type getsw       string -> onoff -> o.
type toggle      string ->          o -> o.

sw Name on.

setsw  Name OnOff G :- sw Name OnOff => G.
getsw  Name OnOff   :- sw Name X, !, OnOff = X.
toggle Name       G :- getsw Name on,  setsw Name off G;
                       getsw Name off, setsw Name on  G.
```

Figure 9.2: An implementation of named switches.

```
module registers.

local reg    string -> int -> o.
type setreg  string -> int -> o -> o.
type getreg  string -> int -> o.

reg    Name 0.
setreg Name N G :- reg Name N => G.
getreg Name N   :- reg Name M, !, N = M.

type modifyreg   string -> (int -> int) -> o -> o.
type incremreg   string                 -> o -> o.

modifyreg Name F G :- getreg Name V, U is (F V),
                      setreg Name U G.
incremreg Name   G :- modifyreg Name (x\ x + 1) G.
```

Figure 9.3: An implementation of named registers.

this switch, and `V` is the value of the switch, either `on` or `off`. The only access that is given to a switch is via the `setsw`, `getsw`, and `toggle`. The first of these predicates is used to set the switch to a particular value, the second is used to read the value of the switch, and the third toggles the switch. Notice that the two predicates that can change a switch's value are written in a *continuation-passing style* similar to that discussed briefly in Section 8.1. As a switch's value is changed, more and more assumptions about it's setting are made; of course, it is only the most recent setting which is the one that is ever desired: hence, the cut (`!`) in the definition of `getsw` is necessary.

The code in Figure 9.3 is similar: in this case, registers store integer values and these registers can be changed either by setting them to arbitary values (using `setreg`, doing an increment (using `incremreg`), or by applying an arbitrary integer valued function (using `modifyreg`).

9.4.2 Assuming computed clauses

Figure 9.4 contains an example of computing a program clause that is then assumed. Given the clauses in that figure, the goal

```
?- define s (natjoin r (iter t (1::2::nil))) (s 1 X).
```

would first translate the "specification"

```
(natjoin r (iter t (1::2::nil)))
```

using `trans_spec` to the predicate expression

```
x\y\ sigma z\
  r x z,
  pi aux\
     pi U\(aux nil U U),
```

```
module reltrans.

type r, s, t     int -> int -> o.

r 1 2.
r 2 3.

t 2 3.
t 3 4.

type trans_spec      (A -> A -> o) -> (A -> A -> o) -> o.
type iter            (A -> A -> o) -> list A -> (A -> A -> o) -> o.
type natjoin, union  (A -> A -> o) -> (A -> A -> o) -> A -> A -> o.

trans_spec r r.
trans_spec t t.
trans_spec (natjoin R S) x\y\(sigma z\(G x z, H z y)) :-
   trans_spec R G, trans_spec S H.
trans_spec (union R S) x\y\(G x z; H z y) :-
   trans_spec R G, trans_spec S H.
trans_spec (iter R L)
   x\y\( pi aux\( pi U\(aux nil U U) &
                  pi U\(pi V\(pi W\(pi Z\(pi M\(
           aux (U::V) W Z :- G U W M, aux V M Z))))))
         => (aux L x y))
  :- trans_spec R G.

type define   (A -> A -> o) ->  (A -> A -> o) -> o -> o.

define Name Spec Goal :-
  trans_spec Spec G, (pi X\(pi Y\(Name X Y :- G X Y))) => Goal.
```

Figure 9.4: A translator for relational calculus specifications.

```
     pi U\(pi V\(pi W\(pi Z\(aux (U::V) W Z :-
                              sigma M\(t U W M, aux V M Z))))))
      => aux (1::2::nil) z y
```

The action of the `define` clause will then extend the current program with the clause

```
s X Y :-
  sigma z\ r X Z,
  pi aux\
     ((pi U\ aux nil U U),
      (pi U\ pi V\ pi W\ pi Z\ aux (U::V) W Z :-
                              sigma M\ t U W M, aux V M Z))
       => aux (1::2::nil) Z Y
```

and then evaluate the goal formula `(s 1 X)`, the result of which will bind `X` to `4`.

While this computation seems sensible, there are at least two potential problems with the clause for the `define` predicate in Figure 9.4. First, the first argument of `define` may not be bound to a "name", such as the constant `s` used in the above example. It could be the case that a logic variable is used for the name. In this case, the current program will be extended with a program clause with no fixed head. Such clauses can lead to inconsistent current programs, as was described along with higher-order clauses (Chapter 8). Second, this clause for `define` could produce a run-time error, in the sense that a formula other than a hereditary Harrop formula might get added to the current program.

The first of these two problems can be repaired if we actually intend that the variable `Name` ranges over only names. In that case, we should not give `Name` the type `A -> A -> o` but rather the type `string`, in which case, we could need to use a new non-logical constant `pred` of type `string -> A -> A -> o` to coerce a name into a binary predicate. Thus the `define` clause could be changed to be

```
type define   string -> (A -> A -> o) -> o -> o.

define Name Spec Goal :-
  trans_spec Spec G, (pi X\ pi Y\ pred Name X Y :- G X Y) => Goal.
```

with the additional change that we need to invoke `(pred Name)` instead of simply `Name`.

The second problem, namely the fact that

```
pi X\ pi Y\ pred Name X Y :- G X Y
```

may not be a *hohh* clause is not so easily addressed since it depends on the behaviour of `define`.

9.5 Logical properties of *hohh*$^+$

The higher-order intuitionistic theory of hereditary Harrop formulas is a rich theory. To the extent that λProlog actually implements part of this theory, we can make use of mathematical properties of that theory to conclude results about programs written in λProlog. In this section, we illustrate some examples of using such mathematical properties of logic to infer properties of λProlog programs.

In Section 3.3 we mentioned that the sequent calculus could be used to describe some aspects of the run-time behavior of λProlog programs. As was mentioned there, an idealized interpreter can be seen as building cut-free proofs. A consequence of the cut-elimination result of sequent calculus can be stated as follows: if the goal $M \supset G$ is provable from program $\mathcal{P}_1$ and M is provable from program $\mathcal{P}_2$, then the goal G follows from the program $\mathcal{P}_1 \cup \mathcal{P}_2$. (The formula M must be in the core of the language being considered.) A similar result also holds for universal quantification: if the goal $\forall x\ G$ is provable from program $\langle \Sigma, \mathcal{P} \rangle$ and t is a Σ-term of the same type as x, then the goal $[t/x]G$ follows from $\langle \Sigma, \mathcal{P} \rangle$. Of course, these facts cannot be inferred to be a property of λProlog in general since λProlog is incomplete with respect to provability: it can only be used on those programs for which it can be shown that λProlog will, in fact, be complete.

For a simple example of using these properties, consider the `fohhD` predicate in Figure 7.2. This predicate can be used to determine whether or not its argument is an object-level first-order hereditary Harrop formula. From the definition of this class of formulas (say, in Section 5.1), it is clear that if a universally quantified formula is a *fohh* program clause, so is any instance of that formula. It is a nice observation that this is also true of the predicate that we have specified in Figure 7.2. The proof of this fact goes as follows: Assume that `fohh (all D)` is provable and let `T` be a term of type `term`. Then, since there is only one way this goal can be proved, it must be the case that `pi x\(fohhD (D x))` is provable. Thus, using the property about universal quantification mentioned above, it is the case that `(fohhD (D T))` is provable. Given that the logic we are using contains $\beta\eta$-conversion, the term `(D T)` is convertible (hence, equal) to the result of substituting `T` for the outermost bound variable of `D`. Thus, if `(forall D)` is an object-level *fohh*, then so too is the instance `(D T)`.

For a more interesting example of such reasoning, consider the `subst` predicate in Figure 7.6. We shall show that if `(subst M T S)` is provable, then `S` is equal to the result of substituting `T` for the bound variable in `M`. So, assume that `(subst M T S)` is provable. Then it must be the case that

```
pi x\(copyterm x T => copyform (M x) S).
```

is probable. But clearly, the goal `(copyterm T T)` is also provable. Thus, using the observation about universal quantifiers above, we know that

```
copyterm T T => copyform (M T) S
```

is provable. Using the observation about implication above, we know that

```
copyform (M T) S
```

is provable. But this is provable if and only if the term `(M T)` and `S` are equal (modulo $\beta\eta$-conversion). Thus, `S` is equal to the result of substituting `t` for the bound variable of `M`.

Index

Bibliography

[ACMP84] Peter B. Andrews, Eve Longini Cohen, Dale Miller, and Frank Pfenning. Automating higher order logic. In *Automated Theorem Proving: After 25 Years*, pages 169–192. American Mathematical Society, Providence, RI, 1984.

[And71] Peter B. Andrews. Resolution in type theory. *Journal of Symbolic Logic*, 36:414–432, 1971.

[And86] Peter B. Andrews. *An Introduction to Mathematical Logic and Type Theory*. Academic Press, 1986.

[AvE82] K. R. Apt and M. H. van Emden. Contributions to the theory of logic programming. *Journal of the ACM*, 29(3):841–862, 1982.

[Bar84] Hank Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. Elsevier, revised edition, 1984.

[BR91] Pascal Brisset and Olivier Ridoux. Naïve reverse can be linear. In *Eighth International Logic Programming Conference*, Paris, France, June 1991. MIT Press.

[CH88] Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76(2/3):95–120, February/March 1988.

[Chu40] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[CM84] W. F. Clocksin and C. S. Mellish. *Programming in Prolog*. Springer-Verlag, 1984.

[Ell89] Conal Elliott. Higher-order unification with dependent types. In *Rewriting Techniques and Applications*, volume 355, pages 121–136. Springer-Verlag LNCS, April 1989.

[EP89] Conal Elliott and Frank Pfenning. eLP, a Common Lisp Implementation of λProlog. Implemented as part of the CMU ERGO project, May 1989.

[Fel93] Amy Felty. Encoding the calculus of constructions in a higher-order logic. In M. Vardi, editor, *Eighth Annual Symposium on Logic in Computer Science*, pages 233–244. IEEE, June 1993.

[FM88] Amy Felty and Dale Miller. Specifying theorem provers in a higher-order logic programming language. In *Ninth International Conference on Automated Deduction*, pages 61–80, Argonne, IL, May 1988. Springer-Verlag.

[Gal86] Jean H. Gallier. *Logic for Computer Science: Foundations of Automatic Theorem Proving*. Harper & Row, 1986.

[Gen69] Gerhard Gentzen. Investigations into logical deductions. In M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131. North-Holland Publishing Co., Amsterdam, 1969.

[Gir86] Jean-Yves Girard. The system F of variable types: Fifteen years later. *Theoretical Computer Science*, 45:159–192, 1986.

[Gir87] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[Goe65] Kurt Goedel. On formally undecidable propositions of the principia mathematica and related systems. I. In *Martin Davis, The Undecidable*. Raven Press, 1965.

[GTL89] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and Types*. Cambridge University Press, 1989.

[Han89] Michael Hanus. Polymorphic higher-order programming in prolog. In Giorgio Levi and Maurizio Martelli, editors, *Proceedings of the Sixth International Conference*, pages 382 – 397. MIT Press, 1989.

[Hen50] Leon Henkin. Completeness in the theory of types. *Journal of Symbolic Logic*, 15:81–91, 1950.

[HHP87] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. In *Second Annual Symposium on Logic in Computer Science*, pages 194–204, Ithaca, NY, June 1987.

[HL78] Gérard Huet and Bernard Lang. Proving and applying program transformations expressed with second-order patterns. *Acta Informatica*, 11:31–55, 1978.

[HL94] Pat Hill and John Lloyd. *The Gödel Programming Language*. MIT Press, 1994.

[HM91] Joshua Hodas and Dale Miller. Logic programming in a fragment of intuitionistic linear logic: Extended abstract. In G. Kahn, editor, *Sixth Annual Symposium on Logic in Computer Science*, pages 32–42, Amsterdam, July 1991.

[HM94] Joshua Hodas and Dale Miller. Logic programming in a fragment of intuitionistic linear logic. *Information and Computation*, 110(2):327–365, 1994.

[Hod94] Joshua S. Hodas. *Logic Programming in Intuitionistic Linear Logic: Theory, Design, and Implementation*. PhD thesis, University of Pennsylvania, Department of Computer and Information Science, May 1994. Available as University of Pennsylvania Technical Reports MS-CIS-92-28 or LINC LAB 269.

[HS86] J. Roger Hindley and Jonathan P. Seldin. *Introduction to Combinatory Logic and Lambda Calculus*. Cambridge University Press, 1986.

[Hue75] Gérard Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57, 1975.

[KNW93] Keehang Kwon, Gopalan Nadathur, and Debra Sue Wilson. Implementing a notion of modules in the logic programming language λprolog. In E. Lamma and P. Mello, editors, *Proceedings of the 1992 Workshop on Extensions to Logic Programming*, number 660 in LNCS. Springer-Verlag, 1993.

[KNW94] Keehang Kwon, Gopalan Nadathur, and Debra Sue Wilson. Implementing polymorphic typing in a logic programming language. *Computer Languages*, 20(1):25–42, 1994.

[Lei94] Daniel Leivant. *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 2 of *Handbook of Logic in Artificial Intelligence and Logic Programming*, chapter Higher-order logics, pages 229–321. Oxford University Press, 1994.

[LS86] J. Lambek and P. J. Scott. *Introduction to Higher Order Categorical Logic*. Cambridge University Press, 1986.

[Mil86] Dale Miller. A theory of modules for logic programming. In Robert M. Keller, editor, *Third Annual IEEE Symposium on Logic Programming*, pages 106–114, Salt Lake City, Utah, September 1986.

[Mil87a] Dale Miller. Hereditary harrop formulas and logic programming. In *Proceedings of the VIII International Congress of Logic, Methodology, and Philosophy of Science*, pages 153–156, Moscow, August 1987.

[Mil87b] Dale A. Miller. A compact representation of proofs. *Studia Logica*, 46(4):347–370, 1987.

[Mil90] Dale Miller. Abstractions in logic programming. In Piergiorgio Odifreddi, editor, *Logic and Computer Science*, pages 329–359. Academic Press, 1990.

[Mil91] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4):497–536, 1991.

[Mil92a] Dale Miller. Abstract syntax and logic programming. In *Logic Programming: Proceedings of the First and Second Russian Conferences on Logic Programming*, number 592 in Lecture Notes in Artificial Intelligence, pages 322–337. Springer-Verlag, 1992.

[Mil92b] Dale Miller. Unification under a mixed prefix. *Journal of Symbolic Computation*, pages 321–358, 1992.

[Mil94] Dale Miller. A multiple-conclusion meta-logic. In S. Abramsky, editor, *Ninth Annual Symposium on Logic in Computer Science*, pages 272–281, Paris, July 1994.

[Mil96] Dale Miller. Forum: A multiple-conclusion specification language. *Theoretical Computer Science*, 165(1):201–232, September 1996.

[MM91] John C. Mitchell and Eugenio Moggi. Kripke-style models for typed lambda calculus. *Annals of Pure and Applied Logic*, 51, 1991.

[MN85] Dale Miller and Gopalan Nadathur. A computational logic approach to syntax and semantics. Presented at the Tenth Symposium of the Mathematical Foundations of Computer Science, IBM Japan, May 1985.

[MN86a] Dale Miller and Gopalan Nadathur. Higher-order logic programming. In Ehud Shapiro, editor, *Proceedings of the Third International Logic Programming Conference*, pages 448–462, London, June 1986.

[MN86b] Dale Miller and Gopalan Nadathur. Some uses of higher-order logic in computational linguistics. In *Proceedings of the 24th Annual Meeting of the Association for Computational Linguistics*, pages 247–255. Association for Computational Linguistics, Morristown, New Jersey, 1986.

[MN87a] Dale Miller and Gopalan Nadathur. λProlog Version 2.6. Distribution in C-Prolog sources, August 1987.

[MN87b] Dale Miller and Gopalan Nadathur. A logic programming approach to manipulating formulas and programs. In Seif Haridi, editor, *IEEE Symposium on Logic Programming*, pages 379–388, San Francisco, September 1987.

[MNPS91] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

[MNS87] Dale Miller, Gopalan Nadathur, and Andre Scedrov. Hereditary Harrop formulas and uniform proof systems. In David Gries, editor, *Symposium on Logic in Computer Science*, pages 98–105, Ithaca, NY, June 1987.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

[Nad87] Gopalan Nadathur. *A Higher-Order Logic as the Basis for Logic Programming*. PhD thesis, University of Pennsylvania, May 1987.

[Nip93] Tobias Nipkow. Functional unification of higher-order patterns. In M. Vardi, editor, *Eighth Annual Symposium on Logic in Computer Science*, pages 64–74. IEEE, June 1993.

[NJK95] Gopalan Nadathur, Bharat Jayaraman, and Keehang Kwon. Scoping constructs in logic programming: Implementation problems and their solution. *Journal of Logic Programming*, 25(2):119–161, November 1995.

[NM90] Gopalan Nadathur and Dale Miller. Higher-order Horn clauses. *Journal of the ACM*, 37(4):777–814, October 1990.

[NP92] Gopalan Nadathur and Frank Pfenning. The type system of a higher-order logic programming language. In Frank Pfenning, editor, *Types in Logic Programming*, pages 245–283. MIT Press, 1992.

[Pau86] Lawrence C. Paulson. Natural deduction as higher-order resolution. *Journal of Logic Programming*, 3:237–258, 1986.

[Pau90] Lawrence C. Paulson. Isabelle: The next 700 theorem provers. In Piergiorgio Odifreddi, editor, *Logic and Computer Science*, pages 361–386. Academic Press, 1990.

[PE88] Frank Pfenning and Conal Elliot. Higher-order abstract syntax. In *Proceedings of the ACM-SIGPLAN Conference on Programming Language Design and Implementation*, pages 199–208. ACM Press, June 1988.

[Pfe91] Frank Pfenning. Unification and anti-unification in the Calculus of Constructions. In G. Kahn, editor, *Sixth Annual Symposium on Logic in Computer Science*, pages 74–85. IEEE, July 1991.

[Pfe92] Frank Pfenning, editor. *Types in Logic Programming*. MIT Press, 1992.

[Pym90] David Pym. *Proofs, Search and Computation in General Logic*. PhD thesis, LFCS, University of Edinburgh, 1990.

[Qia93] Zhenyu Qian. Linear unification of higher-order patterns. In J.-P. Jouannaud, editor, *Proc. 1993 Coll. Trees in Algebra and Programming*. Springer Verlag LNCS, 1993.

[Sha85] Steward Shapiro. Second-order languages and mathematical practice. *Journal of Symbolic Logic*, 50(3):714–742, September 1985.

[SS86] Leon Sterling and Ehud Shapiro. *The Art of Prolog: Advanced Programming Techniques*. MIT Press, Cambridge MA, 1986.

[Sta79] Richard Statman. The typed λ calculus is not elementary recursive. *Theoretical Computer Science*, 9:73–81, 1979.

[Tak67] M. Takahashi. A proof of cut-elimination theorem in simple type-theory. *Journal of the Mathematical Society of Japan*, 19:399–410, 1967.

[Tar92] Paul Tarau. Program transformations and WAM-support for the compilation of definite metaprograms. In *Logic Programming: Proceedings of the First and Second Russian Conferences on Logic Programming*, number 592 in LNAI, pages 462–473. Springer-Verlag, 1992.

[Wad91] William W. Wadge. Higher-order Horn logic programming. In *Proceedings of the 1991 International Symposium on Logic Programming*, pages 289–303, October 1991.