CS-1994-38

Higher-Order Logic Programming

Gopalan Nadathur
Dale Miller

Department of Computer Science

Duke University

Durham, North Carolina 27708–0129

December 1994

Higher-Order Logic Programming[†]

Gopalan Nadathur[‡]
Computer Science Department, Duke University
Durham, NC 27706
gopalan@cs.duke.edu
Phone: +1 (919) 660-6545, Fax: +1 (919) 660-6519

Dale Miller
Computer Science Department, University of Pennsylvania
Philadelphia, PA 19104-6389 USA
dale@saul.cis.upenn.edu
Phone: +1 (215) 898-1593, Fax: +1 (215) 898-0587

[†] This paper is to appear in the *Handbook of Logic in Artificial Intelligence and Logic Programming*, D. Gabbay, C. Hogger and A. Robinson (eds.), Oxford University Press.

[‡] This address is functional only until January 1, 1995. After this date, please use the following address: Department of Computer Science, University of Chicago, Ryerson Laboratory, 1100 E. 58th Street, Chicago, IL 60637, Email: gopalan@cs.uchicago.edu.

Contents

1 Introduction

Modern programming languages such as Lisp, Scheme and ML permit procedures to be encapsulated within data in such a way that they can subsequently be retrieved and used to guide computations. The languages that provide this kind of an ability are usually based on the functional programming paradigm, and the procedures that can be encapsulated in them correspond to functions. The objects that are encapsulated are, therefore, of higher-order type and so also are the functions that manipulate them. For this reason, these languages are said to allow for *higher-order programming.* This form of programming is popular among the users of these languages and its theory is well developed.

The success of this style of encapsulation in functional programming makes is natural to ask if similar ideas can be supported within the logic programming setting. Noting that procedures are implemented by predicates in logic programming, higher-order programming in this setting would correspond to mechanisms for encapsulating predicate expressions within terms and for later retrieving and invoking such stored predicates. At least some devices supporting such an ability have been seen to be useful in practice. Attempts have therefore been made to integrate such features into Prolog (see, for example, [War82]), and many existing implementations of Prolog provide for some aspects of higher-order programming. These attempts, however, are unsatisfactory in two respects. First, they have relied on the use of *ad hoc* mechanisms that are at variance with the declarative foundations of logic programming. Second, they have largely imported the notion of higher-order programming as it is understood within functional programming and have not examined a notion that is intrinsic to logic programming.

In this chapter, we develop the idea of higher-order logic programming by utilizing a higher-order logic as the basis for computing. There are, of course, many choices for the higher-order logic that might be used in such a study. If the desire is only to emulate the higher-order features found in functional programming languages, it is possible to adopt a "minimalist" approach, *i.e.*, to consider extending the logic of first-order Horn clauses — the logical basis of Prolog — in as small a way as possible to realize the additional functionality. The approach that we adopt here, however, is to enunciate a notion of higher-order logic programming by describing an analogue of Horn clauses within a rich higher-order logic, namely, Church's Simple Theory of Types [Chu40]. Following this course has a number of pleasant outcomes. First, the extension to Horn clause logic that results from it supports, in a natural fashion, the usual higher-order programming capabilities that are present in functional programming languages. Second, the particular logic that is used provides the elegant mechanism of λ-terms as a means for constructing descriptions of predicates and this turns out to have benefits at a programming level. Third, the use of a higher-order logic blurs the distinction between predicates and functions — predicates correspond, after all, to their characteristic functions — and this makes it natural both to quantify over functions and to extend the mechanisms for constructing predicate expressions to the full class of functional expressions. As a consequence of the last aspect, our higher-order extension to Horn clauses contains within it a convenient means for representing and manipulating objects whose structures incorporate a notion of binding. The abilities that it provides in this respect are not present in *any* other programming paradigm, and, in general, our higher-order Horn clauses lead to a substantial enrichment of the notion of computation within logic programming.

The term "higher-order logic" is often a source of confusion and so it is relevant to explain the sense in which it is used here. There are 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, 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 and we use the term only in the second and third senses here. Note that these two senses are distinct. Thus, a logic can be a higher-order one 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. One such proposal appears in [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].

Developing a theory of logic programming within higher-order logic has another fortunate outcome. The clearest and strongest presentations of the central results regarding higher-order Horn clauses require the use of the sequent calculus: resolution and model theory based methods that are traditionally used in the analysis of first-order logic programming are either not useful or not available in the higher-order context. It turns out that the sequent calculus is an apt tool for characterizing the intrinsic role of logical connectives within logic programming and a study such as the one undertaken here illuminates this fact. This observation is developed in a more complete fashion in [MNPS91] and [Mil94] and in the chapter by Loveland and Nadathur in this volume of the Handbook.

2 Motivating a Higher-Order Extension to Horn Clauses

We are concerned in this chapter with providing a principled basis for higher-order features in logic programming. Before addressing this concern, however, it is useful to understand the higher-order additions to this paradigm that are motivated by programming considerations. We explore this issue in this section by discussing possible enhancements to the logic of first-order Horn clauses that underlies usual logic programming languages.

Horn clauses are traditionally described as the universal closures of disjunctions of literals that contain at most one positive literal. They are subdivided into *positive* Horn clauses that contain exactly one positive literal and *negative* Horn clauses that contain no positive literals. This form of description has its origins in work on theorem-proving within classical first-order logic, a realm from which logic programming has evolved. Clausal logic has been useful within this context because its simple syntactic structure simplifies the presentation of proof procedures. Its use is dependent on two characteristics of classical first-order logic: the ability to convert arbitrary formulas to sets of clauses that are equivalent from the perspective of unsatisfiability, and the preservation of clausal form under logical operations like substitution and resolution. However, these properties do not hold in all logics that might be of interest. The conversion to clausal form is, for instance, not possible within the framework of intuitionistic logic. In a similar vein, substitution into a disjunction of literals may not produce another clause in a situation where predicates are permitted to be variables; as we shall see in Subsection 3.5, substitution for a predicate variable has the potential to change the top-level logical structure of the original formula. The latter observation is especially relevant in the present context: a satisfactory higher-order generalization of Horn clauses must surely allow for predicates to be variables and must therefore be preceded by a different view of Horn clauses themselves.

Fortunately there is a description of Horn clauses that is congenial to their programming application and that can also serve as the basis for a higher-order generalization. Let A be a syntactic variable for an atomic formula in first-order logic. Then we may identify the classes of (first-order) G- and D-formulas by the following rules:

$$\begin{array}{lcl} G & ::= & A \mid G \wedge G \mid G \vee G \mid \exists x\, G \\ D & ::= & A \mid G \supset A \mid \forall x\, D \end{array}$$

These formulas are related to Horn clauses in the following sense: within the framework of classical logic, the negation of a G-formula is equivalent to a set of negative Horn clauses and, similarly, a D-formula is equivalent to a set of positive Horn clauses. We refer to the D-formulas as *definite clauses*, an alternative name in the literature for positive Horn clauses, and to the G-formulas as *goal formulas* because they function as goals within the programming paradigm of interest. These names in fact motivate the symbol chosen to denote members of the respective classes of formulas.

The programming interpretation of these formulas is dependent on treating a collection of closed definite clauses as a program and a closed goal formula as a query to be evaluated; definite clauses and goal formulas are, for this reason, also called *program clauses* and *queries*, respectively. The syntactic structures of goal formulas and definite clauses are relevant to their being interpreted in this fashion. The matrix of a definite clause is a formula of the form A or $G \supset A$, and this is intended to correspond to (part of) the definition of a procedure whose name is the predicate head of A. Thus, an atomic goal formula that "matches" with A may be solved immediately or by solving the corresponding instance of G, depending on the case of the definite clause. The outermost existential quantifiers in a goal formula (that are usually left implicit) are interpreted as a request to find values for the quantified variables that produce a solvable instance of the goal formula. The connectives $\wedge$ and $\vee$ that may appear in goal formulas typically have search interpretations in the programming context. The first connective specifies an AND branch in a search, i.e. it is a request to solve both components of a conjoined goal. The goal formula $G_1 \vee G_2$ represents a request to solve either G_1 or G_2 independently, and $\vee$ is thus a primitive for specifying OR branches in a search. This primitive is not provided for in the traditional description of Horn clauses, but it is nevertheless present in most implementations of Horn clause logic. Finally, a search related interpretation can also be accorded to the existential quantifier. This quantifier can be construed as a primitive for specifying an infinite OR branch in a search, where each branch is parameterized by the choice of substitution for the quantified variable.[1]

We illustrate some of the points discussed above by considering a program that implements the "append" relation between three lists. The first question to be addressed in developing this program is that of the representation to be used for lists. A natural choice for this is one that uses the constant *nil* for empty lists and the binary function symbol *cons* to construct a list out of a "head" element and a "tail" that is a list. This representation makes use of the structure of first-order terms and is symptomatic of another facet of logic programming: the terms of the underlying logic provides the data structures of the resulting programming language. Now, using this representation of lists, a program for appending lists is given by the following definite clauses:

$$\forall L\, append(nil, L, L),$$
$$\forall X\, \forall L_1\, \forall L_2\, \forall L_3 (append(L_1, L_2, L_3) \supset append(cons(X, L_1), L_2, cons(X, L_3))).$$

We assume that *append* in the definite clauses above is a predicate name, coincidental, in fact, with the name of the procedure defined by the clauses. The declarative intent of these formulas is apparent from their logical interpretation: the result of appending the empty list to any other list is the latter list and the result of appending a list with head X and tail L_1 to (another list) L_2 is a list with head X and tail L_3, provided L_3 is the result of appending L_1 and L_2. From a programming perspective, the two definite clauses have the following import: The first clause defines *append* as a procedure that succeeds immediately if its first argument can be matched with *nil*; the precise notion of matching here is, of course, first-order unification. The second clause pertains to the situation where the first argument is a non-empty list. To be precise, it stipulates that one way of solving an "append" goal whose arguments unify with the terms $cons(X, L_1)$, L_2 and $cons(X, L_3)$ is by solving another "append" goal whose arguments are the relevant instantiations of L_1, L_2 and L_3.

The definition of the *append* procedure might now be used to answer relevant questions about the relation of appending lists. For example, suppose that a, b, c and d are constant symbols. Then, a question that might be of interest is the one posed by the following goal formula:

$$\exists L\, append(cons(a, cons(b, nil)), cons(c, cons(d, nil)), L).$$

Consistent with the interpretation of goal formulas, this query corresponds to a request for a value for the variable L that makes

[1] Our concern, at the moment, is only with the *programming* interpretation of the logical symbols. It is a nontrivial property about Horn clauses that this programming interpretation of these symbols is compatible with their logical interpretation. For instance, even if we restrict our attention to classical or intuitionistic logic, the provability of a formula of the form $\exists x\, F$ from a set of formulas Γ does not in general entail the existence of an instance of $\exists x\, F$ that is provable from Γ.

$$append(cons(a, cons(b, nil)), cons(c, cons(d, nil)), L)$$

a solvable goal formula. A solution for this goal formula may be sought by using the procedural interpretation of *append*, leading to the conclusion that the only possible answer to it is the value

$$cons(a, cons(b, cons(c, cons(d, nil))))$$

for L. A characteristic of this query is that the "solution path" and the final answer for it are both deterministic. This is not, however, a necessary facet of every goal formula. As a specific example, the query

$$\exists L_1 \, \exists L_2 \, append(L_1, L_2, cons(a, cons(b, cons(c, cons(d, nil)))))$$

may be seen to have five different answers, each of which is obtained by choosing differently between the two clauses for *append* in the course of a solution. This aspect of nondeterminism is a hallmark of logic programming and is in fact a manifestation of its ability to support the notion of search in a primitive way.

Our current objective is to expose possible extensions to the syntax of the first-order G- and D-formulas that permit higher-order notions to be realized within logic programming. One higher-order ability that is coveted in programming contexts is that of passing procedures as arguments to other procedures. The above discussion makes apparent that the mechanism used for parameter passing in logic programming is unification and that passing a value to a procedure in fact involves the binding of a variable. A further point to note is that there is a duality in logic programming between predicates and procedures; the same object plays one or the other of these roles, depending on whether the point-of-view is logical or that of programming. From these observations, it is clear that the ability to pass procedures as arguments must hinge on the possibility of quantifying over predicates.

Predicate quantification is, in fact, capable of providing at least the simplest of programming abilities available through higher-order programming. The standard illustration of the higher-order capabilities of a language is the possibility of defining a "mapping" function in it. Such "functions" can easily be defined within the paradigm being considered with the provision of predicate variables. For example, suppose that we wish to define a function that takes a function and a list as arguments and produces a new list by applying the given function to each element of the former list. Given the relational style of programming prevalent in logic programming, such a function corresponds to the predicate *mappred* that is defined by the following definite clauses:

$$\forall P \, mappred(P, nil, nil),$$
$$\forall P \, \forall L_1 \, \forall L_2 \, \forall X \, \forall Y \, ((P(X, Y) \wedge mappred(P, L_1, L_2)) \supset mappred(P, cons(X, L_1), cons(Y, L_2))).$$

The representation that is used for lists here is identical to the one described in the context of the *append* program. The clauses above involve a quantification over P which is evidently a predicate variable. This variable can be instantiated with the name of an actual procedure (or predicate) and would lead to the invocation of that procedure (with appropriate arguments) in the course of evaluating a *mappred* query. To provide a particular example, let us assume that our program also contains a list of clauses defining the ages of various individuals, such as the following:

$$age(bob, 24),$$
$$age(sue, 23).$$

The procedure *mappred* can then be invoked with *age* and a list of individuals as arguments and may be expected to return a corresponding list of ages. For instance, the query

$$\exists L\, mappred(age, cons(bob, cons(sue, nil)), L)$$

should produce as an answer the substitution $cons(24, cons(23, nil))$ for L. Tracing a successful solution path for this query reveals that, in the course of producing an answer, queries of the form $age(bob, Y_1)$ and $age(sue, Y_2)$ have to be solved with suitable instantiations for Y_1 and Y_2.

The above example involves an instantiation of a simple kind for predicate variables — the substitution of a name of a predicate. A question to consider is if it is useful to permit the instantiation of such variables with more complex predicate terms. One ability that seems worthwhile to support is that of creating new relations by changing the order of arguments of given relations or by projecting onto some of their arguments. There are several programming situations where it is necessary to have "views" of a given relation that are obtained in this fashion, and it would be useful to have a device that permits the generation of such views without extensive additions to the program. The operations of abstraction and application that are formalized by the λ-calculus provide for such a device. Consider, for example, a relation that is like the *age* relation above, except that it has its arguments reversed. Such a relation can be represented by the predicate term $\lambda X\, \lambda Y\, age(Y, X)$. As another example, the expression $\lambda X\, age(X, 24)$ creates from *age* a predicate that represents the set of individuals whose age is 24.

An argument can thus be made for enhancing the structure of terms in the language by including the operations of abstraction and application. The general rationale is that it is worthwhile to couple the ability to treat predicates as values with devices for creating new predicate valued terms. Now, there are mechanisms for combining predicate terms, namely the logical connectives and quantifiers, and the same argument may be advanced for including these as well. To provide a concrete example, let us assume that our program contains the following set of definite clauses that define the "parent" relation between individuals:

$$\begin{array}{l} parent(bob, john), \\ parent(john, mary), \\ parent(sue, dick), \\ parent(dick, kate). \end{array}$$

One may desire to create a grandparent relation based on these clauses. This relation is, in fact, implemented by the term

$$\lambda X\, \lambda Y\, \exists Z\, (parent(X, Z) \wedge parent(Z, Y)).$$

An existential quantifier and a conjunction are used in this term to "join" two relations in obtaining a new one. Relations of this sort can be used in meaningful ways in performing computations. For example, the query

$$\exists L\, mappred(\lambda X\, \lambda Y\, \exists Z\, (parent(X, Z) \wedge parent(Z, Y)), cons(bob, cons(sue, nil)), L)$$

illustrates the use of the relation shown together with the *mappred* predicate in asking for the grandparents of the individuals in a given list.

Assuming that we do allow logical symbols to appear in terms, it is relevant to consider whether the occurrences of these symbols should be restricted in any way. The role that these symbols are intended to play eventually indicates the answer to this question. Predicate terms are to instantiate predicate variables that get invoked as queries after being supplied with appropriate arguments. The logical connectives and quantifiers that appear in terms therefore become primitives that direct the search in the course of a computation. This observation, when coupled with our desire to preserve the essential character of Horn clause programming while providing for higher-order features, leads to the conclusion that only those logical symbols should be permitted in terms that can appear in the top-level structure of goal formulas. This argues specifically for the inclusion of only conjunctions, disjunctions and existential quantifications. This restriction can, of course, be relaxed if we are dealing with a logic whose propositional and quantificational structure is richer than that of Horn clauses. One such logic is outlined in Section 9 and is studied in detail in [MNPS91].

Our consideration of higher-order aspects began with the provision of predicate variables. In a vein similar to that for logical symbols, one may ask whether there are practical considerations limiting the occurrences of these variables. In answering this question, it is useful to consider the structure of definite clauses again. These formulas are either of the form $\forall \bar{x}\, A$ or $\forall \bar{x}\, (G \supset A)$. An intuitive justification can be provided for permitting predicate variables in at least two places in such formulas: in "argument" positions in A and as the "heads" of atoms in G. The possibility for predicate variables to appear in these two locations is, in fact, what supports the ability to pass procedures as arguments and to later invoke them as goals. Now, there are two other forms in which a predicate variable can appear in the formulas being considered, and these are as an argument of an atom in G and as the head of A. An examination of the definition of the *mappred* procedure shows that it is useful to permit predicate variables to appear within the arguments of atomic goal formulas. With regard to the other possibility, we recall that, from a programming perspective, a definite clause is to be viewed as the definition of a procedure whose name it given by the head of A. A consideration of this fact indicates that predicate variables are not usefully permitted in the head position in A.

We have been concerned up to this point with only predicate variables and procedures. There is, however, another higher-order facet that should be considered and this is the possibility of functions to be variables. There is a similarity between predicates and functions — predicates are, eventually, boolean valued functions — and so it seems reasonable to permit a quantification over the latter if it is permitted over the former. There is, of course, the question of whether permitting such quantifications results in any useful and different higher-order abilities. To answer this question, let us consider the "functional" counterpart of *mappred* that is defined by the following definite clauses:

$$\forall F\, mapfun(F, nil, nil),$$
$$\forall F\, \forall L_1\, \forall L_2\, \forall X\, (mapfun(F, L_1, L_2) \supset mapfun(F, cons(X, L_1), cons(F(X), L_2))).$$

Reading these clauses declaratively, we see that *mapfun* relates a function and two lists just in case the second list is obtained by applying the function to each element of the first list. Notice that the notion of function application involved here is quite different from that of solving a goal. For example, if our terms are chosen to be those of some version of the λ-calculus, function evaluation would be based on β-conversion. Thus, the query

$$\exists L\, mapfun(\lambda X\, h(1, X), cons(1, cons(2, nil)), L)$$

would apply the term $\lambda X\, h(1, X)$ to each of 1 and 2, eventually producing the list

$$cons(h(1,1), cons(h(1,2), nil))$$

as an instantiation for L. (We assume that the symbol h that appears in the query is a constant.) By placing suitable restrictions on the λ-terms that we use, we can make the operation of β-conversion a relatively weak one, and, in fact, strong enough only to encode the substitution operation. Such a choice of terms makes it conceivable to run queries like the one just considered in "reverse." In particular, a query such as

$$\exists F\, mapfun(F, cons(1, cons(2, nil)), cons(h(1,1), cons(h(1,2), nil)))$$

could be posed with the expectation of generating the substitution $\lambda X\, h(1, X)$ for F. The ability to find such solutions is dependent critically on using a weak notion of functions and function evaluation and finding predicate substitutions through an apparently similar process of solving goals is *not* a feasible possibility. To see this, suppose that we are given the query

$$\exists P\, mappred(P, cons(bob, cons(sue, nil)), cons(24, cons(23, nil))).$$

It might appear that a suitable answer can be provided to this query and that this might, in fact, be the value *age* for P. A little thought, however, indicates that the query is an ill-posed one. There are too many predicate terms that hold of *bob* and 24 and of *sue* and 23 — consider, for example, the myriad ways for stating the relation that holds of any two objects — and enumerating these does not seem to be a meaningful computational task.

The above discussion brings out the distinction between quantification over only predicates and quantification over both predicates and functions. This does not in itself, however, address the question of usefulness of permitting quantification over functions. This question is considered in detail in Sections 8 and 9 and so we provide only a glimpse of an answer to it at this point. For this purpose, we return to the two $mapfun$ queries above. In both queries the new ability obtained from function variables and λ-terms is that of analyzing the process of substitution. In the first query, the computation involved is that of performing substitutions into a given structure, namely $h(1, X)$. The second query involves finding a structure from which two different structures can be obtained by substitutions; in particular, a structure which yields $h(1, 1)$ when 1 is substituted into it and $h(1, 2)$ when 2 is used instead. Now, there are several situations where this ability to analyze the structures of terms via the operation of substitution is important. Furthermore, in many of these contexts, objects that incorporate the notion of binding will have to be treated. The terms of a λ-calculus and the accompanying conversion rules are, as we shall see, appropriate tools for correctly formalizing the idea of substitution in the presence of bound variables. Using function variables and λ-terms can, therefore, provide for programming primitives that are useful in these contexts.

We have already argued, using the $mappred$ example, for the provision of predicate variables in the arguments of (atomic) goal formulas. This argument can be strengthened in light of the fact that predicates are but functions of a special kind. When predicate variables appear as the heads of atomic goals, *i.e.*, in "extensional" positions, they can be instantiated, thereby leading to the computation of new goals. However, as we have just observed, it is not meaningful to contemplate finding values for such variables. When predicate variables appear in the arguments of goals, *i.e.* in "intensional" positions, values can be found for them by structural analyses in much the same way as for other function variables. These two kinds of occurrences of predicate variables can thus be combined to advantage: an intensional occurrence of a predicate variable can be used to form a query whose solution is then sought via an extensional occurrence of the same variable. We delay the consideration of specific uses of this facility till Section 7.

3 A Higher-Order Logic

A principled development of a logic programming language that incorporates the features outlined in the previous section must be based on a higher-order logic. The logic that we use for this purpose is derived from Church's formulation of the Simple Theory of Types [Chu40] principally by the exclusion of the axioms concerning infinity, extensionality for propositions, choice and description. Church's logic is particularly suited to our purposes since it is obtained by superimposing logical notions over the calculus of λ-conversion. Our omission of certain axioms is based on a desire for a logic that generalizes first-order logic by providing a stronger notion of variable and term, but that, at the same time, encompasses only the most primitive logical notions that are relevant in this context; only these notions appear to be of consequence from the perspective of computational applications. Our logic is closely related to that of [And71], the only real differences being the inclusion of η-conversion as a rule of inference and the incorporation of a larger number of propositional connectives and quantifiers as primitives. In the subsections that follow, we describe the language of this logic and clarify the intended meanings of expressions by the presentation of a deductive calculus as well as a notion of models. There are several similarities between this logic and first-order logic, especially in terms of proof-theoretic properties. However, the richer syntax of the higher-order logic make the interpretation and usefulness of these properties in the two contexts different. We dwell on this aspect in the last subsection below.

3.1 The Language

The language underlying the formal system that we utilize in this chapter is that of a typed λ-calculus. There are two major syntactic components to this language: the types and the terms. The purpose of the types is to categorize the terms based on a functional hierarchy. From a syntactic perspective, the types constitute the more primitive notion and the formation rules for terms identify a type with each term.

The types that are employed are often referred to as *simple* types. They are determined by a set $\mathcal{S}$ of *sorts* and a set $\mathcal{C}$ of *type constructors*. We assume that $\mathcal{S}$ contains the sort o that is the type of propositions and at least one other sort, and that each member of $\mathcal{C}$ has associated with it a unique positive arity. The class of types is then the smallest collection that includes (i) every sort, (ii) $(c\ \sigma_1\ \ldots\ \sigma_n)$, for every $c \in \mathcal{C}$ of arity n and $\sigma_1, \ldots, \sigma_n$ that are types, and (iii) $(\sigma \to \tau)$ for every σ and τ that are types. Understood intuitively, the type $(\sigma \to \tau)$ corresponds to the set of "function" terms whose domains and ranges are given by σ and τ respectively. In keeping with this intuition, we refer to the types obtained by virtue of (i) and (ii) as *atomic types* and to those obtained by virtue of (iii) as *function* types.

We will employ certain notational conventions with respect to types. To begin with, we will use the letters σ and τ, perhaps with subscripts, as metalanguage variables for types. Further, the use of parentheses will be minimized by assuming that $\to$ associates to the right. Using this convention, every type can be written in the form $(\sigma_1 \to \cdots \to \sigma_n \to \tau)$ where τ is an atomic type. We will refer to $\sigma_1, \ldots, \sigma_n$ as the *argument* types and to τ as the *target* type of the type when it is written in this form. This terminology is extended to atomic types by permitting the argument types to be an empty sequence.

The class of terms is obtained by the operations of *abstraction* and *application* from given sets of constants and variables. We assume that the constants and variables are each specified with a type and that these collections meet the following additional conditions: there is at least one constant and a denumerable number of variables of each type and the variables of each type are distinct from the constants and the variables of any other type. The *terms* or *formulas* are then specified together with an associated type in the following fashion:

(1) A variable or a constant of type σ is a term of type σ.

(2) If x is a variable of type σ and F is a term of type τ then $(\lambda x\, F)$ is a term of type $\sigma \to \tau$, and is referred to as an *abstraction* that *binds* x and whose *scope* is F.

(3) If F_1 is a term of type $\sigma \to \tau$ and F_2 is a term of type σ then $(F_1\ F_2)$, referred to as the *application* of F_1 to F_2, is a term of type τ.

Once again, certain notational conventions will be employed in connection with terms. When talking about an arbitrary term, we will generally denote this by an uppercase letter that possibly has a subscript and a superscript. It will sometimes be necessary to display abstractions and we will usually depict the variable being abstracted by a lowercase letter that may be subscripted. When it is necessary to present specific (object-level) terms, we will explicitly indicate the symbols that are to be construed as constants and variables. In writing terms, we will omit parenthesis by using the convention that abstraction is right associative and application is left associative. This usage will occur at both the object- and the meta-level. As a further shorthand, the abstraction symbol in a sequence of abstractions will sometimes be omitted: thus, the term $\lambda x_1 \ldots \lambda x_n\, T$ may be abbreviated by $\lambda x_1, \ldots, x_n\, T$. Finally, although each term is specified only in conjunction with a type, we will seldom mention the types of terms explicitly. These omissions will be justified on the basis that the types can either be inferred from the context or are inessential to the discussion at hand.

The rules of formation for terms serve to identify the well-formed subparts of any given term. Specifically, a term G is said to *occur* in, or to be a *subterm* or *subformula* of, a term F if (a) G is F, or (b) F is of the form $(\lambda x\, F_1)$ and G occurs in F_1, or (c) F is of the form $(F_1\ F_2)$ and G occurs in either F_1 or F_2. An occurrence of a variable x in F is either *bound* or *free* depending on whether it is or is not an occurrence in the scope of an abstraction that binds x. A variable x is a bound (free) variable of F if it has at least one bound (free) occurrence in F. F is a *closed* term just in case it has no free variables. We write $\mathcal{F}(F)$ to denote the set of free variables of F. This notation is generalized to sets of terms and sets of pairs of terms in the following way: $\mathcal{F}(\mathcal{D})$ is $\bigcup\{\mathcal{F}(F) \mid F \in \mathcal{D}\}$ if $\mathcal{D}$ is a set of terms and $\bigcup\{\mathcal{F}(F_1) \cup \mathcal{F}(F_2) \mid \langle F_1, F_2\rangle \in \mathcal{D}\}$ if $\mathcal{D}$ is a set of pairs of terms.

Example 1. Let $int \in \mathcal{S}$ and $list \in \mathcal{C}$ and assume that the arity of $list$ is 1. Then the following are legitimate types: int, $(list\ int)$ and $int \to (list\ int) \to (list\ int)$. The argument types of the last of these types are int and $(list\ int)$ and its target type is $(list\ int)$. Let $cons$ be a constant of type $int \to (list\ int) \to (list\ int)$, let 1 and 2 be constants of type int and let l be a variable of type $(list\ int)$. Then $\lambda l\, (cons\ 1\ (cons\ 2\ l))$ is a term. A cursory inspection reveals that the type of this term must be $(list\ int) \to (list\ int)$. The above term has one bound variable and no free variables, *i.e.*, it is a closed term. However, it has as subterm the term $(cons\ 1\ (cons\ 2\ l))$ in which the variable l appears free.

The language presented thus far gives us a means for describing functions of a simple sort: abstraction constitutes a device for representing function formation and application provides a means for representing the evaluation of such functions. Our desire, however, is for a language that allows not only for the representation of functions, but also for the use of connectives and quantifiers in describing relations between such functions. Such a capability can be achieved in the present context by introducing a set of constants for representing these logical operations. To be precise, we henceforth assume that the set of constants is partitioned into the set of *parameters* or *nonlogical constants* and the set of *logical constants* with the latter comprising the following infinite list of symbols: $\top$ of type o, $\neg$ of type $o \to o$, $\wedge$, $\vee$ and $\supset$ of type $o \to o \to o$, and, for each σ, $\exists$ and $\forall$ of type $(\sigma \to o) \to o$. We recall that o is intended to be the type of propositions. The logical constants are to be interpreted in the following manner: $\top$ corresponds to the tautologous proposition, the *(propositional) connectives* $\neg$, $\vee$, $\wedge$, and $\supset$ correspond, respectively, to negation, disjunction, conjunction, and implication, and the family of constants $\exists$ and $\forall$ are, respectively, the existential and universal quantifiers. The correspondence between $\exists$ and $\forall$ and the quantifiers familiar from first-order logic may be understood from the following: the existential and universal quantification of x over P is written as $(\exists\ (\lambda x\, P))$ and $(\forall\ (\lambda x\, P))$ respectively. Under this representation, the dual aspects of binding and predication that accompany the usual notions of quantification are handled separately by abstractions and constants that are propositional functions of propositional functions. The constants that are used must, of course, be accorded a suitable interpretation for this representation to be a satisfactory one. For example, the meaning assigned to $\forall$ must be such that $(\forall\ (\lambda x\, P))$ holds just in case $(\lambda x\, P)$ corresponds to the set of all objects of the type of x.

Certain conventions and terminology are motivated by the intended interpretations of the type o and the logical constants. In order to permit a distinction between arbitrary terms and those of type o, we reserve the word "formula" exclusively for terms of type o. Terms of type $\sigma_1 \to \cdots \to \sigma_n \to o$ correspond to n-ary relations on terms and, for this reason, we will also refer to them as predicates of n-arguments. In writing formulas, we will adopt an infix notation for the symbols $\wedge$, $\vee$ and $\supset$; *e.g.*, we will write $(\wedge\ F\ G)$ as $(F \wedge G)$. In a similar vein, the expressions $(\exists x\, F)$ and $(\forall x\, F)$ will be used as abbreviations for $(\exists\ (\lambda x\, F))$ and $(\forall\ (\lambda x\, F))$ Parallel to the convention for abstractions, we will sometimes write the expressions $\exists x_1 \ldots \exists x_n\, F$ and $\forall x_1 \ldots \forall x_n\, F$ as $\exists x_1, \ldots, x_n\, F$ and $\forall x_1, \ldots, x_n\, F$ respectively. In several cases it will only be important to specify that the "prefix" contains a sequence of some length. In such cases, we will use $\bar{x}$ an abbreviation for a sequence of variables and write $\lambda \bar{x}\, F$, $\exists \bar{x}\, F$ or $\forall \bar{x}\, F$, as the case might be.

Our language at this point has the ability to represent functions as well as logical relations between functions. However, the sense in which it can represent these notions is still informal and needs to be made precise. We do this in the next two subsections, first by describing a formal system that clarifies the meaning of abstraction and application and then by presenting a sequent calculus that bestows upon the various logical symbols their intended meanings.

3.2 Equality between Terms

The intended interpretations of abstraction and application are formalized by the rules of λ-conversion. To define these rules, we need the operation of replacing all free occurrences of a variable x in the term T_1 by a term T_2 of the same type as x. This operation is denoted by $S_x^{T_2} T_1$ and is made explicit as follows:

(i) If T_1 is a variable or a constant, then $S_x^{T_2} T_1$ is T_2 if T_1 is x and T_1 otherwise.

(ii) If T_1 is of the form $(\lambda y\, C)$, then $S_x^{T_2} T_1$ is T_1 if y is x and $(\lambda y\, S_x^{T_2} C)$ otherwise.

(iii) If T_1 is of the form $(C\ D)$, then $S_x^{T_2} T_1 = (S_x^{T_2} C\ S_x^{T_2} D)$.

In performing this operation of replacement, there is the danger that the free variables of T_2 become bound inadvertently. The term "T_2 *is substitutable for* x *in* T_1" describes the situations in which the operation is logically correct, *i.e.* those situations where x does not occur free in the scope of an abstraction in T_1 that binds a free variable of T_2. The rules of *α-conversion*, *β-conversion* and *η-conversion* are then, respectively, the following operations and their converses on terms:

(1) Replacing a subterm $(\lambda x\, T)$ by $(\lambda y\, S_x^y T)$ provided y is substitutable for x in T and not free in T.

(2) Replacing a subterm $((\lambda x\, T_1)\ T_2)$ by $S_x^{T_2} T_1$ provided T_2 is substitutable for x in T_1, and vice versa.

(3) Replacing a subterm $(\lambda x\,(T\ x))$ by T provided x is not free in T, and vice versa.

The rules above, referred to collectively as the λ-conversion rules, may be used to define the following relations between terms:

Definition 1. T *λ-conv* (*β-conv*, $\equiv$) S just in case there is a sequence of applications of the λ-conversion (respectively α- and β-conversion, α-conversion) rules that transforms T into S.

The three relations thus defined are easily seen to be equivalence relations. They correspond, in fact, to notions of equality between terms based on the following informal interpretation of the λ-conversion rules: α-conversion asserts that the choice of name for the variable bound by an abstraction is unimportant, β-conversion relates an application to the result of evaluating the application, and η-conversion describes a notion of extensionality for function terms (the precise nature of which will become clear in Subsection 3.4). We use the strongest of these notions in our discussions below, *i.e.* we shall consider T and S equal just in case T λ-conv S.

It is useful to identify the equivalence classes of terms under the relations just defined with canonical members of each class. Towards this end, we say that a term is in *β-normal form* if it does not have a subterm of the form $((\lambda x\, A)\ B)$. If T is in β-normal form and S β-conv T, then T is said to be a β-normal form for S. For our typed language, it is known that a β-normal form exists for every term [And71]. By the celebrated Church-Rosser Theorem for β-conversion [Bar81], this form must be unique up to a renaming of bound variables. We may therefore use terms in β-normal form as representatives of the equivalence classes under the β-conv relation. We note that each such term has the structure

$$\lambda x_1 \ldots \lambda x_n\,(A\ T_1\ \ldots\ T_m)$$

where A is a constant or variable, and, for $1 \le i \le m$, T_i also has the same structure. We refer to the sequence $x_1, \ldots, x_n$ as the *binder*, to A as the *head* and to $T_1, \ldots, T_m$ as the *arguments* of such a term; in particular instances, the binder may be empty, and the term may also have no arguments. Such a term is said to be *rigid* if its head, *i.e.* A, is either a constant or a variable that appears in the binder, and *flexible* otherwise.

In identifying the canonical members of the equivalence classes of terms with respect to λ-conv, there are two different approaches that might be followed. Under one approach, we say that a term is in η-normal form if it has no subterm of the form $(\lambda x\,(A\ x))$ in which x is not free in A, and we say a term is in λ-normal form if it is in both β- and η-normal form. The other alternative is to say that a term is in λ-normal form if it can be written in the form $\lambda\bar{x}\,(A\ T_1\ \ldots\ T_m)$ where A is a constant or variable of type $\sigma_1 \to \ldots \to \sigma_m \to \tau$ with τ being an atomic type and, further, each T_i can also be written in a similar form. (Note that the term must be in β-normal form for this to be possible.) In either case we say that T is a λ-normal form for S if S λ-conv T and T is in λ-normal form. Regardless of which definition is chosen, it is known that a λ-normal form exists for every term in our typed language and that this form is unique up to a renaming of bound variables (see [Bar81] and also the discussion in [Nad87]). We find it convenient to use the latter definition in this chapter and we will write $\lambda norm(T)$ to denote a λ-normal form for T under it. To obtain such a form for any given term, the following procedure may be used: First convert the term into β-normal form by repeatedly replacing every subterm of the form $((\lambda x\, A)\ B)$ by $S_x^B A$ preceded, perhaps, by some α-conversion steps. Now, if the resulting term is of the form $\lambda x_1, \ldots, x_m\,(A\ T_1\ \ldots\ T_n)$ where A is of type $\sigma_1 \to \ldots \sigma_n \to \sigma_{n+1} \to \ldots \to \sigma_{n+r} \to \tau$, then replace it by the term

$$\lambda x_1, \ldots, x_m, y_1, \ldots, y_r\,(A\ T_1\ \ldots\ T_n\ y_1\ \ldots\ y_r)$$

where $y_i, \ldots, y_r$ are distinct variables of appropriate types that are not contained in $\{x_1, \ldots, x_m\}$. Finally repeat this operation of "fluffing-up" of the arguments on the terms $T_1, \ldots, T_n, y_1, \ldots, y_r$.

A λ-normal form of a term T as we have defined it here is unique only up to a renaming of bound variables (*i.e.*, up to α-conversions). While this is sufficient for most purposes, we will occasionally need to talk of a unique normal form. In such cases, we use $\rho(F)$ to designate what we call the *principal normal form* of F. Determining this form essentially requires a naming convention for bound variables and a convention such as the one in [And71] suffices for our purposes.

The existence of a λ-normal form for each term is useful for two reasons. First, it provides a mechanism for determining whether two terms are equal by virtue of the λ-conversion rules. Second, it permits the properties of terms to be discussed by using a representative from each of the equivalence classes that has a convenient structure. Of particular interest to us is the structure of formulas in λ-normal form. For obvious reasons, we call a formula whose leftmost non-parenthesis symbol is either a variable or a parameter an *atomic formula*. Then one of the following is true of a formula in λ-normal form: (i) it is $\top$, (ii) it is an atomic formula, (iii) it is of the form $\neg F$, where F is a formula in λ-normal form, (iv) it is of the form $(F \vee G)$, $(F \wedge G)$, or $(F \supset G)$ where F and G are formulas in λ-normal form, or (v) it is of the form $(\exists x\, F)$ or $(\forall x\, F)$, where F is a formula in λ-normal form.

The β-conversion rule provides a convenient means for defining the operation of substitution on terms. A substitution is formally a (type preserving) mapping on variables that is the identity everywhere except at a finitely many explicitly specified points. Thus, a substitution is represented by a set of the form $\{\langle x_i, T_i\rangle \mid 1 \le i \le n\}$, where, for $1 \le i \le n$, x_i is a distinct variable and T_i is a term that is of the same type as x_i but that is distinct from x_i. The application of a substitution to a term requires this mapping to be extended to the class of all terms and can be formalized as follows: if $\theta = \{\langle x_i, T_i\rangle \mid 1 \le i \le n\}$ and S is a term, then

$$\theta(G) = \rho(((\lambda x_1\ \ldots \lambda x_n\, S)\ T_1\ \ldots\ T_n)).$$

It can be seen that this definition is independent of the order in which the pairs are taken from θ and that it formalizes the idea of replacing the free occurrences of $x_1, \ldots, x_n$ in S simultaneously by the terms $T_1, \ldots, T_n$. We often have to deal with substitutions that are given by singleton sets and we introduce a special notation for the application of such substitutions: if θ is $\{\langle x, T\rangle\}$, then $\theta(S)$ may also be written as $[T/x]S$.

Certain terminology pertaining to substitutions will be used later in the chapter. Given two terms T_1 and T_2, we say T_1 is an *instance* of T_2 if it results from applying a substitution to T_2. The *composition* of two substitutions θ_1 and θ_2, written as $\theta_1 \circ \theta_2$, is precisely the composition of θ_1 and θ_2 when these are viewed as mappings: $\theta_1 \circ \theta_2(G) = \theta_1(\theta_2(G))$. The *restriction* of a substitution θ to a set of variables $\mathcal{V}$, denoted by $\theta \uparrow \mathcal{V}$, is given as follows

$$\theta \uparrow \mathcal{V} = \{\langle x, F\rangle \mid \langle x, F\rangle \in \theta \text{ and } x \in \mathcal{V}\}.$$

Two substitutions, θ_1 and θ_2 are said to be *equal* relative to a set of variables $\mathcal{V}$ if it is the case that $\theta_1 \uparrow \mathcal{V} = \theta_2 \uparrow \mathcal{V}$ and this relationship is denoted by $\theta_1 =_{\mathcal{V}} \theta_2$. θ_1 is said to be *less general than* θ_2 relative to $\mathcal{V}$, a relationship denoted by $\theta_1 \preceq_{\mathcal{V}} \theta_2$, if there is a substitution σ such that $\theta_1 =_{\mathcal{V}} \sigma \circ \theta_2$. Finally, we will sometimes talk of the result of applying a substitution to a set of formulas and to a set of pairs of formulas. In the first case, we mean the set that results from applying the substitution to each formula in the set, and, in the latter case, we mean the set of pairs that results from the application of the substitution to each element in each pair.

3.3 The Notion of Derivation

The meanings of the logical symbols in our language may be clarified by providing an abstract characterization of proofs for formulas containing these symbols. One convenient way to do this is by using a sequent calculus. We digress briefly to summarize the central notions pertaining to such calculi. The basic unit of assertion within these calculi is a *sequent.* A sequent is a pair of finite (possibly empty) sets of formulas $\langle \Gamma, \Theta\rangle$ that is usually written as $\Gamma \longrightarrow \Theta$. The first element of this pair is referred to as the *antecedent* of the sequent and the second is called its *succedent.* A sequent corresponds intuitively to the assertion that in each situation in which all the formulas in its antecedent hold, there is at least one formula in its succedent that also holds. In the case that the succedent is empty, the sequent constitutes an assertion of contradictoriness of the formulas in its antecedent. A *proof* for a sequent is a finite tree constructed using a given set of *inference figures* and such that the root is labeled with the sequent in question and the leaves are labeled with designated *initial sequents.* Particular sequent systems are characterized by their choice of inference figures and initial sequents.

Our higher-order logic is defined within this framework by the inference figure schemata contained in Figure 1. Actual inference figures are obtained from these by instantiating Γ, Θ and Δ by sets of formulas, B, D, and P by formulas, x by a variable, T by a term and c by a parameter. There is, in addition, a proviso on the choice of parameter for c: it should not appear in any formula contained in the lower sequent in the same figure. Also, in the inference figure schemata λ, the sets Δ and Δ' and the sets Θ and Θ' differ only in that zero or more formulas in them are replaced by formulas that can be obtained from them by using the λ-conversion rules. The initial sequents of our sequent calculus are all the sequents of the form $\Gamma \longrightarrow \Theta$ where either $\top \in \Theta$ or for some atomic formulas $A \in \Delta$ and $A' \in \Theta$ it is the case that $A \equiv A'$.

Expressions of the form B, Δ and Θ, B that are used in the inference figure schemata in Figure 1 are to be treated as abbreviations for $\Delta \cup \{B\}$ and $\Theta \cup \{B\}$ respectively. Thus, a particular set of formulas may be viewed as being of the form Θ, B even though $B \in \Theta$. As a result of this view, a formula that appears in the antecedent or succedent of a sequent really has an arbitrary multiplicity.

$$\frac{B, D, \Delta \longrightarrow \Theta}{B \wedge D, \Delta \longrightarrow \Theta}\ \wedge\text{-L} \qquad \frac{\Delta \longrightarrow \Theta, B \qquad \Delta \longrightarrow \Theta, D}{\Delta \longrightarrow \Theta, B \wedge D}\ \wedge\text{-R}$$

$$\frac{B, \Delta \longrightarrow \Theta \qquad D, \Delta \longrightarrow \Theta}{B \vee D, \Delta \longrightarrow \Theta}\ \vee\text{-L}$$

$$\frac{\Delta \longrightarrow \Theta, B}{\Delta \longrightarrow \Theta, B \vee D}\ \vee\text{-R} \qquad \frac{\Delta \longrightarrow \Theta, D}{\Delta \longrightarrow \Theta, B \vee D}\ \vee\text{-R}$$

$$\frac{\Delta \longrightarrow \Theta, B}{\neg B, \Delta \longrightarrow \Theta}\ \neg\text{-L} \qquad \frac{B, \Delta \longrightarrow \Theta}{\Delta \longrightarrow \Theta, \neg B}\ \neg\text{-R}$$

$$\frac{\Delta \longrightarrow \Theta, B \qquad D, \Delta \longrightarrow \Gamma}{B \supset D, \Delta \longrightarrow \Theta \cup \Gamma}\ \supset\text{-L} \qquad \frac{B, \Delta \longrightarrow \Theta, D}{\Delta \longrightarrow \Theta, B \supset D}\ \supset\text{-R}$$

$$\frac{[T/x]P, \Delta \longrightarrow \Theta}{\forall x\, P, \Delta \longrightarrow \Theta}\ \forall\text{-L} \qquad \frac{\Delta \longrightarrow \Theta, [T/x]P}{\Delta \longrightarrow \Theta, \exists x\, P}\ \exists\text{-R}$$

$$\frac{[c/x]P, \Delta \longrightarrow \Theta}{\exists x\, P, \Delta \longrightarrow \Theta}\ \exists\text{-L} \qquad \frac{\Delta \longrightarrow \Theta, [c/x]P}{\Delta \longrightarrow \Theta, \forall x\, P}\ \forall\text{-R}$$

$$\frac{\Delta' \longrightarrow \Theta'}{\Delta \longrightarrow \Theta}\ \lambda$$

Figure 1: Inference Figure Schemata

This interpretation allows us to eliminate the structural inference figures called contraction that typically appear in sequent calculi; these inference figures provide for the multiplicity of formulas in sequents in a situation where the antecedents and succedents are taken to be lists instead of sets. Our use of sets also allows us to drop another structural inference figure called exchange that is generally employed to ensure that the order of formulas is unimportant. A third kind of structural inference figure, commonly referred to as thinning or weakening, permits the addition of formulas to the antecedent or succedent. This ability is required in calculi where only one formula is permitted in the antecedent or succedent of an initial sequent. Given the definition of initial sequents in our calculus, we do not have a need for such inference figures.

Any proof that can be constructed using our calculus is referred to as a **C**-proof. A **C**-proof in which every sequent has at most one formula in its succedent is called an **I**-proof. We write $\Gamma \vdash_C B$ to signify that $\Gamma \longrightarrow B$ has a **C**-proof and $\Gamma \vdash_I B$ to signify that it has an **I**-proof. These relations correspond, respectively, to provability in higher-order classical and intuitionistic logic. Our primary interest in this chapter is in the classical notion, and unqualified uses of the words "derivation" or "proof" are to be read as **C**-proof. We note that this notion of derivability is identical to the notion of provability in the system $\mathcal{T}$ of [And71] augmented by the rule of η-conversion. The reader familiar with [Gen69] may also compare the calculus LK and the one described here for **C**-proofs. One difference between these two is the absence of the inference figures of the form

$$\frac{\Gamma \longrightarrow \Theta, B \qquad\qquad B, \Delta \longrightarrow \Lambda}{\Gamma \cup \Delta \longrightarrow \Theta \cup \Lambda}$$

from our calculus. These inference figures are referred to as the *Cut* inference figures and they occupy a celebrated position within logic. Their omission from our calculus is justified by a rather deep result for the higher-order logic under consideration, the so-called *cut-elimination theorem.* This theorem asserts that the same set of sequents have proofs in the calculi with and without the Cut inference figures. A proof of this theorem for our logic can be modelled on the one in [And71] (see [Nad87] for details). The only other significant difference between our calculus and the one in [Gen69] is that our formulas have a richer syntax and the λ inference figures have been included to manipulate this syntax. This difference, as we shall observe presently, has a major impact on the process of searching for proofs.

3.4 A Notion of Models

An alternative approach to clarifying the meanings of the logical symbols in our language is to specify the role they play in determining the denotations of terms from abstract domains of objects. This may be done in a manner akin to that in [Hen50]. In particular, we assume that we are given a family of domains $\{D_\sigma\}_\sigma$, each domain being indexed by a type. The intention is that a term of type σ will, under a given interpretation for the parameters and an assignment for the variables, denote an object in the domain D_σ. There are certain properties that we expect our domains to satisfy at the outset. For each atomic type σ other than o, we assume that D_σ is some set of *individual objects* of that type. The domain D_o will correspond to a set of truth values. However, within our logic and unlike the logic dealt with in [Hen50], distinctions must be made between the denotations of formulas even when they have the same propositional content: as an example, it should be possible for $Q \vee P$ to denote a different object from $P \vee Q$, despite the fact that these formulas share a truth value under identical circumstances. For this reason, we take D_o to be a domain of *labelled* truth values. Formally, we assume that we are given a collection of labels, $\mathcal{L}$ and that D_o is a subset of $\mathcal{L} \times \{T, F\}$ that denotes a function from the labels $\mathcal{L}$ to the set $\{T, F\}$. For each function type, we assume that the domain corresponding to it is a collection of functions over the relevant domains. Thus, we expect $D_{\sigma\to\tau}$ to be a collection of functions from D_σ to D_τ.

We refer to a family of domains $\{D_\sigma\}_\sigma$ that satisfies the above constraints as a *frame*. We assume now that I is a mapping on the parameters that associates an object from D_σ with a parameter of type σ. The behavior of the logical symbols insofar as truth values are concerned is determined by their intended interpretation as we shall presently observe. However their behavior with respect to labels is open to choice. For the purpose of fixing this in a given context we assume that we are given a predetermined label $\top_l$ and the following mappings:

$$\begin{array}{llll} \neg_l : \mathcal{L} \to \mathcal{L} & \vee_l : \mathcal{L} \to \mathcal{L} \to \mathcal{L} & \wedge_l : \mathcal{L} \to \mathcal{L} \to \mathcal{L} & \supset_l : \mathcal{L} \to \mathcal{L} \to \mathcal{L} \\ \exists_l^\sigma : D_{\sigma\to o} \to \mathcal{L} & \forall_l^\sigma : D_{\sigma\to o} \to \mathcal{L}; & & \end{array}$$

the last two are actually a family of mappings, parameterized by types. Let C be the set containing $\top_l$ and these various mappings. Then the tuple $\langle \mathcal{L}, \{D_\sigma\}_\sigma, I, C\rangle$ is said to be a *pre-structure* or *pre-interpretation* for our language.

A mapping ϕ on the variables is an *assignment* with respect to a pre-structure $\langle \mathcal{L}, \{D_\sigma\}_\sigma, I, C\rangle$ just in case ϕ maps each variable of type σ to D_σ. We wish to extend ϕ to a "valuation" function V_ϕ on all terms. The desired behavior of V_ϕ on a term H is given by induction over the structure of H:

(1) H is a variable of a constant. In this case

- (i) if H is a variable then $V_\phi(H) = \phi(H)$,
- (ii) if H is a parameter then $V_\phi(H) = I(H)$,
- (iii) if H is $\top$ then $V_\phi(H) = \langle \top_l, T\rangle$
- (iv) if H is $\neg$ then $V_\phi(H)\langle l, p\rangle = \langle \neg_l(l), q\rangle$, where q is F if p is T and T otherwise,
- (v) if H is $\vee$ then $V_\phi(H)\langle l_1, p\rangle\langle l_2, q\rangle = \langle \vee_l(l_1)(l_2), r\rangle$, where r is T if either p or q is T and F otherwise,
- (vi) if H is $\wedge$ then $V_\phi(H)\langle l_1, p\rangle\langle l_2, q\rangle = \langle \wedge_l(l_1)(l_2), r\rangle$, where r is F if either p or q is F and T otherwise,
- (vii) if H is $\supset$ then $V_\phi(H)\langle l_1, p\rangle\langle l_2, q\rangle = \langle \supset_l (l_1)(l_2), r\rangle$, where r is T if either p is F or q is T and F otherwise,
- (viii) if H is $\exists$ of type $((\sigma \to o) \to o)$ then, for any $p \in D_{\sigma\to o}$, $V_\phi(H)(p) = \langle \exists_l^\sigma(p), q\rangle$, where q is T if there is some $t \in D_\sigma$ such that $p(t) = \langle l, T\rangle$ for some $l \in \mathcal{L}$ and q is F otherwise, and
- (ix) if H is $\forall$ of type $((\sigma \to o) \to o)$ then, for any $p \in D_{\sigma\to o}$, $V_\phi(H)(p) = \langle \forall_l^\sigma(p), q\rangle$, where q is T if for every $t \in D_\sigma$ there is some $l \in \mathcal{L}$ such that $p(t) = \langle l, T\rangle$ and q is F otherwise.

(2) H is $(H_1\ H_2)$. In this case, $V_\phi(H) = V_\phi(H_1)(V_\phi(H_2))$.

(3) H is $(\lambda x\, H_1)$. Let x be of type σ and, for any $t \in D_\sigma$, let $\phi(x := t)$ be the assignment that is identical to ϕ except that it maps x to t. Then $V_\phi(H) = p$ where p is the function on D_σ such that $p(t) = V_{\phi(x:=t)}(H_1)$.

The definition of pre-structure is, however, not sufficiently restrictive to ensure that $V_\phi(H) \in D_\sigma$ for every term H of type σ. The solution to this problem is to deem that only certain pre-structures are acceptable for the purpose of determining meanings, and to identify these pre–structures and the meaning function simultaneously. Formally, we say a pre-structure $\langle \mathcal{L}, \{D_\sigma\}_\sigma, I, C \rangle$ is a *structure* or *interpretation* only if, for any assignment ϕ and any term H of type σ, $V_\phi(H) \in D_\sigma$ where V_ϕ is given by the conditions above. It follows, of course, that $V_\phi(H)$ is well defined relative to a structure. For a closed term H, $V_\phi(H)$ is independent of the assignment ϕ and may, therefore, be thought of as the denotation of H relative to the given structure.

The idea of a structure as defined here is similar to the notion of a *general model* in [Hen50], the chief difference being that we use a domain of labelled truth values for D_o. Note that our structures degenerate to general models in Henkin's sense in the case that the the set of labels $\mathcal{L}$ is restricted to a two element set. It is of interest also to observe that the domains $D_{\sigma\to\tau}$ in our structures are collections of functions from D_σ to D_τ as opposed to functions in combination with a way of identifying them, and the semantics engendered is extensional in this sense. The axiom of extensionality is not a formula of our logic since its vocabulary does not include the equality symbol. However, it is easily seen that the effect of this axiom holds in our structures at a meta-level: two elements of the domain $D_{\sigma\to\tau}$ are equal just in case they produce the same value when given identical objects from D_σ. As a final observation, in the special case that the domains $D_{\sigma\to\tau}$ include all the functions from D_σ to D_τ we obtain a structure akin to the *standard* models of [Hen50].

Various logical notions pertaining to formulas in our language can be explained by recourse to the definition of V_ϕ. We say, for instance, that a formula H is satisfied by a structure and an assignment ϕ if $V_\phi(H) = \langle l, T \rangle$ relative to that structure. A *valid* formula is one that is satisfied by every structure and every assignment. Given a set of formulas Γ, and a formula A, we say that A is a logical consequence of Γ, written $\Gamma \models A$, just in case A is satisfied by every structure and assignment that also satisfies each member of Γ.

Given a finite set of formulas Θ, let $\vee\Theta$ denote the disjunction of the formulas in Θ if Θ is nonempty and the formula $\neg\top$ otherwise. The following theorem relates the model-theoretic semantics that is presented in this subsection for our higher-order logic to the proof-theoretic semantics presented in the previous subsection. The proof of this theorem and a fuller development of the ideas here may be found in [Nad94].

Theorem 1 *Let Γ,Θ be finite sets of formulas. Then $\Gamma \longrightarrow \Theta$ has a* **C***-proof if and only if $\Gamma \models (\vee\Theta)$.*

3.5 Predicate Variables and the Subformula Property

As noted in Subsection 3.3, the proof systems for first-order logic and our higher-order logic look similar: the only real differences are, in fact, in the presence of the λ-conversion rules and the richer syntax of formulas. The impact of these differences is, however, nontrivial. An important property of formulas in first-order logic is that performing substitutions into them preserves their logical structure — the resulting formula is in a certain precise sense a subformula of the original formula (see [Gen69]). A similar observation can unfortunately not be made about formulas in our higher-order logic. As an example, consider the formula $F = ((p\ a) \supset (Y\ a))$; we assume that p

and a are parameters of suitable type here and that Y is a variable. Now let θ be the substitution

$$\{\langle Y, \lambda z\,((p\ z) \wedge \forall x\,((z\ x) \supset (b\ x)))\rangle\},$$

where x, y and z are variables and b is a parameter. Then

$$\theta(F) \equiv (p\ a) \supset ((p\ a) \wedge \forall x\,((a\ x) \supset (b\ x))).$$

As can be seen from this example, applying a substitution to a formula in which a predicate variable appears free has the potential for dramatically altering the top-level logical structure of the formula.

The above observation has proof-theoretic consequences that should be mentioned. One consequence pertains to the usefulness of cut-elimination theorems. These theorems have been of interest in the context of logic because they provide an insight into the nature of deduction. Within first-order logic, for instance, this theorem leads to the subformula property: if a sequent has a proof, then it has one in which every formula in the proof is a subformula of some formula in the sequent being proved. Several useful structural properties of deduction in the first-order context can be observed based on this property. From the example presented above, it is clear that the subformula property does not hold under any reasonable interpretation for our higher-order logic even though it admits a cut-elimination theorem; predicate terms containing connectives and quantifiers may be generalized upon in the course of a derivation and thus intermediate sequents may have formulas whose structure cannot be predicted from the formulas in the final one. For this reason, the usefulness of cut-elimination as a device for teasing out the structural properties of derivations in higher-order logic has generally been doubted.

A related observation concerns the automation of deduction. The traditional method for constructing a proof of a formula in a logic that involves quantification consists, in a general sense, of substituting expressions for existentially quantified variables and then verifying that the resulting formula is a tautology. In a logic where the propositional structure remains invariant under substitutions, the search for a proof can be based on this structure and the substitution (or, more appropriately, unification) process may be reduced to a constraint on the search. However, the situation is different in a logic in which substitutions can change the propositional structure of formulas. In such logics, the construction of a proof often involves finding the "right" way in which to change the propositional structure as well. As might be imagined, this problem is a difficult one to solve in general, and no good method that is also complete has yet been described for determining these kinds of substitutions in our higher-order logic. The existing theorem-provers for this logic either sacrifice completeness [Ble79, ACMP84] or are quite intractable for this reason [Hue73a, And89].

In the next section we describe a certain class of formulas from our higher-order logic. Our primary interest in these formulas is that they provide a logical basis for higher-order features in logic programming. There is an auxiliary interest, however, in these formulas in the light of the above observations. The special structure of these formulas enables us to obtain useful information about derivations concerning them from the cut-elimination theorem for higher-order logic. This information, in turn, enables the description of a proof procedure that is complete and that at the same time finds substitutions for predicate variables almost entirely through unification. Our study of these formulas thus also demonstrates the utility of the cut-elimination theorem even in the context of a higher-order logic.

4 Higher-Order Horn Clauses

In this section we describe a logical language that possesses all the enhancements to first-order Horn clauses that were discussed in Section 2. The first step in this direction is to identify a subcollection of the class of terms of the higher-order logic that was described in the previous section.

Definition 2. A *positive* term is a term of our higher-order language that does not contain occurrences of the symbols $\neg$, $\supset$ and $\forall$. The collection of positive terms that are in λ-normal form is called the *positive Herbrand Universe* and is denoted by $\mathcal{H}^+$.

The structure of positive terms clearly satisfies the requirement that we wish to impose on the arguments of atomic formulas: these terms are constructed by the operations of application and abstraction and function (and predicate) variables may appear in them as also the logical symbols $\top$, $\vee$, $\wedge$ and $\exists$. As we shall see presently, $\mathcal{H}^+$ provides the domain of terms used for describing the results of computations. It thus plays the same role in our context as does the first-order Herbrand Universe in other discussions of logic programming.

A higher-order version of Horn clauses is now identified by the following definition.

Definition 3. A *positive atomic formula* is a formula of the form $(p\ T_1\ \ldots\ T_n)$, where p is a parameter or a variable and, for $1 \leq i \leq n$, $T_i \in \mathcal{H}^+$. Recall that such a formula is rigid just in case p is a parameter. Let A and A_r be symbols that denote positive and rigid positive atomic formulas respectively. Then the (higher-order) goal formulas and definite clauses are the G- and D-formulas given by the following inductive rules:

$$\begin{array}{rcl} G & ::= & \top \mid A \mid G \wedge G \mid G \vee G \mid \exists x\, G \\ D & ::= & A_r \mid G \supset A_r \mid \forall x\, D \end{array}$$

A finite collection of closed definite clauses is referred to as a *program* and a goal formula is also called a *query*.

There is an alternative characterization of the goal formulas just defined: they are, in fact, the terms in $\mathcal{H}^+$ of type o. The presentation above is chosen to exhibit the correspondence between the higher-order formulas and their first-order counterparts. The first-order formulas are contained in the corresponding higher-order formulas under an implicit encoding that essentially assigns types to the first-order terms and predicates. To be precise, let i be a sort other that o. The encoding then assigns the type i to variables and parameters, the type $i \to \cdots \to i \to i$, with $n+1$ occurrences of i, to each n-ary function symbol, and the type $i \to \cdots \to i \to o$, with n occurrences of i, to each n-ary predicate symbol. Looked at differently, our formulas contain within them a many-sorted version of first-order definite clauses and goal formulas. In the reverse direction, the above definition is but a precise description of the generalization outlined informally in Section 2. Of particular note are the restriction of arguments of atoms to positive terms, the requirement of a specific name for the "procedure" defined by a definite clause and the ability to quantify over predicates and functions. The various examples discussed in Section 2 can be rendered almost immediately into the current syntax, the main difference being the use of a curried notation.

Example 2. Let *list* be a unary type constructor, let *int* and i be sorts. Further, let *nil* and *nil'* be parameters of type $(list\ i)$ and $(list\ int)$ and let *cons* and *cons'* be parameters of type $i \to (list\ i) \to (list\ i)$ and $int \to (list\ int) \to (list\ int)$. The following formulas constitute a program under the assumption that *mappred* is a parameter of type $(i \to int \to o) \to (list\ i) \to (list\ int) \to o$ and that P, L_1, L_2, X and Y are variables of the required types:

$$\forall P\,(mappred\ P\ nil\ nil'),$$
$$\forall P, L_1, L_2, X, Y\,(((P\ X\ Y) \wedge (mappred\ P\ L_1\ L_2)) \supset (mappred\ P\ (cons'\ X\ L_1)\ (cons\ Y\ L_2))).$$

Assuming that *age* is a parameter of type $i \rightarrow int \rightarrow o$, *bob* and *sue* are parameters of type i and L is a variable of type $(list\ int)$, the formula

$$(mappred\ age\ (cons\ bob\ (cons\ sue\ nil))\ L)$$

constitutes a query. If $mapfun$ is a parameter of type $(int \rightarrow int) \rightarrow (list\ int) \rightarrow (list\ int) \rightarrow o$ and F, L_1, L_2 and X are variables of appropriate types, the following formulas, once again, constitute a program:

$$\forall F\,(mapfun\ F\ nil\ nil),$$
$$\forall F, L_1, L_2, X\,((mapfun\ F\ L_1\ L_2) \supset (mapfun\ F\ (cons\ X\ L_1)\ (cons\ (F\ X)\ L_2))).$$

If 1, 2, and h are parameters of type int, int, and $int \rightarrow int \rightarrow int$ respectively and L, X and F are variables of suitable types, the following are queries:

$$(mapfun\ (\lambda X\,(h\ 1\ X))\ (cons\ 1\ (cons\ 2\ nil))\ L), \quad \text{and}$$
$$(mapfun\ F\ (cons\ 1\ (cons\ 2\ nil))\ (cons\ (h\ 1\ 1)\ (cons\ (h\ 1\ 2)\ nil))).$$

Higher-order definite clauses and goal formulas are intended to provide for a programming paradigm in a manner analogous to their first-order counterparts. The notion of computation corresponds as before to solving a query from a given program. The desire to preserve the essential character of Horn clause logic programming dictates the mechanism for carrying out such a computation: an abstract interpreter for our language must perform a search whose structure is guided by the top-level logical symbol in the query being solved. There is, however, a potential problem in the description of such an interpreter caused by the possibility for predicate variables to appear in extensional positions in goal formulas. We recall from Section 3 that substitutions for such variables have the ability to alter the top-level logical structure of the original formula. In the specific context of interest, we see that goal formulas do not remain goal formulas under arbitrary instantiations. For example, consider the instantiation of the goal formula $\exists P\,(P\ a)$ with the term $\lambda x\,\neg(q\ x)$; we assume that P and x are variables here and that a and q are parameters. This instantiation produces the formula $\neg(q\ a)$ which is obviously not a goal formula. If such instantiations must be performed in the course of solving queries similar to the given one, an interpreter that proceeds by examining the top-level structure of only goal formulas cannot be described for our language. The computational mechanism for our language would, therefore, have to diverge even in principle from the one used in the first-order context.

Fortunately this problem has a solution that is adequate at least from a pragmatic perspective. The essential idea is to consider the domain of our computations to be limited to that of positive terms. In particular, instantiations with only positive terms will be used for definite clauses and goal formulas in the course of solving queries. Now, it is easily seen that "positive" instantiations of quantifiers in goal formulas and definite clauses yield formulas that are themselves goal formulas and definite clauses respectively. Problems such as those just discussed would, therefore, not arise in this context. We adopt this solution in our discussions below. Although this solution is adequate in a practical sense, we note that there is a question about its acceptability from a logical perspective; in particular, using it may be accompanied by a loss in logical completeness. We discuss this issue in the next section.

In presenting an abstract interpreter for our language, we find a notation for positive instances of a set of definite clauses useful. This notation is described below.

Definition 4. A (closed) *positive substitution* is a substitution whose range is a set of (closed) terms contained in $\mathcal{H}^+$. Let D be a closed definite clause. Then the collection of its closed positive instances, denoted by $[D]$, is

(i) $\{D\}$ if D is of the form A or $G \supset A$, and

(ii) $\bigcup\{[\varphi(D')] \mid \varphi$ is a closed positive substitution for $x\}$ if D is of the form $\forall x\, D'$.

This notation is extended to programs as follows: if $\mathcal{P}$ is a program, $[\mathcal{P}] = \bigcup\{[D] \mid D \in \mathcal{P}\}$.

We now specify the abstract interpreter in terms of the desired search related interpretation for each logical symbol.

Definition 5. Let $\mathcal{P}$ be a program and let G be a closed goal formula. We use the notation $\mathcal{P} \vdash_O G$ to signify that our abstract interpreter succeeds on G when given $\mathcal{P}$; the subscript on $\vdash_O$ acknowledges the "operational" nature of the notion. Now, the success/failure behavior of the interpreter on closed goal formula is specified as follows:

(i) $\mathcal{P} \vdash_O \top$,

(ii) $\mathcal{P} \vdash_O A$ where A is an atomic formula if and only if $A \equiv A'$ for some $A' \in [\mathcal{P}]$ or for some $G \supset A' \in [\mathcal{P}]$ such that $A \equiv A'$ it is the case that $\mathcal{P} \vdash_O G$,

(iii) $\mathcal{P} \vdash_O G_1 \vee G_2$ if and only if $\mathcal{P} \vdash_O G_1$ or $\mathcal{P} \vdash_O G_2$,

(iv) $\mathcal{P} \vdash_O G_1 \wedge G_2$ if and only if $\mathcal{P} \vdash_O G_1$ and $\mathcal{P} \vdash_O G_2$, and

(v) $\mathcal{P} \vdash_O \exists x\, G$ if and only if $\mathcal{P} \vdash_O \varphi(G)$ for some φ that is a closed positive substitution for x.

The description of the abstract interpreter has two characteristics which require further explanation. First, it specifies the behavior of the interpreter only on closed goal formulas, whereas a query is, by our definition, an arbitrary goal formula. Second, while it defines what a computation should be, it leaves unspecified what the *result* of such a computation is. The explanations of these two aspects are, in a certain sense, related. The typical scenario in logic programming is one where a goal formula with some free variables is to be solved relative to some program. The calculation that is intended in this situation is that of solving the existential closure of the goal formula from the program. If this calculation is successful, the result that is expected is a set of substitutions for the free variables in the given query that make it so. We observe that the behavior of our abstract interpreter accords well with this view of the outcome of a computation: the success of the interpreter on an existential query entails its success on a particular instance of the query and so it is reasonable to require a specific substitution to be returned as an "answer."

Example 3. Suppose that our language includes all the parameters and variables described in Example 2. Further, suppose that our program consists of the definite clauses defining *mappred* in that example and the following in which 24 and 23 are parameters if type *int*:

$(age\ bob\ 24)$,
$(age\ sue\ 23)$.

Then, the query

$$(mappred\ age\ (cons\ bob\ (cons\ sue\ nil))\ L)$$

in which L is a free variable actually requires the goal formula

$$\exists L\,(mappred\ age\ (cons\ bob\ (cons\ sue\ nil))\ L).$$

to be solved. There is a solution to this goal formula that, in accordance with the description of the abstract interpreter, involves solving the following "subgoals" in sequence:

$$(mappred\ age\ (cons\ bob\ (cons\ sue\ nil))\ (cons'\ 24\ (cons'\ 23\ nil')))$$
$$(age\ bob\ 24) \wedge (mappred\ age\ (cons\ sue\ nil)\ (cons'\ 23\ nil'))$$
$$(age\ bob\ 24)$$
$$(mappred\ age\ (cons\ sue\ nil)\ (cons'\ 23\ nil'))$$
$$(age\ sue\ 23) \wedge (mappred\ age\ nil\ nil')$$
$$(age\ sue\ 23)$$
$$(mappred\ age\ nil\ nil').$$

The answer to the original query that is obtained from this solution is the substitution

$$(cons'\ 24\ (cons'\ 23\ nil'))$$

for L.

As another example, assume that our program consists of the clauses for $mapfun$ in Example 2 and that the query is now the goal formula

$$(mapfun\ F\ (cons\ 1\ (cons\ 2\ nil))\ (cons\ (h\ 1\ 1)\ (cons\ (h\ 1\ 2)\ nil)))$$

in which F is a free variable. Once again we can construct the goal formula whose solution is implicitly called for by this query. Further, a successful solution path may be traced to show that an answer to the query is the value $\lambda x\,(h\ 1\ x)$ for F.

We have, at this stage, presented a higher-order generalization to the Horn clauses of first-order logic and we have outlined, in an abstract fashion, a notion of computation in the context of our generalization that preserves the essential character of the notion in the first order case. We have, through this discussion, provided a framework for higher-order logic programming. However, there are two respects in which the framework that we have presented is incomplete. First, we have not provided a justification based on logic for the idea of computation that we have described. We would like to manifest a connection with logic to be true to the spirit of logic programming in general and to benefit from possible declarative interpretations of programs in particular. Second, the abstract interpreter that has been described is not quite adequate as a basis for a practical programming language. As evidenced in Example 3, an unacceptable degree of clairvoyance is needed in determining if a given query has a successful solution — an answer to the query must be known at the outset. A viable evaluation procedure therefore needs to be described for supporting the programming paradigm outlined here. We undertake these two tasks in Sections 5 and 6 respectively.

5 The Meaning of Computations

We may attempt to explain the meaning of a computation as described in the previous section by saying that a query succeeds from a given program if and only if its existential closure is provable from, or a logical consequence of, the program. Accepting this characterization without further argument is, however, problematic. One concern is the treatment of quantifiers. From Definition 5 we see that only positive instances of definite clauses are used and success on existentially quantified goal formulas depends only on success on a closed positive instance of the goal. It is unclear that these restrictions carry over to the idea of provability as well. A related problem concerns the search semantics accorded to the logical connectives. We note, for instance, that success on a goal formula of the form $G_1 \lor G_2$ depends on success on either G_1 or G_2. This property does not hold of provability in general: a disjunctive formula may have a proof without either of the disjuncts being provable.

A specific illustration of the above problems is provided by the following derivation of the goal formula $\exists Y\,(p\ Y)$ from the definite clause $\forall X\,(X \supset (p\ a))$; we assume that p, a and b are parameters here and that X and Y are variables.

$$\dfrac{\dfrac{\dfrac{(p\ b) \longrightarrow (p\ b)}{(p\ b) \longrightarrow \exists Y\,(p\ Y)}\ \exists\text{-R}}{\longrightarrow \exists Y\,(p\ Y), \neg(p\ b)}\ \neg\text{-R} \qquad \dfrac{(p\ a) \longrightarrow (p\ a)}{(p\ a) \longrightarrow \exists Y\,(p\ Y)}\ \exists\text{-R}}{\dfrac{\neg(p\ b) \supset (p\ a) \longrightarrow \exists Y\,(p\ Y)}{\forall X\,(X \supset (p\ a)) \longrightarrow \exists Y\,(p\ Y)}\ \forall\text{-L}}\ \supset\text{-L}$$

The penultimate sequent in this derivation is

$$\neg(p\ b) \supset (p\ a) \longrightarrow \exists Y\,(p\ Y). \qquad (*)$$

The antecedent of this sequent is obtained by substituting a term that is not positive into a definite clause. This sequent obviously has a derivation. There is, however, no term T such that

$$\neg(p\ b) \supset (p\ a) \longrightarrow (p\ T)$$

has a derivation. This is, of course, a cause for concern. If all derivations of

$$\forall X\,(X \supset (p\ a)) \longrightarrow \exists Y\,(p\ Y)$$

involve the derivation of $(*)$, or of sequents similar to $(*)$, then the idea of proving $\exists Y\,(p\ Y)$ would diverge from the idea of solving it, at least in the context where the program consists of the formula $\forall X\,(X \supset (p\ a))$.

We show in this section that problems of the sort described in the previous paragraph do not arise, and that the notions of success and provability in the context of our definite clauses and goal formulas coincide. The method that we adopt for demonstrating this is the following. We first identify a **C'**-proof as a **C**-proof in which each occurrence of $\forall$-L and $\exists$-R constitutes a generalization upon a closed term from $\mathcal{H}^+$. In other words, in each appearance of figures of the forms

$$\frac{[T/x]P, \Delta \longrightarrow \Theta}{\forall x\, P, \Delta \longrightarrow \Theta} \qquad\qquad \frac{\Delta \longrightarrow \Theta, [T/x]P}{\Delta \longrightarrow \Theta, \exists x\, P}$$

it is the case that T is instantiated by a closed term from $\mathcal{H}^+$. We shall show then that if Γ consists only of closed definite clauses and Δ consists only of closed goal formulas, then the sequent $\Gamma \longrightarrow \Delta$ has a **C**-proof only if has a **C'**-proof. Now **C'**-proofs of sequents of the kind described have the following characteristic: every sequent in the derivation has an antecedent consisting solely of closed definite clauses and a succedent consisting solely of closed goal formulas. This structural property of the derivation can be exploited to show rather directly that the existence of a proof coincides with the possibility of success on a goal formula.

5.1 Restriction to Positive Terms

We desire to show that the use of only **C'**-proofs, *i.e.*, the focus on positive terms, does not restrict the relation of provability as it pertains to definite clauses and goal formulas. We do this by describing a transformation from an arbitrary **C**-proof to a **C'**-proof. The following mapping on terms is useful for this purpose.

Definition 6. Let x and y be variables of type o and, for each σ, let z_σ be a variable of type $\sigma \to o$. Then the function *pos* on terms is defined as follows:

(i) If T is a constant or a variable

$$pos(T) = \begin{cases} \lambda x\, \top & \text{if } T \text{ is } \neg \\ \lambda x\, \lambda y\, \top, & \text{if } T \text{ is } \supset \\ \lambda z_\sigma\, \top & \text{if } T \text{ is } \forall \text{ of type } (\sigma \to o) \to o \\ T & \text{otherwise.} \end{cases}$$

(ii) $pos((T_1\ T_2)) = (pos(T_1)\ pos(T_2))$.

(iii) $pos(\lambda w\, T) = \lambda w\, pos(T)$.

Given a term T, the λ-normal form of $pos(T)$ is denoted by T^+.

The mapping defined above is a "positivization" operation on terms as made clear in the following lemma whose proof is obvious.

Lemma 2 *For any term T, $T^+ \in \mathcal{H}^+$. Further, $\mathcal{F}(T^+) \subseteq \mathcal{F}(T)$. In particular, if T is closed, then T^+ is a closed positive term. Finally, if $T \in \mathcal{H}^+$ then $T = T^+$.*

Another property of the mapping defined above is that it commutes with λ-conversion. This fact follows easily from the lemma below.

Lemma 3 *For any terms T_1 and T_2, if T_1 λ-converts to T_2 then $pos(T_1)$ also λ-converts to $pos(T_2)$.*

Proof. We use an induction on the length of the conversion sequence. The key is in showing the lemma for a sequence of length 1. It is easily seen that if T_2 is obtained from T_1 by a single application of the α- or η-conversion rule, then $pos(T_2)$ results from $pos(T_1)$ by a similar rule. Now, let A be substitutable for x in B. Then an induction on the structure of B confirms that $pos(S_x^A B) = S_x^{pos(A)} pos(B)$. Thus, if R_1 is $((\lambda x\, B)\ A)$ and R_2 is $S_x^A B$, it must be the case that $pos(R_1)$ β-converts to $pos(R_2)$. An induction on the structure of T_1 now verifies that if T_2 results from it by a single β-conversion step, then $pos(T_1)$ β-converts to $pos(T_2)$.

$\square$

We will need to consider a sequence of substitutions for variables in a formula in the discussions that follow. In these discussions it is notationally convenient to assume that substitution is a right associative operation. Thus $[R_2/x_2][R_1/x_1]T$ is to be considered as denoting the term obtained by first substituting R_1 for x_1 in T and then substituting R_2 for x_2 in the result.

Lemma 4 *If T is a term in $\mathcal{H}^+$ and $R_1, \ldots, R_n$ are arbitrary terms ($n \geq 0$), then*

$$([R_n/x_n] \ldots [R_1/x_1]T)^+ = [(R_n)^+/x_n] \ldots [(R_1)^+/x_1]T.$$

In particular, this is true when T is an atomic goal formula or an atomic definite clause.

Proof. We note first that for any term $T \in \mathcal{H}^+$, $pos(T) = T$ and thus $T^+ = T$. Using Lemma 3, it is easily verified that

$$([R_n/x_n] \ldots [R_1/x_1]T)^+ = [(R_n)^+/x_n] \ldots [(R_1)^+/x_1]T^+.$$

The lemma follows from these observations.

□

The transformation of a **C**-proof into a **C'**-proof for a sequent of the kind that we are interested in currently can be described as the result of a recursive process that works upwards from the root of the proof and uses the positivization operation on the terms generalized upon in the $\forall$-L and $\exists$-R rules. This recursive process is implicit in the proof of the following theorem.

Theorem 5 *Let Δ be a program and let Θ be a finite set of closed goal formulas. Then $\Delta \longrightarrow \Theta$ has a **C**-proof only if it also has a **C'**-proof.*

Proof. We note first that all the formulas in $\Delta \longrightarrow \Theta$ are closed. Hence, we may assume that the $\forall$-L and $\exists$-R inference figures that are used in the **C**-proof of this sequent generalize on closed terms (*i.e.*, they are obtained by replacing T in the corresponding schemata by closed terms). The standard technique of replacing all occurrences of a free variable in a proof by a parameter may be used to ensure that this is actually the case.

Given a set of formulas Γ of the form

$$\{[R^1_{l_1}/z^1_{l_1}] \ldots [R^1_1/z^1_1]F_1, \ldots, [R^r_{l_r}/z^r_{l_r}] \ldots [R^r_1/z^r_1]F_r\},$$

where $r, l_1, \ldots, l_r \geq 0$, we shall use the notation Γ^+ to denote the set

$$\{[(R^1_{l_1})^+/z^1_{l_1}] \ldots [(R^1_1)^+/z^1_1]F_1, \ldots, [(R^r_{l_r})^+/z^r_{l_r}] \ldots [(R^r_1)^+/z^r_1]F_r\}.$$

Now let Δ be a set of the form

$$\{[T^1_{n_1}/x^1_{n_1}] \ldots [T^1_1/x^1_1]D_1, \ldots, [T^t_{n_t}/x^t_{n_t}] \ldots [T^t_1/x^t_1]D_t\},$$

i.e., a set of formulas, each member of which is obtained by performing a sequence of substitutions into a definite clause. Similarly, let Θ be a set of the form

$$\{[S^1_{m_1}/y^1_{m_1}] \ldots [S^1_1/y^1_1]G_1, \ldots, [S^s_{m_s}/y^s_{m_s}] \ldots [S^s_1/y^s_1]G_s\},$$

i.e., a set obtained by performing sequences of substitutions into goal formulas. We claim that if $\Delta \longrightarrow \Theta$ has a **C**-proof in which all the $\forall$-L and the $\exists$-R figures generalize on closed terms, then $\Delta^+ \longrightarrow \Theta^+$ must have a **C'**-proof. The theorem is an obvious consequence of this claim.

The claim is proved by an induction on the height of **C**-proofs for sequents of the given sort. If this height is 1, the given sequent must be an initial sequent. There are, then, two cases to consider. In the first case, for some i such that $1 \leq i \leq s$ we have that $[S^i_{m_i}/y^i_{m_i}] \ldots [S^i_1/y^i_1]G_i$ is $\top$. But then G_i must be an atomic formula. Using Lemma 4 we then see that $[(S^i_{m_i})^+/y^i_{m_i}] \ldots [(S^i_1)^+/y^i_1]G_i$ must also be $\top$. In the other case, for some i, j such that $1 \leq i \leq s$ and $1 \leq j \leq t$, we have that

$$[R^j_{n_j}/x^j_{n_j}] \ldots [R^j_1/x^j_1]D_j \equiv [S^i_{m_i}/y^i_{m_i}] \ldots [S^i_1/y^i_1]G_i$$

and, further, that these are atomic formulas. From the last observation, it follows that D_j and G_i are atomic formulas. Using Lemma 4 again, it follows that

$$[(R^j_{n_j})^+/x^j_{n_j}] \ldots [(R^j_1)^+/x^j_1]D_j \equiv [(S^i_{m_i})^+/y^i_{m_i}] \ldots [(S^i_1)^+/y^i_1]G_i.$$

Thus in either case it is clear that $\Delta^+ \longrightarrow \Theta^+$ is an initial sequent.

We now assume that the claim is true for derivations of sequents of the requisite sort that have height h, and we verify it for sequents with derivations of height $h + 1$. We argue by considering the possible cases for the last inference figure in such a derivation. We observe that substituting into a definite clause cannot yield a formula that has $\wedge$, $\vee$, $\neg$ or $\exists$ as its top-level connective. Thus the last inference figure cannot be an $\wedge$-L, an $\vee$-L, a $\neg$-L or a $\exists$-L. Further, a simple induction on the heights of derivations shows that if a sequent consists solely of formulas in λ-normal form, then any **C**-proof for it that contains the inference figure λ can be transformed into a shorter **C**-proof in which λ does not appear. Since each formula in $\Delta \longrightarrow \Theta$ must be in λ-normal form, we may assume that the last inference figure in its **C**-proof is not a λ. Thus, the only figures that we need to consider are $\supset$-L, $\forall$-L, $\wedge$-R, $\vee$-R, $\neg$-R, $\supset$-R, $\exists$-R, and $\forall$-R.

Let us consider first the case for an $\wedge$-R, *i.e.*, when the last inference figure is of the form

$$\frac{\Delta \longrightarrow \Theta', B \qquad\qquad \Delta \longrightarrow \Theta', D}{\Delta \longrightarrow \Theta', B \wedge D}$$

In this case $\Theta' \subseteq \Theta$ and for some i, $1 \leq i \leq s$, $[S^i_{m_i}/y^i_{m_i}] \ldots [S^i_1/y^i_1]G_i = B \wedge D$. Our analysis breaks up into two parts depending on the structure of G_i:

(1) If G_i is an atomic formula, we obtain from Lemma 4 that

$$[(S^i_{m_i})^+/y^i_{m_i}] \ldots [(S^i_1)^+/y^i_1]G_i = (B \wedge D)^+ = B^+ \wedge D^+.$$

Now B and D can be written as $[B/y]y$ and $[D/y]y$, respectively. From the hypothesis it thus follows that $\Delta^+ \longrightarrow (\Theta')^+, B^+$ and $\Delta^+ \longrightarrow (\Theta')^+, D^+$ have **C'**-proofs. Using an $\wedge$-R inference figure in conjunction with these, we obtain a **C'**-proof for $\Delta^+ \longrightarrow \Theta^+$.

(2) If G_i is not an atomic formula then it must be of the form $G^1_i \wedge G^2_i$. But then $B = [S^i_{m_i}/y^i_{m_i}] \ldots [S^i_1/y^i_1]G^1_i$ and $D = [S^i_{m_i}/y^i_{m_i}] \ldots [S^i_1/y^i_1]G^2_i$. It follows from the hypothesis that **C'**-proofs exist for

$$\Delta^+ \longrightarrow (\Theta')^+, [(S^i_{m_i})^+/y^i_{m_i}] \ldots [(S^i_1)^+/y^i_1]G^1_i \text{ and } \Delta^+ \longrightarrow (\Theta')^+, [(S^i_{m_i})^+/y^i_{m_i}] \ldots [(S^i_1)^+/y^i_1]G^2_i.$$

A proof for $\Delta^+ \longrightarrow \Theta^+$ can be obtained from these by using an $\wedge$-R inference figure.

An analogous argument can be provided when the last figure is $\vee$-R. For the case of $\supset$-L, we observe first that if the result of performing a sequence of substitutions into a definite clause D is a formula of the form $B \supset C$, then D must be of the form $G \supset A_r$ where G is a goal formula and A_r is a rigid positive atom. An analysis similar to that used for $\wedge$-R now verifies the claim.

Consider now the case when the last inference figure is a $\neg$-R, *i.e.*, of the form

$$\frac{B, \Delta \longrightarrow \Theta'}{\Delta \longrightarrow \Theta', \neg B.}$$

We see in this case that for some suitable i, $[S^i_{m_i}/y^i_{m_i}]\ldots[S^i_1/y^i_1]G_i = \neg B$. But then G_i must be an atomic goal formula and by Lemma 4

$$[(S^i_{m_i})^+/y^i_{m_i}]\ldots[(S^i_1)^+/y^i_1]G_i = (\neg B)^+ = \top.$$

Thus, $\Delta^+ \longrightarrow \Theta^+$ is an initial sequent and the claim follows trivially. Similar arguments can be supplied for $\supset$-R and $\forall$-R.

The only remaining cases are those when the last inference figure is a $\exists$-R or a $\forall$-L. In the former case the last inference figure must have the form

$$\frac{\Delta \longrightarrow \Theta', [T/w]P}{\Delta \longrightarrow \Theta', \exists w\, P}$$

where $\Theta' \subseteq \Theta$ and for some i, $1 \le i \le s$, $[S^i_{m_i}/y^i_{m_i}]\ldots[S^i_1/y^i_1]G_i = \exists w\, P$. We assume, without loss of generality, that w is distinct from the variables $y^i_1, \ldots, y^i_{m_i}$ as well as the variables that are free in $S^i_1, \ldots, S^i_{m_i}$. There are once again two subcases based on the the structure of G_i:

(1) If G_i is an atomic formula, it follows from Lemma 4 that

$$[(S^i_{m_i})^+/y^i_{m_i}]\ldots[(S^i_1)^+/y^i_1]G_i = (\exists w\, P)^+ = \exists w\, (P)^+.$$

Writing $[T/w]P$ as $[T/w][P/u]u$ and invoking the hypothesis, we see that a **C'**-proof must exist for $\Delta^+ \longrightarrow (\Theta')^+, [T^+/w]P^+$. Adding a $\exists$-R figure below this yields a derivation for

$$\Delta^+ \longrightarrow (\Theta')^+, \exists w\, (P)^+,$$

which is identical to $\Delta^+ \longrightarrow \Theta^+$. Further, this must be a **C'**-proof since T is a closed term by assumption and hence, by Lemma 2, T^+ must be a closed positive term.

(2) If G_i is a non-atomic formula, it must be of the form $\exists x\, G'_i$ where G'_i is a goal formula. But now $P = [S^i_{m_i}/y^i_{m_i}]\ldots[S^i_1/y^i_1]G'_i$. Thus, $\Delta^+ \longrightarrow (\Theta')^+, [T^+/w][(S^i_{m_i})^+/y^i_{m_i}]\ldots[(S^i_1)^+/y^i_1]G'_i$ has a **C'**-proof by the hypothesis. By adding a $\exists$-R inference figure below this, we obtain a **C'**-proof for $\Delta^+ \longrightarrow (\Theta')^+, \exists w\, ([(S^i_{m_i})^+/y^i_{m_i}]\ldots[(S^i_1)^+/y^i_1]G'_i)$. Noting that

$$\exists w\, ([(S^i_{m_i})^+/y^i_{m_i}]\ldots[(S^i_1)^+/y^i_1]G'_i) \equiv [(S^i_{m_i})^+/y^i_{m_i}]\ldots[(S^i_1)^+/y^i_1]\exists w\, G'_i,$$

we see that the claim must be true.

The argument for $\forall$-L is similar to the one for the second subcase of $\exists$-R.

$\square$

As mentioned already, Theorem 5 implicitly describes a transformation on **C**-proofs. It is illuminating to consider the result of this transformation on the derivation presented at the beginning of this section. We invite the reader to verify that this derivation will be transformed into the following:

$$\dfrac{\longrightarrow\ \exists Y\,(p\ Y), \top \qquad \dfrac{\dfrac{(p\ a)\ \longrightarrow\ (p\ a)}{(p\ a)\ \longrightarrow\ \exists Y\,(p\ y)}\ \exists\text{-R}}{}}{\dfrac{\top \supset (p\ a)\ \longrightarrow\ \exists Y\,(p\ Y)}{\forall X\,(X \supset (p\ a))\ \longrightarrow\ \exists y\,(p\ Y)}\ \forall\text{-L}}\ \supset\text{-L}$$

Notice that there is no ambiguity about the answer substitution that should be extracted from this derivation for the existentially quantified variable Y.

5.2 Provability and Operational Semantics

We now show that the possibility of success on a goal formula given a program coincides with the existence of a proof for that formula from the program. Theorem 5 allows us to focus solely on **C'**-proofs in the course of establishing this fact, and we do this implicitly below.

The following lemma shows that any set of definite clauses is consistent.

Lemma 6 *There can be no derivation for a sequent of the form $\Delta \longrightarrow$ where Δ is a program.*

Proof. Suppose the claim is false. Then there is a least h and a program Δ such that $\Delta \longrightarrow$ has a derivation of height h. Clearly $h > 1$. Considering the cases for the last inference figure ($\supset$-L or $\forall$-L with the latter generalizing on a closed positive term), we see a sequent of the same sort must have a shorter proof.

□

The lemma below states the equivalence of classical and intuitionistic provability in the context of Horn clauses. This observation is useful in later analysis.

Lemma 7 *Let Δ be a program and let G be a closed goal formula. Then $\Delta \longrightarrow G$ has a derivation only if it has one in which there is at most one formula in the succedent of each sequent.*

Proof. We make a stronger claim: a sequent of the form $\Delta \longrightarrow G_1, \ldots, G_n$, where Δ is a program and $G_1, \ldots, G_n$ are closed goal formulas, has a derivation only if for some i, $1 \leq i \leq n$, $\Delta \longrightarrow G_i$ has a derivation in which at most one formula appears in the succedent of each sequent.

The claim is proved by an induction on the heights of derivations. It is true when the height is 1 by virtue of the definition of an initial sequent. In the situation when the height is $h+1$, we consider the possible cases for the last inference figure in the derivation. The argument for $\wedge$-R and $\vee$-R are straightforward. For instance, in the former case the last inference figure in the derivation must be of the form

$$\frac{\Delta \longrightarrow \Theta, G_j^1 \qquad \Delta \longrightarrow \Theta, G_j^2}{\Delta \longrightarrow \Theta, G_j^1 \wedge G_j^2}$$

where for some j, $1 \leq j \leq n$, $G_j = G_j^1 \wedge G_j^2$ and $\Theta \subseteq \{G_1, \ldots, G_n\}$. Applying the inductive hypothesis to the upper sequents of this figure, it follows that there is a derivation of the requisite sort for $\Delta \longrightarrow G$ for some $G \in \Theta$ or for both $\Delta \longrightarrow G_j^1$ and $\Delta \longrightarrow G_j^2$. In the former case the claim follows directly, and in the latter case we use the two derivations together with an $\wedge$-R inference figure to construct a derivation of the required kind for $\Delta \longrightarrow G_j^1 \wedge G_j^2$.

Similar arguments can be provided for the cases when the last inference figure is $\exists$-R or $\forall$-L. The only additional observation needed in these cases is that the restriction to **C'**-proofs ensures that the upper sequent in these cases has the form required for the hypothesis to apply.

We are left only with the case of $\supset$ -L. In this case, the last inference figure has the form

$$\frac{\Delta' \longrightarrow \Gamma_2, G \qquad\qquad A, \Delta' \longrightarrow \Gamma_1}{G \supset A, \Delta' \longrightarrow \Gamma_1 \cup \Gamma_2}$$

where $\Delta = \{(G \supset A)\} \cup \Delta'$ and $\Theta = \Gamma_1 \cup \Gamma_2$. From Lemma 6 it follows that $\Gamma_1 \neq \emptyset$. By the hypothesis, we see that there is a derivation of the required kind for either $\Delta' \longrightarrow G_1$ for some $G_1 \in \Gamma_2$ or for $\Delta' \longrightarrow G$. In the former case, by adding $G \supset A$ to the antecedent of every sequent of the derivation, we obtain one for $\Delta \longrightarrow G_1$. In the latter case, another use of the inductive hypothesis tells us that there is a derivation of the desired kind for $A, \Delta' \longrightarrow G_2$ for some $G_2 \in \Gamma_1$. This derivation may be combined with the one for $\Delta' \longrightarrow G$ to obtain one for $\Delta \longrightarrow G_2$.

$\square$

The proof of the above lemma is dependent on only one fact: every sequent in the derivations being considered has a program as its antecedent and a set of closed goal formulas as its succedent. This observation (or one closely related to it) can be made rather directly in any situation where quantification over predicate variables appearing in extensional positions is not permitted. It holds, for instance, in the case when we are dealing with only first-order formulas. Showing that the observation also applies in the case of our higher-order formulas requires much work, as we have already seen.

Definition 7. The *length* of a derivation Ξ is defined by recursion on its height as follows:

(i) It is 1 if Ξ consists of only an initial sequent.

(ii) It is $l + 1$ if the last inference figure in Ξ has a single upper sequent whose derivation has length l.

(iii) It is $l_1 + l_2 + 1$ if the last inference figure in Ξ has two upper sequents whose derivations are of length l_1 and l_2 respectively.

The main result of this subsection requires the extraction of a successful computation from a proof of a closed goal formula from a program. The following lemma provides a means for achieving this end.

Lemma 8 *Let Δ be a program, let G be a closed goal formula and let $\Delta \longrightarrow G$ have a derivation of length l. Then one of the following is true:*

(i) *G is $\top$.*

(ii) *G is an atomic formula and either $G \equiv A$ for some A in $[\Delta]$ or for some $G' \supset A \in [\Delta]$ such that $G \equiv A$ it is the case that $\Delta \longrightarrow G'$ has a derivation of length less than l.*

(iii) *G is $G_1 \wedge G_2$ and there are derivations for $\Delta \longrightarrow G_1$ and $\Delta \longrightarrow G_2$ of length less than l.*

(iv) G *is* $G_1 \vee G_2$ *and there is a derivation for either* $\Delta \longrightarrow G_1$ *or* $\Delta \longrightarrow G_2$ *of length less than* l*.*

(v) G *is* $\exists x\, G_1$ *and for some closed positive term* T *it is the case that* $\Delta \longrightarrow [T/x]G_1$ *has a derivation of length less than* l*.*

Proof. We use an induction on the lengths of derivations. The lemma is obviously true in the case this length is 1: G is either $\top$ or G is atomic and $G \equiv A$ for some $A \in \Delta$. When the length is $l+1$, we consider the cases for the last inference figure. The argument is simple for the cases of $\wedge$-R, $\vee$-R and $\exists$-R. For the case of $\supset$-L, *i.e.*, when the last inference figure is of the form

$$\frac{\Delta' \longrightarrow G' \qquad\qquad A, \Delta' \longrightarrow G}{G' \supset A, \Delta' \longrightarrow G}$$

where $\Delta = \{G' \supset A\} \cup \Delta'$, the argument depends on the structure of G. If G is an atomic formula distinct from $\top$, the lemma follows from the hypothesis applied to $A, \Delta' \longrightarrow G$ except in the case when $G' \equiv A$. But in the latter case we see that $(G' \supset A) \in [\Delta]$ and a derivation for $\Delta \longrightarrow G'$ of length less than $l+1$ can be obtained from the one for $\Delta' \longrightarrow G'$ by adding $(G' \supset A)$ to the antecedent of every sequent in that derivation. If G is $G_1 \wedge G_2$, we see first that there must be derivations for $A, \Delta' \longrightarrow G_1$ and $A, \Delta' \longrightarrow G_2$ of smaller length than that for $A, \Delta' \longrightarrow G$. But using the derivation for $\Delta' \longrightarrow G'$ in conjunction with these we obtain derivations for $G' \supset A, \Delta' \longrightarrow G_1$ and $G' \supset A, \Delta' \longrightarrow G_2$ whose lengths must be less than $l+1$. Analogous arguments may be provided for the other cases for the structure of G. Finally a similar (and in some sense simpler) argument can be provided for the case when the last inference figure is a $\forall$-L.

$\square$

The equivalence of provability and the operational semantics defined in the previous section is the content of the following theorem.

Theorem 9 *If* Δ *is a program and* G *is a closed goal formula, then* $\Delta \vdash_C G$ *if and only if* $\Delta \vdash_O G$*.*

Proof. Using Lemma 8 and an induction on the length of a derivation it follows easily that $\Delta \vdash_O G$ if $\Delta \vdash_C G$. In the converse direction we use an induction on the length of the successful computation. If this length is 1, $\Delta \longrightarrow G$ must be an initial sequent. Consider now the case where G is an atomic formula that is solved by finding a $G' \supset A \in [\Delta]$ such that $G \equiv A$ and then solving G'. By the hypothesis, $\Delta \longrightarrow G'$ has a derivation as also does $A \longrightarrow G$. Using an $\supset$-L in conjunction with these, we get a derivation for $G' \supset A, \Delta \longrightarrow G$. Appending a sequence of $\forall$-L inference figures below this, we get a derivation for $\Delta \longrightarrow G$. The argument for the remaining cases is simpler and is left to the reader.

$\square$

6 Towards a Practical Realization

A practical realization of the programming paradigm that we have described thus far depends on the existence of an efficient procedure for determining whether a query succeeds or fails relative to a given program. The abstract interpreter that is described in Section 4 provides the skeleton for such a procedure. However, this interpreter is deficient in an important practical respect: it requires a prior knowledge of suitable instantiations for the existential quantifiers in a goal formula. The technique that is generally used in the first-order context for dealing with this problem is that of delaying the instantiations of such quantifiers till a time when information is available for making an appropriate choice. This effect is achieved by replacing the quantified variables by placeholders whose values are determined at a later stage through the process of unification. Thus, a goal formula of the form $\exists x\, G(x)$ is transformed into one of the form $G(X)$ where X is a new "logic" variable that may be instantiated at a subsequent point in the computation. The attempt to solve an atomic goal formula A involves looking for a definite clause $\forall \bar{y}\,(G' \supset A')$ such that A unifies with the atomic formula that results from A' by replacing the universally quantified variables with logic variables. Finding such a clause results in an instantiation of the logic variables in A and the next task becomes that of solving a suitable instance of G'.

The approach outlined above is applicable to the context of higher-order Horn clauses as well. The main difference is that we now have to consider the unification of λ-terms in a situation where equality between these terms is based on the rules of λ-conversion. This unification problem has been studied by several researchers and in most extensive detail by [Hue75]. In the first part of this section we expose those aspects of this problem and its solution that are pertinent to the construction of an actual interpreter for our language. We then introduce the notion of a $\mathcal{P}$-derivation as a generalization to the higher-order context of the SLD-derivations that are used relative to first-order Horn clauses [AvE82]. At one level, $\mathcal{P}$-derivations are syntactic objects for demonstrating success on a query and our discussions indicate their correctness from this perspective. At another level, they provide the basis for an actual interpreter for our programming paradigm: a symbol manipulating procedure that searches for $\mathcal{P}$-derivations would, in fact, be such an interpreter. Practicality requires that the ultimate procedure conduct such a search in a deterministic manner. Through our discussions here we expose those choices in search that play a critical role from the perspective of completeness and, in the final subsection, discuss ways in which these choices may be exercised by an actual interpreter.

6.1 The Higher-Order Unification Problem

Let us call a pair of terms of the same type a *disagreement pair*. A *disagreement set* is then a finite set, $\{\langle T_i, S_i\rangle \mid 1 \leq i \leq n\}$, of disagreement pairs, and a *unifier* for the set is a substitution θ such that, for $1 \leq i \leq n$, $\theta(T_i) = \theta(S_i)$. The *higher-order unification problem* can, in this context, be stated as the following: Given any disagreement set, to determine whether it has unifiers, and to explicitly provide a unifier if one exists.

The problem described above is a generalization of the well-known unification problem for first-order terms. The higher-order unification problem has certain properties that are different from those of the unification problem in the first-order case. For instance, the question of whether or not a unifier exists for an arbitrary disagreement set in the higher-order context is known to be undecidable [Gol81, Hue73b, Luc72]. Similarly, it has been shown that most general unifiers do not always exist for unifiable higher-order disagreement pairs [Gou76]. Despite these characteristics of the problem, a systematic search can be made for unifiers of a given disagreement set, and we discuss this aspect below.

Huet, in [Hue75], describes a procedure for determining the existence of unifiers for a given disagreement set and, when unifiers do exist, for enumerating some of them. This procedure utilizes the fact that there are certain disagreement sets for which at least one unifier can easily be provided and, similarly, there are other disagreement sets for which it is easily manifest that no unifiers can exist. Given an arbitrary disagreement set, the procedure attempts to reduce it to a disagreement set of one of these two kinds. This reduction proceeds by an iterative use of two simplifying functions, called SIMPL and MATCH, on disagreement sets. The basis for the first of these functions is provided by the following lemma whose proof may be found in [Hue75]. In this lemma, and in the rest of this section, we use the notation $\mathcal{U}(\mathcal{D})$ to denote the set of unifiers for a disagreement set $\mathcal{D}$.

Lemma 10 *Let $T_1 = \lambda\bar{x}\,(H_1\ A_1\ \ldots\ A_r)$ and $T_2 = \lambda\bar{x}\,(H_2\ B_1\ \ldots\ B_s)$ be two rigid terms of the same type that are in λ-normal form. Then $\theta \in \mathcal{U}(\{\langle T_1, T_2\rangle\})$ if and only if*

(i) $H_1 = H_2$ *(and, therefore, $r = s$), and*

(ii) $\theta \in \mathcal{U}(\{\langle \lambda\bar{x}\, A_i, \lambda\bar{x}\, B_i\rangle \mid 1 \le i \le r\})$.

Given any term T and any substitution θ, it is apparent that $\theta(T) = \theta(\lambda norm(T))$. Thus the question of unifying two terms can be reduced to unifying their λ-normal forms. Let us say that T is rigid (flexible) just in case $\lambda norm(T)$ is rigid (flexible), and let us refer to the arguments of $\lambda norm(T)$ as the arguments of T. If T_1 and T_2 are two terms of the same type, their λ-normal forms must have binders of the same length. Furthermore, we may, by a sequence of α-conversions, arrange their binders to be identical. If T_1 and T_2 are both rigid, then Lemma 10 provides us a means for either determining that T_1 and T_2 have no unifiers or reducing the problem of finding unifiers for T_1 and T_2 to that of finding unifiers for the arguments of these terms. This is, in fact, the nature of the simplification effected on a given unification problem by the function SIMPL.

Definition 8. The function SIMPL on sets of disagreement pairs is defined as follows:

(1) If $\mathcal{D} = \emptyset$ then SIMPL($\mathcal{D}$) $= \emptyset$.

(2) If $\mathcal{D} = \{\langle T_1, T_2\rangle\}$, then the forms of T_1 and T_2 are considered.

- (a) If T_1 is a flexible term then SIMPL($\mathcal{D}$) $= \mathcal{D}$.
- (b) If T_2 is a flexible term then SIMPL($\mathcal{D}$) $= \{\langle T_2, T_1\rangle\}$.
- (c) Otherwise T_1 and T_2 are both rigid terms. Let $\lambda\bar{x}\,(C_1\ A_1\ \ldots\ A_r)$ and $\lambda\bar{x}\,(C_2\ B_1\ \ldots\ B_s)$ be λ-normal forms for T_1 and T_2. If $C_1 \neq C_2$ then SIMPL($\mathcal{D}$) $= \mathbf{F}$; otherwise

$$\text{SIMPL}(\mathcal{D}) = \text{SIMPL}(\{\langle \lambda\bar{x}\, A_i, \lambda\bar{x}\, B_i\rangle \mid 1 \le i \le r\}).$$

(3) Otherwise $\mathcal{D}$ has at least two members. Let $\mathcal{D} = \{\langle T_1^i, T_2^i\rangle \mid 1 \le i \le n\}$.

- (a) If SIMPL($\{\langle T_1^i, T_2^i\rangle\}$) $= \mathbf{F}$ for some i then SIMPL($\mathcal{D}$) $= \mathbf{F}$;

(b) Otherwise $\mathrm{SIMPL}(\mathcal{D}) = \bigcup_{i=1}^{n} \mathrm{SIMPL}(\{\langle T_1^i, T_2^i \rangle\})$.

Clearly, SIMPL transforms a given disagreement set into either the marker **F** or a disagreement set consisting solely of "flexible-flexible" or "flexible-rigid" terms. By an abuse of terminology, we shall regard **F** as a disagreement set that has no unifiers. The intention, then, is that SIMPL transforms the given set into a simplified set that has the same unifiers. The lemma below that follows from the discussions in [Hue75] shows that SIMPL achieves this purpose in a finite number of steps.

Lemma 11 SIMPL *is a total computable function on sets of disagreement pairs. Further, if* $\mathcal{D}$ *is a set of disagreement pairs then* $\mathcal{U}(\mathcal{D}) = \mathcal{U}(\mathrm{SIMPL}(\mathcal{D}))$.

The first phase in the process of finding unifiers for a given disagreement set $\mathcal{D}$ thus consists of evaluating $\mathrm{SIMPL}(\mathcal{D})$. If the result of this is **F**, $\mathcal{D}$ has no unifiers. On the other hand, if the result is a set that is either empty or has only flexible-flexible pairs, at least one unifier can be provided easily for the set, as we shall see in the proof of Theorem 14. Such a set is, therefore, referred to as a *solved set.* If the set has at least one flexible-rigid pair, then a substitution needs to be considered for the head of the flexible term towards making the heads of the two terms in the pair identical. There are essentially two kinds of "elementary" substitutions that may be employed for this purpose. The first of these is one that makes the head of the flexible term "imitate" that of the rigid term. In the context of first-order terms this is, in fact, the only kind of substitution that needs to be considered. However, if the head of the flexible formula is a function variable, there is another possibility: one of the arguments of the flexible term can be "projected" into the head position in the hope that the head of the resulting term becomes identical to the head of the rigid one or can be made so by a subsequent substitution. There are, thus, a set of substitutions, each of which may be investigated separately as a component of a complete unifier. The purpose of the function MATCH that is defined below is to produce these substitutions.

Definition 9. Let $\mathcal{V}$ be a set of variables, let T_1 be a flexible term, let T_2 be a rigid term of the same type as T_1, and let $\lambda\bar{x}\,(F\ A_1\ \ldots\ A_r)$, and $\lambda\bar{x}\,(C\ B_1\ \ldots\ B_s)$ be λ-normal forms of T_1 and T_2. Further, let the type of F be $\sigma_1 \to \cdots \to \sigma_r \to \tau$, where τ is atomic and, for $1 \leq i \leq r$, let w_i be a variable of type σ_i. The functions IMIT, PROJ, and MATCH are then defined as follows:

(i) If C is a variable (appearing also in $\bar{x}$), then $\mathrm{IMIT}(T_1, T_2, \mathcal{V}) = \emptyset$; otherwise

$$\mathrm{IMIT}(T_1, T_2, \mathcal{V}) = \{\{\langle F, \lambda w_1, \ldots, w_r\,(C\ (H_1\ w_1\ \ldots\ w_r)\ \ldots\ (H_s\ w_1\ \ldots\ w_r))\rangle\}\},$$

where $H_1, \ldots, H_s$ are variables of appropriate types not contained in $\mathcal{V} \cup \{w_1, \ldots, w_r\}$.

(ii) For $1 \leq i \leq r$, if σ_i is not of the form $\tau_1 \to \ldots \to \tau_t \to \tau$ then $\mathrm{PROJ}_i(T_1, T_2, \mathcal{V}) = \emptyset$; otherwise,

$$\mathrm{PROJ}_i(T_1, T_2, \mathcal{V}) = \{\{\langle f, \lambda w_1, \ldots, w_r\,(w_i\ (H_1\ w_1\ \ldots\ w_r)\ \ldots\ (H_t\ w_1\ \ldots\ w_r))\rangle\}\},$$

where $H_1, \ldots, H_t$ are variables of appropriate types not contained in $\mathcal{V} \cup \{w_1, \ldots, w_r\}$.

(iii) $\mathrm{MATCH}(T_1, T_2, \mathcal{V}) = \mathrm{IMIT}(T_1, T_2, \mathcal{V}) \cup (\bigcup_{1 \leq i \leq r} \mathrm{PROJ}_i(T_1, T_2, \mathcal{V}))$.

The purpose of MATCH is to suggest a set of substitutions that may form "initial segments" of unifiers and, in this process, bring the search for a unifier closer to resolution. That MATCH achieves this effect is the content of the following lemma whose proof may be found in [Hue75] or [Nad87].

Lemma 12 *Let $\mathcal{V}$ be a set of variables, let T_1 be a flexible term and let T_2 be a rigid term of the same type as T_1. If there is a substitution $\theta \in \mathcal{U}(\{\langle T_1, T_2\rangle\})$ then there is a substitution $\varphi \in \mathrm{MATCH}(T_1, T_2, \mathcal{V})$ and a corresponding substitution θ' such that $\theta =_{\mathcal{V}} \theta' \circ \varphi$. Further, there is a mapping π from substitutions to natural numbers,* i.e., *a measure on substitutions, such that $\pi(\theta') < \pi(\theta)$.*

A unification procedure may now be described based on an iterative use of SIMPL and MATCH. The structure of this procedure is apparent from the above discussions and its correctness can be ascertained by using Lemmas 11 and 12. A procedure that searches for a $\mathcal{P}$-derivation, a notion that we describe next, actually embeds such a unification procedure within it.

6.2 $\mathcal{P}$-Derivations

Let the symbols $\mathcal{G}$, $\mathcal{D}$, θ and $\mathcal{V}$, perhaps with subscripts, denote sets of formulas of type o, disagreement sets, substitutions and sets of variables, respectively. The following definition describes the notion of one tuple of the form $\langle \mathcal{G}, \mathcal{D}, \theta, \mathcal{V}\rangle$ being derivable from another similar tuple relative to a program $\mathcal{P}$.

Definition 10. Let $\mathcal{P}$ be a program. We say a tuple $\langle \mathcal{G}_2, \mathcal{D}_2, \theta_2, \mathcal{V}_2\rangle$ is *$\mathcal{P}$-derivable* from another tuple $\langle \mathcal{G}_1, \mathcal{D}_1, \theta_1, \mathcal{V}_1\rangle$ if $\mathcal{D}_1 \neq \mathbf{F}$ and, in addition, one of the following situations holds:

(1) (*Goal reduction step*) $\theta_2 = \emptyset$, $\mathcal{D}_2 = \mathcal{D}_1$, and there is a goal formula $G \in \mathcal{G}_1$ such that

   (a) G is $\top$ and $\mathcal{G}_2 = \mathcal{G}_1 - \{G\}$ and $\mathcal{V}_2 = \mathcal{V}_1$, or

   (b) G is $G_1 \wedge G_2$ and $\mathcal{G}_2 = (\mathcal{G}_1 - \{G\}) \cup \{G_1, G_2\}$ and $\mathcal{V}_2 = \mathcal{V}_1$, or

   (c) G is $G_1 \vee G_2$ and, for $i = 1$ or $i = 2$, $\mathcal{G}_2 = (\mathcal{G}_1 - \{G\}) \cup \{G_i\}$ and $\mathcal{V}_2 = \mathcal{V}_1$, or

   (d) $G \equiv \exists x\, P$ and for some variable $y \notin \mathcal{V}_1$ it is the case that $\mathcal{V}_2 = \mathcal{V}_1 \cup \{y\}$ and $\mathcal{G}_2 = (\mathcal{G}_1 - \{G\}) \cup \{[y/x]P\}$.

(2) (*Backchaining step*) Let $G \in \mathcal{G}_1$ be a rigid positive atomic formula, and let $D \in \mathcal{P}$ be such that $D \equiv \forall \bar{x}\,(G' \supset A)$ for some sequence of variables $\bar{x}$ no member of which is in $\mathcal{V}_1$. Then $\theta_2 = \emptyset$, $\mathcal{V}_2 = \mathcal{V}_1 \cup \{\bar{x}\}$, $\mathcal{G}_2 = (\mathcal{G}_1 - \{G\}) \cup \{G'\}$, and $\mathcal{D}_2 = \mathrm{SIMPL}(\mathcal{D}_1 \cup \{\langle G, A\rangle\})$. (Here, $\{\bar{x}\}$ denotes the set of variables occurring in the list $\bar{x}$.)

(3) *(Unification step)* $\mathcal{D}_1$ is not a solved set and for some flexible-rigid pair $\langle T_1, T_2\rangle \in \mathcal{D}_1$, either $\mathrm{MATCH}(T_1, T_2, \mathcal{V}_1) = \emptyset$ and $\mathcal{D}_2 = \mathbf{F}$, or there is a $\varphi \in \mathrm{MATCH}(T_1, T_2, \mathcal{V}_1)$ and it is the case that $\theta_2 = \varphi$, $\mathcal{G}_2 = \varphi(\mathcal{G}_1)$, $\mathcal{D}_2 = \mathrm{SIMPL}(\varphi(\mathcal{D}_1))$, and, assuming φ is the substitution $\{\langle x, S\rangle\}$ for some variable x, $\mathcal{V}_2 = \mathcal{V}_1 \cup \mathcal{F}(S)$.

Let us call a finite set of goal formulas a *goal set*, and a disagreement set that is $\mathbf{F}$ or consists solely of pairs of positive terms a *positive disagreement set.* If $\mathcal{G}_1$ is a goal set and $\mathcal{D}_1$ is a positive disagreement set then it is clear, from Definitions 8, 9 and 10 and the fact that a positive term remains a positive term under a positive substitution, that $\mathcal{G}_2$ is a goal set and $\mathcal{D}_2$ a positive disagreement set for any tuple $\langle \mathcal{G}_2, \mathcal{D}_2, \theta_2, \mathcal{V}_2 \rangle$ that is $\mathcal{P}$-derivable from $\langle \mathcal{G}_1, \mathcal{D}_1, \theta_1, \mathcal{V}_1 \rangle$.

Definition 11. Let $\mathcal{G}$ be a goal set. Then we say that a sequence $\langle \mathcal{G}_i, \mathcal{D}_i, \theta_i, \mathcal{V}_i \rangle_{1 \le i \le n}$ is a $\mathcal{P}$-*derivation sequence* for $\mathcal{G}$ just in case $\mathcal{G}_1 = \mathcal{G}$, $\mathcal{V}_1 = \mathcal{F}(\mathcal{G}_1)$, $\mathcal{D}_1 = \emptyset$, $\theta_1 = \emptyset$, and, for $1 \le i < n$, $\langle \mathcal{G}_{i+1}, \mathcal{D}_{i+1}, \theta_{i+1}, \mathcal{V}_{i+1} \rangle$ is $\mathcal{P}$-derivable from $\langle \mathcal{G}_i, \mathcal{D}_i, \theta_i, \mathcal{V}_i \rangle$.

From our earlier observations it follows rather easily that, in a $\mathcal{P}$-derivation sequence for a goal set $\mathcal{G}$, each $\mathcal{G}_i$ is a goal set and each $\mathcal{D}_i$ is a positive disagreement set. We make implicit use of this fact in our discussions below. In particular, we intend unqualified uses of the symbols $\mathcal{G}$ and $\mathcal{D}$ to be read as syntactic variables for goal sets and positive disagreement sets, respectively.

Definition 12. A $\mathcal{P}$-derivation sequence $\langle \mathcal{G}_i, \mathcal{D}_i, \theta_i, \mathcal{V}_i \rangle_{1 \le i \le n}$ terminates, *i.e.*, is not contained in a longer sequence, if

(a) $\mathcal{G}_n$ is either empty or is a goal set consisting solely of flexible atoms and $\mathcal{D}_n$ is either empty or consists solely of flexible-flexible pairs, or

(b) $\mathcal{D}_n = \mathbf{F}$.

In the former case we say that it is a *successfully terminated* sequence. If this sequence also happens to be a $\mathcal{P}$-derivation sequence for $\mathcal{G}$, then we call it a $\mathcal{P}$-*derivation of* $\mathcal{G}$ and we say that $\theta_n \circ \cdots \circ \theta_1$ is its *answer substitution.*[2] If $\mathcal{G} = \{G\}$ then we also say that the sequence is a $\mathcal{P}$-derivation of G.

Example 4. Let $\mathcal{P}$ be the set of definite clauses defining the predicate *mapfun* in Example 2. Further, let G be the goal formula

$$(mapfun\ F_1\ (cons\ 1\ (cons\ 2\ nil))\ (cons\ (h\ 1\ 1)\ (cons\ (h\ 1\ 2)\ nil)))$$

in which F_1 is a variable and all other symbols are parameters as in Example 2. Then the tuple $\langle \mathcal{G}_1, \mathcal{D}_1, \emptyset, \mathcal{V}_1 \rangle$ is $\mathcal{P}$-derivable from $\langle \{G\}, \emptyset, \emptyset, \{F_1\} \rangle$ by a backchaining step, if

$$\begin{array}{rcl} \mathcal{V}_1 & = & \{F_1, F_2, L_1, L_2, X\}, \\ \mathcal{G}_1 & = & \{(mapfun\ F_2\ L_1\ L_2)\}, \quad \text{and} \\ \mathcal{D}_1 & = & \{\langle F_1, F_2 \rangle, \langle X, 1 \rangle, \langle (F_1\ X), (h\ 1\ 1) \rangle, \langle L_1, (cons\ 2\ nil) \rangle, \langle L_2, (cons\ (h\ 1\ 2)\ nil) \rangle\}, \end{array}$$

where F_2, L_1, L_2, and X are variables. Similarly, if

$$\begin{array}{rcl} \mathcal{V}_2 & = & \mathcal{V}_1 \cup \{H_1, H_2\}, \\ \mathcal{G}_2 & = & \{(mapfun\ F_2\ L_1\ L_2)\}, \\ \theta_2 & = & \{\langle F_1, \lambda w\,(h\ (H_1\ w)\ (H_2\ w)) \rangle\}, \quad \text{and} \\ \mathcal{D}_2 & = & \{\langle L_1, (cons\ 2\ nil) \rangle, \langle L_2, (cons\ (h\ 1\ 2)\ nil) \rangle, \langle X, 1 \rangle, \\ & & \langle (H_1\ X), 1 \rangle, \langle (H_2\ X), 1 \rangle, \langle F_2, \lambda w\,(h\ (H_1\ w)\ (H_2\ w)) \rangle\}, \end{array}$$

[2] This is somewhat distinct from what might be construed as the result of a computation. The latter is obtained by taking the the final goal and disagreement sets into account and restricting the substitution to the free variables in the original goal set. We discuss this matter in Subsection 6.3.

then the tuple $\langle \mathcal{G}_2, \mathcal{D}_2, \theta_2, \mathcal{V}_2 \rangle$ is $\mathcal{P}$-derivable from $\langle \mathcal{G}_1, \mathcal{D}_1, \emptyset, \mathcal{V}_1 \rangle$ by a unification step. (H_1, H_2 and w are additional variables here.) It is, in fact, obtained by picking the flexible-rigid pair $\langle (F_1\ X), (h\ 1\ 1) \rangle$ from $\mathcal{D}_1$ and using the substitution provided by IMIT for this pair. If the substitution provided by PROJ_1 was picked instead, we would obtain the tuple $\langle \mathcal{G}_2, \mathbf{F}, \{\langle F_1, \lambda w\, w \rangle\}, \mathcal{V}_1 \rangle$.

There are several $\mathcal{P}$-derivations of G, and all of them have the same answer substitution: $\{\langle F_1, \lambda w\, (h\ 1\ w) \rangle\}$.

Example 5. Let $\mathcal{P}$ be the program containing only the clause $\forall X\, (X \supset (p\ a))$, where X is a variable of type o and p and a are parameters of type $i \to o$ and i, respectively. Then, the following sequence of tuples constitutes a $\mathcal{P}$-derivation of $\exists Y\, (p\ Y)$:

$$\langle \{\exists Y\, (p\ Y)\}, \emptyset, \emptyset, \emptyset \rangle,\ \langle \{(p\ Y)\}, \emptyset, \emptyset, \{Y\} \rangle,\ \langle \{X\}, \{\langle Y, a \rangle\}, \emptyset, \{Y, X\} \rangle,\ \langle \{X\}, \emptyset, \{\langle Y, a \rangle\}, \{Y, X\} \rangle.$$

Notice that this is a successfully terminated sequence, even though the final goal set contains a flexible atom. We shall presently see that a goal set that contains only flexible atoms can be "solved" rather easily. In this particular case, for instance, the final goal set may be solved by applying the substitution $\{\langle X, \top \rangle\}$ to it.

A $\mathcal{P}$-derivation of a goal G is intended to show that G succeeds in the context of a program $\mathcal{P}$. The following lemma is useful in proving that $\mathcal{P}$-derivations are true to this intent. A proof of this lemma may be found in [Nad87] or [NM90]. The property of the $\mathcal{P}$-derivability relation that it states should be plausible at an intuitive level, given Lemma 11 and the success/failure semantics for goals.

Lemma 13 *Let $\langle \mathcal{G}_2, \mathcal{D}_2, \theta_2, \mathcal{V}_2 \rangle$ be $\mathcal{P}$-derivable from $\langle \mathcal{G}_1, \mathcal{D}_1, \theta_1, \mathcal{V}_1 \rangle$, and let $\mathcal{D}_2 \neq \mathbf{F}$. Further let $\varphi \in \mathcal{U}(\mathcal{D}_2)$ be a positive substitution such that $\mathcal{P} \vdash_O G$ for every G that is a closed positive instance of a formula in $\varphi(\mathcal{G}_2)$. Then*

(i) $\varphi \circ \theta_2 \in \mathcal{U}(\mathcal{D}_1)$, *and*

(ii) $\mathcal{P} \vdash_O G'$ *for every G' that is a closed positive instance of a formula in $\varphi \circ \theta_2(\mathcal{G}_1)$.*

We now show the correctness of $\mathcal{P}$-derivations. An interesting aspect of the proof of the theorem below is that it describes the construction of a substitution that simultaneously solves a set of flexible atoms and unifies a set of flexible-flexible disagreement pairs.

Theorem 14 (Soundness of $\mathcal{P}$-derivations) *Let $\langle \mathcal{G}_i, \mathcal{D}_i, \theta_i, \mathcal{V}_i \rangle_{1 \leq i \leq n}$ be a $\mathcal{P}$-derivation of G, and let θ be its answer substitution. Then there is a positive substitution φ such that*

(i) $\varphi \in \mathcal{U}(\mathcal{D}_n)$, *and*

(ii) $\mathcal{P} \vdash_O G'$ *for every G' that is a closed positive instances of the goal formulas in $\varphi(\mathcal{G}_n)$.*

Further, if φ is a positive substitution satisfying (i) and (ii), then $\mathcal{P} \vdash_O G''$ for every G'' that is a closed positive instance of $\varphi \circ \theta(G)$.

Proof. The second part of the theorem follows easily from Lemma 13 and a backward induction on i, the index of each tuple in the given $\mathcal{P}$-derivation sequence. For the first part we exhibit a substitution — that is a simple modification of the one in Lemma 3.5 in [Hue75] — and then show that it satisfies the requirements.

Let H_σ be a chosen variable for each atomic type σ. Then for each type τ we identify a term $\hat{E}_\tau$ in the following fashion:

(a) If τ is o, then $\hat{E}_\tau = \top$.

(b) If τ is an atomic type other than o, then $\hat{E}_\tau = H_\tau$.

(c) If τ is the function type $\tau_1 \to \cdots \to \tau_k \to \tau_0$ where τ_0 is an atomic type, then $\hat{E}_\tau = \lambda x_1 \ldots \lambda x_k\, \hat{E}_{\tau_0}$, where, for $1 \le i \le k$, x_i is a variable of type τ_i that is distinct from H_{τ_i}.

Now let $\varphi = \{\langle Y, \hat{E}_\tau\rangle \mid Y$ is a variable of type τ and $Y \in \mathcal{F}(\mathcal{G}_n) \cup \mathcal{F}(\mathcal{D}_n)\}$.

Any goal formula in $\mathcal{G}_n$ is of the form $(P\ S_1\ \ldots\ S_l)$ where P is a variable whose type is of the form $\sigma_1 \to \cdots \to \sigma_l \to o$. From this it is apparent that if $G \in \mathcal{G}_n$ then any ground instance of $\varphi(G)$ is identical to $\top$. Thus, it is clear that φ satisfies (ii). If $\mathcal{D}_n$ is empty then $\varphi \in \mathcal{U}(\mathcal{D}_n)$. Otherwise, let $\langle T_1, T_2\rangle \in \mathcal{D}_n$. Since T_1 and T_2 are two flexible terms, it may be seen that $\varphi(T_1)$ and $\varphi(T_2)$ are of the form $\lambda y_1^1 \ldots \lambda y_{m_1}^1\, \hat{E}_{\tau_1}$, and $\lambda y_1^2 \ldots \lambda y_{m_2}^2\, \hat{E}_{\tau_2}$ respectively, where τ_i is a primitive type and $\hat{E}_{\tau_i} \notin \{y_1^i, \ldots, y_{m_i}^i\}$ for $i = 1, 2$. Since T_1 and T_2 have the same types and substitution is a type preserving mapping, it is clear that $\tau_1 = \tau_2$, $m_1 = m_2$ and, for $1 \le i \le m_1$, y_i^1 and y_i^2 are variables of the same type. But then $\varphi(T_1) = \varphi(T_2)$.

□

We would like to show a converse to the above theorem that states that searching for a $\mathcal{P}$-derivation is adequate for determining whether a goal G succeeds or fails in the context of a program $\mathcal{P}$. The crux of such a theorem is showing that if the goals in the last tuple of a $\mathcal{P}$-derivation sequence succeed and the corresponding disagreement set has a unifier, then that sequence can be extended to a successfully terminated sequence. This is the content of the following lemma, a proof of which may be found in [Nad87] and [NM90]. The lemma should, in any case, be intuitively acceptable, given the success/failure semantics for goals and Lemmas 11 and 12.

Lemma 15 *Let $\langle \mathcal{G}_1, \mathcal{D}_1, \theta_1, \mathcal{V}_1\rangle$ be a tuple that is not a terminated $\mathcal{P}$-derivation sequence and for which $\mathcal{F}(\mathcal{G}_1) \cup \mathcal{F}(\mathcal{D}_1) \subseteq \mathcal{V}_1$. In addition, let there be a positive substitution $\varphi_1 \in \mathcal{U}(\mathcal{D}_1)$ such that, for each $G_1 \in \mathcal{G}_1$, $\varphi_1(G_1)$ is a closed goal formula and $\mathcal{P} \vdash_O \varphi_1(G_1)$. Then there is a tuple $\langle \mathcal{G}_2, \mathcal{D}_2, \theta_2, \mathcal{V}_2\rangle$ that is $\mathcal{P}$-derivable from $\langle \mathcal{G}_1, \mathcal{D}_1, \theta_1, \mathcal{V}_1\rangle$ and a positive substitution φ_2 such that*

(i) $\varphi_2 \in \mathcal{U}(\mathcal{D}_2)$,

(ii) $\varphi_1 =_{\mathcal{V}_1} \varphi_2 \circ \theta_2$,

(iii) for each $G_2 \in \mathcal{G}_2$, $\varphi_2(G_2)$ is a closed goal formula such that $\mathcal{P} \vdash_O \varphi_2(G_2)$.

Further, there is a mapping $\kappa_{\mathcal{P}}$ from goal sets and substitutions to ordinals, i.e., *a measure on pairs of goal sets and substitutions, relative to $\mathcal{P}$ such that $\kappa_{\mathcal{P}}(\mathcal{G}_2, \varphi_2) < \kappa_{\mathcal{P}}(\mathcal{G}_1, \varphi_1)$. Finally, when there are several tuples that are $\mathcal{P}$-derivable from $\langle \mathcal{G}_1, \mathcal{D}_1, \theta_1, \mathcal{V}_1\rangle$, such a tuple and substitution exist for every choice of (1) the kind of step, (2) the goal formula in a goal reduction or backchaining step, and (3) the flexible-rigid pair in a unification step.*

In using this lemma to prove the completeness of $\mathcal{P}$-derivations, we need the following observation that is obtained by a routine inspection of Definition 11.

Lemma 16 *Let $\langle \mathcal{G}_2, \mathcal{D}_2, \theta_2, \mathcal{V}_2 \rangle$ be $\mathcal{P}$-derivable from $\langle \mathcal{G}_1, \mathcal{D}_1, \theta_1, \mathcal{V}_1 \rangle$ and let $\mathcal{D}_2 \neq \mathbf{F}$. Then $\mathcal{V}_1 \subseteq \mathcal{V}_2$ and if $\mathcal{F}(\mathcal{G}_1) \cup \mathcal{F}(\mathcal{D}_1) \subseteq \mathcal{V}_1$, then $\mathcal{F}(\mathcal{G}_2) \cup \mathcal{F}(\mathcal{D}_2) \subseteq \mathcal{V}_2$.*

Theorem 17 (Completeness of $\mathcal{P}$-derivations) *Let φ be a closed positive substitution for the free variables of G such that $\mathcal{P} \vdash_O \varphi(G)$. Then there is a $\mathcal{P}$-derivation of G with an answer substitution θ such that $\varphi \preceq_{\mathcal{F}(G)} \theta$.*

Proof. From Lemmas 15 and 16 and the assumption of the theorem, it is evident that there is a $\mathcal{P}$-derivation sequence $\langle \mathcal{G}_i, \mathcal{D}_i, \theta_i, \mathcal{V}_i \rangle_{1 \leq i}$ for $\{G\}$ and a sequence of substitutions γ_i such that

(i) $\gamma_1 = \varphi$,

(ii) γ_{i+1} satisfies the equation $\gamma_i =_{\mathcal{V}_i} \gamma_{i+1} \circ \theta_{i+1}$,

(iii) $\gamma_i \in \mathcal{U}(\mathcal{D}_i)$, and

(iv) $\kappa_{\mathcal{P}}(\mathcal{G}_{i+1}, \gamma_{i+1}) < \kappa_{\mathcal{P}}(\mathcal{G}_i, \gamma_i)$.

From (iv) it follows that the sequence must be finite. From (iii) and Lemmas 11 and 12 it is evident, then, that it must be a successfully terminated sequence, *i.e.* a $\mathcal{P}$-derivation of G. Suppose that the length of the sequence is n. From (i), (ii), Lemma 16 and an induction on n, it can be seen that that $\varphi \preceq_{\mathcal{V}_1} \theta_n \circ \cdots \circ \theta_1$. But $\mathcal{F}(G) = \mathcal{V}_1$ and $\theta_n \circ \cdots \circ \theta_1$ is the answer substitution for the sequence.

□

6.3 Designing an Actual Interpreter

An interpreter for a language based on our higher-order formulas may be described as a procedure which, given a program $\mathcal{P}$ and a query G, attempts to construct a $\mathcal{P}$-derivation of G. The search space for such a procedure is characterized by a set of states each of which consists of a goal set and a disagreement set. The initial state in the search corresponds to the pair $\langle \{G\}, \emptyset \rangle$. At each stage in the search, the procedure is confronted with a state that it must attempt to simplify. The process of simplification involves either trying to find a solution to the unification problem posed by the disagreement set, or reducing the goal set to an empty set or a set of flexible atomic goal formulas. Given a particular state, there are several ways in which an attempt might be made to bring the search closer to a resolution. However, Definition 10 indicates that the possible choices are finite, and Theorem 17 assures us that if these are tried exhaustively then a solution is bound to be found if one exists at all. Further, a solution path may be augmented by substitutions from which an answer can be extracted in a manner akin to the first-order case.

An exhaustive search, while being necessary for completeness, is clearly an inappropriate basis for an interpreter for a programming language, and a trade-off needs to be made between completeness and practicality. From Lemma 15 we understand that certain choices can be made by the interpreter without adverse effects. However, the following choices are critical:

(i) the disjunct to be added to the goal set in a goal reduction step involving a disjunctive goal formula,

(ii) the definite clause to be used in a backchaining step, and

(iii) the substitution to be used in a unification step.

When such choices are encountered, the interpreter can, with an accompanying loss of completeness, explore them in a depth-first fashion with the possibility of backtracking. The best order in which to examine the options is a matter that is to be settled by experimentation. This issue has been explored in actual implementations of our language [BR92, MN88, EP91], and we comment briefly on the insights that have been gained from these implementations.

The description of the search space for the interpreter poses a natural first question: Should the unification problem be solved first or should a goal reduction step be attempted? A reasonable strategy in this context appears to be that of using a unification step whenever possible and attempting to reduce the goal set only after a solved disagreement set has been produced. There are two points that should be noted with regard to this strategy. First, an interpreter that always tries to solve an unsolved unification problem before looking at the goal set would function in a manner similar to standard Prolog interpreters which always solve the (straightforward) first-order unification problems first. Second, the attempt to solve a unification problem stops short of looking for unifiers for solved disagreement sets. The search for unifiers for such sets can be rather expensive [Hue75], and may be avoided by carrying the solved disagreement sets forward as constraints on the remaining search. In fact, it appears preferable not to look for unifiers for these sets even after the goal set has been reduced to a "solved" set. When the search reaches such a stage, the answer substitution and the final solved disagreement and goal sets may be produced as a response to the original query. From Theorem 14 we see that composing any substitution that is a unifier for this disagreement set and that also "solves" the corresponding goal set with the answer substitution produces a complete solution to the query and the response of the interpreter may be understood in this sense. In reality, the presentation of the answer substitution and the mentioned sets can be limited to only those components that have a bearing on the substitutions for the free variables in the original query since it is these that constitute the result of the computation.

In attempting to solve the unification problem corresponding to a disagreement set, a flexible-rigid pair in the set may be picked arbitrarily. Invoking MATCH on such a pair produces a set of substitutions in general. One of these needs to be picked to progress the search further, and the others must be retained as alternatives to backtrack to in case of a failure. Certain biases may be incorporated in choosing from the substitutions provided by MATCH, depending on the kinds of solutions that are desired first. For example, consider the unification problem posed by the disagreement set $\{\langle (F\ 2), (cons\ 2\ (cons\ 2\ nil))\rangle\}$ where F is a variable of type $int \rightarrow (list\ int)$ and the other symbols are as in Example 2. There are four unifiers for this set:

$$\{\langle F, \lambda x\,(cons\ x\ (cons\ x\ nil))\rangle\},\ \{\langle F, \lambda x\,(cons\ 2\ (cons\ x\ nil))\rangle\},$$
$$\{\langle F, \lambda x\,(cons\ x\ (cons\ 2\ nil))\rangle\}, \text{ and } \{\langle F, \lambda x\,(cons\ 2\ (cons\ 2\ nil))\rangle\}.$$

If the substitutions provided by PROJ_is are chosen first at each stage, then these unifiers will be produced in the order that they appear above, perhaps with the second and third interchanged. On the other hand, choosing the substitution provided by IMIT first results in these unifiers being produced in the reverse order. Now, the above unification problem may arise in a programming context out of the following kind of desire: We wish to unify the function variable F with the result of "abstracting" out all occurrences of a particular constant, which is 2 in this case, from a given data structure, which is an integer list here. If this is the desire, then it is clearly preferable to choose the PROJs substitutions before the substitution provided by IMIT. In a slightly more elaborate scheme, the user may be given a means for switching between these possibilities.

In attempting to solve a goal set, a nonflexible goal formula from the set may be picked arbitrarily. If the goal formula is either conjunctive or existential, then there is only one way in which it can be simplified. If the goal formula picked is $\{G_1 \vee G_2\}$ and the remaining goal set is $\mathcal{G}$, then, for the sake of completeness, the interpreter should try to solve $\{G_1\} \cup \mathcal{G}$ and $\{G_2\} \cup \mathcal{G}$ simultaneously. In practice, the interpreter may attempt to solve $\{G_1\} \cup \mathcal{G}$ first, returning to $\{G_2\} \cup \mathcal{G}$ only in case of failure. This approach, as the reader might notice, is in keeping with the one used in Prolog. In the case that the goal formula picked is atomic, a backchaining step must be used. In performing such a step, it is enough to consider only those definite clauses in the program of the form $\forall \bar{x}\, A$ or $\forall \bar{x}\, (G \supset A)$ where the head of A is identical to the head of the goal formula being solved, since all other cases will cause the disagreement set to be reduced to $\mathbf{F}$ by SIMPL. For completeness, it is necessary to use each of these definite clauses in a breadth-first fashion in attempting to solve the goal formula. Here again the scheme that is used by standard Prolog interpreters might be adopted: the first appropriate clause might be picked based on some predetermined ordering and others may be returned to only if this choice leads to failure.

The above discussion indicates how an interpreter can be constructed based on the notion of $\mathcal{P}$-derivations. An interpreter based on only these ideas would still be a fairly simplistic one. Several specific improvements (such as recognizing and handling special kinds of unification problems) and enhancements (such as those for dealing with polymorphism, a necessary practical addition) can be described to this interpreter. An examination of these aspects is beyond the scope of this chapter. For the interested reader, we mention that a further discussion of some of these aspects may be found in [Nad87], that a detailed presentation of a particular interpreter appears in [EP91], that a class of λ-terms with interesting unifiability properties is studied in [Mil91] and that ideas relevant to compilation are explored in [BR91, KNW94, NJK94, NJW93].

7 Examples of Higher-Order Programming

In this section, we present some programs in our higher-order language that use predicate variables and λ-terms of predicate type. We begin this discussion by describing a concrete syntax that will be used in the examples in this and later sections. We then present a set of simple programs that illustrate the idea of higher-order programming. In Subsection 7.3 we describe a higher-order logic programming approach to implementing goal-directed theorem proving based on the use of *tactics* and *tacticals*. We conclude this section with a brief comparison of the notion of higher-order programming in the logic programming and the functional programming settings.

7.1 A Concrete Syntax for Programs

The syntax that we shall use is adapted from that of the language λProlog. In the typical programming situation, it is necessary to identify three kinds of objects: the sorts and type constructors, the constants and variables with their associated types and the definite clauses that define the various predicate symbols. We present the devices for making these identifications below and simultaneously describe the syntax for expressions that use the objects so identified.

We assume that the two sorts `o` and `int` corresponding to propositions and integers and the unary list type constructor `list` are available in our language at the outset. This collection can be enhanced by declarations of the form

```
kind   <Id>   type -> ... -> type.
```

in which `<Id>` represents a token that is formed out of a sequence of alphanumeric characters and that begins with a lowercase letter. Such a declaration identifies `<Id>` as a type constructor whose arity is one less than the number of occurrences of `type` in the declaration. A declaration of this kind may also be used to add new sorts, given that these are, in fact, nullary type constructors. As specific examples, if `int` and `list` were not primitive to the language, then they might be added to the available collections by the declarations

```
kind   int    type.
kind   list   type -> type.
```

The sorts and type constructors available in a given programming context can be used as expected in forming type expressions. Such expressions might also use the constructor for function types that is written in concrete syntax as `->`. Thus, the type $int \rightarrow (list\ int) \rightarrow (list\ int)$ seen first in Example 1 would be written now as `int -> (list int) -> (list int)`.

The logical constants are rendered into concrete syntax as follows: $\top$ is represented by `true`, $\wedge$ and $\vee$ are denoted by the comma and semicolon respectively, implication is rendered into `:-` after being written in reverse order (*i.e.*, $G \supset A$ is denoted by `A :- G` where `A` and `G` are the respective translations of A and G), $\neg$ is not used, and $\forall$ and $\exists$ of type $(\tau \rightarrow o) \rightarrow o$ are represented by `pi` and `sigma` respectively. The last two constants are polymorphic in a sense that is explained below. To reduce the number of parentheses in expressions, we assume that conjunction and disjunction are right associative operators and that they have narrower scope than implication. Thus, the formula $(F \wedge (G \wedge H)) \supset A$ will be denoted by an expression of the form `A :- F,G,H`.

Nonlogical constants and variables are represented by tokens formed out of sequences of alphanumeric characters or sequences of "sign" characters. Symbols that consist solely of numeric characters are treated as nonlogical constants that have the type `int` associated with them. For other constants, a type association is achieved by a declaration of the form

```
type   <Id>   <Type>.
```

in which <Id> represents a constant and <Type> represents a type expression. As an example, if `i` has been defined to be a sort, then

```
type   f    (list i) -> i.
```

is a valid type declaration and results in `f` being identified as a constant of type `(list i) -> i`. The types of variables will be left implicit in our examples with the intention that they be inferred from the context.

It is sometimes convenient to identify a family of constants whose types are similar through *one* declaration. To accommodate this possibility, our concrete syntax allows for variables in type expressions. Such variables are denoted by capital letters. Thus, `A -> (list A) -> (list A)` is a valid type expression. A type declaration in which variables occur in the type is to be understood in the following fashion: It represents an infinite number of declarations each of which is obtained by substituting, in a uniform manner, closed types for the variables that occur in the type. For instance, the quantifier symbol `sigma` can be thought to be given by the type declaration

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

This declaration represents, amongst others, the declarations

```
type   sigma   (int -> o) -> o.
type   sigma   ((int -> int) -> o) -> o.
```

The tokens `sigma` that appear in each of these (virtual) declarations are, of course, distinct and might be thought to be subscripted by the type chosen in each case for the variable `A`. As another example, consider the declarations

```
type   nil    (list A).
type   ::     A -> (list A) -> (list A).
```

The symbols `nil` and `::` that are identified by them serve as polymorphic counterparts of the constants *nil* and *cons* of Example 2: using "instances" of `nil` and `::` that are of type `(list int)` and `int -> (list int) -> (list int)` respectively, it is possible to construct representations of lists of objects of type `int`. These two symbols are treated as pre-defined ones of our language and we further assume that `::` is an infix and right-associative operator.

The symbol `\` is used as an infix and right-associative operator that denotes abstraction and juxtaposition with some intervening white-space serves as an infix and left-associative operator that represents application. Parentheses may be omitted in expressions by assuming that abstractions bind more tightly than applications. Thus, the expressions

`x\x`, `(x\x Y\Y)` and `x\y\z\(x z (y z))`

denote, respectively, the terms $\lambda x\, x$, $((\lambda x\, x)\ (\lambda Y\, Y))$ and $\lambda x\, \lambda y\, \lambda z\, ((x\ z)\ (y\ z))$. Within formulas, tokens that are not explicitly bound by an abstraction are taken to be variables if they begin with an uppercase letter. A collection of definite clauses that constitute a program will be depicted as in Prolog by writing them in sequence, each clause being terminated by a period. Variables that are not given an explicit scope in any of these clauses are assumed to be universally quantified over the entire clause in which they appear. As an example, assuming that the declaration

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

gives the type of the predicate parameter `append`, the following definite clauses define the append relation of Prolog:

```
append nil L L.
(append (X::L1) L2 (X::L3)) :- (append L1 L2 L3).
```

Notice that, in contrast to Prolog and in keeping with the higher-order nature of our language, our syntax for formulas and terms is a curried one. We depict a query by writing a formula of the appropriate kind preceded by the token `?-` and followed by a period. As observed in Section 4, the free variables in a query are implicitly existentially quantified over it and represent a means for extracting a result from a computation. Thus, the expression

```
?- (append (1::2::nil) (3::4::nil) L).
```

represents a query that asks for the result of appending the two lists `1::2::nil` and `3::4::nil`.

In conjunction with the `append` example, we see that variables may occur in the types corresponding to the constants and variables that appear in a "definite clause". Such a clause is, again, to be thought of as a schema that represents an infinite set of definite clauses, each member of this set being obtained by substituting closed types for the type variables that occur in the schema. Such a substitution is, of course, to be constrained by the fact that the resulting instance must constitute a well-formed definite clause. Thus, consider the first of the definite clause "schemata" defining `append`. Writing the types of `nil`, `L` and `append` as `(list B)`, `C` and

```
    (list A) -> (list A) -> (list A) -> o,
```

respectively, we see that `C` must be instantiated by a type of the form `(list D)` and, further, that the same closed type must replace the variables `A`, `B` and `D`. Consistent with this viewpoint, the invocation of a clause such as this one must also be accompanied by a determination of the type instance to be used. In practice this choice can be delayed through unification at the level of types. This possibility is discussed in greater detail in [NP92] and we assume below that it is used in the process of solving queries.

A final comment concerns the inference of types for variables. Obvious requirements of this inference is that the overall term be judged to be well-formed and that every occurrence of a variable that is (implicitly or explicitly) bound by the same abstraction be accorded the same type. However, these requirements are, of themselves, not sufficiently constraining. For example, in the first definite clause for `append`, these requirements can be met by assigning the variable `L` the type `(list int)` or the type `(list D)` as indicated above. Conflicts of this kind are to be resolved by choosing a type for each variable that is most general in the sense that every other acceptable type is an instance of it. This additional condition can be sensibly imposed and it ensures that types can be inferred for variables in an unambiguous fashion [DM82].

```
type   mappred    (A -> B -> o) -> (list A) -> (list B) -> o.
type   forsome    (A -> o) -> (list A) -> o.
type   forevery   (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).

(forsome P (X::L)) :- (P X).
(forsome P (X::L)) :- (forsome P L).

(forevery P nil).
(forevery P (X::L)) :- (P X), (forevery 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 2: Definition of some higher-order predicates

7.2 Some Simple Higher-Order Programs

Five higher-order predicate constants are defined through the declarations in Figure 2. The intended meanings of these higher-order predicates are the following: A closed query of the form `(mappred P L K)` is to be solvable if `P` is a binary predicate, `L` and `K` are lists of equal length and corresponding elements of `L` and `K` are related by `P`; `mappred` is, thus, a polymorphic version of the predicate *mappred* of Example 2. A closed query of the form `(forsome P L)` is to be solvable if `L` is a list that contains at least one element that satisfies the predicate `P`. In contrast, a closed query of the form `(forevery P L)` is to be solvable only if *all* elements of `L` satisfy the predicate `P`. Assuming that `R` is a closed term denoting a binary predicate and `X` and `Y` are closed terms denoting two objects, the query `(trans R X Y)` is to be solvable just in case the objects given by `X` and `Y` are related by the transitive closure of the relation given by `R`; notice that the subterm `(trans R)` of this query is also a predicate. Finally, a closed query of the form `(sublist P L K)` is to be solvable if `L` is a sublist of `K` all of whose elements satisfy `P`.

Figure 3 contains some declarations defining a predicate called `age` that can be used in conjunction with these higher-order predicates. An interpreter for our language that is of the kind described in Section 6 will succeed on the query

```
?- mappred x\y\(age x y) (ned::bob::sue::nil) L.
```

relative to the clauses in Figures 2 and 3, and will have an answer substitution that binds `L` to the

```
kind   person   type.
type   bob      person.
type   sue      person.
type   ned      person.
type   age      person -> int -> o.

(age bob 24).
(age sue 23).
(age ned 23).
```

Figure 3: A database of people and their ages

list `(23::24::23::nil)`. This is, of course, the list of ages of the individuals `ned`, `bob` and `sue`, respectively. If the query

```
?- mappred x\y\(age y x) (23::24::nil) K.
```

is invoked relative to the same set of definite clauses, two substitutions for `K` can be returned as answers: `(sue::bob::nil)` and `(ned::bob::nil)`. Notice that within the form of higher-order programming being considered, non-determinism is supported naturally. Support is also provided in this context for the use of "partially determined" predicates, *i.e.*, predicate expressions that contain variables whose values are to be determined in the course of computation. The query

```
?- (forevery x\(age x A) (ned::bob::sue::nil)).
```

illustrates this feature. Solving this query requires determining if the predicate `x\(age x A)` is true of `ned`, `bob` and `sue`. Notice that this predicate has a variable `A` appearing in it and a binding for `A` will be determined in the course of computation, causing the predicate to become further specified. The given query will fail relative to the clauses in Figures 2 and 3 because the three individuals in the list do not have a common age. However, the query

```
?- (forevery x\(age x A) (ned::sue::nil)).
```

will succeed and will result in `A` being bound to `23`. The last two queries are to be contrasted with the query

```
?- (forevery x\(sigma Y\(age x Y)) (ned::bob::sue::nil)).
```

in which the predicate `x\(sigma Y\(age x Y))` is completely determined. This query will succeed relative to the clauses in Figures 2 and 3 since all the individuals in the list

```
    (ned::bob::sue::nil)
```

have an age defined for them.

An interpreter for our language that is based on the ideas in Section 6 solves a query by using a succession of goal reduction, backchaining and unification steps. None of these steps result in a flexible goal formula being selected. Flexible goal formulas remain in the goal set until substitutions performed on them in the course of computation make them rigid. This may never happen and the computation may terminate with such goal formulas being left in the goal set. Thus, consider the query

```
?- (P sue 23).
```

relative to the clauses in Figure 3. Our interpreter will succeed on this immediately because the only goal formula in its initial goal set is a flexible one. It is sensible to claim this to be a successful computation because there is at least one substitution — in particular, the substitution `x\y\true` for `P` — that makes this query a solvable one. It might be argued that there are meaningful answers for this query relative to the given program and that these should be provided by the interpreter. For example, it may appear that the binding of `P` to the term `x\y\(age x y)` (that is equal by virtue of η-conversion to `age`) should be returned as an answer to this query. However, many other similarly suitable terms can be offered as bindings for `P`; for example, consider the terms `x\y\(age ned 23)` and `x\y\((age x 23), (age ned y))`. There are, in fact, far too many "suitable" answer substitutions for this kind of a query for any reasonable interpreter to attempt to generate. It is for this reason that flexible goal formulas are never selected in the course of constructing a $\mathcal{P}$-derivation and that the ones that persist at the end are presented as such to the user along with the answer substitution and any remaining flexible-flexible pairs.

Despite these observations, flexible goal formulas can have a meaningful role to play in programs because the range of acceptable substitutions for predicate variables can be restricted by other clauses in the program. For example, while it is not sensible to ask for the substitutions for `R` that make the query

```
?- (R john mary).
```

a solvable one, a programmer can first describe a restricted collection of predicate terms and then ask if any of these predicates can be substituted for `R` to make the given query a satisfiable one. Thus, suppose that our program contains the definite clauses that are presented in Figure 4. Then, the query

```
?- (rel R), (R john mary).
```

is a meaningful one and is solvable only if the term

```
x\y\(sigma Z\((wife x Z), (mother Z y))).
```

is substituted for `R`. The second-order predicate `rel` specifies the collection of predicate terms that are relevant to consider as substitutions in this situation.

Our discussions pertaining to flexible queries have been based on a certain logical view of predicates and the structure of predicate terms. However, this is not the only tenable view. There is, in fact, an alternative viewpoint under which a query such as

```
?- (P sue 23).
```

can be considered meaningful and for which the only legitimate answer is the substitution of `age` for `P`. We refer the reader to [CKW93] for a presentation of a logic that justifies this viewpoint and for the description of a programming language that is based on this logic.

```
type   primrel   (person -> o) -> o.
type   rel       (person -> o) -> o.
type   mother    person -> o.
type   wife     person -> o.

(primrel mother).
(primrel wife).
(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 4: Restricting the range of predicate substitutions

7.3 Implementing Tactics and Tacticals

To provide another illustration of higher-order programming, we consider the task of implementing the *tactics* and *tacticals* that are often used in the context of (interactive) theorem proving systems. As described in [GMW79], a tactic is a primitive method for decomposing a goal into other goals whose achievement or satisfaction ensures the achievement or satisfaction of the original goal. A tactical is a high-level method for composing tactics into meaningful and large scale problem solvers. The functional programming language ML has often been used to implement tactics and tacticals. We show here that this task can also be carried out in a higher-order logic programming language. We use ideas from [FM88, Fel93] in this presentation.

The task that is at hand requires us, first of all, to describe an encoding within terms for the notion of a goal in the relevant theorem proving context. We might use `g` as a sort for expressions that encode such goals; this sort, that corresponds to "object-level" goals is to be distinguished from the sort `o` that corresponds to "meta-level" goals, *i.e.*, the queries of our programming language. The terms that denote primitive object-level goals will, in general, be determined by the problem domain being considered. For example, if the desire is to find proofs for formulas in first order logic, then the terms of type `g` will have to incorporate an encoding of such formulas. Alternatively, if it is sequents of first-order logic that have to be proved, then terms of type `g` should permit an encoding of sequents. Additional constructors over the type `g` might be included to support the encoding of compound goals. For instance, we will use below the constant `truegoal` of type `g` to denote the trivially satisfiable goal and the constant `andgoal` of type `g -> g -> g` to denote a goal formed out of the conjunction of two other goals. Other combinations such as the disjunction of two goals are also possible and can be encoded in a similar way.

A tactic in our view is a binary relation between a primitive goal and another goal, either compound or primitive. Thus tactics can be encoded by predicates of type `g -> g -> o`. Abstractly, if a tactic `R` holds of `G1` and `G2`, *i.e.*, if `(R G1 G2)` is solvable from a presentation of primitive tactics as a set of definite clauses, then satisfying the goal `G2` in the object-language should suffice to satisfy goal `G1`.

An illustration of these ideas can be provided by considering the task of implementing a proof procedure for propositional Horn clauses. For simplicity of presentation, we restrict the proposi-

```
type   p   g.
type   q   g.
type   r   g.
type   s   g.

type   cl1tac   g -> g -> o.
type   cl2tac   g -> g -> o.
type   cl3tac   g -> g -> o.
type   cl4tac   g -> g -> o.

(cl1tac p (andgoal r s)).
(cl2tac q r).
(cl3tac s (andgoal r q)).
(cl4tac r truegoal).
```

Figure 5: A tactic-style encoding of some propositional definite clauses

tional goal formulas that will be considered to be conjunctions of propositions. The objective will, of course, be to prove such formulas. Each primitive object-level goal therefore corresponds to showing some atomic proposition to be true, and such a goal might be encoded by a constant of type `g` whose name is identical to that of the proposition. Now, if `p` and `q` are two atomic propositions, then the goal of showing that their conjunction is true can be encoded in the object-level goal `(andgoal p q)`. The primitive method for reducing such goals is that of backchaining on (propositional) definite clauses. Thus, the tactics of interest will have the form `(R H G)`, where `H` represents the head of a clause and `G` is the goal corresponding to its body. The declarations in Figure 5 use these ideas to provide a tactic-style encoding of the four propositional clauses

```
p :- r,s.
q :- r.
s:- r,q.
r.
```

The tactics `cl1tac`, `cl2tac`, `cl3tac` and `cl4tac` correspond to each of these clauses respectively.

The declarations in Figure 6 serve to implement several general tacticals. Notice that tacticals are higher-order predicates in our context since they take tactics that are themselves predicates as arguments. The tacticals in Figure 6 are to be understood as follows. The `orelse` tactical is one that succeeds if either of the two tactics it is given can be used to successfully reduce the given goal. The `try` tactical forms the reflexive closure of a given tactic: if `R` is a tactic then `(try R)` is itself a tactic and, in fact, one that always succeeds, returning the original goal if it is unable to reduce it by means of `R`. This tactical uses an auxiliary tactic called `idtac` whose meaning is self-evident. The `then` tactical specifies the natural join of the two relations that are its arguments and is used to compose tactics: if `R1` and `R2` are closed terms denoting tactics, and `G1` and `G2` are closed terms representing (object-level) goals, then the query `(then R1 R2 G1 G2)` succeeds just in case the application of `R1` to `G1` produces `G3` and the application of `R2` to `G3` yields the goal `G2`.

```
type then        (g -> g -> o) -> (g -> g -> o) -> g -> g -> o.
type orelse      (g -> g -> o) -> (g -> g -> o) -> g -> g -> o.
type maptac      (g -> g -> o) -> g -> g -> o.
type repeat      (g -> g -> o) -> g -> g -> o.
type try         (g -> g -> o) -> g -> g -> o.
type complete    (g -> g -> o) -> g -> g -> o.
type idtac       g -> g -> o.
type goalreduce  g -> g -> o.

(orelse R1 R2 G1 G2) :- (R1 G1 G2).
(orelse R1 R2 G1 G2) :- (R2 G1 G2).

(try R G1 G2) :- (orelse R idtac G1 G2).

(idtac G G).

(then R1 R2 G1 G2) :- (R1 G1 G3), (maptac R2 G3 G2).

(maptac R truegoal truegoal).
(maptac R (andgoal G1 G2) (andgoal G3 G4)) :- (maptac R G1 G3), (maptac R G2 G4).
(maptac R G1 G2) :- (R G1 G2).

(repeat R G1 G2) :- (orelse (then R (repeat R)) idtac G1 G2).

(complete R G1 truegoal) :- (R G1 G2), (goalreduce G2 truegoal).

(goalreduce (andgoal truegoal G1) G2) :- (goalreduce G1 G2).
(goalreduce (andgoal G1 truegoal) G2) :- (goalreduce G1 G2).
(goalreduce G G).
```

Figure 6: Some simple tacticals

The `maptac` tactical is used in carrying out the application of the second tactic in this process since `G2` may be not be a primitive object-level goal: `maptac` maps a given tactic over all the primitive goals in a compound goal. The `then` tactical plays a fundamental role in combining the results of step-by-step goal reduction. The `repeat` tactical is defined recursively using `then`, `orelse`, and `idtac` and it repeatedly applies the tactic it is given until this tactic is no longer applicable. Finally, the `complete` tactical succeeds if the tactic it is given completely solves the goal it is given. The completely solved goal can be written as `truegoal` and `(andgoal truegoal truegoal)` and in several other ways, and so the auxiliary predicate `goalreduce` is needed to reduce all of these to `truegoal`. Although the `complete` tactical is the only one that uses the predicate `goalreduce`, the other tacticals can be modified so that they also use it to simplify the structure of the goal they produce whenever this is possible.

Tacticals, as mentioned earlier, can be used to combine tactics to produce large scale problem solvers. As an illustration of this, consider the following definite clause

```
(depthfirst G) :-
  (complete (repeat (orelse cl1tac (orelse cl2tac (orelse cl3tac cl4tac))))
            G truegoal).
```

in conjunction with the declarations in Figures 5 and 6. Assuming an interpreter for our language of the kind described in Section 6, this clause defines a procedure that attempts to solve an object-level goal by a depth-first search using the given propositional Horn clauses. The query

```
?- (depthfirst p).
```

has a successful derivation and it follows from this that the proposition `p` is a logical consequence of the given Horn clauses.

7.4 Comparison with Higher-Order Functional Programming

The examples of higher-order programming considered in this section have similarities to higher-order programming in the functional setting. In the latter context, higher-order programming corresponds to the treatment, perhaps in a limited fashion, of functions as values. For example, in a language like ML [MTH90], function expressions can be constructed using the abstraction operation, bound to variables, and applied to arguments. The semantics of such a language is usually extensional, and so intensional operations on functions like that of testing whether two function descriptions are identical are not supported by it. We have just seen that our higher-order logic programming language supports the ability to build predicate expressions, to bind these to variables, to apply them to arguments and, finally, to invoke the resulting expressions as queries. Thus, the higher-order capabilities present in functional programming languages are matched by ones available in our language. In the converse direction, we note that there are some completely new higher-order capabilities present in our language. In particular, this language is based on logic that has intensional characteristics and so can support computations on the *descriptions* of predicates and functions. This aspect is not exploited in the examples in this section but will be in those in the next.

In providing a more detailed comparison of the higher-order programming capabilities of our language that are based on an extensional view with those of functional programming languages, it is useful to consider the function `maplist`, the "canonical" higher-order function of functional programming. This function is given by the following equations:

```
(maplist f nil) = nil
(maplist f (x::l)) = (f x)::(maplist f l)
```

There is a close correspondence between `maplist` and the predicate `mappred` defined in Subsection 7.2. In particular, let $\mathcal{P}$ be a program, let `q` be a binary predicate and let `f` be a functional program such that `(q t s)` is provable from $\mathcal{P}$ if and only if `(f t)` evaluates to `s`; that is, the predicate `q` represents the function `f`. Further, let $\mathcal{P}'$ be $\mathcal{P}$ extended with the two clauses that define `mappred`. (We assume that no definite clauses defining `mappred` appear in $\mathcal{P}$.) Then, for any functional programming language that is reasonably pure, we can show that `(maplist f l)` evaluates to `k` if and only if `(mappred q l k)` is provable from $\mathcal{P}'$. Notice, however, that `mappred` is "richer" than `mapfun` in that its first argument can be a non-functional relation. In particular, if `q` denotes a non-functional relation, then `(mappred q)` is an acceptable predicate and itself denotes a non-functional relation. This aspect was illustrated earlier in this section through the query

```
?- (mappred x\y\(age y x) (23::24::nil) K).
```

In a similar vein, it is possible for a predicate expression to be partially specified in the sense that it contains logic variables that get instantiated in the course of using the expression in a computation. An illustration of this possibility was provided by the query

```
?- (forevery x\(age x A) (ned::sue::nil)).
```

These additional characteristics of higher-order logic programming are, of course, completely natural and expected.

8 Using λ-Terms as Data Structures

As noted in Section 2, the expressions that are permitted to appear as the arguments of atomic formulas in a logic programming language constitute the data structures of that language. The data structures of our higher-order language are, therefore, the terms of a λ-calculus. There are certain programming tasks that involve the manipulation of objects that embody a notion of binding; the implementation of theorem proving and program transformation systems that perform computations on quantified formulas and programs are examples of such tasks. Perspicuous and succinct representations of such objects can be provided by using λ-terms. The data structures of our language also incorporate a notion of equality based on λ-conversion and this provides useful support for logical operations that might have to be performed on the objects being represented. For example, following the discussion in Subsection 3.2, β-conversion can be used to realize the operation of substitution on these objects. In a similar sense, our language permits a quantification over function variables, leading thereby to the provision of higher-order unification as a primitive for taking apart λ-terms. The last feature is truly novel to our language and a natural outcome of considering the idea of higher-order programming in logic programming in its full generality: logic programming languages typically support intensional analyses of objects that can be represented in them, and the manipulation of λ-terms through higher-order unification corresponds to such examinations of functional expressions.

The various features of our language that are described above lead to several important applications for it in the realm of manipulating the syntactic objects of other languages such as formulas and programs. We refer to such applications as *meta-programming* ones and we provide illustrations of them in this section. The examples we present deal first with the manipulation of formulas and then with the manipulation of programs. Although our language has several features that are useful from this perspective, it is still lacking in certain respects as a language for meta-programming. We discussion this aspect in the last subsection below as a prelude to an extension of it that is considered in Section 9.

Before embarking on the main discussions of this section, it is useful to recapitulate the kinds of computations that can be performed on functional objects in our language. For this purpose, we consider the predicate `mapfun` that is defined by the following declarations.

```
type   mapfun   (A -> B) -> (list A) -> (list B) -> o.

(mapfun F nil nil).
(mapfun F (X::L) ((F X)::K)) :- (mapfun F L K).
```

This predicate is a polymorphic version of the predicate *mapfun* presented in Example 2 and it relates a term of functional type to two lists of equal length if the elements of the second list can be obtained by applying the functional term to the corresponding elements of the first list. The notion of function application that is relevant here is, of course, the one given by the λ-conversion rules. Thus, suppose that `h` is a nonlogical constant of type `int -> int -> int`. Then the answer to the query

```
?- (mapfun x\(h 1 x) (1::2::nil) L).
```

is one that entails the substitution of the term `((h 1 1)::(h 1 2)::nil)` for `L`. In computing this answer, an interpreter for our language would have to form the terms `((x\(h 1 x)) 1)` and ((x\(h 1 x)) 2) that may subsequently be simplified using the λ-conversion rules. As another example, consider the query

```
?- (mapfun F (1::2::nil) ((h 1 1)::(h 1 2)::nil)).
```

Any $\mathcal{P}$-derivation for this query would have an associated answer substitution that contains the pair ⟨F, x\(h 1 x)⟩. A depth-first interpreter for our language that is of the kind described in Section 6 would have to consider unifying the terms (F 1) and (h 1 1). There are four incomparable unifiers for these two terms and these are the ones that require substituting the terms

x\(h x x),   x\(h 1 x),   x\(h x 1),   and   x\(h 1 1)

respectively for F. Now, the terms (F 2) and (h 1 2) can be unified only under the second of these substitutions. Thus, if the interpreter selects a substitution for F distinct from the second one above, it will be forced to backtrack to reconsider this choice when it attempts to unify the latter two terms. As a final example, consider the query

```
?- (mapfun F (1::2::nil) (3::4::nil)).
```

This query does not succeed in the context of our language. The reason for this is that there is no λ-term of the kind used in our language that yields 3 when applied to 1 and 4 when applied to 2. Note that there are an infinite number of functions that map 1 to 3 and 2 to 4. However, none of these functions can be expressed by our our terms.

As a final introductory remark, we observe that it is necessary to distinguish in the discussions below between the programs and formulas that are being manipulated and the ones in our language that carry out thes manipulations. We do this by referring the former as object-level ones and the latter as ones of the meta-level.

8.1 Implementing an Interpreter for Horn Clauses

Formulas in a quantificational logic can be represented naturally within the terms of our language. The main concern in encoding these objects is that of capturing the nature of quantification. As noted in Subsection 3.1, the binding and predication aspects of a quantifier can be distinguished and the former can be handled by the operation of abstraction. Adopting this approach results in direct support for certain logical operations on formulas. For example, the equivalence of two formulas that differ only in the names used for their bound variables is mirrored directly in the equality of λ-terms by virtue of α-conversion. Similarly, the operation of instantiating a formula is realized easily through β-conversion. Thus, suppose the expression (all x\T) represents a universally quantified formula. Then the instantiation of this formula by a term represented by S is effected simply by writing the expression ((x\T) S); this term is equal, via β-conversion, to the term [S/x]T. The upshot of these observations is that a programmer using our language does not need to explicitly implement procedures for testing for alphabetic equivalence, for performing substitution or for carrying out other similar logical operations on formulas. The benefits of this are substantial since implementing such operations correctly is a non-trivial task.

We provide a specific illustration of the aspects discussed above by considering the problem of implementing an interpreter for the logic of first-order Horn clauses. The first problem to address here is the representation of object-level formulas and terms. We introduce the sorts form and term for this purpose; λ-terms of these two types will represent the objects in question. Object-level

```
kind   term   type.
kind   form   type.

type   false   form.
type   truth   form.
type   and     form -> form -> form.
type   or      form -> form -> form.
type   imp     form -> form -> form.
type   all     (term -> form) -> form.
type   some    (term -> form) -> form.

type   a      term.
type   b      term.
type   c      term.
type   f      term -> term.
type   path   term -> term -> form.
type   adj    term -> term -> form.
type   prog   form -> o.

prog (and (adj a b) (and (adj b c) (and (adj c (f 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: Specification of an object-level logic and some object-level definite clauses

```
type   interp      form -> form -> o.
type   backchain   form -> form -> form -> o.

(interp Cs (some B))  :- (interp Cs (B T)).
(interp Cs (and B C)) :- (interp Cs B), (interp Cs C).
(interp Cs (or B C))  :- (interp Cs B).
(interp Cs (or B C))  :- (interp Cs C).
(interp Cs A)         :- (backchain Cs Cs A).

(backchain Cs (and D1 D2) A) :- (backchain Cs D1 A).
(backchain Cs (and D1 D2) A) :- (backchain Cs D2 A).
(backchain Cs (all D) A)     :- (backchain Cs (D T) A).
(backchain Cs A A).
(backchain Cs (imp G A) A)   :- (interp Cs G).
```

Figure 8: An interpreter for first-order Horn clauses

logical and nonlogical constants, functions, and predicates will be represented by relevant meta-level nonlogical constants. Thus, suppose we wish to represent the following list of first-order definite clauses:

$adj(a, b)$,
$adj(b, c)$,
$adj(c, f(c))$,
$\forall x \forall y (adj(x, y) \supset path(x, y))$ and
$\forall x \forall y \forall z (adj(x, y) \wedge path(y, z) \supset path(x, z))$.

We might do this by using the declarations in Figure 7. In this program, a representation of a list of these clauses is eventually "stored" by using the (meta-level) predicate `prog`. Notice that universal and existential quantification at the object-level are encoded by using, respectively, the constants `all` and `some` of second-order type and that variable binding at the object-level is handled as expected by meta-level abstraction.

The clauses in Figure 8 implement an interpreter for the logic of first-order Horn clauses assuming the representation just described. Thus, given the clauses in Figures 7 and 8, the query

```
?- (prog Cs), (interp Cs (path a X)).
```

is solvable and produces three distinct answer substitutions, binding `X` to `b`, `c` and `(f c)`, respectively. Notice that object-level quantifiers need to be instantiated at two places in an interpreter of the kind we desire: in dealing with an existentially quantified goal formula and in generating an instance of a universally quantified definite clause. Both kinds of instantiation are realized in the clauses in Figure 8 through (meta-level) β-conversion and the specific instantiation is delayed in both cases through the use of a logic variable.

In understanding the advantages of our language, it is useful to consider the task of implementing an interpreter for first-order Horn clause logic in a pure first-order logic programming language.

```
kind   tm   type.

type   abs    (tm -> tm) -> tm.
type   app    tm -> tm -> tm.
type   eval   tm -> tm -> o.

(eval (abs R) (abs R)).
(eval (app M N) V) :- (eval M (abs R)), (eval N U), (eval (R U) V).
```

Figure 9: An encoding of untyped λ-terms and call-by-value evaluation

This task is a much more involved one since object-level quantification will have to be represented in a different way and substitution will have to be explicitly encoded in such a language. These problems are "finessed" in the usual meta-interpreters written in Prolog (such as those in [SS86]) by the use of non-logical features; in particular, by using the "predicate" `clause`.

8.2 Dealing with Functional Programs as Data

Our second example of meta-programming deals with the problem of representing functional programs and manipulating these as data. Some version of the λ-calculus usually underlies a functional programming language and, towards dealing with the representation issue, we first show how untyped λ-terms might be encoded in simply typed λ-terms. Figure 9 contains some declarations that are relevant in this context. These declarations identify the sort `tm` that corresponds to the encodings of object-level λ-terms and the two constants `abs` and `app` that serve to encode object-level abstraction and application. As illustrations of the representation scheme, the untyped λ-terms $\lambda x\, x$, $\lambda x\, \lambda y\, (x\ y)$, and $\lambda x\, (x\ x)$ are denoted by the simply typed λ-terms `(abs x\x)`, `(abs x\(abs y\(app x y)))`, and `(abs x\(app x x))`, respectively. A notion of evaluation based on β-reduction usually accompanies the untyped λ-terms. Such an evaluation mechanism is not directly available under the encoding. However, it can be realized through a collection definite clauses. The clauses defining the predicate `eval` in Figure 9 illustrate how this might be done. As a result of these clauses, this predicate relates two meta-level terms if the second encodes the result of evaluating the object-level term that is encoded by the first; the assumption in these clauses is that any "value" that is produced must be distinct from an (object-level) application. We note again that application and β-conversion at the meta-level are used in the second clause for `eval` to realize the needed substitution at the object-level of an actual argument for a formal one of a functional expression.

Most functional programming languages enhance the underlying λ-calculus by including a collection of predefined constants and functions. In encoding such constants a corresponding set of nonlogical constants might be used. For the purposes of the discussions below, we shall assume the collection presented in Figure 10 whose purpose is understood as follows:

(1) The constants `fixpt` and `cond` encode the fixed-point and conditional operators of the object-language being considered; these operators play an essential role in realizing recursive schemes.

(2) The constants `truth` and `false` represent the boolean values and `and` represents the binary operation of conjunction on booleans.

(3) The constant `c` when applied to an integer yields the encoding of that integer; thus `(c 0)` encodes the integer 0. The constants `+`, `*`, `-`, `<` and `=` represent the obvious binary operators on integers. Finally, the constant `intp` encodes a function that recognizes integers: `(intp e)` encodes an expression that evaluates to true if `e` is an encoding of an integer and to false otherwise.

(4) The constants `cons` and `nill` encode list constructors, `consp` and `null` encode recognizers for nonempty and empty lists, respectively, and the constants `car` and `cdr` represent the usual list destructors.

(5) The constant `pair` represents the pair constructor, `pairp` represents the pair recognizer, and `first` and `second` represent the obvious destructors of pairs.

(6) the constant `error` encodes the error value.

Each of the constants and functions whose encoding is described above has a predefined meaning that is used in the evaluation of expressions in the object-language. These meanings are usually clarified by a set of equations that are satisfied by them. For instance, if *fixpt* and *cond* are the fixed-point and conditional operators of the functional programming language being encoded and *true* and *false* are the boolean values, then the meanings of these operators might be given by the following equations:

$$\forall x\ ((fixpt\ x) = (x\ (fixpt\ x))) \quad \forall x \forall y\ ((cond\ true\ x\ y) = x) \quad \forall x \forall y\ ((cond\ false\ x\ y) = y)$$

The effect of such equations on evaluation can be captured in our encoding by augmenting the set of definite clauses for `eval` contained in Figure 9. For example, the definite clause

```
(eval (fixpt F) V) :- (eval (F (fixpt F)) V).
```

can be added to this collection to realize the effect of the first of the equations above. We do not describe a complete set of equations or an encoding of it here, but the interested reader can find extended discussions of this and other related issues in [HM92] and [Han93]. A point that should be noted about an encoding of the kind described here is that it reflects the effect of the object-language equations only in the evaluation relation and does not affect the equality relation of the meta-language. In particular, the notions of equality and unification for our typed λ-terms, even those containing the new nonlogical constants, are still governed only by the λ-conversion rules.

The set of nonlogical constants that we have described above suffices for representing several recursive functional programs as λ-terms. For example, consider a tail-recursive version of the factorial function that might be defined in a functional programming language as follows:

$$fact\ n\ m \ = \ (cond\ (n = 0)\ m\ (fact\ (n-1)\ (n * m))).$$

(The factorial of a non-negative integer n is to be obtained by evaluating the expression $(fact\ n\ 1)$.) The function $fact$ that is defined in this manner can be represented by the λ-term

```
(fixpt fact\(abs n\(abs m\(cond (= n (c 0)) m
                              (app (app fact (- n (c 1))) (* n m)))))).
```

```
type   cond    tm -> tm -> tm -> tm.
type   fixpt   (tm -> tm) -> tm.

type   truth   tm.
type   false   tm.
type   and     tm -> tm -> tm.

type   c      int -> tm.
type   +      tm -> tm -> tm.
type   -      tm -> tm -> tm.
type   *      tm -> tm -> tm.
type   <      tm -> tm -> tm.
type   =      tm -> tm -> tm.
type   intp   tm -> tm

type   nill    tm.
type   cons    tm -> tm -> tm.
type   null    tm -> tm.
type   consp   tm -> tm.
type   car     tm -> tm.
type   cdr     tm -> tm.

type   pair     tm -> tm -> tm.
type   pairp    tm -> tm.
type   first    tm -> tm.
type   second   tm -> tm.

type   error   tm.
```

Figure 10: Encodings for some constants and functions of a functional programming language

We assume below a representation of this kind for functional programs and we describe manipulations on these programs in terms of manipulations on their representation in our language.

As an example of manipulating programs, let us suppose that we wish to transform a recursive program that expects a single pair argument into a corresponding curried version. Thus, we would like to transform the program given by the term

```
(fixpt fact\(abs p\
 (cond (and (pairp p) (= (first p) (c 0)))
       (second p)
       (cond (pairp p) (app fact (pair (- (first p) (c 1))
                                       (* (first p) (second p))))
                     error))))
```

into the factorial program presented earlier. Let the argument of the given function be `p` as in the case above. If the desired transformer is implemented in a language such as Lisp, ML, or Prolog then it would have to be a recursive program that descends through the structure of the term representing the functional program, making sure that the occurrences of the bound variable `p` in it are within expressions of the form `(pairp p)`, `(first p)`, or `(second p)` and, in this case, replacing these expressions respectively by a true condition, and the first and second arguments of the version of the program being constructed. Although this does not happen with the program term displayed above, it is possible for this descent to enter a context in which the variable `p` is bound locally, and care must be exercised to ensure that occurrences of `p` in this context are not confused with the argument of the function. It is somewhat unfortunate that the *names* of bound variables have to be considered explicitly in this process since the choice of these names has no relevance to the meanings of programs. However, this concern is unavoidable if our program transformer is to be implemented in one of the languages under consideration since a proper understanding of bound variables is simply not embedded in them.

The availability of λ-terms and of higher-order unification in our language permits a rather different kind of solution to the given problem. In fact, the following (atomic) definite clause provides a concise description of the desired relationship between terms representing the curried and uncurried versions of recursive programs:

```
(curry (fixpt q1\(abs x\(A (first x) (second x) (pairp x)
                          (r\s\(app q1 (pair r s))))))
       (fixpt q2\(abs y\(abs z\(A y z truth q2))))).
```

The first argument of the predicate `curry` in this clause constitutes a "template" for programs of the type we wish to transform: For a term representing a functional program to unify with this term, it must have one argument that corresponds to `x`, every occurrence of this argument in the body of the program must be within an expression of the form `(first x)`, `(second x)` or `(pairp x)`, and every recursive call in the program must involve the formation of a pair argument. (The expression `r\s\(app q1 (pair r s))` represents a function of two arguments that applies `q1` to the pair formed from its arguments.) The recognition of the representation of a functional program as being of this form results in the higher-order variable `A` being bound to the result of extracting the expressions `(first x)`, `(second x)`, `(pairp x)`, and `r\s\(app q1 (pair r s))` from the term corresponding to the body of this program. The desired transformation can be effected merely by replacing the extracted expressions with the two arguments of the curried version, `truth` and a recursive call, respectively. Such a replacement is achieved by means of application and β-conversion in the second argument of curry in the clause above.

To illustrate the computation described abstractly above, let us consider solving the query

```
?- (curry (fixpt fact\(abs p\
            (cond (and (pairp p) (= (first p) (c 0)))
                  (second p)
                  (cond (pairp p) (app fact (pair (- (first p) (c 1))
                                                  (* (first p) (second p))))
                                  error))))
         NewProg).
```

using the given clause for curry. The described interpreter for our language would, first of all, instantiate the variable A with the term

```
u\v\p\q\(cond (and p (= u (c 0))) v (cond p (q (- u (c 1)) (* u v)) error).
```

(This instantiation for A is unique.) The variable NewProg will then be set to a term that is equal via λ-conversion to

```
(fixpt q2\y\z\(cond (and truth (= y (c 0))) z
                    (cond truth (q2 (- y (c 1)) (* y z)) error))).
```

Although this is not identical to the term we wished to produce, it can be reduced to that form by using simple identities pertaining to the boolean constants and operations of the function programming language. A program transformer that uses these identities to simplify functional programs can be written relatively easily.

As a more complex example of manipulating functional programs, we consider the task of recognizing programs that are tail-recursive; such a recognition step might be a prelude to transforming a given program into one in iterative form. The curried version of the factorial program is an example of a tail-recursive program as also is the program for summing two non-negative integers that is represented by the following λ-term:

```
(fixpt sum\(abs n\(abs m\(cond (= n (c 0)) m
                               (app (app sum (- n (c 1))) (+ m (c 1)))))))).
```

Now, the tail-recursiveness of these programs can easily be recognized by using higher-order unification in conjunction with their indicated representations. Both are, in fact, instances of the term

```
(fixpt f\(abs x\(abs y\(cond (C x y) (H x y) (app (app f (F1 x y)) (F2 x y))))))
```

Further, the representations of only tail-recursive programs are instances of the last term: Any closed term that unifies with the given "second-order template" must be a representation of a recursive program of two arguments whose body is a conditional and in which the only recursive call appears in the second branch of the conditional and, that too, as the head of the expression constituting that branch. Clearly any functional program that has such a structure must be tail-recursive. Notice that all the structural analysis that has to be performed in this recognition process is handled completely within higher-order unification.

Templates of the kind described above have been used by Huet and Lang [HL78] to describe certain transformations on recursive programs. Templates are, however, of limited applicability when used alone since they can only recognize restricted kinds of patterns. For instance, the template that is shown above for recognizing tail-recursiveness will not identify tail-recursive programs that contain more than one recursive call or that have more than one conditional in their body. An example of such a program is the one for finding the greatest common denominator of two numbers that is represented by the following λ-term:

```
(fixpt gcd\(abs x\(abs y\
  (cond (= (c 1) x) (c 1)
        (cond (= x y) x
              (cond (< x y) (app (app gcd y) x) (app (app gcd (- x y)) y)))))))
```

This program is tail-recursive but its representation is not an instance of the template presented above. Worse still, there is no second-order term all of whose closed instances represent tail-recursive programs and that also has the representations of the factorial, the sum and the greatest common denominator progams as its instances.

A recursive specification of a class of tail-recursive program terms that includes the representations of all of these programs can, however, be provided by using definite clauses in addition to second-order λ-terms and higher-order unification. This specification is based on the following observations:

(1) A program is obviously tail-recursive if it contains no recursive calls. The representation of such a program can be recognized by the template `(fixpt f\(abs x\(abs y\(H x y))))` in which `H` is a variable.

(2) A program that consists solely of a recursive call with possibly modified arguments is also tail-recursive. The second-order term `(fixpt f\(abs x\(abs y\(f (H x y) (G x y)))))` in which `H` and `G` are variables unifies with the representations of only such programs.

(3) Finally, a program is tail-recursive if its body consists of a conditional in which there is no recursive call in the test and whose left and right branches themselves satisfy the requirements of tail-recursiveness. Assuming that `C`, `H1` and `H2` are variables of appropriate type, the representations of only such programs unify with the λ-term

    ```
    (fixpt f\(abs x\(abs y\(cond (C x y) (H1 f x y) (H2 f x y)))))
    ```

    in a way such that `(fixpt H1)` and `(fixpt H2)` represent tail-recursive programs under the instantiations determined for `H1` and `H2`.

These observations can be translated immediately into the definition of a one place predicate that recognizes tail-recursive functional programs of two arguments and this is done in Figure 11. It is easily verified that all three tail-recursive programs considered in this section are recognized to be so by `tailrec`. The definition of `tailrec` can be augmented so that it also transforms programs that it recognizes to be tail-recursive into iterative ones in an imperative language. We refer the reader to [MN87] for details.

```
type   tailrec   tm -> o.

(tailrec (fixpt f\(abs x\(abs y\(H x y))))).
(tailrec (fixpt f\(abs x\(abs y\(f (H x y) (G x y)))))).
(tailrec (fixpt f\(abs x\(abs y\(cond (C x y) (H1 f x y) (H2 f x y)))) :-
     (tailrec (fixpt H1)), (tailrec (fixpt H2)).
```

Figure 11: A recognizer for tail-recursive functional programs of two arguments

8.3 A Shortcoming of Horn Clauses for Meta-Programming

Higher-order variables and an understanding of λ-conversion result in the presence of certain interesting meta-programming capabilities in our language. This language has a serious deficiency, however, with respect to this kind of programming that can be exposed by pushing our examples a bit further. We do this below.

We exhibited an encoding for a first-order logic in Subsection 8.1 and we presented an interpreter for the Horn clause fragment of this logic. A natural question to ask in this context is if we can define a (meta-level) predicate that recognizes the representations of first-order Horn clauses. It is an easy matter to write predicates that recognize first-order terms, atomic formulas, and quantifier-free Horn clauses. However, there is a problem when we attempt to deal with quantifiers. For example, when does a term of the form `(all B)` represent an object-level Horn clause? If we had chosen to represent formulas using only first-order terms, then universal quantification might have been represented by an expression of the form `forall(x,B)`, and we could conclude that such an expression corresponds to a definite clause just in case `B` corresponds to one. Under the representation that we actually use, `B` is a higher-order term and so we cannot adopt this simple strategy of "dropping" the quantifier. We might try to mimic it by instantiating the quantifier. But with what should we perform the instantiation? The ideal solution is to use a new constant, say `dummy`, of type `term`; thus, `(all B)` would be recognized as a definite clause if `(B dummy)` is a definite clause. The problem is that our language does not provide a logical way to generate such a new constant. An alternative is to use just any constant. This strategy will work in the present situation, but there is some arbitrariness to it and, in any case, there are other contexts in which the newness of the constant is critical.

A predicate that recognizes tail-recursive functional programs of two arguments was presented in Subsection 8.2. It is natural to ask if it is possible to recognize tail-recursive programs that have other arities. One apparent answer to this question is that we mimic the clauses in Figure 11 for each arity. Thus, we might add the clauses

```
(tailrec (fixpt f\(abs x\(H x)))).
(tailrec (fixpt f\(abs x\(f (H x) (G x))))).
(tailrec (fixpt f\(abs x\(cond (C x) (H1 f x) (H2 f x))))) :-
     (tailrec (fixpt H1)), (tailrec (fixpt H2)).
```

to the earlier program to obtain one that also recognizes tail-recursive functional programs of arity 1. However, this is not really a solution to our problem. What we desire is a *finite* set of clauses that can be used to recognize tail-recursive programs of *arbitrary* arity. If we maintain our encoding of functional programs, a solution to this problem seems to require that we descend through the abstractions corresponding to the arguments of a program in the term representing the program discharging each of these abstractions with a new constant, and that we examine the structure that results from this for tail-recursiveness. Once again we notice that our language does not provide us with a principled mechanism for creating the new constants that are needed in implementing this approach.

The examples above indicate a problem in defining predicates in our language for recognizing terms of certain kinds. This problem becomes more severe when we consider defining relationships between terms. For example, suppose we wish to define a binary predicate called `prenex` that relates the representations of two first-order formulas only if the second is a prenex normal form of the first. One observation useful in defining this predicate is that a formula of the form $\forall x\, B$ has $\forall x\, C$ as a prenex normal form just in case B has C as a prenex normal form. In implementing this observation, it is necessary, once again, to consider dropping a quantifier. Using the technique of substitution with a dummy constant to simulate this, we might translate our observation into the definite clause

```
(prenex (all B) (all C)) :- (prenex (B dummy) (C dummy)).
```

Unfortunately this clause does not capture our observation satisfactorily and describes a relation on formulas that has little to do with prenex normal forms. Thus, consider the query

```
?- (prenex (all x\(p x x)) E).
```

We expect that that the only answer to this query be the one that binds `E` to `(all x\(p x x))`, the prenex normal form of `(all x\(p x x))`. Using the clause for `prenex` above, the given goal would be reduced to

```
?- (prenex (p dummy dummy) (C dummy)).
```

with `E` being set to the term `(all C)`. This goal should succeed only if `(p dummy dummy)` and `(C dummy)` are equal. However, as is apparent from the discussions in Section 6, there are four substitutions for `C` that unify these two terms and only one of these yields an acceptable solution to the original query.

The crux of the problem that is highlighted by the examples above is that a language based exclusively on higher-order Horn clauses does not provide a principled way to generate new constants and, consequently, to descend under abstractions in λ-terms. The ability to carry out such a descent, and, thereby, to perform a general recursion over objects containing bound variables that are represented by our λ-terms, can be supported by enhancing our this language with certain new logical symbols. We consider such an enhancement in the next section.

9 Hereditary Harrop Formulas

The deficiency of our language that was discussed in Subsection 8.3 arises from the fact that higher-order Horn clauses provide no abstraction or scoping capabilities at the level of predicate logic to match those present at the level of terms. It is possible to extend Horn clause logic by allowing occurrences of implications and universal quantifiers within goal formulas, thereby producing the logic of *hereditary Harrop formulas* [MNPS91]. This enhancement to the logic leads to mechanisms for controlling the availability of the names and the clauses that appear in a program. It has been shown elsewhere that these additional capabilities can be used to realize notions such as modular programming, hypothetical reasoning, and abstract data types within logic programming [Mil89b, Mil90, NM88]. We show in this section that they can also be used to overcome the shortcoming of our present language from the perspective of meta-programming.

9.1 Permitting Universal Quantifiers and Implications in Goals

Let Σ be a set of nonlogical constants and let $\mathcal{K}_\Sigma$ denote the set of λ-normal terms that do not contain occurrences of the logical constants $\supset$ and $\bot$ or of nonlogical constants that are not elements of Σ; notice that the only logical constants that *are* allowed to appear in terms in $\mathcal{K}_\Sigma$ are $\top$, $\wedge$, $\vee$, $\forall$, and $\exists$. Further, let A and A_r be syntactic variables for atomic formulas and rigid atomic formulas in $\mathcal{K}_\Sigma$, respectively. Then the *higher-order hereditary Harrop* formulas and the corresponding goal formulas relative to Σ are the D- and G-formulas defined by the following mutually recursive syntax rules:

$$\begin{array}{rcl} D & ::= & A_r \mid G \supset A_r \mid \forall x\, D \mid D \wedge D \\ G & ::= & \top \mid A \mid G \wedge G \mid G \vee G \mid \forall x\, G \mid \exists x\, G \mid D \supset G. \end{array}$$

Quantification in these formulas, as in the context of higher-order Horn clauses, may be over function and predicate variables. When we use the formulas described here in programming, we shall think of a closed D-formula as a program clause relative to the signature Σ, a collection of such clauses as a program, and a G-formula as a query. There are considerations that determine exactly the definitions of the D- and G-formulas that are given here, and these are described in [Mil90, MNPS91].

An approach similar to that in Section 4 can be employed here as well in explaining what it means to solve a query relative to a program and what the result of a solution is to be. The main difference is that, in explaining how a closed goal formula is to be solved, we now have to deal with the additional possibilities for such formulas to contain universal quantifiers and implications. The attempt to solve a universally quantified goal formula can result in the addition of new nonlogical constants to an existing signature. It will therefore be necessary to consider generating instances of program clauses relative to different signatures. The following definition provides a method for doing this.

Definition 13. Let Σ be a set of nonlogical constants and let a closed positive Σ-*substitution* be one whose range is a set of closed terms contained in $\mathcal{K}_\Sigma$. Now, if D is a program clause, then the collection of its closed positive Σ-instances is denoted by $[D]_\Sigma$ and is given as follows:

(i) if D is of the form A or $G \supset A$, then it is $\{D\}$,

(ii) if D is of the form $D' \wedge D''$ then it is $[D']_\Sigma \cup [D'']_\Sigma$, and

(iii) if D is of the form $\forall x\, D'$ then it is $\bigcup\{[\varphi(D')]_\Sigma \mid \varphi$ is a closed positive Σ-substitution for $x\}$.

This notation is extended to programs as follows: if $\mathcal{P}$ is a program, $[\mathcal{P}]_\Sigma = \bigcup\{[D]_\Sigma \mid D \in \mathcal{P}\}$.

The attempt to solve an implicational goal formula will lead to an augmentation of an existing program. Thus, in describing an abstract interpreter for our new language, it is necessary to parameterize the solution of a goal formula by both a signature and a program. We do this in the following definition that modifies the definition of operational semantics contained in Definition 5 to suit our present language.

Definition 14. Let Σ be a set of nonlogical constants and let $\mathcal{P}$ and G be, respectively, a program and a goal formula relative to Σ. We then use the notation $\Sigma; \mathcal{P} \vdash_O G$ to signify that our abstract interpreter succeeds on G when given the signature Σ and the logic program $\mathcal{P}$. The success/failure behavior of the interpreter for hereditary Harrop formulas is itself specified as follows:

(i) $\Sigma; \mathcal{P} \vdash_O \top$,

(ii) $\Sigma; \mathcal{P} \vdash_O A$ where A is an atomic formula if and only if $A \equiv A'$ for some $A' \in [\mathcal{P}]_\Sigma$ or for some $(G \supset A') \in [\mathcal{P}]_\Sigma$ such that $A \equiv A'$ it is the case that $\Sigma; \mathcal{P} \vdash_O G$,

(iii) $\Sigma; \mathcal{P} \vdash_O G_1 \wedge G_2$ if and only if $\Sigma; \mathcal{P} \vdash_O G_1$ and $\Sigma; \mathcal{P} \vdash_O G_2$,

(iv) $\Sigma; \mathcal{P} \vdash_O G_1 \vee G_2$ if and only if $\Sigma; \mathcal{P} \vdash_O G_1$ or $\Sigma; \mathcal{P} \vdash_O G_2$,

(v) $\Sigma; \mathcal{P} \vdash_O \exists x\, G$ if and only if $\Sigma; \mathcal{P} \vdash_O [t/x]G$ for some term $t \in \mathcal{K}_\Sigma$ with the same type as x,

(vi) $\Sigma; \mathcal{P} \vdash_O D \supset G$ if and only if $\Sigma; \mathcal{P} \cup \{D\} \vdash_O G$, and

(vii) $\Sigma; \mathcal{P} \vdash_O \forall x\, G$ if and only if $\Sigma \cup \{c\}; \mathcal{P} \vdash_O [c/x]G$ where c is a nonlogical constant that is not already in Σ and that has the same type as x.

Let $\mathcal{P}$ be a program, let G be a closed goal formula and let Σ be any collection of nonlogical constants that includes those occurring in $\mathcal{P}$ and G. Using techniques similar to those in Section 5, it can then be shown that $\mathcal{P} \vdash_I G$ if and only if $\Sigma; \mathcal{P} \vdash_O G$; see [MNPS91] for details. Thus $\vdash_O$ coincides with $\vdash_I$ in the context of interest. It is easy to see, however, that $\vdash_C$ is a stronger relation than $\vdash_O$. For example, the goal formulas $p \vee (p \supset q)$ and

$$(((r\ a) \wedge (r\ b)) \supset q) \supset \exists x((r\ x) \supset q)$$

(in which p, q, r and a are parameters) are both provable in classical logic but they are not solvable in the above operational sense. (The signature remains unchanged in the attempt to solve both these goal formulas and hence can be elided.) The operational interpretation of hereditary Harrop formulas is based on a natural understanding of the notion of goal-directed search and we shall therefore think of the complementary logical interpretation of these formulas as being given by intuitionistic logic and not classical logic.

The main novelty from a programming perspective of a language based on hereditary Harrop formulas over one based on Horn clauses is the following: in the new language it is possible to restrict the scope of nonlogical constants and program clauses to selected parts of the search for a solution to a query. It is these scoping abilities that provide the basis for notions such as modular

```
type   term   term -> o.
type   atom   form -> o.

(term a).
(term b).
(term c).
(term (f X)) :- (term X).

(atom (path X Y)) :- (term X), (term Y).
(atom (adj X Y)) :- (term X), (term Y).
```

Figure 12: Recognizers for the encodings of object-language terms and atomic formulas

programming, hypothetical reasoning, and abstract data types in this language. As we shall see later in this section, these abilities also provide for logic-level abstractions that complement term-level abstractions.

A (deterministic) interpreter can be constructed for a language based on higher-order hereditary Harrop formulas in a fashion similar to that in the case of higher-order Horn clauses. There are, however, some differences in the two contexts that must be taken into account. First, the solution to a goal formula must be attempted in the context of a specific signature and program associated with that goal formula and not in a global context. Each goal formula must, therefore, carry a relevant signature and program. Second, the permitted substitutions for each logic variable that is introduced in the course of solving an existential query or instantiating a program clause are determined by the signature that is in existence at the time of introduction of the logic variable. It is therefore necessary to encode this signature in some fashion in the logic variable and to use it within the unification process to ensure that instantiations of the variable contain only the permitted nonlogical constants. Several different methods can be used to realize this requirement in practice, and some of these are described in [Mil89a, Mil92, Nad93]).

9.2 Recursion over Structures that Incorporate Binding

The ability to descend under abstractions in λ-terms is essential in describing a general recursion over representations of the kind described in Section 8 for objects containing bound variables. It is necessary to "discharge" the relevant abstractions in carrying out such a descent, and so the ability to generate new constants is crucial to this process. The possibility for universal quantifiers to appear in goal formulas with their associated semantics leads to such a capability being present in a language based on higher-order hereditary Harrop formulas. We show below that this suffices to overcome the problems outlined in Subsection 8.3 for a language based on higher-order Horn clauses.

We consider first the problem of recognizing the representations of first-order Horn clauses. Our solution to this problem assumes that the object-level logic has a fixed signature that is given by the type declarations in Figure 7. Using a knowledge of this signature, it is a simple matter to define predicates that recognize the representations of (object-level) terms and atomic formulas. The clauses for `term` and `atom` that appear in Figure 12 serve this purpose in the present context.

```
type   defcl   form -> o.
type   goal    form -> o.

(defcl (all C))   :- (pi x\((term x) => (defcl (C x)))).
(defcl (imp G A)) :- (atom A), (goal G).
(defcl A)         :- (atom A).

(goal truth).
(goal (and B C)) :- (goal B), (goal C).
(goal (or B C))  :- (goal B), (goal C).
(goal (some C))  :- (pi x\((term x) => (body (C x)))).
(goal A)         :- (atom A).
```

Figure 13: Recognizing representations of Horn clauses in a given first-order object-language

These predicates are used in Figure 13 in defining the predicate `defcl` and `goal` that are intended to be recognizers of encodings of object-language definite clauses and goal formulas respectively. We use the symbol `=>` to represent implications in (meta-level) goal formulas in the program clauses that appear in this figure. Recall that the symbol `pi` represents the universal quantifier.

In understanding the definitions presented in Figure 13, it is useful to focus on the first clause for `defcl` that appears in it. This clause is not an acceptable one in the Horn clause setting because an implication and a universal quantifier are used in its "body". An attempt to solve the query

```
?- (defcl (all x\(all y\(all z\(imp (and (adj x y) (path y z)) (path x z)))))).
```

will result in this clause being used. This will, in turn, result in the variable `C` that appears in the clause being instantiated to the term

```
x\(all y\(all z\(imp (and (adj x y) (path y z)) (path x z)))).
```

The way in which this λ-term is to be processed can be described as follows. First, a new constant must be picked to play the role of a name for the bound variable `x`. This constant must be added to the signature of the object-level logic, in turn requiring that the definition of the predicate `term` be extended. Finally, the λ-term must be applied to the new constant and it must be checked if the resulting term represents a Horn clause. Thus, if the new constant that is picked is `d`, then an attempt must be made to solve the goal formula

```
(defcl (all y\(all z\(imp (and (adj d y) (path y z)) (path d z)))))
```

after the program has been augmented with the clause `(term d)`. From the operational interpretation of implications and universal quantifiers described in Definition 14, it is easily seen that this is exactly the computation that is performed with the λ-term under consideration.

One of the virtues of our extended language is that it is relatively straightforward to verify formal properties of programs written in it. For example, consider the following property of the definition of `defcl`: If the query `(defcl (all B))` has a solution and if `T` is an object-level term (that is, `(term T)` is solvable), then the query `(defcl (B T))` has a solution, *i.e.*, the property of

```
type   quantfree   form -> o.

(quantfree false).
(quantfree truth).
(quantfree A)         :- (atom A).
(quantfree (and B C)) :- ((quantfree B), (quantfree C)).
(quantfree (or  B C)) :- ((quantfree B), (quantfree C)).
(quantfree (imp B C)) :- ((quantfree B), (quantfree C)).
```

Figure 14: Recognizing quantifier free formulas in a given object-language

being a Horn clause is maintained under first-order universal instantiation. This property can be seen to hold by the following argument: If the query `(defcl (all B))` is solvable, then it must be the case that the query `pi x\((term x) => (defcl (B x)))` is also solvable. Since `(term T)` is provable and since universal instantiation and modus ponens holds for intuitionistic logic and since the operational semantics of our language coincides with intuitionistic provability, we can conclude that the query `(defcl (B T))` has a solution. The reader will easily appreciate the fact that proofs of similar properties of programs written in other programming languages would be rather more involved than this.

Ideas similar to those above are used in Figure 15 in defining the predicate `prenex`. The declarations in this figure are assumed to build on those in Figure 7 and 12 that formalize the object-language and those in Figure 14 that define a predicate called `quantfree` that recognizes quantifier free formulas. The definition of `prenex` uses an auxiliary predicate called `merge` that raises the scope of quantifiers over the binary connectives. An interpreter for our language should succeed on the query

```
?- (prenex (or (all x\(and (adj x x) (and (all y\(path x y))
                                          (adj (f x) c))))
               (adj a b))
           Pnf).
```

relative to a program consisting of these various clauses and should produce the term

```
all x\(all y\(or (and (adj x x) (and (path x y) (adj (f x) c))) (adj a b)))
```

as the (unique) binding for `Pnf` when it does. The query

```
?- (prenex (and (all x\(adj x x)) (all z\(all y\(adj z y)))) Pnf).
```

is also solvable, but could result in `Pnf` being bound to any one of the terms

`all z\(all y\(and (adj z z) (adj z y)))`,
`all x\(all z\(all y\(and (adj x x) (adj z y))))`,
`all z\(all x\(and (adj x x) (adj z x)))`,
`all z\(all x\(all y\(and (adj x x) (adj z y))))`, and
`all z\(all y\(all x\(and (adj x x) (adj z y))))`.

```
type   prenex   form -> form -> o.
type   merge    form -> form -> o.

(prenex false false).
(prenex truth truth).
(prenex B B) :- (atom B).
(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 (imp B C) D)       :- (prenex B U), (prenex C V), (merge (imp U V) D).
(prenex (all B) (all D))   :- (pi x\((term x) => (prenex (B x) (D x)))).
(prenex (some B) (some D)) :- (pi x\((term x) => (prenex (B x) (D x)))).

(merge (and (all B) (all C)) (all D))   :-
         (pi x\((term x) => (merge (and (B x) (C x)) (D x)))).
(merge (and (all B) C) (all D))         :-
         (pi x\((term x) => (merge (and (B x) C) (D x)))).
(merge (and (some B) C) (some D))       :-
         (pi x\((term x) => (merge (and (B x) C) (D x)))).
(merge (and B (all C)) (all D))         :-
         (pi x\((term x) => (merge (and B (C x)) (D x)))).
(merge (and B (some C)) (some D))       :-
         (pi x\((term x) => (merge (and B (C x)) (D x)))).
(merge (or (some B) (some C)) (some D)) :-
         (pi x\((term x) => (merge (or (B x) (C x)) (D x)))).
(merge (or (all B) C) (all D))          :-
         (pi x\((term x) => (merge (or (B x) C) (D x)))).
(merge (or (some B) C) (some D))        :-
         (pi x\((term x) => (merge (or (B x) C) (D x)))).
(merge (or B (all C)) (all D))          :-
         (pi x\((term x) => (merge (or B (C x)) (D x)))).
(merge (or B (some C)) (some D))        :-
         (pi x\((term x) => (merge (or B (C x)) (D x)))).
(merge (imp (all B) (some C)) (some D)) :-
         (pi x\((term x) => (merge (imp (B x) (C x)) (D x)))).
(merge (imp (all B) C) (some D))        :-
         (pi x\((term x) => (merge (imp (B x) C) (D x)))).
(merge (imp (some B) C) (all D))        :-
         (pi x\((term x) => (merge (imp (B x) C) (D x)))).
(merge (imp B (all C)) (all D))         :-
         (pi x\((term x) => (merge (imp B (C x)) (D x)))).
(merge (imp B (some C)) (some D))       :-
         (pi x\((term x) => (merge (imp B (C x)) (D x)))).
(merge B B)                             :- (quantfree B).
```

Figure 15: Relating the representations of first-order formulas and their prenex normal forms

We now consider the task of defining a predicate that recognizes the representations of tail-recursive functions of arbitrary arity. We assume the same set of predefined constants and functions for our functional programming language here as we did in Subsection 8.2 and the encodings for these are given, once again, by the declarations in Figure 10. Now, our approach to recognizing the λ-terms of interest will, at an operational level, be the following: we shall descend through the abstractions in the representation of a program discharging each of them with a new constant and then check if the structure that results from this process satisfies the requirements of tail-recursiveness. The final test requires the constant that is introduced as the name of the function, *i.e.*, the constant used to instantiate the top-level abstracted variable of `M` in the expression `(fixpt M)`, to be distinguished from the other nonlogical constants. Thus, suppose that the expression that is produced by discharging all the abstractions is of the form `(cond C M N)`. For the overall λ-term to be the representation of a tail-recursive program, it must, first of all, be the case that the constant introduced as the name of the function does not appear in the term `C`. Further, each of `M` and `N` should be a λ-term that contains an occurrence of the constant introduced as the name of the function at most as its head symbol or that represents a conditional that recursively satisfies the requirements being discussed.

The required distinction between constants and the processing described above are reflected in the clauses in Figure 16 that eventually define the desired predicate `tailrec`. Viewed operationally, the clauses for the predicates `tailrec` and `trfn` realize a descent through the abstractions representing the name and arguments of a function. Notice, however, that the program is augmented differently in each of these cases: `headrec` is asserted to hold of the constant that is added in descending through the abstraction representing the name of the function whereas `term` is asserted to hold of the new constants introduced for the arguments. Now, the predicates `term` and `headrec` that are also defined by the clauses in this figure recognize two different classes of λ-terms representing functional programs: (i) those that are constructed using the original set of nonlogical constants and the ones introduced for the arguments of programs and (ii) those in which the constant introduced as the name of the program also appears, but only as the head symbol. The predicate `term` is defined by reflecting the various type declarations in Figure 10 into the meta-language. This definition will, of course, get augmented to realize any extension that occurs to the first category of terms through the addition of new nonlogical constants. The conditions under which the predicate `headrec` should be true of a term can be express as follows: the term must either be identical to the constant introduced as the name of the function or it must be of the form `(app M N)` where `headrec` is true of `M` and `term` is true of `N`. These conditions are obviously captured by the clause for `headrec` in Figure 16 and the one that is added in descending under the abstraction representing the name of the function. The final test for tail-recursiveness can be carried out easily using these predicates and this is, in fact, manifest in the definition of `trbody` in Figure 16.

Our last example in this section concerns the task of assigning types to expressions in our functional programming language. We shall, of course, be dealing with encodings of types and expressions of the object-language, and we need a representation for types to complement the one we already have for expressions. Figure 17 contains declarations that are useful for this purpose. The sort `ty` that is identified by these declarations is intended to be the meta-level type of terms that encode object-level types. We assume three primitive types at the object-level: those for booleans, natural numbers and lists of natural numbers. The nonlogical constants `boole`, `nat` and `natlist` serve to represent these. Further object-level types are constructed by using the function and pair type constructors and these are encoded by the binary constants `arrow` and `pairty`, respectively.

```
type   term      tm -> o.
type   tailrec   tm -> o.
type   trfn      tm -> o.
type   trbody    tm -> o.
type   headrec   tm -> o.

(term (abs R))       :- (pi x\((term x) => (term (R x)))).
(term (app M N))     :- (term M), (term N).
(term (cond M N P)) :- (term M), (term N), (term P).
(term (fixpt R))     :- (pi x\((term x) => (term (R x)))).
(term truth).
(term false).
(term (and M N))     :- (term M), (term N).
(term (c X)).
(term (+ M N))       :- (term M), (term N).
(term (- M N))       :- (term M), (term N).
(term (* M N))       :- (term M), (term N).
(term (< M N))       :- (term M), (term N).
(term (= M N))       :- (term M), (term N).
(term (intp M))      :- (term M).
(term nill).
(term (cons M N))    :- (term M), (term N).
(term (null M))      :- (term M).
(term (consp M))     :- (term M).
(term (car M))       :- (term M).
(term (cdr M))       :- (term M).
(term (pair M N))    :- (term M), (term N).
(term (pairp M))     :- (term M).
(term (first M))     :- (term M).
(term (second M))    :- (term M).
(term error).

(tailrec (fixpt M))     :- (pi f\((headrec f) => (trfn (M f)))).
(trfn (abs R))          :- (pi x\((term x) => (trfn (R x)))).
(trfn R)                :- (trbody R).
(trbody (cond M N P))  :- (term M), (trbody N), (trbody P).
(trbody M)              :- (term M); (headrec M).
(headrec (app M N))     :- (headrec M), (term N).
```

Figure 16: A recognizer for tail-recursive functional programs of arbitrary arity

The program clauses in Figure 17 define the predicate `typeof` that relates the representation of an expression in our functional programming language to the representation of its type. These clauses are, for the most part, self explanatory. We draw the attention of the reader to the first two clauses in this collection that are the only ones that deal with abstractions in the the representation of functional programs. Use is made in both cases of a combination of universal quantification and implication that is familiar by now in order to move the term-level abstraction into the meta-level. We observe, once again, that the manner of definition of the typing predicate in our language makes it easy to establish formal properties concerning it. For example, the following property can be shown to be true of `typeof` by using arguments similar to those used in the case of `defcl`: if the queries `(typeof (abs M) (arrow A B))` and `(typeof N A)` have solutions, then the query `(typeof (M N) B)` also has a solution.

We refer the reader to [Fel93, PM90, HM92, Mil91] for other, more extensive, illustrations of the value of a language based on higher-order hereditary Harrop formulas from the perspective of meta-programming. It is worth noting that all the example programs presented in this section as well as several others that are described in the literature fall within a sublanguage of this language called L_λ. This sublanguage, which is described in detail in [Mil91], has the computationally pleasant property that higher-order unification is decidable in its context and admits of most general unifiers.

```
kind   ty   type.

type   boole     ty.
type   nat       ty.
type   natlist   ty.
type   arrow     ty -> ty -> ty.
type   pairty    ty -> ty -> ty.

type   typeof    tm -> ty -> o.

(typeof (abs M) (arrow A B))     :- (pi x\((typeof x A) => (typeof (M x) B))).
(typeof (fixpt M) A)             :- (pi x\((typeof x A) => (typeof (M x) A))).
(typeof (app M N) B)             :- (typeof M (arrow A B)), (typeof N A).
(typeof (cond C L R) A)          :-
            (typeof C boole), (typeof L A), (typeof R A).
(typeof truth  boole).
(typeof false  boole).
(typeof (and M N)  boole)        :- (typeof M boole), (typeof N boole).
(typeof (c X) nat).
(typeof (+ M N) nat)             :- (typeof M nat), (typeof N nat).
(typeof (- M N) nat)             :- (typeof M nat), (typeof N nat).
(typeof (* M N) nat)             :- (typeof M nat), (typeof N nat).
(typeof (< M N) boole)           :- (typeof M nat), (typeof N nat).
(typeof (= M N) boole)           :- (typeof M nat), (typeof N nat).
(typeof (intp M) boole)          :- (typeof M A).
(typeof nill  natlist).
(typeof (cons M N) natlist)      :- (typeof M nat), (typeof N natlist).
(typeof (null M) boole)          :- (typeof M natlist).
(typeof (consp M) boole)         :- (typeof M natlist).
(typeof (car M) nat)             :- (typeof M natlist).
(typeof (cdr M) natlist)         :- (typeof M natlist).
(typeof (pair M N) (pairty A B)) :- (typeof M A), (typeof N B).
(typeof (pairp M) boole)         :- (typeof M A).
(typeof (first M) A)             :- (typeof M (pair A B)).
(typeof (second M) B)            :- (typeof M (pair A B)).
(typeof error A).
```

Figure 17: A predicate for typing functional programs

10 Conclusion

We have attempted to develop the notion of higher-order programming within logic programming in this chapter. A central concern in this endeavor has been to preserve the declarative style that is a hallmark of logic programming. Our approach has therefore been to identify an analogue of first-order Horn clauses in the context of a higher-order logic; this analogue must, of course, preserve the logical properties of the first-order formulas that are essential to their computational use while incorporating desirable higher-order features. This approach has lead to the description of the so-called higher-order Horn clauses in the simple theory of types, a higher-order logic that is based on a typed version of the lambda calculus. An actual use of these formulas in programming requires that a practically acceptable proof procedure exist for them. We have exhibited such a procedure by utilizing the logical properties of the formulas in conjunction with a procedure for unifying terms of the relevant typed λ-calculus. We have then examined the applications for a programming language that is based on these formulas. As initially desired, this language provides for the usual higher-order programming features within logic programming. This language also supports some unusual forms of higher-order programming: it permits λ-terms to be used in constructing the descriptions of syntactic objects such as programs and quantified formulas, and it allows computations to be performed on these descriptions by means of the λ-conversion rules and higher-order unification. These novel features have interesting uses in the realm of meta-programming and we have illustrated this fact in this chapter. A complete realization of these meta-programming capabilities, however, requires a language with a larger set of logical primitives than that obtained by using Horn clauses. These additional primitives are incorporated into the logic of hereditary Harrop formulas. We have described this logic here and have also outlined some of the several applications that a programming language based on this logic has in areas such as theorem proving, type inference, program transformation, and computational linguistics.

The discussions in this chapter reveal a considerable richness to the notion of higher-order logic programming. We note also that these discussions are not exhaustive. Work on this topic continues along several dimensions such as refinement, modification and extension of the language, implementation, and exploration of applications, especially in the realm of meta-programming.

11 Acknowledgements

We are grateful to Gilles Dowek for his comments on this chapter. Miller's work has been supported in part by the following grants: ARO DAAL03-89-0031, ONR N00014-93-1-1324, NSF CCR91-02753, and NSF CCR92-09224. Nadathur has similarly received support from the NSF grants CCR-89-05825 and CCR-92-08465.

References

[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.

[And89] Peter B. Andrews. On connections and higher-order logic. *Journal of Automated Reasoning*, 5(3):257–291, 1989.

[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.

[Bar81] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* North Holland Publishing Co., 1981.

[Ble79] W. W. Bledsoe. A maximal method for set variables in automatic theorem-proving. In *Machine Intelligence 9*, pages 53–100. John Wiley & Sons, 1979.

[BR91] Pascal Brisset and Olivier Ridoux. Naive reverse can be linear. In *Eighth International Logic Programming Conference*, Paris, France, June 1991. MIT Press.

[BR92] Pascal Brisset and Olivier Ridoux. The architecture of an implementation of λProlog: Prolog/Mali. In Dale Miller, editor, *Proceedings of the 1992 λProlog Workshop*, 1992.

[Chu40] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[CKW93] Weidong Chen, Michael Kifer, and David S. Warren. HiLog: A foundation for higher-order logic programming. *Journal of Logic Programming*, 15(3):187–230, February 1993.

[DM82] Luis Damas and Robin Milner. Principal type schemes for functional programs. In *Proceedings of the Ninth ACM Symposium on Principles of Programming Languages*, pages 207–212. ACM Press, 1982.

[EP91] Conal Elliott and Frank Pfenning. A semi-functional implementation of a higher-order logic programming language. In Peter Lee, editor, *Topics in Advanced Language Implementation*, pages 289 – 325. MIT Press, 1991.

[Fel93] Amy Felty. Implementing tactics and tacticals in a higher-order logic programming language. *Journal of Automated Reasoning*, 11(1):43–81, August 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.

[Gen69] Gerhard Gentzen. Investigations into logical deduction. In M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131. North Holland Publishing Co., 1969.

[GMW79] Michael J. Gordon, Arthur J. Milner, and Christopher P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.

[Gol81] Warren Goldfarb. The undecidability of the second-order unification problem. *Theoretical Computer Science*, 13:225 – 230, 1981.

[Gou76] W. E. Gould. A matching procedure for ω-order logic. Technical Report Scientific Report No. 4, A F C R L, 1976.

[GTL89] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and Types*. Cambridge University Press, 1989.

[Han93] John Hannan. Extended natural semantics. *Journal of Functional Programming*, 3(2):123 – 152, April 1993.

[Hen50] Leon Henkin. Completeness in the theory of types. *Journal of Symbolic Logic*, 15:81 – 91, 1950.

[HL78] Gérard Huet and Bernard Lang. Proving and applying program transformations expressed with second-order patterns. *Acta Informatica*, 11:31–55, 1978.

[HM92] John Hannan and Dale Miller. From operational semantics to abstract machines. *Mathematical Structures in Computer Science*, 2(4):415–459, 1992.

[Hue73a] Gérard Huet. A mechanization of type theory. In *Proceedings of the Third Interational Joint Conference on Articifical Intelligence*, pages 139–146, 1973.

[Hue73b] Gérard Huet. The undecidability of unification in third order logic. *Information and Control*, 22:257 – 267, 1973.

[Hue75] Gérard Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57, 1975.

[KNW94] Keehang Kwon, Gopalan Nadathur, and Debra Sue Wilson. Implementing polymorphic typing in a logic programming language. *Computer Languages*, 20(1):25–42, 1994.

[Luc72] C. L. Lucchesi. The undecidability of the unification problem for third order languages. Technical Report Report C S R R 2059, Department of Applied Analysis and Computer Science, University of Waterloo, 1972.

[Mil89a] Dale Miller. Lexical scoping as universal quantification. In *Sixth International Logic Programming Conference*, pages 268–283, Lisbon, Portugal, June 1989. MIT Press.

[Mil89b] Dale Miller. A logical analysis of modules in logic programming. *Journal of Logic Programming*, 6:79 – 108, 1989.

[Mil90] Dale Miller. Abstractions in logic programming. In Peirgiorgio 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.

[Mil92] 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.

[MN87] 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.

[MN88] Dale Miller and Gopalan Nadathur. λProlog Version 2.7. Distribution in C-Prolog and Quintus sources, July 1988.

[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.

[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, 1987.

[Nad93] Gopalan Nadathur. A proof procedure for the logic of hereditary Harrop formulas. *Journal of Automated Reasoning*, 11(1):115–145, August 1993.

[Nad94] Gopalan Nadathur. A notion of models for higher-order logic. Technical Report CS-1994-34, Department of Computer Science, Duke University, October 1994.

[NJK94] Gopalan Nadathur, Bharat Jayaraman, and Keehang Kwon. Scoping constructs in logic programming: Implementation problems and their solution. Technical Report CS-1994-35, Department of Computer Science, Duke University, October 1994. To appear in *Journal of Logic Programming*.

[NJW93] Gopalan Nadathur, Bharat Jayaraman, and Debra Sue Wilson. Implementation considerations for higher-order features in logic programming. Technical Report CS-1993-16, Department of Computer Science, Duke University, June 1993.

[NM88] Gopalan Nadathur and Dale Miller. An Overview of λProlog. In *Fifth International Logic Programming Conference*, pages 810–827, Seattle, Washington, August 1988. MIT Press.

[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.

[Pau90] Lawrence C. Paulson. Isabelle: The next 700 theorem provers. In Peirgiorgio Odifreddi, editor, *Logic and Computer Science*, pages 361 – 386. Academic Press, 1990.

[PM90] Remo Pareschi and Dale Miller. Extending definite clause grammars with scoping constructs. In David H. D. Warren and Peter Szeredi, editors, *Seventh International Conference in Logic Programming*, pages 373–389. MIT Press, June 1990.

[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, 1986.

[Wad91] William W. Wadge. Higher-order Horn logic programming. In *1991 International Symposium on Logic Programming*, pages 289–303. MIT Press, October 1991.

[War82] David H. D. Warren. Higher-order extensions to prolog: Are they needed? In *Machine Intelligence 10*, pages 441 – 454. Halsted Press, 1982.