ABSTRACTIONS IN LOGIC PROGRAMS

Dale Miller
Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104–6389 USA

February 1993

1. Introduction

Most logic programming languages have the first-order, classical theory of Horn clauses as their logical foundation. Purely proof-theoretical considerations show that Horn clauses are not rich enough to naturally provide the abstraction mechanisms that are common in most modern, general purpose programming languages. For example, Horn clauses do not incorporate the important software abstraction mechanisms of modules, data type abstractions, and higher-order programming.

As a result of this lack, implementers of logic programming languages based on Horn clauses generally add several nonlogical primitives on top of Horn clauses to provide these missing abstraction mechanisms. Although the missing features are often captured in this fashion, formal semantics of the resulting languages are often lacking or are very complex. Another approach to providing these missing features is to enrich the underlying logical foundation of logic programming. This latter approach to providing logic programs with these missing abstraction mechanisms is taken in this paper. The enrichments we will consider have simple and direct operational and proof theoretical semantics.

In Section 2, we present a first-order sorted logic and define sequential proof systems. These proof systems are used to define provability in classical and intuitionistic logics. In Section 3, we present first-order Horn clauses and describe certain aspects of their proof theory. In Section 4, Horn clauses are extended by permitting implications into the bodies of program clauses. The intuitionistic interpretation of clauses of this extension can provide a foundation for developing modular programming facilities for logic programs. In Sections 5 and 6, universal quantifiers, as well as implications, are added to the bodies of program clauses. The addition of this new form of quantification permits constants to be given local scope in the evaluation of logic programs. Such scoping of constants can be exploited to provide forms

To appear in the volume *Logic and Computer Science* edited by P. Odifreddi and published by Academic Press.

of data abstractions. This enrichment of Horn clauses, containing implications and universal quantifiers in the body of program clauses, is known as *hereditary Harrop formulas*.

A higher-order logic is presented in Section 7 and higher-order generalizations to Horn clauses and hereditary Harrop formulas are presented in Section 8. Several examples of higher-order logic programs are provided in Section 9. We conclude in Section 10.

This paper is essentially an overview and summary of the work the author and his colleagues have been engaged in over the past four years. All the main results and theorems reported in this paper have appeared in the papers [20, 21, 22, 25, 26, 27, 28].

2. A First-Order Logic

Let S be a fixed, finite set of *primitive types* (also called *sorts*). We assume that the symbol o is always a member of S. Following Church [4], o is the type for propositions. The set of *types* is the smallest set of expressions that contains the primitive types and is closed under the construction of function types, denoted by the binary, infix symbol $\rightarrow$. The Greek letters τ and σ are used as syntactic variables ranging over types. The type constructor $\rightarrow$ associates to the right: read $\tau_1 \rightarrow \tau_2 \rightarrow \tau_3$ as $\tau_1 \rightarrow (\tau_2 \rightarrow \tau_3)$.

Let τ be the type $\tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow \tau_0$ where $\tau_0 \in S$ and $n \geq 0$. (By convention, if $n = 0$ then τ is simply the type τ_0.) The types $\tau_1, \ldots, \tau_n$ are the *argument types of* τ while the type τ_0 is the *target type of* τ. The order of a type τ is defined as follows: If $\tau \in S$ then τ has order 0; otherwise, the order of τ is one greater than the maximum order of the argument types of τ. Thus, τ has order 1 exactly when τ is of the form $\tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow \tau_0$ where $n \geq 1$ and $\{\tau_0, \tau_1, \ldots, \tau_n\} \subseteq S$. We say, however, that τ *is a first-order type* if the order of τ is either 0 or 1 and that no argument type of τ is o. The target type of a first-order type may be o.

For each type τ, we assume that there are denumerably many constants and variables of that type. Constants and variables do not overlap and if two constants (or variables) have different types, they are different constants (or variables). A *signature (over S)* is a finite set Σ of constants. We often enumerate signatures by listing their members as pairs, written $a{:}\,\tau$, where a is a constant of type τ. Although attaching a type in this way is redundant, it makes reading signatures easier. A signature is *first-order* if all its constants are of first-order type.

We can now define the first-order logic $\mathcal{F}$. The *logical constants* of $\mathcal{F}$ are the symbols $\wedge$ (conjunction), $\vee$ (disjunction), $\supset$ (implication), $\top$ (truth), $\bot$ (absurdity), and for every $\tau \in S - \{o\}$, $\forall_\tau$ (universal quantification over type

τ), and $\exists_\tau$ (existential quantification over type τ). Thus, $\mathcal{F}$ has only a finite number of logical constants. Negation will not be of much interest in this paper, but when needed, the negation of a formula B is written as $B \supset \bot$.

Let τ be a type of the form $\tau_1 \to \cdots \to \tau_n \to \tau_0$ where τ_0 is a primitive type and $n \geq 0$. If τ_0 is o, a constant of type τ is a *predicate constant of arity* n. If τ_0 is not o, then a constant of type τ is either an *individual constant* if $n = 0$ or a *function constant of arity* n if $n \geq 1$. Similarly, we can define *predicate variable of arity* n, *individual variable*, and *function variable of arity* n.

Boldface letters are used for syntactic variables as follows: $\mathbf{a}, \mathbf{b}, \mathbf{c}$ range over individual constants; $\mathbf{x}, \mathbf{y}, \mathbf{z}$ range over individual variables; $\mathbf{f}, \mathbf{g}, \mathbf{h}$ range over function constants; and $\mathbf{p}, \mathbf{q}$ range over predicate constants. It is not until Section 6 that we are interested in function and predicate variables.

Let τ be a primitive type different from o. A *first-order term of type* τ is either a constant or variable of type τ, or of the form $(\mathbf{f}\ \mathbf{t}_1 \dots \mathbf{t}_n)$ where $\mathbf{f}$ is a function constant of type $\tau_1 \to \cdots \to \tau_n \to \tau$ and, for $i = 1, \dots, n$, $\mathbf{t}_i$ is a term of type τ_i. In the latter case, $\mathbf{f}$ is the *head* and $\mathbf{t}_1, \dots, \mathbf{t}_n$ are the *arguments* of this term.

A first-order formula is either *atomic* or *non-atomics*. An atomic formula is of the form $(\mathbf{p}\ \mathbf{t}_1 \dots \mathbf{t}_n)$, where $n \geq 0$, $\mathbf{p}$ is a predicate constant of the first-order type $\tau_1 \to \cdots \to \tau_n \to o$, and $\mathbf{t}_1, \dots, \mathbf{t}_n$ are first-order terms of the types $\tau_1, \dots, \tau_n$, respectively. The predicate constant $\mathbf{p}$ is the *head* of this atomic formula. Non-atomic formulas are of the form $\top$, $\bot$, $B_1 \wedge B_2$, $B_1 \vee B_2$, $B_1 \supset B_2$, $\forall_\tau \mathbf{x}\ B$, or $\exists_\tau \mathbf{x}\ B$, where B, B_1, and B_2 are formulas and τ is a primitive type different from o. The usual notions of free and bound variables and of open and closed terms and formulas are assumed.

The boldface letters $\mathbf{t}, \mathbf{s}$ range over terms; the roman letters B, C range over formulas; A ranges over atomic formulas; and the Greek letters Γ, Δ range over sets of formulas.

Let $\mathbf{s}$ be a first-order term of type τ and let $\mathbf{x}$ be a variable of type τ. The operation of *substituting* $\mathbf{s}$ *for free occurrences of* $\mathbf{x}$ is written as $[\mathbf{x} \mapsto \mathbf{s}]$. Bound variables are assumed to be changed in a systematic fashion in order to avoid variable capture. Simultaneous substitution is written as the operator $[\mathbf{x}_1 \mapsto \mathbf{s}_1, \dots, \mathbf{x}_n \mapsto \mathbf{s}_n]$.

Let Σ be a first-order signature. A Σ-*term* is a closed term all of whose constants are members of Σ. Likewise, a Σ-*formula* is a closed formula all of whose nonlogical constants are members of Σ.

Provability for $\mathcal{F}$ is given in terms of sequent calculus proofs [9]. A *sequent* of $\mathcal{F}$ is a triple $\Sigma\ ;\ \Gamma \longrightarrow \Delta$, where Σ is a first-order signature over S and Γ and Δ are finite (possibly empty) sets of Σ-formulas. The set Γ is this sequent's *antecedent* and Δ is its *succedent*. The expressions Γ, B

$$\frac{\Sigma\ ;\ \Gamma\ \longrightarrow\ \Delta, B \qquad \Sigma\ ;\ \Gamma\ \longrightarrow\ \Delta, C}{\Sigma\ ;\ \Gamma\ \longrightarrow\ \Delta, B \wedge C}\ \wedge\text{-R} \qquad \frac{\Sigma\ ;\ B, C, \Delta\ \longrightarrow\ \Theta}{\Sigma\ ;\ B \wedge C, \Delta\ \longrightarrow\ \Theta}\ \wedge\text{-L}$$

$$\frac{\Sigma\ ;\ B, \Delta\ \longrightarrow\ \Theta \qquad \Sigma\ ;\ C, \Delta\ \longrightarrow\ \Theta}{\Sigma\ ;\ B \vee C, \Delta\ \longrightarrow\ \Theta}\ \vee\text{-L}$$

$$\frac{\Sigma\ ;\ \Gamma\ \longrightarrow\ \Delta, B}{\Sigma\ ;\ \Gamma\ \longrightarrow\ \Delta, B \vee C}\ \vee\text{-R} \qquad \frac{\Sigma\ ;\ \Gamma\ \longrightarrow\ \Delta, C}{\Sigma\ ;\ \Gamma\ \longrightarrow\ \Delta, B \vee C}\ \vee\text{-R}$$

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

$$\frac{\Sigma\ ;\ \Gamma, [\mathbf{x} \mapsto \mathbf{t}]B\ \longrightarrow\ \Theta}{\Sigma\ ;\ \Gamma, \forall_\tau x\ B\ \longrightarrow\ \Theta}\ \forall\text{-L} \qquad \frac{\Sigma\ ;\ \Gamma\ \longrightarrow\ \Theta, [\mathbf{x} \mapsto \mathbf{t}]B}{\Sigma\ ;\ \Gamma\ \longrightarrow\ \Theta, \exists x\ B}\ \exists\text{-R}$$

$$\frac{\Sigma \cup \{\mathbf{c}\colon \tau\}\ ;\ \Gamma, [\mathbf{x} \mapsto \mathbf{c}]B\ \longrightarrow\ \Theta}{\Sigma\ ;\ \Gamma, \exists_\tau \mathbf{x}\ B\ \longrightarrow\ \Theta}\ \exists\text{-L} \qquad \frac{\Sigma \cup \{\mathbf{c}\colon \tau\}\ ;\ \Gamma\ \longrightarrow\ \Theta, [\mathbf{x} \mapsto \mathbf{c}]B}{\Sigma\ ;\ \Gamma\ \longrightarrow\ \Theta, \forall_\tau \mathbf{x}\ B}\ \forall\text{-R}$$

$$\frac{\Sigma\ ;\ \Gamma\ \longrightarrow\ \Theta, \bot}{\Sigma\ ;\ \Gamma\ \longrightarrow\ \Theta, B}\ \bot\text{-R}$$

Figure 1: Inference rules for $\mathcal{F}$

and B, Γ denote the set $\Gamma \cup \{B\}$; this notation is used even if $B \in \Gamma$. The inference rules for sequents are presented in Figure 1. The following provisos are also attached to the four inference rules for quantifier introduction: in $\forall$-R and $\exists$-L, the constant $\mathbf{c}$ is not in Σ, and, in $\forall$-L and $\exists$-R, $\mathbf{t}$ is a Σ-term of type τ.

A *proof* of the sequent $\Sigma\ ;\ \Gamma\ \longrightarrow\ \Theta$ is a finite tree constructed using these inference rules such that the root is labeled with $\Sigma\ ;\ \Gamma\ \longrightarrow\ \Theta$ and the leaves are labeled with *initial sequents*, that is, sequents $\Sigma'\ ;\ \Gamma'\ \longrightarrow\ \Theta'$ such that either $\top$ is a member of Θ' or the intersection $\Gamma' \cap \Theta'$ contains either $\bot$ or an atomic formula.

Sequent systems generally have three *structural* rules that are not listed here. Two such rules, interchange and contraction, are not necessary here because the antecedents and succedents of sequents are sets instead of lists. Hence, the order and multiplicity of formulas in sequents are not made explicit. The third common structural rule is that of thinning: from a given se-

quent one may add any additional formulas to the succedent and antecedent. Thinning could be added as a derived inference rule, but it is not needed in this paper.

Any proof is also called a **C**-*proof*. Any **C**-proof in which the succedent of every sequent in it is a singleton set is also called an **I**-*proof*. Furthermore, an **I**-proof in which no instance of the $\bot$-R inference rule appears is also called an **M**-*proof*. Sequent proofs in classical, intuitionistic, and minimal logics are represented by, respectively, **C**-proofs, **I**-proofs, and **M**-proofs. Finally, let Σ be a given first-order signature over S, let Γ be a finite set of Σ-formulas, and let B be a Σ-formula. We write $\Sigma; \Gamma \vdash_C B$, $\Sigma; \Gamma \vdash_I B$, and $\Sigma; \Gamma \vdash_M B$ if the sequent $\Sigma \,;\, \Gamma \longrightarrow B$ has, respectively, a **C**-proof, an **I**-proof, or an **M**-proof. It follows immediately that $\Sigma; \Gamma \vdash_M B$ implies $\Sigma; \Gamma \vdash_I B$, and this in turn implies $\Sigma; \Gamma \vdash_C B$.

The notions of provability defined here are not equivalent to the more usual presentations of classical, intuitionistic, and minimal logic [7, 9, 32, 36] in which signatures are not made explicit and substitution terms (the terms used in $\forall$-L and $\exists$-R) are not constrained to be taken from such signatures. The main reason they are not equivalent is illustrated by the following example. Let S be the set $\{i, o\}$ and consider the sequent

$$\{p\colon i \to o\} \,;\, \forall_i x\,(px) \;\longrightarrow\; \exists_i x\,(px).$$

This sequent has no proof even though $\exists_i x\,(px)$ follows from $\forall_i x\,(px)$ in the traditional presentations of classical, intuitutionistic, and minimal logics. The reason for this difference is that there are no $\{p\colon i \to o\}$-terms of type i: that is, the type i is *empty* in this signature. Thus we need an additional definition: the signature Σ *inhabits* the set of primitive types S if for every $\tau \in S$ different than o, there is a Σ-term of type τ. When Σ inhabits S, the notions of provability defined above coincide with the more traditional presentations.

Let $\mathcal{D}_0$ be the finite set of formulas that satisfy the inductive definition of D-formulas given by

$$D ::= A \mid B \supset A \mid \forall_\tau \mathbf{x}\; D \mid D_1 \wedge D_2,$$

where the syntactic variable B ranges over arbitrary first-order formulas. In this paper, first-order logic programs are always subsets of $\mathcal{D}_0$. In particular, the various first-order logic programming languages presented here are defined by simply restricting the set of formulas over which B is permitted to range. The formula B is called the *body* of the clause $B \supset A$.

Let Γ be a finite subset of $\mathcal{D}_0$. The set of formulas $|\Gamma|_\Sigma$ is defined to be the smallest subset of $\mathcal{D}_0$ such that

- $\Gamma \subseteq |\Gamma|_\Sigma$,
- if $D_1 \wedge D_2 \in |\Gamma|_\Sigma$ then $\{D_1, D_2\} \subseteq |\Gamma|_\Sigma$, and
- if $\forall_\tau \mathbf{x}\, D \in |\Gamma|_\Sigma$ and $\mathbf{t}$ is a Σ-term, then $[\mathbf{x} \mapsto \mathbf{t}]D \in |\Gamma|_\Sigma$.

If Γ is a set of Σ-formulas, then so is $|\Gamma|_\Sigma$.

Whenever the antecedent of a sequent is a subset of $\mathcal{D}_0$, it is convenient to introduce the *backchaining* inference rule, shown in Figure 2. The proviso for this rule is that $B \supset A \in |\Gamma|_\Sigma$. We also extend the class of initial formulas to include those sequents $\Sigma\ ;\ \Gamma \longrightarrow \Delta, A$ such that $A \in |\Gamma|_\Sigma$. Notice that any **M**-proof, **I**-proof, or **C**-proof containing the BC inference rule or this new kind of initial sequent can be directly converted to an **M**-proof, **I**-proof, or **C**-proof without these extensions by replacing them with repeated uses of the $\forall$-L, $\wedge$-L, and $\supset$-L inference rules.

$$\frac{\Sigma\ ;\ \Gamma \longrightarrow \Delta, B}{\Sigma\ ;\ \Gamma \longrightarrow \Delta, A}\ \text{BC}$$

Figure 2: The backchaining inference rule

The function *pred* associate to every formula in $\mathcal{D}_0$ a set of predicate constants. This function is defined by induction as follows: $pred(D_1 \wedge D_2) = pred(D_1) \cup pred(D_2)$; $pred(\forall_\tau \mathbf{x}\ D) = pred(D)$; $pred(B \supset A) = pred(A)$; and $pred(A)$, for atomic A, is the singleton set containing the predicate constant at the head of A.

3. First-order Horn clauses

Consider the two classes of first-order formulas defined by the following recursive definitions for the syntactic variables G and D:

$$G ::= \top \mid A \mid G_1 \wedge G_2 \mid G_1 \vee G_2 \mid \exists_\tau \mathbf{x}\, G$$
$$D ::= A \mid G \supset A \mid \forall_\tau \mathbf{x}\ D \mid D_1 \wedge D_2.$$

Any formula satisfying the recursion for G is called a *goal formula*, while any formula satisfying the recursion for D is called a *Horn clause*. Horn clauses are also called *definite formulas*. The classifications into goal and definite formulas are both extended to incorporate additional formulas in later sections. In the literature on the theory of Horn clauses, goal formulas generalize "negative" Horn clauses while what are called Horn clauses here generalize "positive" Horn clauses (see, for example, [3]). Finite sets of Horn clauses, denoted by the syntactic variable $\mathcal{P}$, are called *Horn clause programs*.

Assume that Σ is some fixed, first-order signature over S. Computations with Horn clauses can be specified as follows. Let $\mathbf{x}_1, \ldots, \mathbf{x}_n (n \geq 0)$ be a list of individual variables of primitive type $\tau_1, \ldots, \tau_n$, respectively. Let $\mathcal{P}$ be a finite set of Horn clauses that are also Σ-formulas. Let G be a formula whose nonlogical constants are in the signature Σ and all of whose free variables are in the list $\mathbf{x}_1, \ldots, \mathbf{x}_n$. A *result of solving G with respect to $\mathcal{P}$* is a list of Σ-terms $\mathbf{t}_1, \ldots, \mathbf{t}_n$ such that for $i = 1, \ldots, n$, $\mathbf{t}_i$ has type τ_i and the sequent

$$\Sigma \; ; \; \mathcal{P} \; \longrightarrow \; [\mathbf{x}_1 \mapsto \mathbf{t}_1, \ldots, \mathbf{x}_n \mapsto \mathbf{t}_n]G$$

has a **C**-proof. Clearly, there may be no such result or many, including infinitely many.

One way to understand how Horn clauses can be used in computations is to examine the structure of proofs involving Horn clauses and goal formulas. The following theorem provides a computationally useful characterization of such proofs.

Theorem 1. *Let Σ be a first-order signature over S that also inhabits S, let $\mathcal{P}$ be a Horn clause program, and let G be a goal formula. Also assume that $\mathcal{P} \cup \{G\}$ is a set of Σ-formulas. Then, $\Sigma; \mathcal{P} \vdash_C G$ if and only if the sequent $\Sigma \; ; \; \mathcal{P} \; \longrightarrow \; G$ has a proof Ξ satisfying the following three conditions.*

(i) The succedents of all sequents occurring in Ξ contain just a single formula.

(ii) If a sequent occurrence in Ξ has a non-atomic formula in its succedent, that sequent occurrence is the conclusion of the inference rule that introduces the logical connective into the succedent.

(iii) If a sequent occurrence in Ξ has an atomic formula in its succedent, that sequent occurrence is either initial or is the conclusion of the BC inference rule.

This characterization of proofs is intentionally abstract since we shall use it in situations that extend Horn clauses. The following statements are all immediate conclusions of this theorem.

- All sequents occurring in Ξ are of the form $\Sigma \; ; \; \mathcal{P} \; \longrightarrow \; G'$, where G' is some goal formula. Thus, in proving a goal formula, different signatures and antecedents (programs) need not be considered.
- $\Sigma; \mathcal{P} \vdash_C G$ if and only if $\Sigma; \mathcal{P} \vdash_I G$ if and only if $\Sigma; \mathcal{P} \vdash_M G$.
- $\Sigma; \mathcal{P} \vdash_C G_1 \wedge G_2$ if and only if $\Sigma; \mathcal{P} \vdash_C G_1$ and $\Sigma; \mathcal{P} \vdash_C G_2$.
- $\Sigma; \mathcal{P} \vdash_C G_1 \vee G_2$ if and only if $\Sigma; \mathcal{P} \vdash_C G_1$ or $\Sigma; \mathcal{P} \vdash_C G_2$.
- $\Sigma; \mathcal{P} \vdash_C \exists_\tau \mathbf{x}\, G$ if and only if there is a Σ-term $\mathbf{t}$ such that $\Sigma; \mathcal{P} \vdash_C [\mathbf{x} \mapsto \mathbf{t}]G$.

Any proof that satisfies the three conditions of Theorem 1 is called a *uniform proof*. The fact that uniform proofs are complete for the classical

theory of Horn clauses makes it possible to implement very simple theorem provers (also called interpreters) to do computations involving Horn clauses. In particular, the search for uniform proofs can be done in a goal-directed manner. That is, if a goal (the succedent of the sequent) is non-atomic, then the top-level logical connective of the goal determines which inference rule can be used to prove that sequent. For example, an attempt to prove a conjunction means that the interpreter should attempt an AND search to prove both conjuncts; to prove a disjunction means that the interpreter should attempt an OR search to prove either disjunct; and to prove an existential means that the interpreter should attempt to find a substitution instance. If the sequent is $\Sigma\,;\,\mathcal{P} \longrightarrow A$, where A is atomic, the program $\mathcal{P}$ must be consulted via the backchaining inference rule. The set $|\mathcal{P}|_\Sigma$ must contain some member, say D, such that $\mathit{pred}(D)$ is the set containing only the predicate that is the head of A. If no such D exists, then there is no proof of this sequent. Otherwise, D can be taken to be either of the form A, in which case, the sequent is proved immediately, or of the form $G' \supset A$, in which case, this sequent is provable if the sequent $\Sigma\,;\,\mathcal{P} \longrightarrow G'$ is provable It has been argued elsewhere [25] that uniform proofs can provide a general and abstract characterization of which logical systems make suitable logic programming languages.

We can now modify the definition of a result of a Horn clause program. Let the set of Σ-formulas $\mathcal{P}$ be a Horn clause program and let the goal formula $\exists_{\tau_1}\mathbf{x}_1 \ldots \exists_{\tau_n}\mathbf{x}_n G$ $(n \geq 0)$ be a Σ-formula. Given a uniform proof of the sequent

$$\Sigma\,;\,\mathcal{P} \longrightarrow \exists_{\tau_1}\mathbf{x}_1 \ldots \exists_{\tau_n}\mathbf{x}_n G,$$

a result of solving G with respect of $\mathcal{P}$ can be read off of the proof: by Theorem 1, the last n inference rules of this proof are instances of the $\exists$-R inference rule; a result is, therefore, the list of the substitution terms used in these $\exists$-R rules.

If Horn clauses were extended into a richer collection of formulas, uniform proofs would not generally be complete for even minimal logic. For example, let p be a predicate constant of type $i \to o$ and let a and b be constants of type i. If $(p\ a) \vee (p\ b)$ was allowed to be a program clause then from this program clause both the disjunct $(p\ a) \vee (p\ b)$ and the existential $\exists_i x\,(p\ x)$ have **M**-proofs but neither have uniform proofs. Thus, the interpretation of a disjunctive goal as a disjunctive computation specification would break down in the presence of such a program clause. Also, proofs would fail to yield results for computations as before since existential goals cannot be proved directly by the $\exists$-R inference rule.

In the next several sections, we present extensions to Horn clauses and, in each case, we require that uniform proofs are complete since we desire

that any logical foundation for a logic programming language should admit goal-directed interpretation. Theorem 1, therefore, can be interpreted as stating that the classical theory of Horn clauses is a suitable (although weak) foundation for a logic programming language. There are, fortunately, other more expressive foundations.

4. Providing Scope to Program Clauses

Horn clause programs do not possess any mechanisms for providing scope to program clauses. During a computation (a search for a uniform proof) involving Horn clauses, the antecedents of all sequents remain unchanged: in a sense, the logic program is global. Thus, if several programs (sets of Horn clauses) are needed for a particular computation, those sets must all be unioned together at the start of the computation. This has the obvious disadvantage that, in large programs, there may be name clashes between different parts of a program, that is, two different programs may use the same constant in two different and mutually inconsistent fashions. This lack of modularization makes many aspects of program development and verification difficult to address.

Prolog implementations have dealt with this lack in their logical foundations by providing several nonlogical and side-effect primitives [34]. For example, the `consult` program primitive loads program clauses from disk storage and adds them to the current program. There is no intended scope to this augmentation: it is permanent until some additional side-effect is used to undo it. The `assert` primitive of Prolog takes a term and, translating it as a program clause, adds the latter to the current program, again with no intended scope to the addition. The added clause is available until the nonlogical primitive `retract` removes it. In essence, the pair, `assert` and `retract`, are editing commands for the current program space. Since they are so general, scoping facilities for Prolog can be built using them. This situation in Prolog is rather unfortunate since it should be possible, as in most programming languages that have scoping mechanisms, to separate scoping from side-effects.

Fortunately, logic provides a very simple scoping mechanism for program clauses. Using the definition of uniform proofs in the last section, consider a uniform proof of a sequent with succedent $D \supset G$. This sequent must be the conclusion of the inference rule $\supset$-R. Thus, to prove this goal from, say program $\mathcal{P}$, it is necessary to prove G from the program $\mathcal{P} \cup \{D\}$. This use of implication to denote the operation that augments the antecedent (the current program) leads immediately to a stack based scoping mechanism for program clauses. For example, let D_1 and D_2 be two Horn clauses and let

G_1 and G_2 be two goals. A uniform proof of the sequent

$$\Sigma\ ;\quad \longrightarrow\ D_1 \supset ((D_2 \supset G_1) \wedge G_2)$$

would contain uniform proofs of the two sequents

$$\Sigma\ ;\ D_1, D_2\ \longrightarrow\ G_1 \quad \text{and} \quad \Sigma\ ;\ D_1\ \longrightarrow\ G_2.$$

That is, the goals G_1 and G_2 are attempted with respect to two different programs.

In order to capitalize on this use of implications in goal formulas, we must also permit them in the body of clauses. This leads to the following extension of the classes of goal formulas and definite clauses, given by the mutually recursive definitions for the syntactic variables G and D:

$$\begin{array}{c} G ::= \top \mid A \mid G_1 \wedge G_2 \mid G_1 \vee G_2 \mid \exists_\tau \mathbf{x}\ G \mid D \supset G \\ D ::= A \mid G \supset A \mid \forall_\tau \mathbf{x}\ D \mid D_1 \wedge D_2. \end{array} \qquad (*)$$

We will now extend our use of the terms *definite formula* and *goal formula* to be D- and G-formulas in this new sense. Notice that Horn clauses are still definite clauses.

The classical logic interpretation of logical connectives does not support the scoping interpretation of implication that we have just described. For example, there is no uniform proof of the goal formula $p \vee (p \supset q)$ from the empty program since neither does p follow from the empty program nor does q follow from p. This formula, however, is a classical logic tautology since it is truth-functionally equivalent to $(p \supset p) \vee q$. Intuitutionistic and minimal logics, however, do support this scoping interpretation of implications. The following theorem is proved in [20].

Theorem 2. *Let Σ be a first-order signature over S that also inhabits S. With respect to $(*)$ above, let $\mathcal{P}$ be a finite set of definite formulas and let G be a goal formula such that $\mathcal{P} \cup \{G\}$ is a set of Σ-formulas. $\Sigma; \mathcal{P} \vdash_I G$ if and only if the sequent $\Sigma\ ;\ \mathcal{P}\ \longrightarrow\ G$ has a uniform proof.*

Similar to the previous section, all the following statements are immediate conclusions of Theorem 2.

- All sequents occurring in a uniform proof of the sequent

  $$\Sigma\ ;\ \mathcal{P}\ \longrightarrow\ G$$

  are of the form $\Sigma\ ;\ \mathcal{P}'\ \longrightarrow\ G'$, where G' is some goal formula and $\mathcal{P}'$ is a set of definite formulas containing $\mathcal{P}$. The signatures do not change.
- $\Sigma; \mathcal{P} \vdash_I G$ if and only if $\Sigma; \mathcal{P} \vdash_M G$.
- $\Sigma; \mathcal{P} \vdash_I G_1 \wedge G_2$ if and only if $\Sigma; \mathcal{P} \vdash_I G_1$ and $\Sigma; \mathcal{P} \vdash_I G_2$.

- $\Sigma; \mathcal{P} \vdash_I G_1 \vee G_2$ if and only if $\Sigma; \mathcal{P} \vdash_I G_1$ or $\Sigma; \mathcal{P} \vdash_I G_2$.
- $\Sigma; \mathcal{P} \vdash_I \exists_\tau \mathbf{x}\, G$ if and only if there is a Σ-term $\mathbf{t}$ such that $\Sigma; \mathcal{P} \vdash_I [\mathbf{x} \mapsto \mathbf{t}]G$.
- $\Sigma; \mathcal{P} \vdash_I D \supset G$ if and only if $\Sigma; \mathcal{P} \cup \{D\} \vdash_I G$.

The following example illustrates the use of an implication in the body of a program clause. Let S be the set $\{i, l, o\}$ and let Σ be some first-order signature that contains the signature

$$\{nil\colon l,\ cons\colon i \to l \to l,\ a\colon i,\ b\colon i,\ reverse\colon l \to l \to o,\ r\colon l \to l \to o\}.$$

Lists of the individual constants a and b are denoted by Σ-terms of type l. For example, the term

$$(cons\ a\ (cons\ b\ nil))$$

denotes the list with first member a and second member b. To make terms denoting lists more compact and suggestive of the lists they denote, we use the usual Prolog syntax for lists [34]: the expression $[a_1, \ldots, a_n|k]$ denotes the term

$$(cons\ a_1\ \ldots\ (cons\ a_n\ k)\ldots).$$

Finally, the expression $[a_1, \ldots, a_n|nil]$ is simply written as $[a_1, \ldots, a_n]$. Thus $[\,]$ denotes *nil*. Below is a program clause that defines the binary relation of two lists being *reverses* of each other.

$$\forall_l l \forall_l k \big(\{ [(r\ [\,]\ k) \wedge \forall_i x \forall_l m \forall_l n ((r\ n\ [x|m]) \supset (r\ [x|n]\ m))] \supset (r\ l\ [\,])\} \supset (reverse\ l\ k) \big)$$

Name this one clause as D_r and consider using this clause to check that the reverse of the list $[a, b]$ is the list $[b, a]$. This is done by attempting a uniform proof of the sequent

$$\Sigma\ ;\ D_r \longrightarrow (reverse\ [a, b]\ [b, a]).$$

This sequent has a uniform proof only if the sequent

$$\Sigma\ ;\ D_r,\ (r\ [\,]\ [b, a]),\ \forall_i x \forall_l m \forall_l n [(r\ n\ [x|m]) \supset (r\ [x|n]\ m)] \longrightarrow (r\ [a, b]\ [\,])$$

has a uniform proof. A uniform proof of this sequent essentially involves using only backchaining over the two Horn clauses that define the meaning of the r predicate.

This way of defining *reverse* is very natural and symmetric: two lists, say terms $\mathbf{t}$ and $\mathbf{s}$, are reverses of each other if we can move from the pair of terms $([\,], \mathbf{s})$ to the pair $(\mathbf{t}, [\,])$ by successively moving the first list member of the second component to the front of the first component.

From the definition of *reverse*, it is clear that the embedded two clauses for r are available only during the search of proofs for goals of the form $(reverse\ \mathbf{t}\ \mathbf{s})$. It is not possible, however, to guarantee that during such a search, only those two clause for r are available. If the program was larger, that is, of the form $\mathcal{P} \cup \{D_r\}$, and if $pred(\mathcal{P})$ contained the predicate r, then the atomic goal $(r\ [a, b]\ [\,])$ will be attempted with more than the two clauses intended to define the predicate r. In certain programs, this accumulation may be exactly what is intended. In other programs, such as the one presented here, it serves the wrong purpose. The additional scoping declarations described in the next two sections provides a way to address this problem.

5. Providing Scope to Individual Constants

In both Horn clauses and the extension to them made in the previous section, no provisions have been made for giving scope to constants. This follows from a simple analysis of the structure of uniform proofs involving these formulas. For example, a uniform proof of the sequent $\Sigma\ ;\ \mathcal{P} \longrightarrow G$, where $\mathcal{P}$ is a set of definite clauses and G is a goal (with respect to the definitions in the previous section), contains sequents all with the same signature component, namely Σ. It is not possible to specify programs that make use of different signatures at different parts of a computation (proof): signatures are essentially global. For a constant to be used one place in a computation, it must be present at the start of the computation (in the root sequent); thus, since constants are used to build data structures, it is not possible for a program to build data structures that are used to solve entirely local problems. One part of a program might ascribe special meaning to certain data constructors but there is no formal mechanism to guarantee that other parts of the program would not ascribe it other meanings. These logic programming languages lack support for what is generally called *data abstraction*.

Again, logic provides a very simple mechanism for providing constants with scope. With respect to the notion of definite clauses and goals defined in the previous section, let $\mathcal{P}$ be a set of Σ-formulas that are also definite formulas and let G be a Σ-formula and a goal formula in which $\mathbf{x}$ occurs free. Consider attempting to find a uniform proof for the sequent

$$\Sigma\ ;\ \mathcal{P} \longrightarrow \forall_\tau \mathbf{x}\, G.$$

This has a uniform proof only if it is the conclusion of the $\forall$-R inference rule, that is, only if the sequent

$$\Sigma \cup \{\mathbf{c}{:}\,\tau\}\ ;\ \mathcal{P} \longrightarrow [\mathbf{x} \mapsto \mathbf{c}]G,$$

where $\mathbf{c}$ is a constant not in Σ, has a uniform proof. The search for a proof of this sequent thus has an expanded signature.

This use of the universal quantifier suggests the following enrichment of the notions of goals and definite clauses:

$$\begin{array}{c} G ::= \top \mid A \mid G_1 \wedge G_2 \mid G_1 \vee G_2 \mid \exists_\tau \mathbf{x}\, G \mid D \supset G \mid \forall_\tau \mathbf{y}\, G \\ D ::= A \mid G \supset A \mid \forall_\tau \mathbf{x}\, D \mid D_1 \wedge\ D_2. \end{array} \qquad (**)$$

In [14], Harrop studied a class of formulas that can be defined as follows. Let B be a syntactic variables for arbitrary first-order formulas and let H be defined by

$$H ::= A \mid B \supset H \mid \forall_\tau \mathbf{x}\, H \mid H_1 \wedge\ H_2.$$

An H-formula is often called a *Harrop formula*. Clearly, all definite clauses in $(**)$ are Harrop formulas. Such definite clauses also satisfy an additional constraint: all positive subformula occurrences of a definite clause are also Harrop formulas. For this reason, the definite clauses in $(**)$ are referred to as *first-order hereditary Harrop formulas*. The following theorem is proved in [25].

Theorem 3. *Let Σ be a first-order signature over S that also inhabits S. Let $\mathcal{P}$ be a finite set of hereditary Harrop formulas and let G be a goal formula in $(**)$ such that $\mathcal{P} \cup \{G\}$ is a set of Σ-formulas. Then $\Sigma; \mathcal{P} \vdash_I G$ if and only if the sequent $\Sigma\ ;\ \mathcal{P}\ \longrightarrow\ G$ has a uniform proof.*

This theorem is similar to Theorem 2. All the observations listed after that theorem are true of this extension as well. The following additional observation can be made.

- Let Σ and $\mathcal{P}$ be as in the statement of theorem. Let $\forall_\tau \mathbf{x}\, G$ be a Σ-formula and a goal formula of $(**)$. Then $\Sigma; \mathcal{P} \vdash_I \forall_\tau \mathbf{x}\, G$ if and only if for some constant $\mathbf{c}$ of type τ not in Σ, $\Sigma \cup \{\mathbf{c}\colon \tau\}; \mathcal{P} \vdash_I [\mathbf{x} \mapsto \mathbf{c}]G$.

Thus, the search for a proof of a universally quantified goal leads to a proof in which the signature is augmented. Consider, for example, a uniform proof of a sequent that has the Σ-formula $\exists_\tau \mathbf{x} \forall_\tau \mathbf{y} \exists_\tau \mathbf{z} G$ as its succedent. This proof instantiates $\exists_\tau \mathbf{x}$ with a Σ-term, and instantiates $\exists_\tau \mathbf{z}$ with a $\Sigma \cup \{\mathbf{c}\colon \tau\}$-term, where $\mathbf{c}$ is the constant (not in Σ) that is used to instantiate $\forall_\tau \mathbf{y}$. Thus, the two existential quantifiers are distinguished by the fact that the second one can be instantiated by more substitution terms.

Unification is often used to implement interpreters for logic programming language since it provides a means for delaying the selection of substitution terms. The possibility of such quantifier alternation forces such implementations to use unification algorithms that are slightly more complex than the unification algorithm needed for implementing Horn clauses. If substitution terms are replaced by free variables instead of closed terms (as

required in our proof systems) then the signature from which substitution terms for such variables can be drawn must be somehow attached to those free variables. See [21] for a description of how unification can be modified to directly deal with quantifier alternation.

First-order hereditary Harrop formulas, as presented here, do not provide a strong enough notion of scoping to adequately deal with data abstraction because these formulas only allow the scoping of constants of primitive types. Data types in logic programming involve not only individual constants but also function and predicate constants. The next section extends first-order hereditary Harrop formulas to provide for the required stronger scoping notion.

6. Providing Scopes to Function and Predicate Symbols

There is an unfortunate asymmetry in the use of universal goals to augment signatures: while signatures contain constants with types of both order 0 and 1, universal goals can only be used to introduce new constants of order 0. Thus, it is natural to consider allowing universal goals to also quantify over types of order 1. That is, universal quantifiers should be allowed to quantify over function and predicate constants as well as simply individual constants. Thus, we will simply extend the definition $(**)$ of hereditary Harrop formulas to permit τ in the $\forall_\tau \mathbf{y}\, G$ case to be of order 0 or 1. From the proof-theoretic point-of-view, this is not a serious departure from first-order logic. For example, let σ be a type of order 1. A search for a uniform proof of the sequent

$$\Sigma \,;\, \mathcal{P} \;\longrightarrow\; \forall_\sigma \mathbf{k}\, G$$

(where $\mathbf{k}$ is a syntactic variable for function variables) yields a search for a uniform proof for the sequent

$$\Sigma \cup \{\mathbf{f}\!:\sigma\} \,;\, \mathcal{P} \;\longrightarrow\; [\mathbf{k} \mapsto \mathbf{f}]G,$$

provided $\mathbf{f}$ is not in Σ. The augmented signature, $\Sigma \cup \{\mathbf{f}\!:\sigma\}$, is still a first-order signature.

With this extension of hereditary Harrop formulas, we can improve on the definition of *reverse* from the Section 4. Consider the following definite clause.

$$\begin{aligned}\forall_l l \forall_l k \big[\forall_{l\to l\to o} r \big(&(r\ [\,]\ k) \wedge \\ &\forall_i x \forall_l m \forall_l n [(r\ n\ [x|m]) \supset (r\ [x|n]\ m)] \supset (r\ l\ [\,])\big) \\ &\supset (reverse\ l\ k)\big].\end{aligned}$$

This clause is a formula over the signature

$$\{nil\colon l,\ cons\colon i \to l \to l,\ a\colon i,\ b\colon i,\ reverse\colon l \to l \to o\},$$

that is, no auxiliary predicate constant is needed in this definition of *reverse*: just those constants needed to build lists and to designate the reverse program are required to build this clause. During a computation involving this clause, an auxiliary constant is introduced via the $\forall_{l\to l\to o} r$ quantifier, but this constant must be new to the signature current at the time it is introduced. Thus, there is no way for the environment (the current list of definite clauses or signature) to interfere with the internal workings of this version of *reverse*.

Before we illustrate how universal goals can be used to provide for data abstraction, it is convenient to make the following sugared extension to definite clauses. It would be very useful if the scoping that universal quantifiers provide to goals could also be provided to definite clauses. Consider, for example, the following three definite (Horn) clauses. Here, the primitive sort s informally denotes the type for the data type of *stacks*.

$$\forall_i x \forall_s s(pop\ x\ (stk\ x\ s)\ s)$$
$$\forall_i x \forall_s s(push\ x\ s\ (stk\ x\ s))$$
$$empty(emp)$$

With such an implementation of stacks, it might be desirable if the constants *emp*, for the empty stack, and *stk*, for the constructor that adds a stack element, were local to this definition. That is, goals and other definite clauses written using this implementation of stacks should formally be excluded from using these two constants: stack manipulations would need to be done exclusively via the three predicates *pop*, *push*, and *empty*.

Existential quantification over definite clauses can provide for this style of local declaration. For example, if D_s denotes the conjunction of the above three clauses that implement stacks, then $\exists_{i\to s\to s} stk\ \exists_s emp\, D_s$ should capture the desired notion of local scoping. Such a use of existential quantifiers, however, is not formally permitted in the definition of hereditary Harrop formulas above for a good reason. When existential quantifiers are allowed over program clauses, uniform proofs are no longer complete. For example, the sequent

$$\{p\colon i \to o\}\ ;\ \longrightarrow\ \exists_\tau x\ (p\ x) \supset \exists_\tau x\ (p\ x)$$

has an **M**-proof but has no uniform proof. Thus, the proposal to use existential quantifiers in program clauses must be qualified carefully.

Let D_1 and $\forall_\tau \mathbf{x}\, D_2$ be closed definite clauses and let G be a goal formula. While the formula $(D_1 \wedge \exists_\tau \mathbf{x}\, D_2) \supset G$ is not a goal in the extended logic of

this section, it is intuitutionistic (and minimally) equivalent to the formula $\forall_\tau \mathbf{x}((D_1 \wedge D_2) \supset G)$, which is a legal goal formula. This suggests that existential quantifiers can be allowed in a definite clause only if they are either at the top level of that definite clause or are in the scope of either conjunctions or other existential quantifiers. Such occurrences of existential quantifiers can rewritten in this fashion to be given a greater scope and made into occurrences of universal quantifiers over a goal formula. Thus the sequent above should be identified with the sequent

$$\{p{:}\, i \to o\} \; ; \;\; \longrightarrow \;\; \forall_\tau x \; ((p \; x) \supset \exists_\tau x \; (p \; x)),$$

which has a uniform proof.

If we return to the example of stacks above and use this identification, the search for a uniform proof of the sequent

$$\Sigma \; ; \; \mathcal{P} \;\; \longrightarrow \;\; (\exists_{i \to s \to s} stk \; \exists_s emp \, D_s) \supset G$$

would lead to the search for a uniform proof of the sequent

$$\Sigma \cup \{stk'{:}\, i \to s \to s, emp'{:}\, s\} \; ; \; \mathcal{P}, [stk \mapsto stk', emp \mapsto emp']D_s \;\; \longrightarrow \;\; G$$

where stk' and emp' are constants assumed not to be in Σ. The fact that stk' and emp' are "new" constants formally guarantees that the only meaning given to these constants is given to them by the clauses in $[stk \mapsto stk', emp \mapsto emp']D_s$.

7. A Higher-Order Logic

A form of abstraction we have not yet considered is that of higher-order programming. In the functional programming setting, this style of programming requires that functions be treated as values; that is, they can be constructed, applied, and bound to variables. In the logic programming setting, higher-order programming is characterized similarly except predicates instead of functions need to be treated as values. The logic $\mathcal{F}$ does not support computations with predicates since during the construction of a proof in $\mathcal{F}$ logic, the only substitution terms allowed are terms of primitive type, and such primitive types are different from o.

In the remainder of this paper we consider logic programming within a logic that permits predicate quantification and the construction of substitution terms for compound predicate expressions. This higher-order logic is essentially a sublogic of Church's Simple Theory of Types [4]. Terms are permitted to contain λ-abstractions and quantification is allowed over all types.

Let S be a set of primitive types, again containing the type o. The terms and formula of our higher-order logic, named $\mathcal{T}$, are defined with respect to signatures of arbitrary order over S. The logical constants of $\mathcal{T}$ are like those of $\mathcal{F}$ except that the quantifiers, $\forall_\tau$ and $\exists_\tau$, are taken to be constants instead of special binding operators. We also assume that there are denumerably many constants and variables for every type.

In $\mathcal{F}$, the class of terms and formulas intersect in the class of atomic formulas, that is, an atomic formula is essentially a term of type o. Because terms will now contain λ-abstractions, it is possible to defined the class of terms in such a way that they contain all formulas. In particular, a formula is defined to be a term of type o. To achieve this uniform treatment, the logical constants are given the following types: $\wedge, \vee, \supset$ are all of type $o \to o \to o$; $\top, \bot$ are of type o; and $\forall_\tau$ and $\exists_\tau$ are of type $(\tau \to o) \to o$, for all types τ. These latter two symbols are now considered to be constants and not binders as they were in $\mathcal{F}$. The binding associated to quantifiers will be captured by λ-abstractions.

A constant or variable of type τ is a term of type τ. If $\mathbf{t}$ is a term of type $\tau \to \sigma$ and $\mathbf{s}$ is a term of type τ, then the application $(\mathbf{t}\ \mathbf{s})$ is a term of type σ. Application associates to the left, that is, the expression $(\mathbf{t}_1\ \mathbf{t}_2\ \mathbf{t}_3)$ is read as $((\mathbf{t}_1\ \mathbf{t}_2)\ \mathbf{t}_3)$. Finally, if $\mathbf{x}$ is a variable of type τ and $\mathbf{t}$ is a term of type σ, then the abstraction $\lambda\mathbf{x}\ \mathbf{t}$ is a term of type $\tau \to \sigma$.

A formula is a term of type o. The logical constants $\wedge, \vee, \supset$ are written in the familiar infix form whenever they occur in an expression with two arguments present. The expressions $\forall_\tau(\lambda\mathbf{z}\ \mathbf{t})$ and $\exists_\tau(\lambda\mathbf{z}\ \mathbf{t})$ are written simply as $\forall_\tau\mathbf{z}\ \mathbf{t}$ and $\exists_\tau\mathbf{z}\ \mathbf{t}$.

If $\mathbf{x}$ and $\mathbf{s}$ are terms of the same type then $[\mathbf{x} \mapsto \mathbf{s}]$ denotes the operation of substituting $\mathbf{s}$ for all free occurrences of $\mathbf{x}$, systematically changing bound variables in order to avoid variable capture.

Terms are related to other terms by following conversion rules.

- The term $\mathbf{s}$ α-converts to the term $\mathbf{s}'$ if $\mathbf{s}$ contains a subformula occurrence of the form $\lambda\mathbf{x}\ \mathbf{t}$ and $\mathbf{s}'$ arises from replacing that subformula occurrence with $\lambda\mathbf{y}\ [\mathbf{x} \mapsto \mathbf{y}]\mathbf{t}$, provided $\mathbf{y}$ is not free in $\mathbf{t}$.
- The term $\mathbf{s}$ β-converts to the term $\mathbf{s}'$ if $\mathbf{s}$ contains a subformula occurrence of the form $(\lambda\mathbf{x}\ \mathbf{t})\mathbf{t}'$ and $\mathbf{s}'$ arises from replacing that subformula occurrence with $[\mathbf{x} \mapsto \mathbf{t}']\mathbf{t}$.
- The term $\mathbf{s}$ η-converts to $\mathbf{s}'$ if $\mathbf{s}$ contains a subformula occurrence of the form $\lambda\mathbf{x}\ (\mathbf{t}\ \mathbf{x})$, where $\mathbf{x}$ is not free in $\mathbf{t}$, and $\mathbf{s}'$ arises from replacing that subformula occurrence with $\mathbf{t}$.

The binary relation *conv*, denoting λ-*conversion*, is defined so that $\mathbf{t}$ *conv* $\mathbf{s}$ if there is a list of terms $\mathbf{t}_1, \ldots, \mathbf{t}_n$, with $n \geq 1$, $\mathbf{t}$ equal to $\mathbf{t}_1$, $\mathbf{s}$ equal to $\mathbf{t}_n$, and for $i = 1, \ldots, n-1$, either $\mathbf{t}_i$ converts to $\mathbf{t}_{i+1}$ or $\mathbf{t}_{i+1}$

converts to $\mathbf{t}_i$ by α, β, or η. Expressions of the form $\lambda\mathbf{x}\ (\mathbf{t}\ \mathbf{x})$ are called η-redexes (provide $\mathbf{x}$ is not free in $\mathbf{t}$) while expressions of the form $(\lambda\mathbf{x}\ \mathbf{t})\mathbf{s}$ are called β-redexes. A term is in λ-normal form if it contains no β or η-redexes. Every term can be converted to a λ-normal term, and that normal term is unique up to the name of bound variables. See [15] for a fuller discussion of these basic properties of the simply typed λ-calculus.

Let Σ be a signature of unrestricted order. A term is a Σ-*term* if all of its nonlogical constants are members of Σ. Similarly, a formula is a Σ-*formula* if all of its nonlogical constants are members of Σ. It should be clear that if Σ is a first-order signature, then every Σ-term and Σ-formula of $\mathcal{F}$ corresponds directly to a λ-normal Σ-term and Σ-formula of $\mathcal{T}$. The substitution operation in $\mathcal{F}$ is naturally extended by the substitution operation of $\mathcal{T}$.

All the inference rules given in Section 2 can be interpreted as inference rules for $\mathcal{T}$, where, of course, the signatures allowed in sequents is now permitted to be of arbitrary order. To have a complete inference system for $\mathcal{T}$ we need only add the inference rule

$$\frac{\Sigma\ ;\ \Gamma\ \longrightarrow\ \Delta}{\Sigma\ ;\ \Gamma'\ \longrightarrow\ \Delta'}\ \lambda,$$

where the pair Γ and Γ' and the pair Δ and Δ' different only modulo λ-conversion.

A λ-normal term, say $\mathbf{t}$, is of the form

$$\lambda\mathbf{x}_1 \ldots \lambda\mathbf{x}_n(\mathbf{h}\ \mathbf{t}_1 \cdots \mathbf{t}_m)$$

where $n, m \geq 0$, $\mathbf{h}$ is either a constant or variable, and the terms $\mathbf{t}_1, \ldots, \mathbf{t}_n$ are λ-normal. The *head* of $\mathbf{t}$ is $\mathbf{h}$, the *arguments* of $\mathbf{t}$ are the terms $\mathbf{t}_1, \ldots, \mathbf{t}_m$, and the *binder* of $\mathbf{t}$ is the list of variables $\mathbf{x}_1, \ldots, \mathbf{x}_n$. If B is a λ-normal formula, then its binder is empty and its head is either a logical constant, a nonlogical constant, or a variable. If the head of B is not a logical constant then B is an *atomic* formula. An atomic formula is *rigid* if its head is a nonlogical constant, otherwise the head is a variable and it is *flexible*.

$\mathcal{T}$ is much more complex than $\mathcal{F}$ for several reasons. By virtue of the λ inference rule, the notion of equality of terms is that of λ-conversion, which is a much richer notion of equality than the simple syntactic identity used in $\mathcal{F}$. Also, since quantification can now be over types of any order and formulas can contain flexible atoms, it is possible for the structure of formulas to change drastically under substitutions. For example, consider the formula $\forall_{i\to o} p \forall_i x(p\ x \supset q\ x)$, where q is a constant of type $i \to o$. Let B be some formula in which the variable x may be free. The result of substituting $\lambda x\, B$ for p, and then normalizing, yields the formula $\forall_i x(B \supset q\ x)$. Thus, unlike

in $\mathcal{F}$, the number of occurrences of logical constants can increase arbitrarily as a result of universal instantiation. This fact alone makes theorem proving in $\mathcal{T}$ particularly difficult. As we observe in the next section, this aspect of substitution makes higher-order extensions to logic programming languages more expressive and significantly complicates the proof that uniform proofs can be complete. For more about higher-order logics similar to $\mathcal{T}$, see [1, 4].

8. Two Higher-Order Logic Programming Languages

There are several way to generalize Horn clauses from $\mathcal{F}$ to $\mathcal{T}$. The approach used here is to permit quantification over all occurrences of function symbols and some occurrences of predicate symbols, and to replace first-order terms by simply typed λ-terms within which there may be embedded occurrences of logical connectives. Thus, predicates *and* functions are treated as first-class values. As we mentioned earlier, it is having predicates as values that gives rise to higher-order programming in logic programs. Thus, to achieve our ends, we do not need to permit quantification over function symbols in Horn clause. We will, however, permit such quantification since it does not complicate our proof-theoretic considerations in this sections and since, as is briefly mentioned in Section 10, this extension provides logic programs with programming techniques not easily achieved in other programming languages.

Let $\mathcal{H}_1$ be the set of all λ-normal terms that do not contain occurrences of the logical constants $\supset, \forall$, and $\bot$; that is, the only logical constants these terms may contain are $\top, \wedge, \vee$, and $\exists$. Let the syntactic variable A now denote an atomic formula in $\mathcal{H}_1$. Such a formula must have the form $(\mathbf{h}\,\mathbf{t}_1 \ldots \mathbf{t}_n)$, where $\mathbf{h}$ is either a variable or non-logical constant and $\mathbf{t}_1, \ldots, \mathbf{t}_n$ are members of $\mathcal{H}_1$. The syntactic variable A_r is used for rigid atomic formulas in $\mathcal{H}_1$. A *goal formula* in the logic of *higher-order Horn clauses* is any formula in $\mathcal{H}_1$. Notice that goal formulas satisfy the clause

$$G ::= \top \mid A \mid G_1 \wedge G_2 \mid G_1 \vee G_2 \mid \exists_\tau \mathbf{x}\, G$$

just as is the case with the goals attached to first-order Horn clauses. This clause, however, does not serve to define the entire structure of these goals since it does not reveal the structure of logical connectives that can appear inside atomic formulas. The set of *higher-order Horn clauses* is defined by the following clause:

$$D ::= A_r \mid G \supset A_r \mid D_1 \wedge D_2 \mid \forall_\tau \mathbf{x}\, D.$$

The quantification in both G- and D-formulas may, of course, be over variables of any type.

The use of rigid atoms in the definition of higher-order Horn clauses has two simple motivations. First, if we generalize the function *pred* from first-order to higher-order formulas in the obvious manner, then the restriction to rigid atoms implies that *pred* always return a set of predicate constants. Thus, it is still possible to identify a program clause as partially defining *specific* procedures, namely, those returned by the *pred* function. Second, this requirement also makes it impossible for a collection of higher-order Horn clauses to be inconsistent. As a corollary of Theorem 4 (below), a sequent of the form $\Sigma\ ;\ \mathcal{P}\ \longrightarrow\ A_r$ is provable only if the top-level predicate constant of A_r is a member of $pred(\mathcal{P})$. If the condition on occurrences of predicate variables is relaxed, however, programs can become inconsistent. Arbitrary formulas are provable, for instance, from the set $\{p, \forall x(p \supset x)\}$.

The following theorem was first proved in [26]. Details of the proof can be found either there or in the paper [28].

Theorem 4. *Let Σ be a signature of arbitrary order over S such that Σ inhabits S. Let $\mathcal{P}$ be a finite set of higher-order Horn clauses and let G be a goal formula such that $\mathcal{P} \cup \{G\}$ is a set of Σ-formulas. $\Sigma; \mathcal{P} \vdash_C G$ if and only if the sequent $\Sigma\ ;\ \mathcal{P}\ \longrightarrow\ G$ has a uniform proof.*

A proof of this theorem is not a immediate generalize of the proof for the theorem concerning first-order Horn clauses (Theorem 1) since proofs in the higher-order setting can be very complex. For example, Figure 3 contains a derivation (taken from [28]) of the goal formula $\exists_i y\,(p\ y)$ from the higher-order Horn clause $\forall_o x\ (x \supset p\ a)$. We assume here that p is of type $i \to o$, that a and b are constants of type i, and that q is of type o. (Signatures are not listed in the sequents of Figures 3 and 4 since they are all the same.) This derivation illustrates that the substitution instance of a higher-order Horn clause may not be a higher-order Horn clause. A straightforward induction on the structure of proofs cannot be employed to prove Theorem 4.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{p\ b\ \longrightarrow\ q, p\ b}{\longrightarrow\ p\ b \supset q, p\ b}\ \supset\text{-R} \qquad p\ a\ \longrightarrow\ p\ a}{(p\ b \supset q) \supset p\ a\ \longrightarrow\ p\ a, p\ b}\ \supset\text{-L}}{(p\ b \supset q) \supset p\ a\ \longrightarrow\ p\ a, \exists y\ p\ y}\ \exists\text{-R}}{(p\ b \supset q) \supset p\ a\ \longrightarrow\ \exists y\ p\ y}\ \exists\text{-R}}{\forall x\ (x \supset p\ a)\ \longrightarrow\ \exists y\ p\ y}\ \forall\text{-L}$$

Figure 3: A non-uniform proof

The first step in proving Theorem 4 is to prove the following lemma: In constructing **C**-proofs of the sequents involving higher-order Horn clauses and their corresponding goal formulas, the only substitution terms needed are terms from the set $\mathcal{H}_1$. In other words, $\mathcal{H}_1$ is a kind of Herbrand Universe for higher-order Horn Clauses. Once this lemma is established, it follows that the only substitution instances of higher-order Horn clauses that need to be considered are also higher-order Horn clauses. Given this, the inductive argument used to prove Theorem 1 will work here also. The proof of this lemma employs a proof transformation that maps certain occurrence of implicational subformulas to $\top$. The result of applying this transformation to the proof in Figure 3 yields the proof in Figure 4. While the transformed proof is not uniform, it is easy to extract a uniform proof from it.

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

Figure 4: A modified proof

In the next section we will present several examples of higher-order logic programs. For now, we present two very simple examples. Let *exists* be a constant of type $(i \to o) \to o$ and let *or* be a constant of type $o \to o \to o$. The three clauses

$$\forall_{i\to o} p \forall_i x\ (p\ x \supset exists\ p)$$
$$\forall_o p \forall_o q\ (p \supset or\ p\ q)$$
$$\forall_o p \forall_o q\ (q \supset or\ p\ q)$$

define, in a sense, existential quantification over type i and disjunction in goal formulas. That is, let $\mathcal{P}$ be a higher-order Horn clause program containing these three classes such that for every other member D of $\mathcal{P}$, $pred(D)$ does not contain *exists* or *or*. Then (suppressing signatures) $\mathcal{P} \vdash_C (or\ G_1\ G_2)$ if and only if $\mathcal{P} \vdash_C G_1$ or $\mathcal{P} \vdash_C G_2$. Similarly, $\mathcal{P} \vdash_C (exists\ \lambda x\ G)$ if and only if there exists a term t of type i such that $\mathcal{P} \vdash_C [x \mapsto t]G$. Of course, existential quantifiers of other types could similarly be defined in this way.

We now describe a higher-order version of hereditary Harrop formulas. There are several choices in how such an extension can be made. One critical

choice concerns the richness of the logical expressions that may be embedded in atomic formulas. In making Horn clauses higher-order, the logical connectives $\top, \wedge, \vee$, and $\exists_\tau$ are permitted to be embedded in atomic formulas. These are also the same logical constants allowed as the top-level connectives of the goals attached to Horn clauses. It would be natural to allow this parallel: that is, to permit atomic, higher-order hereditary Harrop formulas to contain embedded logic that included not only the connectives $\top$, $\vee$, $\wedge$, and $\exists_\tau$ that are permitted in $\mathcal{H}_1$ but also the connectives $\supset$ and $\forall_\tau$. As we show below, if implications are permitted to be embedded in atomic formulas, uniform proofs will no longer be complete.

Let $\mathcal{H}_2$ be the set of λ-normal terms that do not contain occurrences of the logical constants $\supset$ and $\bot$. In other words, $\mathcal{H}_2$ only extends $\mathcal{H}_1$ by permitting the constants $\forall_\tau$ (for all types τ). Let the syntactic variable A denote atomic formulas in $\mathcal{H}_2$ and let the syntactic variable A_r denote rigid atomic formulas in $\mathcal{H}_2$. Define the notions of goal formulas and definite formulas by the following mutual recursion:

$$G ::= \top \mid A \mid G_1 \wedge G_2 \mid G_1 \vee G_2 \mid \forall x\, G \mid \exists x\, G \mid D \supset G$$
$$D ::= A_r \mid G \supset A_r \mid \forall x\, D \mid D_1 \wedge D_2.$$

These D-formulas are called *higher-order hereditary Harrop formulas*. Clearly, every higher-order Horn clause is a higher-order hereditary Harrop formula. Similarly, every first-order hereditary Harrop formula is also a higher-order hereditary Harrop formula. It should be pointed out, however, that the definite clauses defined in Section 6 are not instances of higher-order hereditary Harrop formulas: the quantification over predicates that is allowed in Section 6 is richer than what is permitted here.

The proof of the following theorem can be found in [25].

Theorem 5. *Let Σ be a signature of arbitrary order over S such that Σ inhabits S. Let $\mathcal{P}$ be a finite set of higher-order hereditary Harrop formulas and let G be a goal formula such that $\mathcal{P} \cup \{G\}$ is a set of Σ-formulas. $\Sigma; \mathcal{P} \vdash_I G$ if and only if the sequent $\Sigma\ ;\ \mathcal{P} \longrightarrow G$ has a uniform proof.*

A logic programming language based on higher-order hereditary Harrop formulas does provide both higher-order programming as well as most of the program structuring mechanism described in Sections 4, 5, and 6. As hinted above, these two kinds of programming abstractions cannot be mixed as fully as one might like since uniform, goal-directed proofs are no longer complete. Permitting implications into the terms that appear as arguments of atomic formulas can result in "goal" formulas that are theorems of minimal logic but do not have uniform proofs. In operational terms, if implications are permitted inside atomic formulas then it can no longer be guaranteed that

embedded logical connectives will move into the top-level logical structure of formulas in a manner respecting the syntactic restrictions on goal formulas and definite clauses. For example, consider the following formula (taken from [25]):

$$\exists Q[\forall p \forall q[R(p \supset q) \supset R(Qpq)] \wedge Q(t \vee s)(s \vee t)],$$

where R is a constant of type $o \to o$, s and t are constants of type o, Q is a variable of type $o \to o \to o$, and p and q are variables of type o. This formula has exactly one **M**-proof, which is obtained by substituting $\lambda x \lambda y(x \supset y)$ for the bound variable Q. This proof must contain within it a proof of the sequent $t \vee s \longrightarrow s \vee t$. Since there is no uniform proof of this sequent, there can be no uniform proof for the original sequent. In this example, the subformula $t \vee s$ has an occurrence in a goal formula (inside an atomic formula). Once the substitution of an implication for Q is made, this disjunction appears on the left-hand side of an implication as an "illegitimate" program clause. It is this movement of a disjunction from a positive to a negative occurrence that spoils the completeness of uniform proofs.

One of the possible uses for higher-order features in a programming language is that part of a computation might build programs that later parts of the computation might use. The restriction that requires *rigid* atoms at various positions in D-formulas, however, greatly restricts the possibility of this kind of computation within the logic programming languages presented here. For example, it is impossible for a set of higher-order hereditary Harrop formulas to build new terms and then directly "evaluate" them. Consider, for example, the sequent

$$\mathcal{P} \longrightarrow \exists Q[(\mathit{compile}\ d\ Q) \wedge (Q \supset g)],$$

where d and g are some terms of type o and Q is a variable of type o. This sequent can be thought of as describing the computation that uses the information in the term d to build (compile) a program clause Q, and then to use that new clause to help solve the goal g. Here, we assume that the program $\mathcal{P}$ contains clauses that describe how to compute the relation *compile*. The succedent of this sequent, however, is not a valid hereditary Harrop formula since the left-hand side of the implication $Q \supset g$ is not a rigid atom. This language, therefore, does not seem to have a feature that, for example, corresponds directly to the *eval* function of Lisp.

In a practical system, both of these restrictions (rigid atoms and no embedded implications) can be a hindrance to a programmer. One way around this would be to simply remove these restrictions and allow sequents like the one above. There is no problem with permitting this sequent as long as the terms that are related by *compile* to d are legal program clauses. This is,

however, a rather deep question about the nature of the user-defined predicate *compile* and such a property can be very hard to establish in general. Instead, an implementation of a higher-order version of hereditary Harrop formulas might not check these syntactic restrictions of goal formulas prior to attempting to interpret them. Such an interpreter would need to be prepared to generate runtime errors if it were ever asked to consider a sequent in which the antecedent was not a collection of higher-order hereditary Harrop formulas. An advantage of using hereditary Harrop formulas exclusively is that Theorem 5 guarantees that there will be no such runtime errors. Such a restriction, however, would disallow meaningful computations.

9. Higher-Order Programming

Below we present several simple examples of higher-order programming using higher-order Horn clauses. Consider the following four constants, each of which is of order 2 and takes a first-order predicate as their first argument.

$$\begin{aligned} \mathit{mappred} &: (i \to i \to o) \to \mathit{list} \to \mathit{list} \to o \\ \mathit{forsome} &: (i \to o) \to \mathit{list} \to o \\ \mathit{forevery} &: (i \to o) \to \mathit{list} \to o \\ \mathit{trans} &: (i \to i \to o) \to i \to i \to o \end{aligned}$$

We assume that these predicates are specified by the following higher-order Horn clauses.

$$\begin{aligned} &\forall P\,(\mathit{mappred}\ P\ [\,]\ [\,]) \\ &\forall P, x, y, l, k\,(P\ x\ y \wedge \mathit{mappred}\ P\ l\ k \supset \mathit{mappred}\ P\ [x|l]\ [y|k]) \end{aligned}$$

$$\begin{aligned} &\forall P, x, l\,(P\ x \supset \mathit{forsome}\ P\ [x|l]) \\ &\forall P, x, l\,(\mathit{forsome}\ P\ l \supset \mathit{forsome}\ P\ [x|l]) \end{aligned}$$

$$\begin{aligned} &\forall P\,(\mathit{forevery}\ P\ [\,]) \\ &\forall P, x, l\,(P\ x \wedge \mathit{forevery}\ P\ l \supset \mathit{forevery}\ P\ [x|l]) \end{aligned}$$

$$\begin{aligned} &\forall R, x, y\,(R\ x\ y \supset \mathit{trans}\ R\ x\ y) \\ &\forall R, x, y, z\,(\mathit{trans}\ R\ x\ y \wedge \mathit{trans}\ R\ y\ z \supset \mathit{trans}\ R\ x\ z) \end{aligned}$$

The intended meaning of these higher-order predicates is very simple. A goal of the form $(\mathit{mappred}\ P\ l\ k)$ is provable given the code above if l and k are lists of equal length and corresponding members of these lists are P-related. This predicate seems to correspond most closely to the *mapcar* function of Lisp. Similarly, the goal $(\mathit{forsome}\ P\ l)$ is provable if l is a list in which some

member satisfies the predicate P. The goal (*forevery* P l) is provable if l is a list all of whose members satisfy the predicate P. A goal of the form (*trans* R x y) is provable given the code above if x and y are members of the transitive closure of the binary relation R.

For a final example of higher-order programming, we consider a specification of tactics and tacticals. As described in [10], a *tactic* is a primitive method for decomposing a goal into a list of goals, and a *tactical* is a high-level method for composing these tactics into meaningful and large scale problem solvers. The functional programming language ML has often been used to implement tactics and tacticals. Below we present a specification (given in [5]) of them as a collection of higher-order Horn clauses.

Let g be a new primitive type that denotes expressions encoding object-level goals (not to be confused with "meta-level" goals, which are terms of type o). The exact nature of the terms of this type depends on the problem domain. For example, if the application is that of attempting to find sequent proofs for formulas in first-order logic, then terms of type g need to encode the succedent and antecedent of a sequent. If the goal is to show that an equality is provable, the goal must encode that equality. In any case, we assume that there are always two constants that can be used to build goals, namely, *truegoal* of type g, which denotes the trivially satisfied goal, and *andgoal* of type $g \to g \to g$, which denotes the conjunction or pairing of two goals. A term of type g that is conjunctive is also called a *compound* goal; otherwise it is a *primitive* goal. In ML implementations of these concepts, nonempty lists of goals are used to denote compound goals and the empty list of goals is used to denote *truegoal*. While this approach can easily be adopted here, it is natural to consider other ways of combining goals besides conjunction and, hence, identifying compound goals with lists of goals is not sufficiently general [5].

In this setting then, a tactic is a binary relation between a primitive goal and another goal, possibly compound or primitive. Thus tactics are coded as predicates of type $g \to g \to o$. Abstractly, if a tactic denotes the relation R, then $R(g_1, g_2)$ is true if satisfying goal g_2 is sufficient to satisfy goal g_1. For example, assume that the formulas of a propositional logic are identified with terms of type *bool*: the constants p, q, r, and s are of type *bool* and denote propositional constants and the constants and, or, imp are of type $bool \to bool \to bool$ and denote conjunction, disjunction, and implication at the propositional level. Let $true$ be a constant of type $bool \to g$. Then an expression of the form $(true\ A)$ denotes the object-level goal of demonstrating that the propositional formula A is true. Given this encoding of propositional logic into (first-order) λ-terms of type *bool*, the following are examples of

some simple tactics.

$\forall A, B\,(andtac\ (true\ (and\ A\ B))\ (andgoal\ (true\ A)\ (true\ B)))$
$\forall A, B\,(ortac\ (true\ (or\ A\ B)\ (true\ A))$
$\forall A, B\,(ortac\ (true\ (or\ A\ B)\ (true\ B))$
$\forall A, B\,(backchain\ (true\ A)\ (andgoal\ (true\ (imp\ B\ A))\ (true\ B)))$

Before presenting the clauses for tacticals, we need to present an auxilary function named *maptac* of type $(g \to g \to o) \to g \to g \to o$ that applies a tactic to all primitive goals in a given goal. In the remaining clauses used to specify aspects of this example, we do not explicitly universally quantify variables around displayed Horn clauses. Instead, when the symbols R and G, with possible subscripts, are present in a clause, we assume that they are implicitly universally quantified over the clause in which they occur.

$maptac\ R\ truegoal\ truegoal$
$maptac\ R\ G_1\ G_3 \wedge maptac\ R\ G_2\ G_4 \supset$
$\qquad maptac\ R\ (andgoal\ G_1\ G_2)\ (andgoal\ G_3\ G_4)$
$R\ G_1\ G_2 \supset maptac\ R\ G_1\ G_2$

Given these clauses, we can now specify several common tacticals. The new constants in the following higher-order Horn clauses have the following types.

$$\begin{aligned} then &: (g \to g \to o) \to (g \to g \to o) \to g \to g \to o \\ orelse &: (g \to g \to o) \to (g \to g \to o) \to g \to g \to o \\ idtac &: g \to g \to o \\ try &: (g \to g \to o) \to g \to g \to o \\ complete &: (g \to g \to o) \to g \to g \to o \\ goalreduce &: g \to g \to o \end{aligned}$$

In the clauses below, the variables R, R_1, R_2 denote tactics while the variables G, G_1, G_2, G_3, G_4 denote object-level goals.

$R_1\ G_1\ G_3 \wedge maptac\ R_2\ G_3\ G_2 \supset then\ R_1\ R_2\ G_1\ G_2$

$R_1\ G_1\ G_2 \supset orelse\ R_1\ R_2\ G_1\ G_2$
$R_2\ G_1\ G_2 \supset orelse\ R_1\ R_2\ G_1\ G_2$

$idtac\ G\ G$

$orelse\ (then\ R\ (repeat\ R))\ idtac\ G_1\ G_2 \supset repeat\ R\ G_1\ G_2$

$orelse\ R\ idtac\ G_1\ G_2 \supset try\ R\ G_1\ G_2$

$R\ G_1\ G_2 \wedge goalreduce\ G_2\ truegoal \supset complete\ R\ G_1\ truegoal$

$$\begin{array}{l} \mathit{goalreduce}\ G_1\ G_2 \supset \mathit{goalreduce}\ (\mathit{andgoal}\ \mathit{truegoal}\ G_1)\ G_2 \\ \mathit{goalreduce}\ G_1\ G_2 \supset \mathit{goalreduce}\ (\mathit{andgoal}\ G_1\ \mathit{truegoal})\ G_2 \\ \mathit{goalreduce}\ G\ G \end{array}$$

The *then* tactical performs the composition of tactics. The tactic R_1 is applied to the (primitive) "input" goal G_1 and then tactic R_2 is applied to the resulting goal. The predicate *maptac* is used in this second application since the application of R_1 can result in a compound goal. This tactical plays a fundamental role in combining the results of step-by-step goal reduction. The *orelse* tactical attempts to apply either the tactic R_1 or the tactic R_2. The third tactical, *idtac*, simply returns the input goal unchanged. The *repeat* tactical is defined recursively using these previous three tacticals. The *try* tactical forms the reflexive closure of a given tactic. In operational terms, the tactic, *try* R, can be used to first reduce a goal using R and, if that fails, to simply return the given goal unchanged. Finally, the *complete* tactical succeeds if its given tactic can completely solve (that is, reduce to *truegoal*) a given goal. If a complete reduction is not possible, the given goal is returned unchanged. This tactical requires the auxiliary procedure *goalreduce* that simplifies compound goal expressions by removing occurrences of *truegoal* from them. Although the *complete* tactical is the only one that requires the use of the *goalreduce* procedure, it is also possible and probably desirable to modify the other tacticals so that they use it to similarly simplify their output goal structures whenever possible.

The examples above demonstrate how predicate variables can be used in logic programs. The higher-order clauses examined in the previous section, however, permitted predicate as well as function variables. Thus we can, for example, write the following higher-order Horn clauses that define the predicate *mapfun* of type $(i \to i) \to \mathit{list} \to \mathit{list} \to o$.

$$\begin{array}{l} \quad \forall f\,(\mathit{mapfun}\ f\ [\,]\ [\,]) \\ \forall f, x, l, k\,(\mathit{mapfun}\ f\ l\ k \supset \mathit{mapfun}\ f\ [x|l]\ [f\ x|k]) \end{array}$$

A goal $(\mathit{mapfun}\ f\ l\ k)$ is provable from these clauses if l and k are lists of the same length and if f is applied to a member in the first list the result is λ-convertible to the corresponding element in the second list. Thus, if g is a constant of type $i \to i \to i$ and a and b are constants of type i, then the goal

$$\exists l\,(\mathit{mapfun}\ \lambda x\,(g\ x\ x)\ [a, b]\ l)$$

has a proof with the existential witness $[(g\ a\ a), (g\ b\ b)]$. Similarly, the goal

$$\exists f\,(\mathit{mapfun}\ f\ [a, b]\ [(g\ a\ a), (g\ b\ b)])$$

has a proof with the existential witness $\lambda x\,(g\ x\ x)$. The kind of pattern matching that this example illustrates does not correspond to any conventional notions of higher-order programming that derive their origins from functional programming languages. In order for an implementation of higher-order Horn clause to be complete and thus find abstractions in this fashion, it must implement unification of simply typed λ-terms containing variables of functional (higher-order) types. Such unification is a significant enrichment of first-order term unification. It has been studied by various researchers (see, for example, [16,35]) and implementations of it have been used in various theorem proving and logic programming settings [2, 27, 29]. Further discussion of the use of functional variables in logic programming is, however, beyond the scope of this paper. The interested reader is referred to the papers [5, 12, 17, 23, 31].

10. Conclusion

In this paper, logic programming has been presented at a very high-level. Our focus has been on the declarative and proof-theoretic meaning of logic connectives and quantifiers. Unification, which plays a crucial role in understanding an actual interpreter or compiler of programming languages such as Prolog, was hardly mentioned in this paper. The reason for this is that our approach for providing Horn clauses with abstractions has been to understand various kinds of abstractions as phenomena of provability. Once abstractions can be understood in terms of provability, traditional theorem proving techniques can be employed to help implement such extensions to logic programs. From this point of view, unification is only an implementation technique used to interpret logic programs. In fact, implementing interpreters for the logics described in this paper requires roughly three different kinds of unification processes. Specifically, the programming languages outlined in Sections 3 and 4 requires the notion of unification of first-order terms. The languages described in Sections 5 and 6 require that usual first-order term unification be modified so that certain constants cannot appear in the substitution terms for certain free variables: this constraint is necessary to guarantee that constants introduced for universal quantifiers in goals are indeed "new." Finally, the languages described in Section 8 require the unification of simply typed λ-terms.

Clearly, there is a rather large gap between presenting logic systems as we have here and developing and implementing real programming languages that incorporate the enhanced logics. The experimental language, λProlog [27], is an attempt to base a Prolog-like language on higher-order hereditary Harrop formulas. λProlog uses a depth-first search discipline to implement

not only clause selection (used also by Prolog) but also unifier selection. Since the unification of simply typed λ-terms with functional variables may yield multiple unifiers, these must also be selected in some order and any choice here may need to be backtracked over. There are currently two implementation of λProlog: one implemented in Prolog [24] and one in Common Lisp [6].

The classes of formulas similar to first-order hereditary Harrop formulas have been developed by various other researchers from different points of view. For example, both Gabbay and Reyle [8] and McCarty [18, 19] are concerned with extending logic programming with hypothetical reasoning. Hallnäs and Schroeder-Heister [11] also use proof-theoretic arguments of a kind different than those used here to similarly extend Horn clauses.

Logics similar to higher-order hereditary Harrop formulas have also been used as meta languages in specifying and implementing theorem provers [5, 29, 30] and program transformation and manipulation systems [12, 13, 23].

Acknowledgements. I am very grateful to Eva Ma for her editorial comments on an earlier draft of this paper and to Felix Wu for catching several typos. The work reported here has been supported in part by grants ONR N00014-88-K-0633, NSF CCR-87-05596, and DARPA N00014-85-K-0018.

11. References

[1] P. Andrews, *An Introduction to Mathematical Logic and Type Theory*, Academic Press, 1986.

[2] P. Andrews, D. Miller, E. Cohen, and F. Pfenning, "Automating Higher-Order Logic" in *Automated Theorem Proving: After 25 Years*, AMS Contemporary Mathematics Series 29, 1984.

[3] K. Apt and M. van Emden, Contributions to the Theory of Logic Programming, Journal of the ACM 29 (1982) 841 – 862.

[4] A. Church, A Formulation of the Simple Theory of Types, Journal of Symbolic Logic 5 (1940) 56 – 68.

[5] A. Felty and D. Miller, Specifying Theorem Provers in a Higher-Order Logic Programming Language, Proceedings of the Ninth International Conference on Automated Deduction, Argonne, IL, 23 – 26 May1988, eds. E. Lusk and R. Overbeek, Springer-Verlag Lecture Notes in Computer Science, Vol. 310, 61 – 80.

[6] C. Elliott and F. Pfenning, eLP, a Common Lisp implementation of λProlog, January 1989.

[7] M. Fitting, *Intuitionistic Logic Model Theory and Forcing*, North-Holland Pub. Co., 1969.

[8] D. Gabbay and U. Reyle, N-Prolog: An Extension to Prolog with Hypothetical Implications. I, Journal of Logic Programming 1 (1984) 319 – 355.

[9] G. Gentzen, Investigations into Logical Deductions, in: M. E. Szabo (ed.), *The Collected Papers of Gerhard Gentzen*, North-Holland Publishing Co., Amsterdam, 1969, 68 – 131.

[10] M. J. Gordon, A. J. Milner, and C. P. Wadsworth, *Edinburgh LCF: A Mechanised Logic of Computation*, Lecture Notes in Computer Science, Vol. 78, Springer-Verlag, 1979.

[11] L. Hallnäs and P. Schroeder-Heister, A Proof-Theoretic Approach to Logic Programming. I. Generalized Horn Clauses (unpublished) 1988.

[12] J. Hannan and D. Miller, Uses of Higher-Order Unification for Implementing Program Transformers, Fifth International Conference and Symposium on Logic Programming, ed. K. Bowen and R. Kowalski, MIT Press, 1988, 942 – 959.

[13] J. Hannan and D. Miller, A Meta Language for Functional Programs, Proceedings of the 1988 Workshop on Meta Programming, Bristol, UK, eds. H. Rogers and H. Abramson, MIT Press (to appear).

[14] R. Harrop, Concerning Formulas of the types $A \rightarrow B \vee C$, $A \rightarrow (Ex)B(x)$ in Intuitionistic Formal Systems, Journal of Symbolic Logic, **25** (1960) 27 — 32.

[15] J. Hindley and J. Seldin, *Introduction to Combinators and λ-calculus*, Cambridge University Press, 1986.

[16] G. Huet, A Unification Algorithm for Typed λ-Calculus, Theoretical Computer Science 1 (1975) 27 – 57.

[17] G. Huet and B. Lang, Proving and Applying Program Transformations Expressed with Second-Order Logic, Acta Informatica 11 (1978) 31–55.

[18] L. McCarty, Clausal Intuitionistic Logic I. Fixed Point Semantics, Journal of Logic Programming 5 (1988) 1 – 31.

[19] L. McCarty, Clausal Intuitionistic Logic II. Tableau Proof Procedure, Journal of Logic Programming, 5 (1988) 93 – 132.

[20] D. Miller, A Logical Analysis of Modules in Logic Programming, Journal of Logic Programming 6 (1989) 79 – 108.

[21] D. Miller, Lexical Scoping as Universal Quantification, Sixth International Logic Programming Conference, Lisbon, June 1989, eds. G. Levi and M. Martelli, MIT Press, 268 – 283.

[22] D. Miller and G. Nadathur, Higher-Order Logic Programming, Proceedings of the Third International Logic Programming Conference, London, June 1986, ed. E. Shapiro, Sringer-Verlag, 448 – 462.

[23] D. Miller and G. Nadathur, A Logic Programming Approach to Manipulating Formulas and Programs, Proceedings of the IEEE Fourth

Symposium on Logic Programming, IEEE Press, 1987, 379 – 388.

[24] D. Miller and G. Nadathur, LP2.6 (August 1987) and LP2.7 (July 1988) implementations of λProlog, distribution in C-Prolog and Quintus Prolog source code.

[25] D. Miller, G. Nadathur, F. Pfenning, and A. Scedrov, Uniform Proofs as a Foundation for Logic Programming, Annals of Pure and Applied Logic (to appear).

[26] G. Nadathur, *A Higher-Order Logic as the Basis for Logic Programming*, Ph. D. dissertation, University of Pennsylvania, May 1987.

[27] G. Nadathur and D. Miller, An Overview of λProlog, Fifth International Conference on Logic Programming, eds. R. Kowlaski and K. Bowen, MIT Press, 1988, 810 – 827.

[28] G. Nadathur and D. Miller, Higher-Order Horn Clauses, Journal of the ACM (to appear).

[29] L. Pauslon, The Foundation of a Generic Theorem Prover, Journal of Automated Reasoning, Vol. 5, September 1989, 363 – 397.

[30] F. Pfenning, Partial Polymorphic Type Inference and Higher-Order Unification, Proceedings of the 1988 ACM Conference on Lisp and Functional Programming, ed. Jerome Chailloux, ACM Press, 153 – 163.

[31] F. Pfenning and C. Elliot, Higher-Order Abstract Syntax, Proceedings of the ACM-SIGPLAN Conference on Programming Language Design and Implementation, ACM Press, 1988, 199 – 208.

[32] D. Prawitz, *Natural Deduction*, Almqvist & Wiksell, Uppsala, 1965.

[33] C. Smorynski, Applications of Kripke models, pp. 324 – 391 in [36].

[34] L. Sterling and E. Shapiro, *The art of Prolog: advanced programming techniques*, MIT Press, Cambridge MA, 1986.

[35] W. Snyder and J. H. Gallier, Higher Order Unification Revisited: Complete Sets of Transformations, Journal of Symbolic Computation, Vol. 8, 1989, 101 – 140.

[36] A. Troelstra, *Metamathematical Investigations of Intuitionistic Arithmetic and Analysis*, Lecture Notes in Mathematics 344, Springer-Verlag, Berlin, 1973.