UNIFORM PROOFS AS A FOUNDATION FOR LOGIC PROGRAMMING

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

Gopalan Nadathur Computer Science Department
Duke University, Durham, NC 27706

Frank Pfenning Computer Science Department
Carnegie Mellon University, Pittsburgh, PA 15213

Andre Scedrov Mathematics Department
University of Pennsylvania, Philadelphia, PA 19104

Abstract: A proof-theoretic characterization of logical languages that form suitable bases for Prolog-like programming languages is provided. This characterization is based on the principle that the declarative meaning of a logic program, provided by provability in a logical system, should coincide with its operational meaning, provided by interpreting logical connectives as simple and fixed search instructions. The operational semantics is formalized by the identification of a class of cut-free sequent proofs called *uniform proofs*. A uniform proof is one that can be found by a goal-directed search that respects the interpretation of the logical connectives as search instructions. The concept of a uniform proof is used to define the notion of an *abstract logic programming language*, and it is shown that first-order and higher-order Horn clauses with classical provability are examples of such a language. Horn clauses are then generalized to *hereditary Harrop formulas* and it is shown that first-order and higher-order versions of this new class of formulas are also abstract logic programming languages if the inference rules are those of either intuitionistic or minimal logic. The programming language significance of the various generalizations to first-order Horn clauses is briefly discussed.

Appear in *Annals of Pure and Applied Logic*, 1991 (**51**), 125–157.

1. Introduction

Most logic programming languages can be thought of as implementations of the classical, first-order theory of Horn clauses. The language Prolog [29], for instance, is generally described using these formulas and its interpreter is based on SLD-resolution [1, 31]. Although the use of first-order Horn clauses in this capacity provides for a programming language that has many interesting features, it also results in a language that lacks most forms of abstractions commonly found in modern programming languages. For example, Horn clauses are not capable of supporting modular programming, abstract data types, or higher-order functions in either a direct or a natural fashion.

There are essentially three broad approaches that can be adopted to provide these missing features in a language such as Prolog. The first approach is to take programming constructs from other languages and to mix them into Horn clauses. For example, a notion of higher-order functions can be provided in Prolog by mixing some of the higher-order mechanisms of Lisp with Horn clauses [32, 33]. The second approach is to modify an existing interpreter in some simple ways so that the resulting interpreter has a behavior that can be utilized to provide aspects of the missing features. This is generally achieved in logic programming languages via the implementation of various non-logical primitives, such as *call*, *univ*, and *functor* [29]. Both of these approaches generally provide for immediate and efficient extensions to the language. However, they have the disadvantage that they clutter up of the semantics of the language, obscure the declarative readings of programs, and move the language far from its basis in logic.

The third approach, which we pursue here, involves extending the logic programming paradigm to include richer logics than Horn clauses in the hope that they provide a logical basis for the missing abstraction mechanisms. While this approach does not always lead to immediate and efficient solutions like the other two approaches, it generally has the advantage that the extended language continues to have a clear semantics. There are, however, greatly differing possibilities with regard to the logical systems that can be used in this context. The theory of Horn clauses appears to be close to one extreme of these possibilities. At the other extreme, there is the possibility of using full and unrestricted quantificational logic with a general purpose theorem prover serving as the interpreter for the resulting language. There is a need, therefore, for a criterion for determining whether a given logical theory is an adequate basis for a logic programming language before this third approach can be brought to fruition.

In this paper, we argue that there is, in fact, a natural criterion for making such a determination. The basic observation here is that logic programs are intended to specify *search behavior* and that this fact is just as central to logic programming as the fact that it makes use of symbolic logic for some of its syntax and metatheory. We attempt to formalize this search behavior through the identification of certain kinds of proofs, called *uniform proofs*, in sequent calculi without the cut inference rule. An *abstract logic programming language* is then characterized in terms of uniform proofs. In abstract logic programming languages, the declarative reading of the logical connectives coincides with the search related reading. The class of such languages includes classical first-order Horn clauses and excludes full quantificational logic as well as some other recently proposed extensions to Horn clauses. Fortunately, there are logics more expressive than first-order Horn clauses and apparently relevant to actual logic programming practice that are also captured by the notion of an abstract logic programming language. Three such logics are described in this paper.

The remainder of this paper is structured as follows. In the next section we provide the formal definition of a uniform proof and of an abstract logic programming language. We follow this up with a presentation of four different abstract logic programming languages: in Section 3, we present the first-order and higher-order versions of positive Horn clauses and in Sections 4 and 5 we present the first-order and higher-order versions of a new class of formulas called *hereditary Harrop formulas.* Section 6 contains a detailed proof that higher-order hereditary Harrop formulas interpreted using either intuitionistic or minimal logic form an abstract logic programming language. Some possible applications within logic programming for our various extensions are outlined in Section 7 and Section 8 contains some concluding remarks. An appendix collects together the various abstract logic programming languages defined in this paper and summarizes some of the observations made about them.

2. Uniform Proofs

The syntax of the logic used here corresponds closely to that of the simple theory of types [2]. In particular, our logic consists of types and simply typed λ-terms. The set of types contains a collection of primitive types or sorts and is closed under the formation of functional types: *i.e.*, if α and β are types, then so is $\alpha \rightarrow \beta$. The type constructor $\rightarrow$ associates to the right. We assume that there are denumerably many variables and constants of each type. Simply typed λ-terms are built up in the usual fashion from these typed constants and variables via abstraction and application. Application associates to the left.

It is assumed that the reader is familiar with most of the basic notions and definitions pertaining to substitution and λ-conversion for this language; only a few are reviewed here. Two terms are equal if they differ by only alphabetic changes in their bound variable names. A term is in *λ-normal form* if it contains no *β-redexes*, *i.e.* it has no subformulas of the form $(\lambda x\ B)C$. Every term can be β-converted to a unique λ-normal form, and we write $\lambda\text{norm}(t)$ to denote the λ-normal form corresponding to the term t. The notation $[C/x]B$ denotes the result of substituting C for each free occurrence of x in B and is defined formally to be $\lambda\text{norm}((\lambda x\ B)C)$.

Logic is introduced into these term structures by including o, a type for propositions, amongst the primitive types, and by requiring that the collection of constants contain the following typed *logical* constants: $\wedge, \vee, \supset$ all of type $o \rightarrow o \rightarrow o$; $\top$, $\bot$ of type o; and, for every type α, $\forall_\alpha$ and $\exists_\alpha$ both of type $(\alpha \rightarrow o) \rightarrow o$. These logical constants are also referred to as *logical connectives.* The type subcript on $\forall$ and $\exists$ will be omitted except when its value is essential in a discussion and cannot be inferred from context. Expressions of the form $\forall(\lambda x\ B)$ and $\exists(\lambda x\ B)$ will be abbreviated by $\forall x\ B$ and $\exists x\ B$, respectively. Terms of propositional type are referred to as *formulas.* The λ-normal form of a formula consists, at the outermost level, of a sequence of applications, and the leftmost symbol in this sequence is called its *top-level* symbol. *First-order formulas* are those formulas in λ-normal form that are obtained by only using variables that are of primitive type and nonlogical constants that are of type $\alpha_1 \rightarrow \ldots \rightarrow \alpha_n \rightarrow \alpha_0$ where $n \geq 0$, $\alpha_1, \ldots, \alpha_n$ are primitive types distinct from o, and α_0 is a primitive type (possibly o).

In various parts of this paper, we shall use the following syntactic variables with the corresponding general connotations:

$\mathcal{D}$	A set of formulas that serves as possible program clauses of some logic programming language.
$\mathcal{G}$	A set of formulas that serves as possible queries or goals for this programming language.
A	An atomic formula; that is, a formula whose top-level symbol is not a logical constant. $\top$ and $\bot$ are not atomic formulas.
A_r	A rigid atomic formula; that is, an atomic formula whose top-level symbol is not a variable.
D	A member of $\mathcal{D}$, referred to as a definite clause or a program clause.
G	A member of $\mathcal{G}$, referred to as a goal or query.
$\mathcal{P}$	A finite subset of formulas from $\mathcal{D}$, referred to as a (logic) program.

One meaningful relation that could be asked for arbitrary sets $\mathcal{D}$ and $\mathcal{G}$ is the following. Given some notion of logical provability (such as classical or intuitionistic provability) denoted by $\vdash$, is it the case that $\mathcal{P} \vdash G$? This notion of provability could be used to state that the goal G succeeds given program $\mathcal{P}$. There are at least two reasons why this very general notion of success is unsatisfactory as a foundation for logic programming.

First, in an abstract sense, computation in the logic programming setting means goal-directed search. Therefore, the primitives of the programming language should specify how to build and traverse a search space. Since we are trying to provide a *logical* foundation for logic programming, these primitives should be the logical connectives. As we shall see, the meaning of logical connectives in a very general provability setting does not easily support a search-related interpretation and we will have to look for a more restricted notion of provability.

Second, the result of a computation in logic programming is generally something that is extracted from the proof of a goal from a program. Typically, this extraction is a substitution or witness, called an *answer substitution*, for the existentially quantified variables in the goal formula. For it to be possible to make such an extraction, the provability relation over programs should satisfy the *existential property*; that is, whenever $\mathcal{P} \vdash \exists x\, G$, there should be some term t such that $\mathcal{P} \vdash [t/x]G$. An answer substitution is then the substitution for the existentially quantified variables of a goal that are contained in a proof of that goal. Again, many provability relations do not satisfy this property if $\mathcal{D}$ and $\mathcal{G}$ are taken to be arbitrary sets of formulas.

The definition of a proof theoretic concept that captures this notion of computation-as-search can be motivated by describing how a simple, non-deterministic interpreter (theorem prover) for programs and goals should function. This interpreter, given the pair $\langle \mathcal{P}, G \rangle$ in its initial state, should either succeed or fail. We shall use the notation $\mathcal{P} \vdash_O G$ to indicate the (meta) proposition that the interpreter succeeds if started in the state $\langle \mathcal{P}, G \rangle$. The subscript on $\vdash_O$ signifies that this describes an "operational semantics" (of an idealized interpreter). The search-related semantics that we want to attribute to the logical constants $\top, \wedge, \vee, \supset, \forall$, and $\exists$ can then be informally specified by associating with them the following six search instructions.

SUCCESS	$\mathcal{P} \vdash_O \top$.
AND	$\mathcal{P} \vdash_O G_1 \wedge G_2$ only if $\mathcal{P} \vdash_O G_1$ and $\mathcal{P} \vdash_O G_2$.
OR	$\mathcal{P} \vdash_O G_1 \vee G_2$ only if $\mathcal{P} \vdash_O G_1$ or $\mathcal{P} \vdash_O G_2$.
INSTANCE	$\mathcal{P} \vdash_O \exists_\alpha x\, G$ only if there is some term t of type α such that $\mathcal{P} \vdash_O [t/x]G$.
AUGMENT	$\mathcal{P} \vdash_O D \supset G$ only if $\mathcal{P} \cup \{D\} \vdash_O G$.
GENERIC	$\mathcal{P} \vdash_O \forall_\alpha x\, G$ only if $\mathcal{P} \vdash_O [c/x]G$, where c is a parameter of type α that is not free in $\mathcal{P}$ or in G.

Thus, the logical constant $\top$ simply signifies a successfully completed search. The logical connectives $\wedge$ and $\vee$ provide for the specification of non-deterministic AND and OR nodes in the interpreter's search space. The quantifier $\exists$ specifies an infinite non-deterministic OR branch where the disjuncts are parameterized by the set of all terms. Implication instructs the interpreter to augment its program, and universal quantification instructs the interpreter to introduce a new parameter and to try to prove the resulting generic instance of the goal.

There are several points to be noted with respect to the search instructions above.

First, they only partially specify the behavior of an idealized interpreter since they do not describe a course of action when atomic goals need to be solved. In each of the examples considered in this paper, a natural choice turns out to be the operation of backchaining. This might not, however, be the most general choice in all cases and building it into the definition could make it unduly restrictive. Second, the search instructions specify only the success/failure semantics for the various connectives and do not address the question of what the result of a computation should be. The abstract interpreter must solve existentially quantified goals by providing specific instances for the existential quantifiers and the instantiations that are used can be provided, as usual, as the result of a computation. As outlined at the end of this paper, an actual interpreter that extracts answer substitutions from uniform proofs can be constructed for each of the logic programming languages considered here. We have chosen not to build in this notion of answer substitution into the description of an abstract interpreter in order to provide as broad a framework as possible. A point of particular interest is that the abstract interpreter is specified in a manner completely independent of any notion of unification: free variables that appear in goals are not variables in the sense that substitutions can be made for them; substitutions are made only when quantifiers are instantiated. Finally, we note that some of the naturalness, from a logical point of view, of the search instructions arises from the fact that their converses are true in most logical systems. They are true, for instance, within minimal logic, the weakest of the logical systems that we consider below.

Our concern with logic programming languages that contain the constant $\perp$ will be minimal in this paper. This is largely because $\perp$ contributes little to our understanding of abstractions in logic programs. This symbol is useful within logic programming to provide a notion of negation: see [15] for a development of this notion. In the present context it is important to point out that the natural tendency to read $\perp$ as failure does not correspond to the role of this symbol within logical systems: in the inference systems corresponding to classical, intuitionistic, and minimal logic that are considered below, $\perp$ is something that is provable when there is a contradiction in the assumptions.

In formalizing the behavior of the idealized interpreter, we shall find sequent-style proof systems a useful tool. We assume, once again, a familiarity with the basic notions of such proof systems and we summarize only a few of these below. A *sequent* is a pair of finite (possibly empty) sets of formulas $\langle \Gamma, \Theta \rangle$ that is written as $\Gamma \longrightarrow \Theta$. Proofs for sequents are constructed by putting them together using inference figures. Figure 1 contains the various inference figures needed in this paper. The proviso that the parameter c is not free in any formula of the lower sequent is assumed for the $\exists$-L and $\forall$-R figures. Also, in the inference figure λ, the sets Γ and Γ' and the sets Θ and Θ' differ only in that zero or more formulas in them are replaced by some formulas to which they are β-convertible. A *proof* for the sequent $\Gamma \longrightarrow \Theta$ is then a finite tree constructed using these inference figures and such that the root is labeled with $\Gamma \longrightarrow \Theta$ and the leaves are labeled with *initial sequents*, *i.e.*, sequents $\Gamma \longrightarrow \Theta$ such that either $\top \in \Theta$ or the intersection $\Gamma \cap \Theta$ contains either $\bot$ or an atomic formula. Sequent systems of this kind generally have three *structural* figures, which we have not listed. Two of these figures, interchange and contraction, are not necessary here because the antecedents and succedents of sequents are taken to be sets instead of lists. Hence, the order and multiplicity of formulas in sequents is not important. If an antecedent is of the form Γ, B, it may be the case that $B \in \Gamma$; that is, a formula in an antecedent or succedent has an arbitrary multiplicity. The third common structural inference figure is that of thinning. The definition of initial sequents above removes the need for this inference figure.

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

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

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

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

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

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

$$\frac{\Gamma \longrightarrow \Theta, \bot}{\Gamma \longrightarrow \Theta, B}\ \bot\text{-R} \qquad \frac{\Gamma' \longrightarrow \Theta'}{\Gamma \longrightarrow \Theta}\ \lambda$$

Figure 1: Inference figures

We define the following three kinds of proofs. An arbitrary proof will be called a **C**-proof. A **C**-proof in which each sequent occurrence has a singleton set for its succedent is also called an **I**-proof. Finally, an **I**-proof that contains no instance of the $\bot$-R inference figure is also called an **M**-proof. We write $\Gamma \vdash_C B$, $\Gamma \vdash_I B$, and $\Gamma \vdash_M B$, if the sequent $\Gamma \longrightarrow B$ has, respectively, a **C**-proof, an **I**-proof, and an **M**-proof. If the set Γ is empty, it will be dropped entirely from the left side of these three relations. The three relations defined here correspond to provability in, respectively, higher-order classical, intuitionistic and minimal logic. More detailed discussions of these kinds of sequent proof systems and their relationship to other presentations of the corresponding logics can be found in [5, 9, 28, 30]. Of particular note here is the use of the cut-elimination theorems for these various logics to identify $\vdash_C, \vdash_I, \vdash_M$ with the customary definitions of these provability relations.

A *uniform proof* is an **I**-proof in which each occurrence of a sequent whose succedent contains a non-atomic formula is the lower sequent of the inference figure that introduces its top-level connective. In other words, a uniform proof is an **I**-proof such that, for each occurrence of a sequent $\Gamma \longrightarrow G$ in it, the following conditions are satisfied:

- If G is $\top$, that sequent is initial.
- If G is $B \wedge C$ then that sequent is inferred by $\wedge$-R from $\Gamma \longrightarrow B$ and $\Gamma \longrightarrow C$.
- If G is $B \vee C$ then that sequent is inferred by $\vee$-R from either $\Gamma \longrightarrow B$ or $\Gamma \longrightarrow C$.
- If G is $\exists x\, P$ then that sequent is inferred by $\exists$-R from $\Gamma \longrightarrow [t/x]P$ for some term t.
- If G is $B \supset C$ then that sequent is inferred by $\supset$-R from $B, \Gamma \longrightarrow C$.
- If G is $\forall x\, P$ then that sequent is inferred by $\forall$-R from $\Gamma \longrightarrow [c/x]P$, where c is a parameter that does not occur in the given sequent.

The notion of a uniform proof reflects the search instructions associated with the logical connectives. We can, in fact, formalize $\vdash_O$ by saying that $\mathcal{P} \vdash_O G$, *i.e.*, the interpreter succeeds on the goal G given the program $\mathcal{P}$, if and only if there is a uniform proof of the sequent $\mathcal{P} \longrightarrow G$. An *abstract logic programming language* is then defined as a triple $\langle \mathcal{D}, \mathcal{G}, \vdash \rangle$ such that for all finite subsets $\mathcal{P}$ of $\mathcal{D}$ and all formulas G of $\mathcal{G}$, $\mathcal{P} \vdash G$ if and only if $\mathcal{P} \vdash_O G$.

We shall presently consider examples of abstract logic programming languages. Before we do this, however, we describe two logical systems that are not included by this definition. First, let us take for $\mathcal{D}$ the set of positive Horn clauses extended by permitting the antecedents of implications to contain negated literals, for $\mathcal{G}$ the existential closure of the conjunctions of atoms, and for $\vdash$ the notion of classical provability (see, for example, [6]). The resulting system fails to be an abstract logic programming language under our definition since

$$p \supset q(a),\ \neg p \supset q(b) \vdash_C \exists x\, q(x)$$

although there is no term t such that $q(t)$ follows from the same program (antecedent). Thus, there is no uniform proof for $\exists x\, q(x)$ from the program $\{p \supset q(a),\ \neg p \supset q(b)\}$. For another example, let $\mathcal{D}$ be the set of positive and negative Horn clauses, let $\mathcal{G}$ be the set of negations of such formulas, and let $\vdash$ once again be classical provability (see, for example, [8]). This system again fails to be an abstract logic programming language since

$$\neg p(a) \vee \neg p(b) \vdash_C \exists x\, \neg p(x)$$

although no particular instance of the existentially quantified goal can be proved.

3. Horn Clauses

Horn clauses are generally defined in the literature 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 presentation of Horn clauses is motivated by the fact that its simple syntactic nature simplifies the description of resolution theorem provers. Our analysis of provability will be based on sequent proofs rather than on resolution refutations. We therefore prefer to use the following more natural definition of this class of formulas.

Let A be a syntactic variable that ranges over atomic, first-order formulas. Let $\mathcal{G}_1$ be the collection of first-order formulas defined by the following inductive rule:

$$G := \top \mid A \mid G_1 \wedge G_2 \mid G_1 \vee G_2 \mid \exists x\, G.$$

Similarly, let $\mathcal{D}_1$ be the collection of first-order formulas defined by the following inductive rule:

$$D := A \mid G \supset A \mid D_1 \wedge D_2 \mid \forall x\, D.$$

Notice that any formula of $\mathcal{D}_1$ can be rewritten as a conjunction of some list of positive Horn clauses by uses of prenexing, anti-prenexing, and deMorgan laws. Similarly, every positive Horn clause can be identified with some formula in $\mathcal{D}_1$. Given this correspondence and the additional fact that we do not allude to negative Horn clauses anywhere in our discussions below, we shall refer to the formulas in $\mathcal{D}_1$ as *first-order Horn clauses.* This definition of (positive) Horn clauses is in fact more appealing than the traditional one for several reasons. First, it is textually closer to the kind of program clauses actually written by Prolog programmers: these programmers do not write lists of signed literals and they often insert disjunctions into the bodies of clauses. Second, it provides for a more compact notation, given that the size of the conjunctive normal form of a formula can be exponentially larger than the size of the formula. Finally, this presentation of Horn clauses will be easier to generalize when, as in the next section, we desire to include additional connectives inside program clauses.

Let *fohc* $= \langle \mathcal{D}_1, \mathcal{G}_1, \vdash_C \rangle$. We then have the following theorem.

Theorem 1. *fohc is an abstract logic programming language.*

Proof. Only a sketch is provided here since most of the details are present in the proof of a more general theorem in Section 6. Let $\mathcal{P}$ and G be an *fohc* program and goal, respectively. If $\mathcal{P} \vdash_O G$, then the uniform proof of $\mathcal{P} \longrightarrow G$ is also a **C**-proof, and thus $\mathcal{P} \vdash_C G$. For the converse, we may assume that $\mathcal{P} \longrightarrow G$ has a **C**-proof, say Ξ, with no instances of the λ inference figure and then show that it must also have a uniform proof. An argument for this is outlined as follows:

(1) Let $\mathcal{A}$ be a finite subset of $\mathcal{D}_1$ and $\mathcal{B}$ be a finite subset of $\mathcal{G}_1$ such that $\mathcal{A} \longrightarrow \mathcal{B}$ has a **C**-proof. A simple inductive argument shows that the antecedent and succedent of each sequent occurrence in this proof are subsets of $\mathcal{D}_1$ and $\mathcal{G}_1 \cup \{\bot\}$, respectively. By virtue of Proposition 3 below, this derivation can be transformed into one in which the inference figure $\bot$-R does not appear, essentially by "omitting" occurrences of $\bot$-R and by "padding" the succedents of sequents above this inference figure appropriately. We may, therefore, assume without loss of generality that the succedent of each sequent in the derivation under consideration is a subset of $\mathcal{G}_1$ and, further, that the inference figure $\bot$-R does not appear in it. An inductive argument now shows that there is some formula G' in $\mathcal{B}$ such that $\mathcal{A} \longrightarrow G'$ has an **M**-proof. From this observation, it follows that there is an **M**-proof Ξ' for $\mathcal{P} \longrightarrow G$; when spelt out in detail, the above argument shows how Ξ' can be obtained from the given derivation Ξ.

(2) Given the nature of the antecedents and succedents of its sequents, it is clear that the only inference figures that appear in Ξ' are $\wedge$-L, $\supset$-L, $\forall$-L, $\wedge$-R, $\vee$-R, and $\exists$-R. If Ξ' is not already a uniform proof, then this is because the introduction rules for connectives on the right might not appear as close to the root as required: instances of $\wedge$-L, $\supset$-L, and $\forall$-L might come between an occurrence of a sequent with a compound formula in its succedent and the inference figure where that formula's top-level connective was introduced. It is possible to prove, again by an inductive argument, that the inference rules $\wedge$-L, $\supset$-L, and $\forall$-L commute with $\wedge$-R, $\vee$-R, and $\exists$-R. Thus, the **M**-proof Ξ' can be converted into a uniform proof of the same sequent, and thus $\mathcal{P} \vdash_O G$. ▮

The transformation outlined above implicitly shows the equivalence of classical, intuitionistic, and minimal provability for sequents of the form $\mathcal{P} \longrightarrow G$ where $\mathcal{P} \subseteq \mathcal{D}_1$ and $G \in \mathcal{G}_1$. It follows, therefore, that the triples $\langle \mathcal{D}_1, \mathcal{G}_1, \vdash_I \rangle$ and $\langle \mathcal{D}_1, \mathcal{G}_1, \vdash_M \rangle$ are also abstract logic programming languages. They are, in fact, the same abstract logic programming languages as *fohc* since the sets of sequents of the form $\mathcal{P} \longrightarrow G$ that have **C**-, **I**-, **M**-, and uniform proofs are the same.

fohc extends the logical language underlying Prolog slightly, since it allows explicit existential quantifiers and does not require normal forms. It is, however, relatively weak as a logic programming language in the sense that there are no cases for the introduction of implications or universal quantifiers into succedents. Put equivalently, an interpreter for this language does not need to implement the search operations AUGMENT and GENERIC. There are many ways in which the languages of programs and goals, *i.e.*, the classes $\mathcal{D}_1$ and $\mathcal{G}_1$, can be strengthened in order to obtain richer abstract logic programming languages, and we examine some of these here. In this section we examine the possibility of permitting higher-order terms and formulas in Horn clauses. In the next section we shall be concerned with allowing implication and universal quantification in goals.

In introducing higher-order notions into Horn clauses, our approach will be 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. To define such a class of formulas precisely, let us first identify $\mathcal{H}_1$ as 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 can 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 $Pt_1 \ldots t_n$, where P is either a variable or non-logical constant. A is said to be *rigid* if P is a constant, and the syntactic variable A_r is used for rigid atomic formulas in $\mathcal{H}_1$. We now let $\mathcal{G}_2$ be the set of all formulas in $\mathcal{H}_1$ and, in a manner akin to the definition of $\mathcal{D}_1$, we let $\mathcal{D}_2$ be the set of formulas satisfying the following rule:

$$D := A_r \mid G \supset A_r \mid D_1 \wedge D_2 \mid \forall x\, D.$$

The quantification here may be over higher-order variables, and G ranges over the formulas of $\mathcal{H}_1$. Notice that a closed, non-atomic formula in $\mathcal{G}_2$ must be either $\top$, or have $\wedge, \vee$, or $\exists$ as its top-level constant. The formulas in $\mathcal{D}_2$ are what we call *higher-order Horn clauses.*

The restriction that the atomic formulas appearing in the "conclusions" of higher-order Horn clauses be rigid is motivated by two considerations. First, given the operational interpretation that is generally accorded to a Horn clause, the predicate symbol of the atom in the conclusion of an implication is the name of a *procedure* that that clause is helping to define. Requiring this predicate symbol to be a constant forces each such implication to be part of the definition of some specific procedure. Second, this requirement also makes it impossible for a collection of higher-order Horn clauses to be inconsistent in the sense that arbitrary formulas can be proved from it. This observation follows from Proposition 3 below. In fact, a sequent of the form $\mathcal{P} \longrightarrow A_r$ is provable only if the top-level predicate constant of A_r is also the top-level predicate constant of the conclusion of some implication in $\mathcal{P}$. If the condition on occurrences of predicate variables is relaxed, however, programs can become inconsistent. For instance, arbitrary formulas are provable from the set $\{p, \forall x(p \supset x)\}$.

Let *hohc* $= \langle \mathcal{D}_2, \mathcal{G}_2, \vdash_C \rangle$. We wish to show then that *hohc* is an abstract logic programming language. The proof that was provided for *fohc* does not carry over immediately to this case: since predicates may be quantified upon, it is possible to construct **C**-proofs in the context of *hohc* that are much more complex than in the first-order case. For example, consider the following derivation of the goal formula $\exists y\ Py$ from the higher-order Horn clause $\forall x\ (x \supset Pa)$. We assume here that q is of type o and that there is some primitive type i such that P is of type $i \rightarrow o$, and a and b are of type i.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{Pb \longrightarrow q, Pb}{\longrightarrow Pb \supset q, Pb}\ \supset\text{-R} \qquad Pa \longrightarrow Pa}{(Pb \supset q) \supset Pa \longrightarrow Pa, Pb}\ \supset\text{-L}}{(Pb \supset q) \supset Pa \longrightarrow Pa, \exists y\ Py}\ \exists\text{-R}}{(Pb \supset q) \supset Pa \longrightarrow \exists y\ Py}\ \exists\text{-R}}{\forall x\ (x \supset Pa) \longrightarrow \exists y\ Py}\ \forall\text{-R}$$

This derivation illustrates that an important observation made in the proof of Theorem 1 is not true in the higher-order case: in a **C**-proof of a sequent of the form $\mathcal{P} \longrightarrow G$, there may appear sequents that contain non-Horn clauses such as $(Pb \supset q) \supset Pa$ in their antecedents. Furthermore, this is a proof of a sequent with an existential formula in its succedent in which the existential quantifier results from generalizing on two different constants; the formula $\exists y\ Py$ in the final sequent is obtained by generalizing on Pa and Pb. To show *hohc* is an abstract logic programming language, we must be able to show that there is an alternative **C**-proof with a single witnessing term for the existentially quantified goal.

The proof of Theorem 1 can be adapted to this case if it is possible to show that in a **C**-proof of a sequent in *hohc* it is only necessary to use substitution terms taken from the set $\mathcal{H}_1$. This would help because of the following observation: if t is a member of $\mathcal{H}_1$ and x is a variable of the same type as t, then $[t/x]s$ is a member of $\mathcal{H}_1$ for each s that is a member of $\mathcal{H}_1$ and, similarly, $[t/x]D$ is a member of $\mathcal{D}_2$ for each D that is a member of $\mathcal{D}_2$. If it could be proved that substitutions from only this set need to be considered, then we can, in fact, restrict our attention to **C**-proofs in which the antecedent and succedent of each sequent occurrence is a subset of $\mathcal{D}_1$ and $\mathcal{G}_1 \cup \{\bot\}$, respectively. The rest of the argument in the proof of Theorem 1 would then carry over to this case as well.

In [22] and [24], it is shown that this restriction on substitutions within **C**-proofs in fact preserves the set of provable sequents. That is, if $\mathcal{P}$ is a set of higher-order Horn clauses and G is a formula in $\mathcal{G}_2$ such that $\mathcal{P} \longrightarrow G$ has a **C**-proof, then this sequent has a **C**-proof in which the substitution terms used in the inference figures $\forall$-L and $\exists$-R are from the set $\mathcal{H}_1$. Hence, $\mathcal{H}_1$ acts as a kind of Herbrand universe for *hohc*. In proving this fact, a procedure is described for transforming arbitrary **C**-proofs to **C**-proofs of this restricted variety. This procedure would, for instance, transform the above problematic proof into the following proof of the same sequent.

$$\dfrac{\dfrac{\dfrac{\dfrac{\longrightarrow\ \top, Pb \qquad Pa\ \longrightarrow\ Pa}{\top \supset Pa\ \longrightarrow\ Pa, Pb}\ \supset\text{-L}}{\top \supset Pa\ \longrightarrow\ Pa, \exists y\ Py}\ \exists\text{-R}}{\top \supset Pa\ \longrightarrow\ \exists y\ Py}\ \exists\text{-R}}{\forall x\ (x \supset Pa)\ \longrightarrow\ \exists y\ Py}\ \forall\text{-R.}$$

The substitution term $Pb \supset q$ has been replaced here by the simpler term $\top$. Although the existential quantifier $\exists y\ Py$ is still instantiated twice in the proof, the arguments in the proof of Theorem 1 permit the removal of the "unnecessary" instance, Pb. These observations are made precise in the following theorem.

Theorem 2. *hohc is an abstract logic programming language.*

Proof. Again, only an outline is provided. Details can be found in [22, 24], and the proof in Section 6 is also similar in spirit.

Let $\mathcal{P}$ be a finite set of higher-order Horn clauses and let G be a formula in $\mathcal{G}_2$. The theorem is trivial in one direction, since a uniform proof of $\mathcal{P}\ \longrightarrow\ G$ is also a **C**-proof. For the other direction, assume that $\mathcal{P}\ \longrightarrow\ G$ has a **C**-proof Ξ. We argue below that this sequent must then also have a uniform proof.

Let Ξ' be the result of dropping all sequents in Ξ that occur above the lower sequent of an instance of either $\supset$-R or $\forall$-R. In general, Ξ' will not be a proof, since its leaves might not be initial sequents. Because of the effect of higher-order substitutions, the formulas that occur in the antecedents of sequents of Ξ' might not be members of $\mathcal{D}_2$. However, let B, L_r, and L be syntactic variables for formulas described in the following manner: B denotes arbitrary formulas, L_r denotes rigid atomic formulas not necessarily in $\mathcal{H}_1$, and L denotes formulas defined inductively by

$$L := L_r \mid B \supset L_r \mid L_1 \wedge L_2 \mid \forall x\ L.$$

It is easily seen then that formulas in the antecedents of sequents in Ξ' are L-formulas.

Now let τ be the mapping on λ-terms that first replaces all occurrences of $\supset$, $\forall$, and $\perp$ with the terms $\lambda x \lambda y\ \top$, $\lambda w\ \top$, and $\top$, respectively, and then converts the result into λ-normal form. Obviously, $\tau(t)$ is a member of $\mathcal{H}_1$ for any term t, and if $t \in \mathcal{H}_1$ then $\tau(t) = t$. Now consider two additional mappings defined using τ. For arbitrary formulas B, let $\tau^+(B) := \tau(B)$; τ^+ maps arbitrary formulas into $\mathcal{G}_2$, while preserving formulas in

$\mathcal{G}_2$. On the class of L-formulas, let τ^- by the mapping given by the following recursion: $\tau^-(L_r) := \tau(L_r)$, $\tau^-(B \supset L_r) := \tau(B) \supset \tau(L_r)$, $\tau^-(L_1 \wedge L_2) := \tau^-(L_1) \wedge \tau^-(L_1)$, and $\tau^-(\forall x\ L) := \forall x\ \tau^-(L)$; τ^- maps L-formulas into $\mathcal{D}_2$, preserving, again, the formulas already in $\mathcal{D}_2$.

Ξ' is now transformed into Ξ'' by the following operation: apply τ^- to each formula in the antecedent of the sequents in Ξ' and apply τ^+ to each formula in the succedent of the sequents in Ξ'. It can be shown that Ξ'' is a **C**-proof of $\mathcal{P} \longrightarrow G$ (since $\tau^+(G) = G$ and $\tau^-(D) = D$ for each $D \in \mathcal{P}$) in which all substitution terms in $\exists$-R and $\forall$-L are from $\mathcal{H}_1$. Furthermore, every sequent in Ξ'' has an antecedent that is a subset of $\mathcal{D}_2$ and a succedent that is a subset of $\mathcal{G}_2 \cup \{\bot\}$. The proof of this theorem can now be completed by arguments that are identical to those used in the proof of Theorem 1. ∎

The transformation implicit in the above proof once again indicates the equivalence of classical, intuitionistic, and minimal provability for sequents of the form $\mathcal{P} \longrightarrow G$ where $\mathcal{P} \subseteq \mathcal{D}_2$ and $G \in \mathcal{G}_2$. Thus replacing the proof relation in the definition of *hohc* by either $\vdash_I$ or $\vdash_M$ results in an abstract logic programming language that is identical to *hohc*.

Before concluding this section, we observe the following Proposition concerning the L-formulas described in the proof of Theorem 2. An immediate consequence of this Proposition is that any finite set of L-formulas is consistent. Since first-order and higher-order Horn clauses are L-formulas, it follows therefore that finite sets of these formulas are also consistent. This fact was used in the proofs of Theorem 1 and 2. We shall use this proposition again in a similar fashion in Section 6.

Proposition 3. *Let Γ be a finite set of L-formulas. Then there can be no* **C***-proof for* $\Gamma \longrightarrow \bot$.

Proof. Consider the set of all **C**-proofs of sequents of the form $\Gamma \longrightarrow \bot$, where Γ is a finite set of L-formulas. Assume this set is non-empty and let Ξ be a proof in this set of minimal height. Ξ cannot be an initial sequent since $\bot$ is not an L-formula. Thus, the last inference figure of Ξ must be either $\supset$-L, $\forall$-L, or $\wedge$-L. This is, however, impossible since, in all of these cases, Ξ would contain a proof of $\bot$ from a finite set of L-formulas as a subproof, contradicting the choice of Ξ. ∎

Although the availability of higher-order terms in *hohc* makes this language more expressive than *fohc* (see Section 7), an interpreter for this language still does not implement the AUGMENT and GENERIC search operations. In the next section, we present our first example of an abstract logic programming language that incorporates these additional search operations.

4. First-Order Hereditary Harrop Formulas

We return to first-order logic in this section in order to present an abstract logic programming language that makes stronger use of logical connectives. We shall presently consider a language in which goal formulas may have occurrences of all six logical constants that have been given an operational interpretation. We note first that unless the occurrences and combinations of these connectives are greatly restricted, classical logic cannot be expected to provide a proof system for an abstract logic programming language with such goal formulas. For example, consider the goal formula $p \vee (p \supset q)$. If this goal was given to our idealized interpreter with the program being empty, the interpreter would fail: there is no uniform proof for either p or q from p. There is, however, a **C**-proof for this formula, namely

$$\cfrac{\cfrac{\cfrac{p \longrightarrow p, q}{\longrightarrow p, p \supset q}\ \supset\text{-R}}{\longrightarrow p \vee (p \supset q), p \supset q}\ \vee\text{-R}}{\longrightarrow p \vee (p \supset q)}\ \vee\text{-R.}$$

Thus classical provability is too strong for specifying the behavior of an interpreter that implements the AUGMENT search operation. For this purpose it is necessary to choose a logic weaker than classical logic, and intuitionistic logic and minimal logic appear to be possible choices.

Let us consider now the class of first-order formulas given by the following recursive definition of the syntactic variable D:

$$D := A \mid B \supset D \mid \forall x\, D \mid D_1 \wedge D_2.$$

We assume here that A denotes atomic first-order formulas and B denotes arbitrary first-order formulas. These D-formulas are known as *Harrop formulas* [11, 30]. The syntax of these formulas can be simplified slightly by noting the following equivalences for minimal logic: $B \supset (D_1 \wedge D_2) \equiv (B \supset D_1) \wedge (B \supset D_2)$ and $B \supset \forall x\, D \equiv \forall x\, (B \supset D)$ provided x is not free in B. Thus, an inductive definition of Harrop formulas that is equivalent in both intuitionistic and minimal logic to the one above can be given by

$$D := A \mid B \supset A \mid \forall x\, D \mid D_1 \wedge D_2.$$

Since this definition resembles the earlier definitions of program clauses, we shall prefer it over the former one. Obviously, all first-order Horn clauses are Harrop formulas.

Let $\mathcal{P}$ be a finite set of Harrop formulas and let B be some non-atomic formula. An important property of Harrop formulas that was shown in [11] is the following: if

$\mathcal{P} \vdash_I B$ then there is an **I**-proof of $\mathcal{P} \longrightarrow B$ in which the last inference rule introduces the logical connective of B. Hence, an **I**-proof of a sequent whose succedent is a set of Harrop formulas can be made uniform "at the root". However, this observation does not imply the existence of a uniform proof whenever there is an **I**-proof since such a proof of $\mathcal{P} \longrightarrow B$ can contain sequents whose antecedents are not sets of Harrop formulas; the above-mentioned property cannot, therefore, hold at these sequents. For example, let $\mathcal{P}$ be the empty set of Harrop formulas and B be the formula $(p \vee q) \supset (q \vee p)$. There is no uniform proof of the resulting sequent. Thus, the triple consisting of Harrop formulas, arbitrary formulas, and intuitionistic logic is not an abstract logic programming language. These observations carry over to the case where the proof relation considered is that of minimal logic.

To motivate the definition of a triple that does constitute an abstract logic programming language, let us consider the behavior of our idealized interpreter when given a program consisting of Harrop formulas and a goal that is an arbitrary formula. As long as the program remains a set of Harrop formulas, the search interpretation of the top-level logical connective of the goal is consistent with the intuitionistic or minimal logic meaning of that connective. The program might, however, become non-Harrop if the AUGMENT rule is used. To avoid this problem, we stipulate that a formula whose top-level connective is an implication may appear as a goal only if the formula to the left of the implication is itself a Harrop formula. A similar restriction must also apply to the antecedent of program clauses whose top-level logical connective is an implication, since the $\supset$-L inference figure moves this into the succedent of the upper sequent. The G- and D-formulas given by the following following mutual recursion meet these conditions:

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

Let $\mathcal{D}_3$ be the set of D-formulas and let $\mathcal{G}_3$ be the set of G-formulas. A formula from $\mathcal{D}_3$ is called a *first-order hereditary Harrop* formula [14], and *fohh* is defined to be the triple $\langle \mathcal{D}_3, \mathcal{G}_3, \vdash_I \rangle$. We have the following theorem concerning this triple.

Theorem 4. *fohh is an abstract logic programming language.*

Proof. Again we only outline the proof. A similar theorem is proved in [15] and the proof in Section 6 extends the proof outlined here.

Let $\mathcal{P}$ be a finite subset of $\mathcal{D}_3$ and let G be a member of $\mathcal{G}_3$. Since any uniform proof is also an **I**-proof, $\mathcal{P} \vdash_O G$ implies $\mathcal{P} \vdash_I G$.

Assume that $\mathcal{P} \vdash_I G$ and let Ξ be an **I**-proof of $\mathcal{P} \longrightarrow G$. The only reason why Ξ might not be a uniform proof is because the introduction rules for the connectives in formulas in the succedent are not as close to the root as required: instances of $\wedge$-L, $\supset$-L, and $\forall$-L might have come between an occurrence of a sequent with a compound formula in its succedent and the inference figure where the top-level connective of that formula was introduced. Now, from the discussion above, it follows that each sequent that occurs in Ξ has an antecedent that is a subset of $\mathcal{D}_3$ and a succedent whose sole element is a member of $\mathcal{G}_3$. Using this observation and the fact that the inference rules $\wedge$-L, $\supset$-L, and $\forall$-L commute with $\wedge$-R, $\vee$-R, $\supset$-R, $\forall$-R, and $\exists$-R, an inductive argument can show how Ξ can be converted to a uniform proof and, therefore, $\mathcal{P} \vdash_O G$. ▮

A point to note is that, by virtue of Proposition 3, the transformation described in the above proof actually produces a uniform proof that is also a minimal proof. Thus the triple $\langle \mathcal{D}_3, \mathcal{G}_3, \vdash_M \rangle$ amounts to the same logic programming language as *fohh*. As another point, we note that an interpreter for *fohh* must implement all six search operations described in Section 2, since the corresponding six logical constants can appear in goal formulas. Such an interpreter would subsume an interpreter for *fohc*.

There is an interesting relationship between *fohh* and *fohc*. Let $\mathcal{M}$ be the intersection of the sets $\mathcal{D}_3$ and $\mathcal{G}_3$. This class can be defined a the set of all formulas satisfying the following inductive definition:

$$M := A \mid M \supset A \mid \forall x\, M \mid M_1 \wedge M_2.$$

Notice that $\mathcal{M}$ contains all formulas of the form $\forall x_1 \ldots \forall x_m [A_1 \wedge \ldots \wedge A_n \supset A_0]$ where $n \geq 0$ and $m \geq 0$. Thus members of $\mathcal{D}_1$ are equivalent (in minimal logic) to members of $\mathcal{M}$. In this sense, Horn clauses can be both program clauses *and* goal formulas in *fohh*. For an example of why this fact might be useful, consider the goal formula $M \wedge G$. This is equivalent (in intuitionistic and minimal logic) to the goal formula $M \wedge (M \supset G)$. Hence, a goal of the form $M \wedge G$ can be replaced by the one of the form $M \wedge (M \supset G)$. Proofs of the latter goal can be much shorter than of the former since the fact that M is provable is "stored" during the attempt to prove G. The clause M could be used several times without the need to prove it each time it is used.

For more examples of the use of *fohh* to write a richer set of programs than is possible with *fohc*, the reader is referred to [7, 15, 17].

5. Higher-Order Hereditary Harrop Formulas

We now wish to describe a higher-order version of hereditary Harrop formulas. There are certain choices that need to be made in order to do this. A choice that is of particular importance concerns the form of logic that is to be allowed to be embedded within atomic formulas. One proposal is to permit all the connectives that could appear as the top-level symbols of *fohh* goal formulas to appear within atomic formulas. As we shall see presently, a language described in this manner does not constitute an abstract logic programming language since it contains theorems of minimal logic that do not have uniform proofs. It is actually instructive to analyze this unsuccessful proposal carefully, and we do this below. Before doing so, however, we present a higher-order extension of hereditary Harrop formulas that is, in fact, successful.

Let $\mathcal{H}_2$ be the set of λ-normal terms that do not contain occurrences of the logical constants $\supset$ and $\perp$. In other words, the only logical constants that terms in $\mathcal{H}_2$ may contain are $\top$, $\land$, $\lor$, $\forall$, and $\exists$. Let the syntactic variable A denote atomic formulas in $\mathcal{H}_2$ and let the syntactic variable A_r denote rigid atomic formulas in $\mathcal{H}_2$. Then $\mathcal{G}_4$ and $\mathcal{D}_4$ are, respectively, the sets of G and D-formulas that are defined by the following mutual recursion:

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

Quantification here may be over higher-order variables. The formulas of $\mathcal{D}_4$ will be called *higher-order hereditary Harrop formulas.* Letting *hohh* $= \langle \mathcal{D}_4, \mathcal{G}_4, \vdash_I \rangle$, we note the following theorem, whose proof is the subject of the next section.

Theorem 5. *hohh is an abstract logic programming language.*

In trying to understand the nature of the abstract logic programming language *hohh*, it is useful to consider briefly the behavior of the idealized interpreter in the context of *hohc*. We note first that atoms with predicate variables as their top-level symbol might appear in goal formulas. By making substitutions for such variables, the interpreter can move embedded logical connectives into the top-level logical structure of these formulas. However, the interpreter cannot similarly alter atomic program clauses or the formulas to the right of the implication symbol in these clauses; in the latter case, the top-level logical structure of only the goal formula to the left of the implication symbol can be altered. Thus, in the *hohc* case, every connective that is permitted in goal formulas may also be allowed to appear within the terms that constitute the arguments of atomic formulas: since only such terms may ultimately be substituted for predicate variables, the interpreter would in this case only produce goal formulas from goal formulas and program clauses from program clauses. This aspect of *hohc* is in fact reflected in the definitions of the goal formulas in $\mathcal{G}_2$ and the universe of "substitution" terms $\mathcal{H}_1$: $\mathcal{G}_2$ is exactly the set of formulas in $\mathcal{H}_1$.

In the definition of higher-order hereditary Harrop formulas, however, this close relationship between goal formulas and substitution terms does not hold: the set $\mathcal{G}_4$ is strictly larger than the set of formulas in $\mathcal{H}_2$. In particular, goal formulas can contain implications while the terms in $\mathcal{H}_2$ cannot. Relaxing this restriction by permitting implications into atomic formulas can result in "goal" formulas that are theorems of minimal logic but which do not have uniform proofs. For example, consider the formula

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

where R is a constant of type $o \rightarrow o$, s and t are constants of type o, Q is a variable of type $o \rightarrow o \rightarrow o$, and p and q are constants of type o. This formula has exactly one **M**-proof, obtained by using $\lambda x \lambda y(x \supset y)$ as the substitution term for the existentially quantified 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. The source of the "problem" in this example can be analyzed as follows. The subformula $t \vee s$ has a positive occurrence in the original formula to be proved. However, substituting the term containing an implication for Q produces a formula in which $t \vee s$ has a negative occurrence. The resulting formula cannot be a goal formula within the framework of *hohh* since it contains a formula with an $\vee$ as its top-level symbol at a place where only program clauses are permitted. The presence of implications in substitution terms may transform positive occurrences of formulas into negative occurrences and can consequently lead to the existence of only non-uniform proofs. One way in which this situation can be prevented from occurring is by making it unnecessary for implications to appear in substitution terms. This can be achieved by not permitting implications inside atomic formulas. This is the reason for not permitting implications in the members of $\mathcal{H}_2$.

One possible use for higher-order features in a programming language is in letting part of a computation build a program 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 abstract logic programming languages described herein. Consider, for example, the sequent

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

where d and g are some (specific) formulas of type o and Q is a variable of type o. Consistent with discussions in this paper, the antecedent of this sequent can be thought of as a program and the succedent as a goal. Let us now assume that the clauses in $\mathcal{P}$ define the relation *compile* between two terms such that the latter term is a program clause obtained by compiling the information in the former term. Then, the above sequent can be thought of as describing the following kind of computation: compile the term d to construct a program clause Q, and use this new clause in trying to solve the goal g. Such a computation would correspond rather directly to some of the computations that can be performed in Lisp because of the presence of the function *eval*. Unfortunately this is not a computation that can be described even in *hohh*, the richest of the languages considered here: the succedent of this sequent is not a valid goal formula of *hohh* since the formula $Q \supset g$ that appears in it has a non-rigid atom to the left of the implication.

The requirement that atoms in certain places be rigid and the restriction that implications not occur embedded in atoms might, in practice, be a hindrance to a programmer. A possible solution to this problem is to remove these restrictions, thus permitting sequents such as the one above to define legal computations. This might seldom conflict with the structure of the abstract interpreter in practice; for instance, in the specific example under consideration it might be the case that the the program $\mathcal{P}$ defines *compile* in such a way that it relates only legal program clauses to d. Whether or not this is true requires, in general, establishing rather deep properties about user-defined programs — in our example it would require establishing a property of *compile* — a task that might be very hard to carry out in general. An implementation of a higher-order "version" of hereditary Harrop formulas might forego making such a determination statically and, instead, signal a runtime error when it encounters sequents whose antecedents are not Harrop formulas; thus, in our example it should signal a runtime error if Q is instantiated with a formula whose top-level symbol is a disjunction or existential quantifier. Looked at in this light, Theorem 5 guarantees that if the language is restricted to being in *hohh*, no such runtime errors will occur.

6. Proof of Uniformity

The objective of this section is to prove Theorem 5, that is, to show that $\langle \mathcal{D}_4, \mathcal{G}_4, \vdash_I \rangle$ is an abstract logic programming language. For this purpose it is convenient to introduce the notion of an $\mathbf{M}'$-proof.

Definition 6. An $\mathbf{M}'$-proof is an $\mathbf{M}$-proof in which each occurrence of a $\forall$-L or an $\exists$-R figure constitutes a generalization upon a term from $\mathcal{H}_2$. In other words, in each appearance of a figure of one of the following two forms,

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

it is the case that $t \in \mathcal{H}_2$. Within $\mathbf{M}'$-proofs, $\mathcal{H}_2$ acts as a kind of Herbrand universe. ∎

In this section, we shall refer to formulas from $\mathcal{D}_4$ as D-formulas and to formulas from $\mathcal{G}_4$ as G-formulas. The discussions below can be understood, in this context, in the following manner. Let Γ be a finite collection of D-formulas and let G be a G-formula such that $\Gamma \longrightarrow G$ has an $\mathbf{I}$-proof. We show then that this $\mathbf{I}$-proof can be transformed into a uniform proof for the same sequent. This transformation is effected by a two step process. The first step consists of obtaining an $\mathbf{M}'$-proof from the given $\mathbf{I}$-proof, and the second step involves the extraction of a uniform proof from the resulting $\mathbf{M}'$-proof. While these respective transformations are not presented explicitly, they will be apparent from the constructive nature of the proofs to Lemmas 10 and 11 that appear below.

In elucidating the first step in the transformation process, the following mapping from arbitrary terms to terms in $\mathcal{H}_2$ is used.

Definition 7. Let x and y be variables of type o. Then the function *pos* on terms is defined as follows:

(i) If F is a constant or a variable

$$pos(F) = \begin{cases} \lambda x \lambda y\ \top, & \text{if } F \text{ is } \supset; \\ \top, & \text{if } F \text{ is } \bot; \\ F, & \text{otherwise.} \end{cases}$$

(ii) $pos([F_1\ F_2]) = [pos(F_1)\ pos(F_2)]$.

(iii) $pos(\lambda z\ F) = \lambda z\ pos(F)$.

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

This "positivization" operation on terms commutes with λ-conversion, as is proved in the following lemma.

Lemma 8. *For any terms F_1 and F_2, if F_1 λ-converts to F_2 then $pos(F_1)$ λ-converts to $pos(F_2)$.*

Proof. Clearly, if F_1 α-converts to F_2, then $pos(F_1)$ α-converts to $pos(F_2)$. Let x be a variable possibly free in B, and let A be a term of the same type as x that is also free for x in B. Let H_2 be the result of replacing all occurrences of a variable x in the term B by the term A. An induction on the structure of B verifies the following: $pos(H_2)$ results from substituting $pos(A)$ for all occurrences of x in $pos(B)$. Thus, if H_2 results by a β-reduction step from H_1, where $H_1 = (\lambda x\ B)A$, then $pos(H_2)$ results from $pos(H_1)$ by a similar step. This observation together with an induction on the structure of F_1 verifies that if F_2 results from F_1 by a single β-reduction step, then $pos(F_2)$ results similarly from $pos(F_1)$. An induction on the conversion sequence now verifies the lemma. ▮

It is necessary to consider below the result of performing a sequence of substitutions into a term. In order to avoid an excessive use of parentheses, we adopt the convention that substitution is a right associative operation: for example, $[t_2/x_2][t_1/x_1]F$ denotes the term that is obtained by first substituting t_1 for x_1 in F and then substituting t_2 for x_2 in the result.

Lemma 9. *If F is a term in $\mathcal{H}_2$ and $t_1, \ldots, t_n$ are arbitrary terms ($n \geq 0$), then*

$$([t_n/x_n] \ldots [t_1/x_1]F)^+ = [(t_n)^+/x_n] \ldots [(t_1)^+/x_1]F.$$

In particular, this is true when F is an atomic G- or D-formula.

Proof. Using the definition of substitution, the properties of λ-conversion and Lemma 8, it is easily seen that

$$([t_n/x_n]\ldots[t_1/x_1]F)^+ = [(t_n)^+/x_n]\ldots[(t_1)^+/x_1]F^+.$$

For any term $F \in \mathcal{H}_2$, it is evident that $pos(F) = F$ and, hence, that $F^+ = F$. ▮

The basis for the first step in the transformation alluded to above is now provided by the following lemma; as mentioned already, the actual mechanism for carrying out this step should be apparent from its proof.

Lemma 10. *If* Γ *is a finite set of* D*-formulas and* G *is a* G*-formula, the sequent* $\Gamma \longrightarrow G$ *has an* **I***-proof only if it also has an* **M′***-proof.*

Proof. Let Δ be a set of the form

$$\{[t^1_{n_1}/x^1_{n_1}]\ldots[t^1_1/x^1_1]D_1,\ldots,[t^r_{n_r}/x^r_{n_r}]\ldots[t^r_1/x^r_1]D_r\},$$

where $r, n_1, \ldots, n_r \geq 0$ and $D_1, \ldots, D_r$ are D-formulas; *i.e.*, Δ is a set of formulas each member of which is obtained by performing a sequence of substitutions into a D-formula. In this context, let Δ^+ denote the set

$$\{[(t^1_{n_1})^+/x^1_{n_1}]\ldots[(t^1_1)^+/x^1_1]D_1,\ldots,[(t^r_{n_r})^+/x^r_{n_r}]\ldots[(t^r_1)^+/x^r_1]D_r\}.$$

Given any G-formula G, we claim that $\Delta \longrightarrow [s_m/y_m]\ldots[s_1/y_1]G$ has an **I**-proof only if $\Delta^+ \longrightarrow [(s_m)^+/y_m]\ldots[(s_1)^+/y_1]G$ has an **M′**-proof. The lemma follows easily from this claim.

The claim is proved by an induction on the height of an **I**-proof for a sequent of the form $\Delta \longrightarrow [s_m/y_m]\ldots[s_1/y_1]G$. If this height is 1, the given sequent must be an initial sequent. It is easily seen that performing substitutions into a D-formula produces an L-formula (see Section 3). Thus Δ is a set of L-formulas and, consequently, $\bot \notin \Delta$. Therefore, $[s_m/y_m]\ldots[s_1/y_1]G$ must be either $\top$ or an atomic formula. But then G must itself be $\top$ or an atomic formula, and so, by Lemma 9,

$$[(s_m)^+/y_m]\ldots[(s_1)^+/y_1]G = ([s_m/y_m]\ldots[s_1/y_1]G)^+.$$

Now if $\Delta \longrightarrow [s_m/y_m]\ldots[s_1/y_1]G$ is an initial sequent because $[s_m/y_m]\ldots[s_1/y_1]G = \top$, then $\Delta^+ \longrightarrow [(s_m)^+/y_m]\ldots[(s_1)^+/y_1]G$ must also be an initial sequent since $(\top)^+ = \top$. Otherwise for some i, $1 \leq i \leq r$, it is the case that

$$[s_m/y_m]\ldots[s_1/y_1]G = [t^i_{n_i}/x^i_{n_i}]\ldots[t^i_1/x^i_1]D_i.$$

Since D_i must be an atomic formula here, it follows, using Lemma 9, that

$$([s_m/y_m]\dots[s_1/y_1]G)^+ = ([t^i_{n_i}/x^i_{n_i}]\dots[t^i_1/x^i_1]D_i)^+ = [(t^i_{n_i})^+/x^i_{n_i}]\dots[(t^i_1)^+/x^i_1]D_i.$$

Thus $\Delta^+ \longrightarrow [(s_m)^+/y_m]\dots[(s_1)^+/y_1]G$ must again be an initial sequent.

For the inductive case, we assume that the claim is true for sequents of the requisite sort that have derivations of height h and then verify it for sequents with derivations of height $h+1$. For this purpose, we consider the possible cases for the last inference figure in such a derivation. This figure cannot be a $\bot$-R: If it were, then $\Delta \longrightarrow \bot$ would have an **I**-proof, contradicting Proposition 3 since Δ, as we have observed, must be a set of L-formulas. Further, the figure in question cannot be an $\vee$-L or an $\exists$-L, since, once again, an L-formula cannot have either an $\vee$ or a $\exists$ as its top-level connective. Finally, a simple induction on the heights of derivations shows that if a sequent consists solely of formulas in λ-normal form, then any **I**-proof for it that contains the inference figure λ can be transformed into a shorter **I**-proof in which λ does not appear. Since each formula in $\Delta \longrightarrow [s_m/y_m]\dots[s_1/y_1]G$ is in λ-normal form, we can assume that the last inference figure in its **I**-proof is not a λ. Thus, the only figures that need to be considered are $\wedge$-L, $\supset$-L, $\forall$-L, $\wedge$-R, $\vee$-R, $\supset$-R, $\forall$-R, and $\exists$-R.

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

$$\frac{\Delta \longrightarrow B \qquad\qquad \Delta \longrightarrow C}{\Delta \longrightarrow B \wedge C}$$

In this case, $B \wedge C = [s_m/y_m]\dots[s_1/y_1]G$. Depending on the structure of G, our analysis breaks up into two parts:

(1) G is an atomic formula. From Lemma 9 it follows that

$$[(s_m)^+/y_m]\dots[(s_1)^+/y_1]G = (B \wedge C)^+ = B^+ \wedge C^+.$$

Now B and C can be written as $[B/y]y$ and $[C/y]y$, respectively. It therefore follows from the hypothesis that $\Delta^+ \longrightarrow B^+$ and $\Delta^+ \longrightarrow C^+$ have $\mathbf{M}'$-proofs. Using an $\wedge$-R figure in conjunction with these, we obtain an $\mathbf{M}'$-proof for $\Delta^+ \longrightarrow B^+ \wedge C^+$.

(2) G is a non-atomic formula. In this case G must be of the form $G_1 \wedge G_2$, where G_1 and G_2 are G-formulas. Hence $B = [s_m/y_m]\dots[s_1/y_1]G_1$ and $C = [s_m/y_m]\dots[s_1/y_1]G_2$. It follows from the hypothesis that the sequents $\Delta^+ \longrightarrow [(s_m)^+/y_m]\dots[(s_1)^+/y_1]G_1$ and $\Delta^+ \longrightarrow [(s_m)^+/y_m]\dots[(s_1)^+/y_1]G_2$ must both have $\mathbf{M}'$-proofs, and therefore

$$\Delta^+ \longrightarrow [(s_m)^+/y_m]\dots[(s_1)^+/y_1]G_1 \wedge [(s_m)^+/y_m]\dots[(s_1)^+/y_1]G_2$$

must also have one. It is now only necessary to note that the succedent of the last sequent is in fact $[(s_m)^+/y_m]\dots[(s_1)^+/y_1]G$ to see that the claim must be true.

An analogous argument can be provided when the last figure is $\vee$-R. If it is an $\supset$-R, then last inference figure in the given **I**-proof must be of the form

$$\frac{B, \Delta \longrightarrow C}{\Delta \longrightarrow B \supset C}$$

where $B \supset C = [s_m/y_m] \dots [s_1/y_1]G$. If G is an atomic formula, then $G^+ = G$ and

$$[(s_m)^+/y_m] \dots [(s_1)^+/y_1]G = (B \supset C)^+ = \top,$$

and $\Delta^+ \longrightarrow [(s_m)^+/y_m] \dots [(s_1)^+/y_1]G$ has a trivial $\mathbf{M}'$-proof. If G is a non-atomic formula, it must be of the form $D' \supset G'$ for some G-formula G' and some D-formula D'. An argument similar to that in (2) above now verifies the claim. For the cases $\wedge$-L and $\supset$-L, we observe first that if the result of performing a sequence of substitutions into a D-formula D is a formula of the form $B \wedge C$, then D must be the conjunction of two D-formulas; if such an instance is of the form $B \supset C$, then D must be of the form $G' \supset D'$ where G' is a G-formula and D' is a D-formula. In each of these cases, the claim is now verified by invoking the hypothesis and by mimicking the argument in (2) above.

If the last figure in the derivation in an $\exists$-R, then it must be of the form

$$\frac{\Delta \longrightarrow [t/x]P}{\Delta \longrightarrow \exists x\ P}$$

where $\exists x\ P = [s_m/y_m] \dots [s_1/y_1]G$. We assume, without loss of generality, that x is distinct from the variables $y_1, \dots, y_m$ as well as the variables that are free in $s_1, \dots, s_m$ and consider, once again, two subcases based on the structure of G. If G is an atomic formula, it follows from Lemma 9 that

$$[(s_m)^+/y_m] \dots [(s_1)^+/y_1]G = (\exists x\ P)^+ = \exists x\ (P)^+.$$

Writing $[t/x]P$ as $[t/x][P/y]y$ and invoking the hypothesis, we see that $\Delta^+ \longrightarrow [t^+/x]P^+$ has an $\mathbf{M}'$-proof. Adding below this an $\exists$-R figure we obtain, as required, an $\mathbf{M}'$-proof for $\Delta^+ \longrightarrow \exists x\ (P)^+$. If, on the other hand, G is a non-atomic formula, it must be of the form $\exists x\ G'$ where G' is a G-formula. But now $P = [s_m/y_m] \dots [s_1/y_1]G'$. Thus, $\Delta^+ \longrightarrow [t^+/x][(s_m)^+/y_m] \dots [(s_1)^+/y_1]G'$ has an $\mathbf{M}'$-proof by the hypothesis, and from this we can obtain one for $\Delta^+ \longrightarrow \exists x\ ([(s_m)^+/y_m] \dots [(s_1)^+/y_1]G')$. Noting that $\exists x\ ([(s_m)^+/y_m] \dots [(s_1)^+/y_1]G')$ is in fact $[(s_m)^+/y_m] \dots [(s_1)^+/y_1]\exists x\, G'$, the claim is verified in this case as well.

The only remaining cases are those when the last inference figure is a $\forall$-L or a $\forall$-R. In both these cases, an argument that is similar to the one for $\exists$-R can be provided: for the case of $\forall$-R, it is only necessary to make the additional observation that if a variable y is free in a formula of the form $[(u_l)^+/z_l]\dots[(u_1)^+/z_1]F$, then it must also be free in $[u_l/z_l]\dots[u_1/z_1]F$. ▮

A direct consequence of the above lemma is the equivalence of provability in intuitionistic and minimal logics in the context of $\mathcal{D}_4$ and $\mathcal{G}_4$ formulas. Of more immediate concern to us is that it permits us to focus on $\mathbf{M}'$-proofs for the sequents of interest in this section. The proof of the following lemma outlines a mechanism for transforming such a derivation into a uniform proof.

Lemma 11. *Let Γ be a finite set of D-formulas and let G be a G-formula. If $\Gamma \longrightarrow G$ has an $\mathbf{M}'$-proof then it also has a uniform proof.*

Proof. The proof of the lemma is based on an observation and a claim. First the observation: In an $\mathbf{M}'$-proof for a sequent of the sort described in the lemma, the antecedent of every sequent is a set of D-formulas and the succedent is a G-formula. The observation may be confirmed by an induction on the height of the $\mathbf{M}'$-proof for such a sequent. It is certainly the case for a derivation of height 1. Given a derivation of height $h+1$ we consider the possibilities for the last inference figure. A routine inspection suffices to confirm the observation in all cases except perhaps for $\forall$-L and $\exists$-R. In the latter two cases it follows by noting that if $t \in \mathcal{H}_2$ then $[t/x]P$ is a D-formula if P is a D-formula, and $[t/x]P$ is a G-formula if P is a G-formula.

Now for the claim: Let Δ be an arbitrary set of D-formulas, and let G' be a G-formula such that $\Delta \longrightarrow G'$ has an $\mathbf{M}'$-proof of height h. Then

(*i*) if $G' = G_1 \wedge G_2$ then $\Delta \longrightarrow G_1$ and $\Delta \longrightarrow G_2$ have $\mathbf{M}'$-proofs of height less than h,

(*ii*) if $G' = G_1 \vee G_2$ then either $\Delta \longrightarrow G_1$ or $\Delta \longrightarrow G_2$ has an $\mathbf{M}'$-proof of height less than h,

(*iii*) if $G' = \exists x\, G_1$ then there is a $t \in \mathcal{H}_2$ such that $\Delta \longrightarrow [t/x]G_1$ has an $\mathbf{M}'$-proof of height less than h,

(*iv*) if $G' = D \supset G_1$ then $\Delta \cup \{D\} \longrightarrow G_1$ has an $\mathbf{M}'$-proof of height less than h, and

(*v*) if $G' = \forall x\, G_1$ then there is a parameter c that appears neither in $\forall x\, G_1$ nor in any of the formulas in Δ for which $\Delta \longrightarrow [c/x]G_1$ has an $\mathbf{M}'$-proof of height less than h.

This claim is proved by inducing again on the height of the derivation for $\Delta \longrightarrow G'$. If this height is 1, it is vacuously true. For the case when the height is $h+1$, we consider the possibilities for the last inference figure. The argument is trivial when this is one of $\wedge$-R, $\vee$-R, $\exists$-R, $\supset$-R, and $\forall$-R. Consider then the case for $\forall$-L, *i.e.*, when the last figure is of the form

$$\frac{[t/x]P, \Theta \longrightarrow G'}{\forall x\, P, \Theta \longrightarrow G'}$$

The argument in this case depends on the structure of G'. For instance, let $G' = \forall x\, G_1$. By the observation, the upper sequent of the above figure is of the requisite sort for the hypothesis to apply. Hence, there is a parameter c that does not appear in any of the formulas in Θ or in $[t/x]P$ or in $\forall x\, G_1$ for which $[t/x]P, \Theta \longrightarrow [c/x]G_1$ has an $\mathbf{M}'$-proof of height less than h. Adding below this derivation a $\forall$-L inference figure, we obtain an $\mathbf{M}'$-proof of height less than $h + 1$ for $\forall x\, P, \Theta \longrightarrow [c/x]G_1$. To verify the claim in this case, it is now only necessary to observe that c cannot be free in $\forall x\, P$ if it is not already free in $[t/x]P$. The analysis for the cases when G' has a different structure follows an analogous pattern. Further, similar arguments can be provided when the last inference figure is an $\wedge$-L or an $\supset$-L, thus completing the proof of the claim.

The lemma follows immediately from the observation and the claim above. In particular, the proof of the claim outlines a mechanism for moving the inference figure that introduces the top-level logical connective in G to the end of the $\mathbf{M}'$-proof. In conjunction with the observation, this amounts to a method for transforming an $\mathbf{M}'$-proof of $\Gamma \longrightarrow G$ into a uniform proof for the same sequent. ▮

Theorem 5, the main result that we sought to establish in this section, is an immediate consequence of Lemmas 10 and 11.

7. Abstractions in Logic Programs

As we mentioned in the introduction, first-order Horn clauses lack any direct and natural expression of the standard abstraction mechanisms that are found in most modern computer programming languages. One of the goals of our investigation into extensions to logic programming was to provide a logical foundation for introducing some of these abstraction mechanisms into the logic programming idiom. Below we briefly describe how the extensions we have considered can help account for three program-level abstraction mechanisms: modules, abstract data types, and higher-order programming. We also describe a new kind of abstraction mechanism, called *term-level* abstraction, that is available in *hohc* and *hohh* because of the possibility of using λ-terms to represent data objects. Our discussions are brief, but we provide references to places where fuller discussions can be found.

Modules. An interpreter for the *fohc* or *hohc* logics need not implement the AUGMENT search operation. Thus, to evaluate a goal in a complex program, all parts of the program must be present at the start of the computation, whether or not the use of bits of code could be localized or encapsulated. For example, let D_0, D_1 and D_2 represent bundles or modules of code. In Horn clause logic programming languages, to attempt goals G_1 and G_2 with respect to these three program modules, the interpreter must be called with the sequent

$$D_0, D_1, D_2 \longrightarrow G_1 \wedge G_2.$$

Even if we, as programmers, know that the code in D_1 is needed only in attempting to solve G_1 and the code in D_2 is needed only in attempting to solve G_2, there is no way to represent this structural fact using Horn clauses.

Implications within goals can, however, be used to provide for this missing structuring mechanism. The following goal could be attempted in either *fohh* or *hohh*:

$$\longrightarrow D_0 \supset (D_1 \supset G_1) \wedge (D_2 \supset G_2).$$

In trying to solve this goal, the interpreter would reduce this sequent into the two sequents

$$D_0, D_1 \longrightarrow G_1 \quad \text{and} \quad D_0, D_2 \longrightarrow G_2.$$

From this it is clear that the attempt to solve the two goals G_1 and G_2 would be made in the context of two different programs. The ability to impose such a structure on code might have a variety of uses. For instance, in the above example we see that the code in modules D_1 and D_2 cannot interfere with each other. This is particularly useful if these two modules are to be written by different programmers: the programmer of D_1 need not be concerned about the predicate names (that is, procedure names) and actual code that appears D_2. This structuring mechanism should also help in establishing formal properties of large programs since explicit use can be made of the fact that certain parts of the code do not interfere with others.

A more detailed description of such an approach to modules in logic programming can be found in [15]. It is of interest to note a peculiar characteristic of the notion of module that is provided by the use of the AUGMENT operation. A facet of many programming languages that provide a module construct is that the meaning of a procedure occurring within a module is given entirely by the code lying within that module. However, in the logic programming setting described here, procedures in modules can always be augmented. For instance, in the example above the code in module D_0 is AUGMENTed with the code in D_1 before an attempt is made to prove G_1. If D_1 contains code for a procedure that also has code in D_0, that procedure would be defined by the accumulation of the code in D_0 and D_1. While there are occasions when this accumulation is desirable, it also gives rise to a notion of modules unlike that found in other programming languages. Modules are not "closures" since it is always possible for a goal to have the current program modules augmented by any legal program clauses. This aspect of modules seems to be related to what is known as the closed-world/open-world problem of logic programming [3]. The simplest proof-theoretic approaches to explicating the meaning of logic programs, such as the ones used here, generally require an open-world interpretation of programs due to the basic monotonicity of the provability relation. In practice, however, programmers often wish to think of the programs they write as being closed. There are several techniques that have been used to impose a closed-world interpretation on logic programs and some of these approaches might be used to force the notion of modules here to be more like the module facilities of other programming languages.

Abstract data types. One problem with traditional logic programming languages is that it is very hard to limit access to the internal structure of data objects. For example, it is not possible to naturally support abstract data types in Horn clause logics. Use of universal quantifiers in goals, however, provides for a degree of security that can be used to support abstract data types [17]. Consider the following concrete example. Let *Sort* be a set of Horn clauses that implements a binary tree sorting algorithm and let us assume that it internally builds a labeled binary trees using the ternary function symbol f and the constant symbol r; thus it might use $f(5, f(3, r, r), f(8, r, r))$ to represent the three-node tree with a root that is labeled with 5 and that has left and right children labeled with 3 and 8, respectively. Now, assume that we attempt the goal $G(z)$ (of the one free variable z) that makes a call to the sorting procedure in *Sort* and that we would like it to be guaranteed that the goal $G(z)$ does not produce an answer (via a substitution term for z) that incorporates the binary tree constructors used within *Sort*. Such a behavior can be produced by letting $D(x, y)$ be the result of replacing every occurrence of the parameters f and r in *Sort* with new free variables x and y, respectively, and then attempting the goal

$$\exists z \forall x \forall y [D(x, y) \supset G(z)].$$

The alternation of quantifiers will not permit the substitution term for z to contain instances of the constants used by the GENERIC search operation. These constants can, however, play the role of being the data constructors of the sorting module.

Higher-order programming. It is possible to specify many of the higher-order operations that are familiar to Lisp and ML programmers in *hohc* and *hohh*. Operations such as mapping a function or predicate over a list or "reducing" one list using a binary function or predicate have very simple definitions. For example, consider the universal closures of the higher-order formulas,

$$Pxy \wedge map\ P\ l\ k \supset map\ P\ (cons\ x\ l)\ (cons\ y\ k) \quad \text{and}$$
$$map\ P\ nil\ nil,$$

where *cons* and *nil* are the constructors for the list data type. The meaning given by these clauses to the predicate *map* is that the goal *map P l k* is provable if l and k are lists of the same length and corresponding members of these lists are P-related. This example is, of course, higher-order because of the quantification over the predicate P. Such higher-order programs are quite common among higher-order functional programming languages. For more examples of higher-order logic programs in this style, see [19, 22].

Data Objects with Variable Bindings. There are many programming tasks that involve the manipulations of structures containing variable bindings. For example, formulas contain internal quantifications and programs contain formal parameters and local variables. In order to write such manipulation programs in a first-order logic programming language, these objects must be encoded as first-order terms. The operations of substitution and notions of bound and free variables must also be implemented prior to writing the various programs desired. If the variable-binding constructs of a language are directly represented in the λ-calculus (in effect using *higher-order abstract syntax*, see [27]), the notions of free and bound variables and the operation of substitution for such terms is part of the meta-theory of the logic programming language itself and would not have to be reimplemented each time it is needed. See [4, 18, 25, 26] for examples of using higher-order logic programming techniques to implement theorem provers and type inference programs, and see [10, 13, 20, 22] for examples of program transformation programs.

8. Conclusion

In this paper we have surveyed our recent attempts at extending the logic programming paradigm by strengthening its logical foundations. The guiding principle in all these attempts has been that a particular success/failure semantics for the logical connectives $\top$, $\wedge$, $\vee$, $\supset$, $\forall$, and $\exists$ is at the heart of logic programming. We have described three extensions to first-order Horn clauses in which this duality in the meaning of the connectives is preserved. From a programming perspective, these extensions are of interest because they lead to the addition of different forms of abstraction to a language such as Prolog.

Although motivated by practical considerations, the discussions in this paper have been largely of a foundational nature and have been a little divorced from the pragmatic issues underlying the design of actual programming languages. It is, in fact, for this reason that the term *abstract logic programming language* is used to describe the extensions proposed herein. There is much work to be done before the ultimate impact that these extensions on the design of Prolog-like languages can be assessed. One aspect whose importance cannot be overemphasized is that of experimenting with actual interpreters for languages based on the extensions. The abstract interpreter used to explicate the search semantics for the logical connectives provides much of the structure of such an interpreter, but there are certain points at which its behavior is described only nondeterministically and at these points the actions of an actual interpreter must be further specified. In some cases, it might be possible to describe devices that permit specific choices to be delayed. The best example of this is the choice of a substitution term in the case of the INSTANCE search operation that can be delayed by the use of free ("logical") variables and the concomitant notion of unification. In other cases, explicit choices might have to be made and mechanisms for doing so need to be specified. Exercising such choices might, in general, result in an interpreter that is incomplete as a proof procedure, and the significance of this fact must be evaluated from a practical standpoint.

It merits mention that issues pertinent to the design of actual interpreters for the various abstract logic programming languages in this paper have received attention elsewhere. A less "abstract" interpreter for *fohc*, the logical language underlying Prolog, is obtained from the well-known notion of SLD-resolution [1] and this forms the basis for implementations of Prolog. Although *hohc* is substantially more complex than *fohc*, a higher-order version of Prolog can be implemented in a fashion that shares many features with first-order Prolog systems. For example, an interpreter for *hohc* can be described in terms of higher-order unification, *i.e.* the unification of simply typed λ-terms perhaps containing higher-order variables, and SLD-resolution [22, 24]. As shown in [22], the usual backtracking mechanism of Prolog can be smoothly integrated with the higher-order unification search process described in [12] to provide interpreters for *hohc*. Similar ideas can be used in implementing *hohh* with the following main differences: there might be a need to solve unification problems that have mixed quantifier prefixes, an aspect dealt with in [16], and it is necessary to consider programs that change in a stack-based fashion. A prototype implementation of most of *hohh* has, in fact, been built based on this approach [23]. It has been used in several programming experiments [4, 10, 18, 20, 26] that have provided an understanding of the usefulness of the various abstraction mechanisms discussed in this paper in actual programming tasks.

As a final note, we observe that the focus in this paper has been on the realization of abstraction mechanisms within logic programming through the use of proof-theoretic techniques. This has led to the ignoring of several programming aspects either not pertinent to the notion of abstraction or not captured directly by proof theory. Included in this category are notions such as control, negation-by-failure, and side-effects. These issues are clearly important in a practical programming language and need to be addressed by further theoretical work.

Acknowledgements. The authors gratefully acknowledge valuable comments on this paper that were provided by D. Leivant, G. Mints, J. Zucker, and an anonymous referee. Support for this work from the following sources is also acknowledged: for Miller from NSF grant CCR-87-05596 and DARPA grant N000-14-85-K-0018, for Nadathur from NSF grant IRI-8805696 and ARO Contract DAAL03-88-K-0082, for Pfenning from the Office of Naval Research under contract N00014-84-K-0415 and from the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 5404, monitored by the Office of Naval Research under the same contract, and for Scedrov from NSF grants DMS85-01522 and CCR-87-05596 and from the University of Pennsylvania Natural Sciences Association Young Faculty Award.

9. References

A preliminary version of this paper appeared as [21]. Theorem 3 of that paper is incorrect. It is corrected by the material in Sections 5 and 6 of the current paper.

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

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

[3] K. Clark, Negation as Failure, in *Logic and Databases*, H. Gallaire and J. Minker (eds.), Plenum Press, New York, 1978, 293 – 322.

[4] 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, Ill., May 1988, 61 – 80.

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

[6] M. Fitting, A Kripke-Kleene Semantics for Logic Programming, Journal of Logic Programming, **4** (1985), 295 – 312.

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

[8] J. Gallier and S. Raatz, Hornlog: A Graph-Based Interpreter for General Horn Clauses, Journal of Logic Programming **4** (1987), 119 – 156.

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

[10] 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.

[11] 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.

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

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

[14] D. Miller, Hereditary Harrop Formulas and Logic Programming, Proceedings of the VIII International Congress of Logic, Methodology, and Philosophy of Science, Moscow, August 1987.

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

[16] D. Miller, Solutions to λ-Term Equations Under a Mixed Prefix, submitted, January 1989.

[17] D. Miller, Lexical Scoping as Universal Quantification, Sixth International Conference on Logic Programming, Lisbon Portugal, June 1989.

[18] D. Miller and G. Nadathur, Some Uses of Higher-Order Logic in Computational Linguistics, Proceedings of the 24th Annual Meeting of the Association for Computational Linguistics, 1986, 247 – 255.

[19] D. Miller and G. Nadathur, Higher-Order Logic Programming, Proceedings of the Third International Logic Programming Conference, London, June 1986, 448 – 462.

[20] D. Miller and G. Nadathur, A Logic Programming Approach to Manipulating Formulas and Programs, Proceedings of the 1987 Symposium on Logic Programming, San Franciso, September 1987, 379 – 388.

[21] D. Miller, G. Nadathur, and A. Scedrov, Hereditary Harrop Formulas and Uniform Proofs Systems, Proceedings of the Second Annual Symposium on Logic in Computer Science, Ithaca, June 1987, 98 — 105.

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

[23] G. Nadathur and D. Miller, An Overview of λProlog, Proceedings of the Fifth International Logic Programming Conference, Seattle, August 1988, 810 – 827.

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

[25] L. Paulson, Natural Deduction as Higher-Order Resolution, Journal of Logic Programming **3** 1986, 237 – 258.

[26] F. Pfenning, Partial Polymorphic Type Inference and Higher-Order Unification, Proceedings of the 1988 ACM Conference on Lisp and Functional Programming, Snowbird, Utah, July 1988, 153 – 163.

[27] F. Pfenning and C. Elliott, Higher-Order Abstract Syntax, Proceedings of the SIGPLAN '88 Symposium on Language Design and Implementation, Atlanta, Georgia, June 1988, 199 – 208.

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

[29] L. Sterling and E. Shapiro, *The Art of Prolog: Advanced Programming Techniques*, MIT Press, 1986.

[30] A. Troelstra, *Metamathematical Investigation of Intuitionistic Arithmetic and Analysis*, LNM 344, Springer-Verlag, 1973.

[31] M. van Emden and R. Kowalski, The Semantics of Predicate Logic as a Programming Language, Journal of the ACM **23** (1976) 733 – 742.

[32] M. van Emden, First-Order Predicate Logic as a Common Basis for Relational and Functional Programming, Proceedings of the Second Annual Symposium on Logic in Computer Science, Ithaca, June 1987, 179 (abstract).

[33] D. H. D. Warren, Higher-Order Extensions to Prolog: Are They Needed? *Machine Intelligence 10*, J. E. Hayes, D. Michie and Y-H. Pao (eds.), Halsted Press, 1982, 441 – 454.

Appendix: A Brief Summary

The definitions of the four abstract logic programming languages discussed in this paper are collected below. These definitions are followed by a brief summary of the observations made about them.

First-order Horn clauses. Let A be a syntactic variable that ranges over first-order atomic formulas. Let $\mathcal{G}_1$ and $\mathcal{D}_1$ be the sets of all first-order G- and D-formulas defined inductively by the following rules:

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

The formulas of $\mathcal{D}_1$ are called *first-order Horn clauses.* The triple $\langle \mathcal{D}_1, \mathcal{G}_1, \vdash_C \rangle$ is *fohc.*

Higher-order Horn clauses. Let $\mathcal{H}_1$ be the set of all λ-normal terms that do not contain occurrences of the logical constants $\supset, \forall,$ and $\perp$. Let A and A_r be syntactic variables denoting, respectively, atomic formulas and rigid atomic formulas in $\mathcal{H}_1$. Let $\mathcal{G}_2$ and $\mathcal{D}_2$ be the sets of all higher-order G and D-formulas defined inductively by the following rules:

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

The formulas of $\mathcal{D}_2$ are called *higher-order Horn clauses.* Notice that $\mathcal{G}_2$ is precisely the set of formulas in $\mathcal{H}_1$. The triple $\langle \mathcal{D}_2, \mathcal{G}_2, \vdash_C \rangle$ is *hohc.*

First-order hereditary Harrop formulas. Let A be a syntactic variable that ranges over first-order atomic formulas. Let $\mathcal{G}_3$ and $\mathcal{D}_3$ be the sets of all first-order G- and D-formulas defined by the following rules:

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

Formulas in $\mathcal{D}_3$ are called *first-order hereditary Harrop* formulas. The triple $\langle \mathcal{D}_3, \mathcal{G}_3, \vdash_I \rangle$ is *fohh*.

Higher-order hereditary Harrop formulas. Let $\mathcal{H}_2$ be the set of all λ-normal terms that do not contain occurrences of the logical constants $\supset$ and $\bot$. Let A and A_r be syntactic variables denoting, respectively, atomic formulas and rigid atomic formulas in $\mathcal{H}_2$. Let $\mathcal{G}_4$ and $\mathcal{D}_4$ be the sets of G- and D-formulas that are defined by the following mutual recursion:

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

The formulas of $\mathcal{D}_4$ are called *higher-order hereditary Harrop* formulas. *hohh* is the triple $\langle \mathcal{D}_4, \mathcal{G}_4, \vdash_I \rangle$.

Some Observations. All four triples, *fohc*, *hohc*, *fohh*, and *hohh*, are abstract programming languages. If the provability relation for *fohc* and *hohc* is weakened from $\vdash_C$ to $\vdash_I$ or $\vdash_M$, the resulting triples would still be abstract logic programming languages. In fact, such a weakening does not change the set of sequents that are provable. If the provability relation for *fohh* and *hohh* is weakened from $\vdash_I$ to $\vdash_M$, the resulting triples would still be abstract logic programming languages, and again, such a weakening does not change the set of sequents that are provable. However, if the provability relation for *fohh* and *hohh* is strengthened to $\vdash_C$, then new sequents would be provable and the resulting triple would not be an abstract programming language.

A set of terms can be classified as a *Herbrand universe* for an abstract logic programming language if provable sequents in the language have uniform proofs in which the terms generalized on in the $\forall$-L and $\exists$-R inference figures are members of this set. The set of first-order terms is a Herbrand universe for both first-order languages, *fohc* and *fohh*. The set $\mathcal{H}_1$ is a Herbrand universe for *hohc* while the set $\mathcal{H}_2$ is a Herbrand universe for *hohh*.

Informally, we can say that one abstract logic programming language is *contained* in another if every goal or program clause of the first is, respectively, a goal or program clause of the second and if each "appropriate" sequent having a uniform proof in one also has a uniform proof in the other. The containment relations among the four languages discussed in this paper is then completely described by noting that *fohc* is contained in all the languages, *hohh* contains all the languages, and *fohh* and *hohc* are not comparable.