Higher-Order Horn-Logic Programming*

William W. Wadge

Abstract

We describe a fragment of higher-order Horn logic which can be used as a higher-order extension of Prolog. It allows the programmer to axiomatize predicates of predicates and operations on predicates.

The restrictions defining the fragment ensure that the higher-order predicates and operations defined are monotonic and continuous—that they are enumeration operators. As a result, the fragment is well behaved both denotationally and operationally. Denotationally, the meaning of a program is its minimal Herbrand model, using standard extensional higher-order model theory. Operationally, it can still be implemented efficiently with standard techniques, using an adapted resolution theorem prover.

The fragment is in fact (almost) a subset of HiLog—a “pure” subset with simple semantics and a straightforward implementation.

1 Introduction

Functional programmers and Logic programmers differ markedly in their attitude to higher-order logic. Functional programmers love higher-order functions; the higher the better! In fact, it would be more accurate to call the Miranda style of programming “higher-order programming”. The first-order subset of the λ-calculus is of practically no interest.

Logic programming, on the other hand, is based on the Horn subset of first-order logic. Higher-order predicate logic has been studied but is notoriously complex: the nicest model-theoretic results (such as compactness) are lost, and the proof systems are all inherently very incomplete. Already at the second order the set of tautologies is not even hyperarithmetic; informally, it is so far from being recursive that we could not decide membership even with Turing machines which can perform a countably infinite number of steps in a finite amount of time.

The attitude of logic programmers has been somewhat ambivalent. On the one hand, Warren [7] has argued that higher-order primitives are of no interest because higher-order programs can be simulated in first-order Prolog with little extra effort. At the same time, there are proposals for higher order to Prolog [4, 2] which are genuinely more powerful but also more complex semantically and/or proof-theoretically.

In this paper, however, we follow the route indicated by [4] and [3] and sacrifice some of the power and complexity of full higher-order logic in return for semantic and proof-theoretic simplicity. Our clauses can be understood declaratively in terms of the standard semantics. Every program has a minimum model, and every type-0 ground term in the model is provable. Warren’s arguments can be understood not as a rebuttal, but as a (partial) implementation technique.

2 Higher-Order Predicate Logic

We will now give a formal definition of the notion of “Higher-Order Predicate Logic” using a type system based on Church’s Simple Theory of Types. In this system, there are only two base types: o, the Boolean domain, and ι, the domain of individuals (data objects).

Informally, we extend first-order Predicate Logic by adding predicates whose arguments can themselves be predicates. These argument predicates are not necessarily first order; their arguments may in turn be predicates, and so on. We also allow operations on predicates: operations whose arguments and/or results are predicates. But we do not allow higher-order operations on first-order operations.

**Definition 1.** A **predicate type** is defined as follows:

1. The Boolean type $\mathcal{O}$ is a predicate type;
2. If $\pi_0, \pi_1, \pi_2, \ldots, \pi_n$ are predicate types, then so is $\pi_0 \times \pi_1 \times \cdots \times \pi_k \rightarrow \pi_n$ (we are assuming for simplicity that predicate arguments precede all individual arguments).

For example, an object of type $(\iota \times \iota \rightarrow \mathcal{O}) \rightarrow \mathcal{O}$ is a predicate on binary (first-order) relations; an object of type $(\iota \times \iota \rightarrow \mathcal{O}) \times (\iota \rightarrow \mathcal{O}) \times \iota \rightarrow (\iota \rightarrow \mathcal{O})$ is an operation which takes a binary relation, a unary relation and a data object and yields a unary relation. Notice that for the sake of uniformity, $\mathcal{O}$ itself is a predicate type. This actually has advantages; for example, the logical connectives are just constants of the appropriate type.

Beside the predicate types we also allow the first-order types over $\iota$.

**Definition 2.** A **HOPL type** is either a predicate type or one of the form $\iota^k \rightarrow \iota$ for some $k \geq 0$ (we identify $\iota^0 \rightarrow \iota$ and $\iota$).

The syntax of HOHL is fairly simple. The main complication is that there are three kinds of syntactically distinguishable symbols: *bound* variables, *free* variables, and *constants*. The bound variables are used in connection with quantifiers, or are implicitly universally quantified (as in Horn clauses). The free variables denote particular but arbitrary individuals or operations as specified by some interpretation. And the constants denote operations which are in some sense the same in all interpretations.

**Definition 3.** The vocabulary of HOPL consists of

1. bound variables of type $\iota$ and of every predicate type;
2. free variables of types $\iota^k \rightarrow \iota$ for all $k \geq 0$ (for $k = 0$ these are individual symbols);
3. free variables of every predicate type;
4. the logical constants $\land, \lor$ and $\rightarrow$ of type $\mathcal{O} \times \mathcal{O} \rightarrow \mathcal{O}$ and not of type $\mathcal{O} \rightarrow \mathcal{O}$;
5. the quantifiers $\forall$ and $\exists$.

Notice that there are no bound variables of any nontrivial function type (such as $\iota \times \iota \rightarrow \iota$). And there are no symbols of any kind of higher functional types, such as $(\iota \times \iota \rightarrow \iota) \rightarrow \iota$.

Expressions are formed in the obvious way, in accordance with the type system. A formula is simply an expression of type $\mathcal{O}$, and a term is an expression of type $\iota$.

**Definition 4.** The family $\{ E_\tau : \tau \text{ a HOPL type} \}$ (of HOPL expressions indexed by type) is defined inductively by

1. every symbol of type $\tau$ is in $E_\tau$;
2. if $F$ is a symbol of type $\tau_0 \times \tau_1 \times \cdots \times \tau_{n-1} \rightarrow \tau_n$ and $A_0, A_1, \ldots, A_{n-1}$ are terms of types $\tau_0, \tau_1, \ldots, \tau_{n-1}$, respectively, then $F(A_0, A_1, \ldots, A_{n-1})$ is in $E_{\tau_n}$;
3. if $A$ is in $E_\mathcal{O}$ and $V$ is a bound variable (of any type), then $\forall V A$ and $\exists V A$ are in $E_\mathcal{O}$.

(Since the logical connectives are constants, we do not need separate clauses for disjunctions, conjunctions, implications, etc.)

The semantics is also straightforward. A model is a nonempty set $U$ (the universe) together with an element which assigns to each free variable or constant a map in the appropriate domain. The appropriate domain for expressions in $E_\tau$ is $R_{U, \tau}$, where
Definition 5. For any nonempty set $U$ and any HOPL types $\tau_0, \tau_1, \ldots, \tau_n$:

1. $R_{U,o} = \{0,1\}$;
2. $R_{U,t} = U$;
3. $R_{U,\tau_0 \times \tau_1 \times \cdots \times \tau_n \rightarrow \tau_n}$ is the set of all functions from $R_{U,\tau_0} \times R_{U,\tau_1} \times \cdots \times R_{U,\tau_n-1} \rightarrow R_{U,\tau_n}$.

The definition merely formalizes the standard, set-theoretic interpretation of the simple theory of types. Notice that in this definition, predicates are not, strictly speaking, sets. For example, an element of higher-order Horn clauses will not have a minimum Herbrand model. At least, we cannot assign a meaning in the same way as for first-order Prolog. In general, a set problem is that as a programming language, higher-order Horn logic is literally meaningless—or $(\tau \rightarrow o)$ Horn clauses are defined by direct analogy to the first-order case; a disjunction of formulas of type $o$, one of which is negated. It is not so easy, however, to define the higher-order analog of Prolog. The problem is that as a programming language, higher-order analog of Prolog. The problem is that as a programming language, higher-order Horn logic is literally meaningless—or at least, we cannot assign a meaning in the same way as for first-order Prolog. In general, a set of higher-order Horn clauses will not have a minimum Herbrand model.

Herbrand models are still defined as those in which the individuals (objects of type $t$) are the ground terms—the terms which can be constructed using free operation variables. Higher-level objects are therefore based on this Herbrand universe; for example, a property of binary relations will denote a set of sets of ordered pairs of elements of the Herbrand universe.

Definition 6. For any HOPL type $\tau$, $H_\tau = R_{U,\tau}$ with $U = H_i$ the ordinary (first-order) Herbrand universe of terms built using free variables of types $t^k \rightarrow t$, $k \geq 0$.

3 Problems with Higher-Order Horn logic

Horn clauses are defined by direct analogy to the first-order case; a disjunction of formulas of type $o$, one of which is negated. It is not so easy, however, to define the higher-order analog of Prolog. The problem is that as a programming language, higher-order analog of Prolog. The problem is that as a programming language, higher-order Horn logic is literally meaningless—or at least, we cannot assign a meaning in the same way as for first-order Prolog. In general, a set of higher-order Horn clauses will not have a minimum Herbrand model.

These are very simple examples of the problem. Let $phi$ be a second-order predicate of type $(t \rightarrow o) \rightarrow o$, that is, a predicate on ordinary unary predicates. Consider the following ‘program’ which axiomatizes $phi$ and two unary first-order relations $p$ and $q$:

$$
p(a) \leftarrow .
q(a) \leftarrow .
phi(p) \leftarrow .
q(b) \leftarrow \phi(q).
$$

One candidate for minimum Herbrand springs to mind, in which $p$ and $q$ are true only of $a$, and $phi$ is true only of $p$. However, this means that $p$ and $q$ have the same extension—they are true of exactly the same individuals, and so themselves are equal. But since $p$ and $q$ are equal, and $phi$ holds for $p$, it must hold for $q$ as well. The fourth rule forces us to add $q(b)$, so that the model becomes $\{p(a), phi(p), q(a), q(b)\}$ (in ad hoc notation).

The only trouble is that $p$ and $q$ are no longer equal, so that $q(b)$ has lost its justification. This is already suspicious. Moreover, we see that by adding, say, $q(c)$ we can avoid the extensionality problem in the first place. The result is another model $\{p(a), phi(p), q(a), q(c)\}$. Of course in this model $q(c)$ is true for no particular reason, which is again suspicious.

Far more serious is the fact that the intersection of these two models, $\{p(a), q(a), phi(p), phi(q)\}$ is not a model—the fourth clause is not satisfied. In fact it is easy to see that the two models given above are minimal but incomparable. We cannot use model theory in the usual way to give a semantics to our little program.

One obvious solution suggests itself, namely abandoning extensionality. Then we could allow $phi$ to be true of $p$ but not $q$, even though $p$ and $q$ have the same extensions. One can make quite
a convincing case for this using the following observation: suppose that \( p \) were true of infinitely many atoms, say of all the natural numbers in successor notation. And suppose that \( \phi(p) \) is the only axiom for \( \phi \). Then \( \phi(q) \) amounts to proving that \( q \) is true of all naturals, which is totally unreasonable.

It is fact possible to give a nonextensional model theory for higher-order logic. Each type is treated as a separate domain and there is a ‘super’ relation to say which elements of a relation domain are true for which arguments. This is the path taken by [4] and [2].

Intensional systems clearly have their uses. We can write queries with a free relation variable, and the implementation gives formulas as answers (using some sort of higher-order unification).

On the other hand, there are many modest but useful applications of higher-order logic programming that do not require free predicate variables or higher-order unification. The following, for example, is a program which transforms a list using a given predicate on each component.

\[
\text{maplist}(R, [], []) :-
\text{maplist}(R, [G | S], [H | T]) :- R(G, H), \text{maplist}(R, S, T).
\]

(We have omitted type information and followed HiLog in using a C-Prolog–like syntax.)

The statements of this program can certainly be understood (declaratively) in terms of extensional notions: \( \text{maplist}(R, L, M) \) says that \( L \) and \( M \) are lists of the same length with corresponding elements related by the binary relation \( R \). The analogous functional program can be understood using a strictly extensional view; why should logic programmers have to give up extensionality just because they want to axiomatize relations with relation parameters?

### 4 The definitional subset of HOHLL

Our plan is instead to retain extensionality by sacrificing some of the generality of full higher-order Horn Logic, and some of the power of nonextensional systems. In other words, we identify a fragment of HOHLL which has the minimum model property and (we should hope) a workable proof theory.

Since we are not changing the semantics, our fragment must rule out the example given above. One of its clauses is not ‘proper’. Suspicion immediately falls on the rule \( \phi(p) \leftarrow \). If \( p \) has an infinite extension, this rule seems to assert an infinite amount of information about \( \phi \). Furthermore, is it part of the definition of \( p \), or part of the definition of \( \phi \)? With two parameters to play with, we should not be surprised that there is no best way to satisfy programs containing this clause.

The solution we offer is very simple: restrict axioms for higher-order predicates so that the left-hand sides refer only to the predicate symbol in question. We do this by requiring that higher-order predicates which appear on the left-hand side of a clause be given a full set of formal parameters for each argument of order one or higher.

**Definition 7.** A higher-order Horn clause is definitional iff

1. in the head of the clause each argument of a predicate type is a variable (which we call a formal parameter) local the clause (implicitly universally quantified);
2. the formal parameters are all distinct;
3. they are the only predicate variables local to the clause.

Notice that there are no restrictions on the formulas in the body of definitional Horn clauses (other than those imposed by the type system). Also, there are no restrictions on the use of variables of type \( \iota \).
For example, suppose that the type of \( \psi \) is such that it expects as its arguments a binary predicate, a unary predicate, and an individual. The following rules are allowed in our fragment:

\[
\begin{align*}
\psi(B, P, g(X, X)) &\leftarrow P(X), B(X, f(X)). \\
\psi(B, P, \text{red}) &\leftarrow \psi(P), B(\text{red}, V), q(V). \\
\psi(B, P, m(Y, Z)) &\leftarrow B(Y, V), B(Z, W), \psi(B, P, m(Y, Z)).
\end{align*}
\]

But the following are excluded:

\[
\begin{align*}
\psi(B, q, X) &\leftarrow B(X, Y), q(Y). \\
\psi(\tau(B), S, 3) &\leftarrow B(3, 3). \\
\psi(B, P, Z) &\leftarrow R(Z), \psi(B, R, 0).
\end{align*}
\]

5 Minimal model semantics

We will now outline the proof that our fragment possesses the minimum model property. The crucial concept is that of \textit{continuity} of a relation or relation operation. The concept is not new, however, and can be traced through the Scott–Strachey semantics all the way back to the ideas of Kleene. For this reason, and to save space, we will only outline the proof.

The proof is based on the fact that the predicate domains over the Herbrand universe are chain-complete partial orders.

\textbf{Definition 8.} For any predicate type \( \pi \), the partial order \( \sqsubseteq_{\pi} \) on \( H_{\pi} \) is as follows:

1. \( \sqsubseteq_o \) is the numerical order on \( \{0, 1\} \);
2. for any predicate type \( \rho \) of the form \( \pi_0 \times \pi_1 \times \cdots \times \pi_k \rightarrow \pi_n \) any \( f \) and \( g \) in \( H_{\rho} \), \( f \sqsubseteq_{\rho} g \) iff \( f(r) \sqsubseteq_{\pi_n} g(r) \) for any \( r \) in \( \pi_0 \times \pi_1 \times \cdots \times \pi_k \).

\textbf{Lemma 1.} For any predicate type \( \pi \), \( (H_{\pi}, \sqsubseteq_{\pi}) \) is a chain complete partial order.

The proof is straightforward. In fact, each \( H_{\pi} \) is a complete partial order. For example, elements of \( H_{(\rightarrow_o \rightarrow o)} \) are (as we mentioned earlier) ‘really’ sets of sets of ground terms; and \( \sqsubseteq_{(\rightarrow_o \rightarrow o)} \) is ‘really’ just the inclusion ordering on these sets of sets.

\textbf{Definition 9.} A Herbrand model is \textit{continuous} iff the meaning it assigns to each free variable of a predicate type is a monotonic and continuous element of the appropriate domain.

Notice that the constants are not necessarily continuous; in particular negation and implication are not even monotonic.

The main technical result is that expressions constructed using continuous primitives denote continuous predicates.

\textbf{Theorem 1.} For any continuous Herbrand model \( \mathcal{M} \), any predicate type \( \pi \) and any expression \( E \) of type \( \pi \): if \( E \) does not use negation, implication or universal quantification, then \( E \) denotes (over \( \mathcal{M} \)) a continuous element of \( H_{\pi} \).

This can be proved by induction on the structure of \( E \) using standard techniques. It is worth noting that it remains true even if we extend our language and allow \( \lambda \)-expressions and finite unions and intersections.

Given this theorem, the minimum Herbrand-model property follows easily.

\textbf{Theorem 2.} Every set of definitional Horn clauses has a minimum Herbrand model, and this model is continuous.
Proof. (Sketch.) Suppose for simplicity that \( \phi, \psi \) and \( \gamma \) are the higher-order operators being defined. The basis idea is to rewrite and combine the clauses defining them into the ‘extended’ HOHLP clauses of the form

\[
\phi \supseteq K_0(\phi, \psi, \omega) \\
\psi \supseteq K_1(\phi, \psi, \omega) \\
\gamma \supseteq K_2(\phi, \psi, \omega)
\]

where \( K_0, K_1 \) and \( K_2 \) are built up using application, union (of predicates), \( \lambda \)-expressions and existential quantification. We can do this in two stages: first, we use \( \lambda \) to eliminate the formal parameters; then, we combine different clauses using unions. For example, if the following are the rules for \( \phi \):

\[
\phi(P) \leftarrow P(1), P(2).
\]

\[
\phi(P) \leftarrow \psi(P), q(X), P(X).
\]

we can combine them as

\[
\phi \supseteq \lambda P \ (P(1), P(2)) \cup \lambda P \exists X (\psi(P), q(X), P(X)).
\]

Since the \( K \)’s denote continuous operations, the usual fixpoint construction gives the minimum model.

The crucial fact is that the inclusions are logically equivalent to the conjunction of the original clauses, so that the minimum model of the inclusions is also the minimum model of the original clauses. Extensionality plays vital rôle here—without it, the transformations could not be justified.

The proof is slightly more complicated by the fact that the higher-order predicate may have zero-order (type \( \iota \)) arguments as well, which on the left side of the clauses are not necessarily simple variables. We handle this by replacing each clause by all its ground instances, and considering each as defining a different higher-order predicate. 

\( \square \)

6 Continuity and extensionality

The concept of continuity also helps explain why extensionality does not cause trouble in our fragment. We see that in the minimum model the denotations of all the higher-order predicates are themselves continuous. This means that they are monotonic and ‘finitary’ in the sense that any finite set of assertions about the result of the operator depends on only a finite amount of information about the argument.

For example, suppose that \( \psi \) is a predicate on unary predicates and that \( \psi \) holds for predicate \( p \). Continuity implies that there must be a finite set of positive facts about \( p \) (for example, \( p(0), p(1), p(3) \)) such that \( \psi \) is true of any other predicate for which the corresponding facts hold. In the example given, if \( r(0), r(1), r(3) \), then \( \psi(r) \).

Now suppose that \( \psi(p) \) holds in the minimum model of our program, and that it happens that \( q \) and \( p \) are coextensional. This means that \( \psi(q) \) holds as well, and that this formula must succeed when it is used as a goal. We assume that the implementation is basically a theorem prover. How can it take extensionality into account?

The answer is that it does not have to. If it can establish \( \psi(p) \), it can do so after verifying only finitely many facts about \( p \). If \( p \) is coextensional with \( q \), it should be able to prove the analogous facts about \( q \), and conclude \( q \). So the extensionality rule is valid, but not needed.

This notion of continuity is well known in recursive function theory, as Fitting remarks [3]. Operations on relations which are continuous in this sense are called enumeration operators. Fitting, incidentally, already pointed out that a calculus of enumeration operators would allow us to add functional programming on top of logic programming. His object language, however, does not have any higher-order entities apart from modules, which are essentially anonymous enumeration operators. Fitting’s modules correspond closely to the \texttt{where} clauses of Landin’s ISWIM.
7 Uses of HOHL

Higher-order operators can be used in logic programming in much the same way as they are used in functional programming.

One use is to express in a single program an algorithm which naturally has a predicate parameter. For example, the predicate ordered defined below:

\[
\text{ordered}(R, []) \leftarrow .
\]
\[
\text{ordered}(R, [X]) \leftarrow .
\]
\[
\text{ordered}(R, [X,Y | T]) \leftarrow R(X,Y), \text{ordered}(R, [Y | T]).
\]

checks that its second parameter (a list) is ordered by its first parameter (a binary relation). In first-order Prolog, we would have to write several clones of these axioms, one for checking numerical order, one for checking lexicographical order, and so on.

Even better, we can define a third-order operation which takes a predicate as its argument and returns the corresponding list-checking predicate:

\[
\text{checker}(R)([]) \leftarrow .
\]
\[
\text{checker}(R)([X]) \leftarrow .
\]
\[
\text{checker}(R)([X,Y | T]) \leftarrow R(X,Y), \text{checker}(R)([Y | T]).
\]

Opportunities for parameterizing algorithms often arise in practice: sorting, searching, forming sums and products, walking trees, and the like.

Another very useful form of higher-order programming is the use of general-purpose combinators to avoid detailed manipulation of individual variables. For example, we can define the relational join, projection, cylindrification and converse operations (on binary predicates) as follows:

\[
\text{join}(P,Q)(X,Y) \leftarrow p(X,M), Q(M,Y). \\
\text{proj0}(P)(X) \leftarrow P(X,Y). \\
\text{cyl1}(U)(X,Y) \leftarrow U(X). \\
\text{conv}(P)(X,Y) \leftarrow P(Y,X). \\
\text{sing}(X)(X) \leftarrow .
\]

and then use these operations to combine relations in the style of relational databases. The sing operation, of type \(\iota \rightarrow (\iota \rightarrow o)\), returns a unary predicate which is true only of its argument.) For example, suppose that we have already defined the following predicates:

\[
\text{produces}(F,P) : \text{firm } F \text{ produces product } P \\
\text{buys}(C,P) : \text{firm } C \text{ buys product } P
\]

Then, given the definitions above, the following expression

\[
\text{proj0} \left( \text{join}(\text{produces, conv(buys)}) \cap \text{cyl1}(\text{sing(acme)} \cup \text{sing(continental)}) \right)
\]

denotes the set of possible customers for products made by acme or continental. Of course, the combinatorial form is not always the clearest. But in HOHLP, the programmer has the choice of using combinators, and can also mix the variable-oriented and combinatorial forms.

There are of course many other interesting uses for higher-order predicates and most of these are well known. C-Prolog, for example, already allows names of predicates to be passed, so some of our examples will already run on an ordinary interpreter. Also, HiLog and other systems have generalized these second-order features and indeed all examples given so far will run unchanged as HiLog. What is new is that we have a criterion to distinguish the more modest (extensional) uses of higher-order concepts from the more ambitious ones; the distinction is not always obvious.
8 Advantages of extensionality

One way of understanding HOHLP is to think of it as a “pure” subset of HiLog. What then is
the point of indentifying a “pure” sublanguage?

One advantage is that we can now write higher-order programs free of the guilt associated with
using murky nonlogical features of Prolog. We can feel comfortable that we are still using normal,
no-nonsense logic and that our programs still have a declarative reading.

Freedom from guilt is fine, of course, but the advantages of extensionality are more than just
psychological. In particular, the ability to read a program declaratively is an immense advantage
of Prolog as compared to (say) Pascal. The only way to understand a Pascal program is by figuring
out what it does. You can also figure out what a Prolog program does, but you do not have to;
you can also (if it is pure Prolog) look at what it says.

Decades of experience have shown that languages with simple semantics are simple in practice
as well as in theory; they are more understandable, more ‘structured’, less error-prone, and so
on. In particular, HOHLP has a simple proof system (not discussed here) which could be used to
verify programs.

Finally, we would like to emphasize the software-engineering significance of extensionality.
Suppose that we have a large program which uses (but does not define) a predicate for sorting
lists. Suppose that separately we have axiomatized a particular sorting predicate s and that our
large program plus the axioms for s are working fine.

Now suppose we come up with new axioms for a predicate q that also sorts lists (and handles
error conditions the same way). We decide to try our software using q rather than s. Will it still
work?

If our software is extensional, we have nothing to worry about. If our software was correct
using s, it will still be correct using q—because as far as correctness goes, the only thing that mat-
ters about s or q are their extensions. Of course, the performance may be different: extensionality
does not mean that there is only one sorting program. But as far as confidence in the results, we
can be sure that coextensional predicates are interchangeable.

Extensionality means exactly that predicates are used as black boxes—and the “black-box”
concept is central to all kinds of engineering.

9 Further extensions

The fragment of HOHL described is in a sense very conservative. Above the first level, it is
essentially functional programming. The rules do not allow a higher-order operation in the head
of clauses, except in their defining clauses. And the rule which allows only the formal-parameter
relation variables in the bodies ensures that no relation variable is ever (speaking operationally)
unsubstantiated.

The main advantage of this conservative approach is that the language is relatively easy to
implement, given an implementation of ordinary first-order Prolog. In fact, some of our examples
will already run under C-Prolog, and almost all will work as HiLog.

This does, however, lead one to ask if there are bigger fragments which make more ambitious
use of higher-order entities but are still implementable. This seems to be the case. In particular,
it seems possible to relax the two restrictions just mentioned.

Consider first queries involving uninstantiated relation variables. At first sight, it would seem
totally unreasonable—the search space (the set of candidate relations) is infinite and not even
countable. Our higher-order predicates, however, are continuous: if a relation satisfies a predicate,
then some finite subset satisfies it. This means that we have to examine only finite relations.

For example, recall the predicate ordered, defined earlier, which checks that its second argument
is a list ordered by its first argument. The query

\[ \text{ordered}(R, [a, b, c, d]) \]
can be solved by assigning to $R$ the predicate which is true of $a$ and $b$, $b$ and $c$, and $c$ and $d$. The implementation might therefore respond

$$R = \{(a, b), (b, c), (c, d)\}$$

Since we only have to examine finite relations, it is possible in principle for an implementation to be complete. The set of finite relations is enumerable, and we can test them one by one. We do not know, however, if there is a more practical approach.

HiLog and other higher-order-logic programming systems allow queries like this but do not deal with them extensionally. Instead, the interpreter generates intensions (formulas) which provably satisfy the query.

It is also possible to relax the restriction that clauses be definitional. Recall that the problem with clauses of the form

$$phi(p) \leftarrow \cdots$$

is that they axiomatize both $phi$ and $p$. In general, there is no ‘best’ (minimal) way of making a rule like this true, because there might be a trade-off between making $phi$ ‘truer’ and making $p$ ‘truer’.

Suppose, however, that $phi$ is constant whose meaning is fixed a priori (independently of the program the clause appears in). In this case, the clause axiomatizes only $p$, and there may be a minimum value of $p$ which satisfies the clause.

As a simple example, suppose that $phi$ asserts its argument is true of 0 and 1. Then the nondefinitional clause $phi(p) \leftarrow$ is equivalent to the conjunction of the clauses $p(0)$ and $p(1)$, which are definitional and cause no problems.

On the other hand, not all higher-order constants can be used this way. For example, if $psi$ asserts that its argument is true of 1 or of 2, the clause $psi(p) \leftarrow$ will not have a minimum model.

Orgun has, in another context [5], identified the property which seems relevant here. A constant like $phi$ must be closed under intersection: that is, if $phi$ is true of each member of a family of predicates, then it must be true of their intersection. There is no relation between the intersective property and continuity. As we have seen, there are simple examples of continuous predicates that are not intersective and cannot be used in the heads of rules defining their arguments.

This suggests a two-level approach to higher-order logic programming. First, we present meta-level rules which define higher-order constants with the intersective property. Then, we use these constants in an object-level (ordinary) program.

For example, our meta-program might have the rules

$$allmembs(P, [H | T]) \rightarrow P(H).$$
$$allmembs(P, [H | T]) \rightarrow allmembs(P, T).$$

The intended meaning of $allmembs(P, L)$ is simply that $P$ is true of all elements of $L$. This meaning is a model of the two defining clauses (notice that we reversed the arrow). We could use $allmembs$ in an object-level clause such that

$$allmembs(student, [tom, dick, harry]) \leftarrow .$$

to assert that $tom$, $dick$ and $harry$ are all students.

The problem is to find syntax and semantics for the metalevel which allow only intersective predicates to be defined. This may involve maximal models, but the matter is still being investigated.

**Acknowledgements**

We would like to thank A. Nait-Abdallah [1] and M. Orgun for some very useful discussions and explanations. The work was supported in part by IBM Canada and by a grant from the Canadian Natural Sciences and Engineering Research Council (NSERC).
References


