Abstract

We propose a purely extensional semantics for higher-order logic programming. Under this semantics, every program has a unique minimum Herbrand model which is the greatest lower bound of all Herbrand models of the program and the least fixed-point of the immediate consequence operator of the program. We also propose an SLD-resolution proof procedure which is sound and complete with respect to the minimum model semantics. In other words, we provide a purely extensional theoretical framework for higher-order logic programming which generalizes the familiar theory of classical (first-order) logic programming.

1 Introduction

The extension of logic programming to support higher-order constructs (in a semantically clean way) is an intriguing research problem. The initial attitude of logic programmers towards this problem was somewhat skeptical: it was argued (see, for example, [8]) that higher-order extensions may not be that necessary since there exist ways of simulating higher-order programming inside Prolog itself. Later on, more genuine approaches were developed. The main such examples are λ-Prolog [6] and Hilog [3]. These two systems share a common idea, namely they are both intensional: two predicates are not considered equal unless their names are the same. The intensional approach has its merits, and this is evidenced by the fact that both of the above systems have continued to develop and to explore various application domains.

There have also been a few attempts to define extensional higher-order logic programming systems. In such a system, two predicates are considered equal if they have the same extensions, namely they are true for the same arguments. The first extensional higher-order approach was proposed in [7]. Subsequently, M. Bezem [2] also considered an alternative extensional approach, which however appears to differ from classical extensionality and has a more proof-theoretical flavor. In [7] it is suggested that by restricting the syntax of higher-order programs, we can get a language in which the defined relations are continuous. Continuity guarantees that every such program has a well-defined meaning which can be computed with standard domain-theoretic techniques. There are two main issues that remained unresolved in [7]:

- The higher-order fragment considered in [7] does not allow uninstantiated higher-order variables to appear in clause bodies or in queries.
- A proof procedure is not provided (but it is conjectured that a sound and complete such procedure exists that even covers the extension with uninstantiated higher-order variables).

In this paper we remedy the above issues and provide the first (to our knowledge) framework for extensional higher-order logic programming that is complete both from a semantic as well as from a proof-theoretic point of view. To demonstrate the key idea of our approach, consider the

*Published in T. Janhunen and I. Niemelä, editors, JELIA 2010, LNAI 6344:91–103, 2010. This work has been partially supported by the University of Athens under the project “Kapodistrias” (grant no. 70/4/5827).
following higher-order program:

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

Consider the query \( \leftarrow \text{ordered}(R, [a, b, c, d]) \). At first sight this appears to be an unreasonable query, since the set of possible solutions is infinite (and actually uncountable). However, at a closer look we realize that all such relations extend the simple relation \( r = \{(a, b), (b, c), (c, d)\} \).

In other words, an implementation that would return the answer \( R \) would be satisfactory in the sense that \( R \) can be taken to represent all its superset relations (whose additional information is of no interest to the programmer). In the fragment that we consider, if a relation satisfies a predicate then there exists a simple (or basic) relation (like \( r \) in our example) that also satisfies the predicate. The set of such simple relations is countable and therefore a proof procedure can be devised that is not only sound but also complete. However, we need to formally characterize which are these basic relations for every possible type.

Fortunately, there exists a branch of domain theory that examines exactly the above issues. The key notion that we need is that of an \( \omega \)-algebraic complete lattice (a special case of an \( \omega \)-algebraic domain, see, for example, [1]), namely a complete lattice which has an enumerable basis of compact elements (which correspond to the simple elements that we have been talking about). Every element of an \( \omega \)-algebraic complete lattice can be represented as the least upper bound of such compact elements. In our example, all the relations (even the infinite ones) for which the \( \text{ordered} \) predicate could be true of, can be represented by some simple relations (like \( r \) above).

The main task of the paper is therefore to develop a semantics for higher-order logic programming that is based on \( \omega \)-algebraic complete lattices. Our semantics refines and extends the work of [7], and has an extra important advantage: it leads to a relatively simple sound and complete proof procedure for higher-order logic programming.

## 2 Algebraic Complete Lattices

In the rest of the paper we assume a basic familiarity with the basic notions regarding partially ordered sets and in particular complete lattices (see, for example, [5]). Given a partially ordered set (poset) \( P \), we write \( \subseteq_P \) (or simply \( \subseteq \)) for the corresponding partial order.

We will be interested in a certain type of complete lattices in which every element can be “created” by using a set of “basic” elements of the lattice:

**Definition 1.** Let \( L \) be a complete lattice. An element \( c \in L \) is called compact if for every directed set \( A \subseteq L \) with \( c \subseteq \bigsqcup A \), there exists \( a \in A \) such that \( c \subseteq a \). The set of all compact elements of \( L \) is denoted by \( \mathcal{K}(L) \).

Let \( P \) be a poset. Given \( B \subseteq P \) and \( x \in P \), we write \( B_{x} = \{ b \in B \mid b \subseteq_P x \} \).

**Definition 2.** A complete lattice \( L \) is called an algebraic complete lattice if for every \( x \in L \), the set \( \mathcal{K}(L)_{\downarrow x} \) is a directed subset of \( L \) with least upper bound \( x \). The set \( \mathcal{K}(L) \) is called the basis of \( L \). If additionally, \( \mathcal{K}(L) \) is countable, then \( L \) is called an \( \omega \)-algebraic complete lattice.

We can now introduce the notion of “step functions”, which form a subset of monotonic functions with interesting properties:

**Definition 3.** Let \( A \) be a poset and \( L \) be an algebraic complete lattice. Let also \( \downarrow_{L} \) be the least element of \( L \). For each \( a \in A \) and \( c \in \mathcal{K}(L) \), we define the step function \( (a \searrow c) : A \rightarrow L \) as

\[
(a \searrow c)(x) = \begin{cases} 
  c, & \text{if } a \subseteq_A x \\
  \downarrow_{L}, & \text{otherwise}
\end{cases}
\]
Given posets $P$, $Q$, we write $[P \rightarrow Q]$ to denote the set of all monotonic functions from $P$ to $Q$. The following lemma, which will prove useful later on, can be easily established:

**Lemma 1.** Let $A$ be a poset and $L$ an algebraic complete lattice. Then, $[A \rightarrow L]$ is an algebraic complete lattice whose basis is the set of all least upper bounds of finitely many step functions from $A$ to $L$. If $A$ is countable and $L$ is an $\omega$-algebraic complete lattice, then $[A \rightarrow L]$ is an $\omega$-algebraic complete lattice.

## 3 The Higher-Order Language $\mathcal{H}$: Syntax

In this section we define the higher-order language $\mathcal{H}$, which will be the basis of the higher-order logic programming language that we will subsequently develop. The language $\mathcal{H}$ is based on a simple type system that supports two base types: $o$, the Boolean domain, and $\iota$, the domain of individuals (data objects). The composite types are partitioned into three classes: functional (assigned to function symbols), argument (assigned to parameters of predicates) and predicate (assigned to predicate symbols).

**Definition 4.** A type $\tau$ can be functional, argument or predicate:

$$
\sigma := \iota | (\iota \rightarrow \sigma)
$$

$$
\rho := \iota | \pi
$$

$$
\pi := o | (\rho \rightarrow \pi)
$$

The binary operator $\rightarrow$ is right-associative. A functional type that is different than $\iota$ will often be written in the form $\iota^n \rightarrow \iota$, $n \geq 1$, and a predicate type that is different than $o$ will be written in the form $\rho_1 \rightarrow \cdots \rightarrow \rho_n \rightarrow o$, $n \geq 1$.

**Definition 5.** The alphabet of the higher-order language $\mathcal{H}$ consists of the following:

1. Predicate variables of every predicate type $\pi$ (such as $p, q, r, \ldots$).
2. Argument variables of every argument type $\rho$ (such as $Q, R, V, X, \ldots$).
3. Individual constant symbols of type $\iota$ (such as $a, b, c, \ldots$).
4. Function symbols of every functional type $\sigma \neq \iota$ (such as $f, g, h, \ldots$).
5. The following logical constant symbols: the propositional constants $0$ and $1$ of type $o$; the equality constant $\approx$; the generalized disjunction and conjunction constants $\lor_\pi$ and $\land_\pi$, for every predicate type $\pi$; the generalized inverse implication constants $\leftarrow_\pi$, for every predicate type $\pi$.
6. The quantifier $\exists$.
7. The parentheses “(” and “)”.

Based on the above alphabet, the expressions of $\mathcal{H}$ can be composed as follows:

**Definition 6.** The set of expressions of the higher-order language $\mathcal{H}$ is recursively defined as follows:

1. Every predicate variable of type $\pi$ is an expression of type $\pi$; every argument variable of type $\rho$ is an expression of type $\rho$; every individual constant symbol of type $\iota$ is an expression of type $\iota$.
2. If $f$ is an $n$-ary function symbol and $E_1, \ldots, E_n$ are expressions of type $\iota$, then $(f \ E_1 \cdots \ E_n)$ is an expression of type $\iota$. 

3
3. If $E_1$ is an expression of type $\rho \rightarrow \pi$ and $E_2$ is an expression of type $\rho$, then $(E_1 \ E_2)$ is an expression of type $\pi$.

4. If $X$ is an argument variable of type $\rho$ and $E$ is an expression of predicate type $\pi$, then $(\lambda X. E)$ is an expression of type $\rho \rightarrow \pi$.

5. If $E_1, E_2$ are expressions of predicate type $\pi$, then $(E_1 \leftarrow \pi \ E_2)$ is an expression of type $o$, and $(E_1 \land_o \ E_2)$ and $(E_1 \pi E_2)$ are expressions of type $\pi$.

6. If $E_1, E_2$ are expressions of type $\iota$, then $(E_1 \approx \ E_2)$ is an expression of type $o$.

7. If $E$ is an expression of type $o$ and $Q$ is an argument variable (of any argument type), then $(\exists Q \ E)$ is an expression of type $o$.

To denote that an expression $E$ has type $\tau$, we will often write $E : \tau$; additionally, we write $\text{type}(E)$ to denote the type of expression $E$. Expressions of type $\iota$ will be called terms and of type $o$ will be called formulas. We will write $\leftarrow, \land$ and $\lor$ instead of $\leftarrow_o, \land_o$ and $\lor_o$. When writing an expression, the usual precedence rules will be used to avoid the excessive use of parentheses.

The notions of free and bound variables of an expression are defined as usual. An expression is called closed if it does not contain any free variables. Given an expression $E$, we denote by $\text{FV}(E)$ the set of all free variables of $E$. By overloading notation, we will also write $\text{FV}(S)$, where $S$ is a set of expressions.

In order to build a programming language based on $\mathcal{H}$, a certain syntactic subset of $\mathcal{H}$ must be considered:

**Definition 7.** A positive expression of $\mathcal{H}$ is one that does not contain $\leftarrow_\pi$.

**Definition 8.** A program clause of $\mathcal{H}$ is an expression of the form $p \leftarrow_\pi E$, where $p$ is a predicate variable called the head of the clause, and $E$ is a closed positive expression of $\mathcal{H}$. A program for $\mathcal{H}$ is a set of clauses.

**Definition 9.** A goal clause of $\mathcal{H}$ is an expression of the form $\leftarrow E$, where $E$ is a positive expression of type $o$. The empty clause is denoted by $\square$.

Notice that goal clauses may contain free argument variables (apart from the argument variables that are existentially quantified). Operationally speaking, the free argument variables that appear in a goal are the ones for which an answer is sought for by the proof procedure.

**Definition 10.** A Horn clause of $\mathcal{H}$ is either a program clause or a goal clause.

**Example 1.** The following is a higher-order program that computes the closure of its input binary relation $R$. The type of closure is $\pi = (\iota \rightarrow \iota \rightarrow o) \rightarrow \iota \rightarrow \iota \rightarrow o$.

\[
\text{closure} \leftarrow_\pi \lambda R. \lambda X. \lambda Y. (R X Y) \\
\text{closure} \leftarrow_\pi \lambda R. \lambda X. \lambda Y. (R X Y) \leftarrow_\pi (R X Y) \land (\text{closure} \ R Z Y)
\]

A possible query could be: "$\leftarrow \text{closure} \ R \ a \ b$" (which intuitively requests for all binary relations such that the pair $(a, b)$ belongs to their transitive closure).

4 The Semantics of $\mathcal{H}$

The semantics of $\mathcal{H}$ is built upon the notion of algebraic complete lattice. We start with the semantics of types and proceed with the semantics of expressions.
4.1 The Semantics of Types

The set-theoretic meaning of the types of $H$ are specified with respect to a set $D$ (where $D$ is later going to be the domain of our interpretations). The fact that a given type $\pi$ denotes a set $[\pi]_D$ will mean that an expression of type $\pi$ denotes an element of $[\pi]_D$. Similarly, an expression of type $\rho$ denotes an element of $[\rho]_D$. In the following definition we define simultaneously (and recursively) two things: the semantics $[\pi]_D$ of a type $\pi$ and the corresponding partial order $\sqsubseteq$.  

**Definition 11.** Let $D$ be a non-empty set. Then:

- $[\iota]_D = D$ and $\sqsubseteq_\iota$ is the binary relation such that $d \sqsubseteq_\iota d$, for all $d \in D$.
- $[\iota^n \to \iota]_D = D^n \to D$. A partial order for this case will not be needed.
- $[0]_D = \{0, 1\}$ and $\sqsubseteq_0$ is the numerical ordering on $\{0, 1\}$.
- $[\iota \to \pi]_D = D \to [\pi]_D$. Moreover, for all $f, g \in [\iota \to \pi]_D$, $f \sqsubseteq_\iota g$ if and only if $f(d) \sqsubseteq_\pi g(d)$, for all $d \in D$.
- $[\pi_1 \to \pi_2]_D = \mathcal{K}(\mathcal{K}([\pi_1]_D)) \to [\pi_2]_D$. Moreover, for all $f, g \in [\pi_1 \to \pi_2]_D$, $f \sqsubseteq_{\pi_1 \to \pi_2} g$ if and only if $f(d) \sqsubseteq_{\pi_1} g(d)$, for all $d \in \mathcal{K}(\mathcal{K}([\pi_1]_D))$.

Obviously each $[\pi]_D$ is an algebraic complete lattice ($\omega$-algebraic if $D$ is countable), due to the fact that the poset $\{0, 1\}$ is an $\omega$-algebraic complete lattice and Lemma 1. The following definition gives us a convenient shorthand that will be used in various places of the paper:

**Definition 12.** Let $D$ be a non-empty set and let $\rho$ be an argument type. Define:

$$F_D(\rho) = \begin{cases} D, &\text{if } \rho = \iota \\ \mathcal{K}([\rho]_D), &\text{otherwise} \end{cases}$$

The set $F_D(\rho)$ will be called the set of basic elements of type $\rho$.

**Example 2.** Consider the type $\iota \to o$ (a first-order predicate with one argument has this type). Then, $[\iota \to o]_D$ is the set of all functions from $D$ to $\{0, 1\}$ (or equivalently, of arbitrary subsets of $D$). Moreover, it can be verified by Definitions 1 and 12 that the set $F_D(\iota \to o)$ is the set of all finite functions from $D$ to $\{0, 1\}$ (or equivalently, of finite subsets of $D$).

As a second example, consider the type $(\iota \to o) \to o$. This is the type of a predicate which takes as its only parameter another predicate which is first-order. Then, $[(\iota \to o) \to o]_D$ is the set of all monotonic functions from finite sets (i.e., elements of $F_D(\iota \to o)$) to $\{0, 1\}$. In other words, in the semantics of the higher-order language that we will develop, a predicate of type $(\iota \to o) \to o$ will denote a monotonic function from finite subsets of $D$ to $\{0, 1\}$, i.e., it will denote a set of finite subsets of $D$ that respects monotonicity. On the other hand, it can be verified that the set $F_D((\iota \to o) \to o)$ contains all the relations that have the following property: each one of them can be written as the union of a finite number of simpler relations each one of which consists of a finite set of elements of $D$ together with all its finite supersets.

4.2 The Semantics of Expressions

We can now proceed to give meaning to the expressions of $H$. This is performed by first defining the notions of interpretation and state for $H$, that are similar to the corresponding notions for first-order languages:

**Definition 13.** An interpretation $I$ of $H$ consists of:

1. a nonempty set $D$, called the domain of $I$;
2. an assignment to each individual constant symbol $c$, of an element $I(c) \in D$;
3. an assignment to each predicate symbol $p : \pi$, of an element $I(p) \in [\pi]_D$;

4. an assignment to each function symbol $f$ of type $\tau^n \rightarrow \tau$, of a function $I(f) \in D^n \rightarrow D$.

**Definition 14.** Let $I$ be a given interpretation with domain $D$. Then, a state $s$ over $I$ is a function that assigns to each argument variable $Q$ of type $\rho$ of $\mathcal{H}$, an element $s(Q) \in F_D(\rho)$.

The key technical difficulty we now have to confront is the definition of the semantics of application. The problem that arises can be explained by an example (written in Prolog-like syntax):

- $p(Q) := Q(0), Q(s(0))$.
- $\text{nat}(0)$.
- $\text{nat}(s(X)) := \text{nat}(X)$.

Consider the query $\leftarrow p(\text{nat})$. The type of $p$ is $(\tau \rightarrow o) \rightarrow o$ while the type of $\text{nat}$ is $\tau \rightarrow o$. Let $I$ be an interpretation of our program with underlying domain $D$. Then, $I(p)$ must be a function from $F_D(\tau \rightarrow o)$ to $\{0,1\}$. According to Example 2, $F_D(\tau \rightarrow o)$ consists of finite sets of elements of $D$. But $I(\text{nat})$ is obviously an infinite set. How can we apply $I(p)$ to $I(\text{nat})$? The key idea is that if a higher-order predicate of our language is true of a relation, then this fact can be established by examining a “finite number of facts” about this relation. In our case, $p$ just examines for its input relation $Q$ whether it is true of 0 and $s(0)$. These remarks suggest that the meaning of $p(\text{nat})$ can be established as follows: we apply $I(p)$ to the “finite approximations” of $I(\text{nat})$, i.e., to all elements of $F_D(\tau \rightarrow o)[I(\text{nat})]$, and then take the least upper bound of the results. In our case $p(\text{nat})$ will be true since there exists a finite fragment of $I(\text{nat})$ for which $I(p)$ is true (namely the set $\{I(0), I(s(0))\}$.

In the following definition, $s[d/X]$ is used to denote a state that is identical to $s$, the only difference being that the new state assigns to $X$ the value $d$.

**Definition 15.** Let $I$ be an interpretation of $\mathcal{H}$, let $D$ be the domain of $I$, and let $s$ be a state over $I$. Then, the semantics of expressions of $\mathcal{H}$, with respect to $I$ and $s$, is defined as follows:

1. $[0]_s(I) = 0$
2. $[1]_s(I) = 1$
3. $[c]_s(I) = I(c)$, for every individual constant $c$
4. $[p]_s(I) = I(p)$, for every predicate variable $p$
5. $[Q]_s(I) = s(Q)$, for every argument variable $Q$
6. $[[f E_1 \cdots E_n]]_s(I) = I(f)[E_1]_s(I) \cdots [E_n]_s(I)$, for every n-ary function symbol $f$
7. $[[E_1 E_2]]_s(I) = \bigsqcup_{b_2 \in B_2}([E_1]_s(I)(b_2))$, where $B_2 = F_D(\text{type}(E_2))[E_1]_s(I)$
8. $[[\lambda X.E]]_s(I) = \lambda d. [E]_{s[d/X]}(I)$, where $d$ ranges over $F_D(\text{type}(X))$
9. $[[E_1 \leftarrow X.E_2]]_s(I) = \begin{cases} 1, & \text{if } [E_2]_s(I) \subseteq [E_1]_s(I) \\ 0, & \text{otherwise} \end{cases}$
10. $[[E_1 \lor X.E_2]]_s(I) = \bigsqcup_{\pi}([E_1]_s(I), [E_2]_s(I))$, where $\bigsqcup_{\pi}$ is the least upper bound function on $[\pi]_D$
11. $[[E_1 \land X.E_2]]_s(I) = \bigsqcup_{\pi}([E_1]_s(I), [E_2]_s(I))$, where $\bigsqcup_{\pi}$ is the greatest lower bound function on $[\pi]_D$
12. $[[E_1 \approx E_2]]_s(I) = \begin{cases} 1, & \text{if } [E_1]_s(I) = [E_2]_s(I) \\ 0, & \text{otherwise} \end{cases}$
4.3 Herbrand Interpretations

Herbrand interpretations are a cornerstone of first-order logic programming. Analogously, we have:

**Definition 17.** The Herbrand universe $U_H$ of $\mathcal{H}$ is the set of all terms that can be formed out of the individual constants and the function symbols of $\mathcal{H}$.

**Definition 18.** A Herbrand interpretation $I$ of $\mathcal{H}$ is an interpretation such that:

1. The domain of $I$ is the Herbrand universe $U_H$ of $\mathcal{H}$.
2. For every individual constant $c$, $I(c) = c$.
3. For every predicate symbol $p$ of type $\pi$, $I(p) \in [\pi]_{U_H}$.
4. For every $n$-ary function symbol $f$ and all $t_1, \ldots, t_n \in U_H$, $I(f)(t_1 \cdots t_n) = f(t_1 \cdots t_n)$.

Since all Herbrand interpretations have the same underlying universe, we will often refer to a “Herbrand state $s$”, meaning a state whose underlying universe is $U_H$. We will often also refer to an “interpretation of a set of formulas $S$” rather than the underlying language $\mathcal{H}$. In this case, we will implicitly assume that the set of individual constants and function symbols are those that appear in $S$. Under this assumption, we will often talk about the Herbrand universe $U_S$ of a set of formulas $S$. The set of Herbrand interpretations of a given program forms a complete lattice:

**Definition 19.** Let $P$ be a program and let $I_P$ be the set of Herbrand interpretations of $P$. We define the following partial order on $I_P$: for all $I, J \in I_P$, $I \subseteq J$ if and only if for every predicate variable $p$ of $P$, $I(p) \subseteq J(p)$.

**Lemma 2.** Let $P$ be a program and let $I_P$ be the set of Herbrand interpretations of $P$. Then $I_P$ is a complete lattice under $\subseteq_{I_P}$.

In the following we denote with $\bot_{I_P}$ the least element of $I_P$, i.e., the interpretation that assigns to each predicate $p : \pi$ of $P$ the element $\bot_{\pi}$.

5 Minimum Herbrand Model Semantics

The basic properties of logic programming extend to the higher-order case:

**Theorem 1** (Model Intersection Theorem). Let $P$ be a program and $\mathcal{M}$ a non-empty set of Herbrand models of $P$. Then, $\bigcap \mathcal{M}$ is a Herbrand model for $P$.

It is straightforward to check that every higher-order program $P$ has at least one Herbrand model $I$, namely the one which for every predicate symbol $p$ and for all basic elements $b_1, \ldots, b_n$ of the appropriate types, $I(p)(b_1 \cdots b_n) = 1$. Therefore, the intersection of all Herbrand models is well-defined, and by the above theorem is a model of the program. We will denote this model by $M_P$.

**Definition 20.** Let $P$ be a higher order program. The immediate consequence operator $T_P : I_P \rightarrow I_P$ is defined as $T_P(I)(p) = \bigsqcup_{(p \rightarrow e) \in P} [E(I)](e)$, for every $p : \pi$ in $P$ and for every $I \in I_P$. 

13. $[\exists Q E](s) = \begin{cases} 1, & \text{if there exists } d \in \mathcal{F}(\text{type}(Q)) \text{ such that } [E](d/\text{Q})(s) = 1 \\ 0, & \text{otherwise} \end{cases}$

For closed expressions $E$ we will often write $[E](I)$ instead of $[E](s)(I)$ (in this case, the meaning of $E$ is independent of $s$). We now define the notion of model:

**Definition 16.** Let $S$ be a set of closed formulas of $\mathcal{H}$ and let $I$ be an interpretation of $\mathcal{H}$. We say that $I$ is a model of $S$ if for every $F \in S$, $[F](I) = 1$. 

13. $[\exists Q E](s) = \begin{cases} 1, & \text{if there exists } d \in \mathcal{F}(\text{type}(Q)) \text{ such that } [E](d/\text{Q})(s) = 1 \\ 0, & \text{otherwise} \end{cases}$

For closed expressions $E$ we will often write $[E](I)$ instead of $[E](s)(I)$ (in this case, the meaning of $E$ is independent of $s$). We now define the notion of model:
Lemma 3. Let \( P \) be a program. Then the mapping \( T_P \) is continuous.

Define now the following sequence of interpretations:

\[
T_P \uparrow 0 = \bot_I
\]

\[
T_P \uparrow (n + 1) = T_P(T_P \uparrow n)
\]

\[
T_P \uparrow \omega = \bigsqcup \{ T_P \uparrow n \mid n < \omega \}
\]

The following theorem is entirely analogous to the one for the first-order case:

Theorem 2. Let \( P \) be a program. Then \( M_P = T_P \uparrow \omega \).

6 Proof Procedure

In this section we propose a sound and complete proof-procedure for extensional higher-order logic programming.

6.1 Basic Expressions

Basic elements (introduced in §4) have played an important rôle in the development of the semantics of our higher-order logic programming language. In order to devise a sound and complete proof procedure for our language, we first need to find a syntactic representation for basic elements:

Definition 21. The set of basic expressions of \( \mathcal{H} \) is recursively defined as follows. The basic expressions of type \( \iota \) are all expressions of \( \mathcal{H} \) of type \( \iota \). The basic expressions of type \( \o \) are \( 0 \) and \( 1 \).

A basic expression of type \( \rho_1 \to \cdots \to \rho_n \to \o \) is a non-empty finite union of \( \lambda \)-abstractions, each of which has one of the following forms:

1. \( \lambda X_1. \cdots \lambda X_n.0 \)
2. \( \lambda X_1. \cdots \lambda X_n.1 \)
3. \( \lambda X_1. \cdots \lambda X_n.A_1 \land \cdots \land A_m \), where \( m > 0 \) and each \( A_i \) is
   
   (a) \( (X_k \approx t) \), if \( X_k \) is of type \( \iota \) and \( t \) is a basic expression of type \( \iota \) whose variables are different from \( X_1, \ldots, X_n \),
   
   (b) \( X_k \), if \( X_k \) is of type \( \o \), or
   
   (c) \( X_k(B_1) \cdots (B_r) \), if \( X_k \) is of type \( \rho'_1 \to \cdots \to \rho'_r \to \o \) and each \( B_j \) is a basic expression of type \( \rho_j \).

In the above definition, if any \( \rho_k \) is of type \( \iota \) then the body of each abstraction within the finite union must either be \( 0 \) or contain exactly one expression of the form \( (X_k \approx t) \).

The proof procedure that will be developed later in this section relies on a special form of basic expressions:

Definition 22. A basic expression \( B \) is called a basic template if in every subexpression of the form \( (X \approx t) \) in \( B \), the term \( t \) is a variable that does not appear in any other place of \( B \).

The following two lemmas suggest that basic expressions are the syntactic analogues of basic elements:

Lemma 4. For every basic expression \( B : \rho \), for every interpretation \( I \) with domain \( D \), and for every state \( s \) over \( I \), \( [B]_s(I) \in \mathcal{F}_D(\rho) \).

The converse of the above lemma holds if we restrict attention to Herbrand interpretations and basic elements over the Herbrand universe.

Lemma 5. Let \( \rho \) be any argument type and let \( b \in \mathcal{F}_{\forall \rho}(\rho) \). Then, there exists a ground basic expression \( B : \rho \) such that for every Herbrand interpretation \( I \), \( [B](I) = b \).
6.2 Substitutions and Unifiers

Definition 23. A substitution \( \theta \) is a finite set \( \{V_1/E_1, \ldots, V_n/E_n\} \), where the \( V_i \)'s are different argument variables of \( H \) and each \( E_i \) is an expression of \( H \) having the same type as \( V_i \). We write \( \text{dom}(\theta) = \{V_1, \ldots, V_n\} \) and \( \text{range}(\theta) = \{E_1, \ldots, E_n\} \). A substitution is called basic if all \( E_i \) are basic expressions.

Definition 24. Let \( \theta \) be a substitution and let \( E \) be a positive expression. Then, \( E\theta \) is an expression obtained from \( E \) as follows:

- \( E\theta = E \), if \( E \) is 0, 1, c or p.
- \( Q\theta = \theta(Q) \) if \( Q \in \text{dom}(\theta) \); otherwise, \( Q\theta = Q \).
- \( (f E_1 \cdots E_n)\theta = (f E_1\theta \cdots E_n\theta) \).
- \( (E_1 E_2)\theta = (E_1\theta E_2\theta) \).
- \( (\lambda X.E_1)\theta = (\lambda Z. (E_1\{X/Z\})\theta) \), where \( Z \not\in \text{FV}(E_1) \cup \text{FV}(\text{dom}(\theta)) \cup \text{FV}(\text{range}(\theta)) \).
- \( (E_1 \lor E_2)\theta = (E_1\theta \lor E_2\theta) \).
- \( (E_1 \land E_2)\theta = (E_1\theta \land E_2\theta) \).
- \( (E_1 \equiv E_2)\theta = (E_1\theta \equiv E_2\theta) \).
- \( (\exists Q E_1)\theta = (\exists Z. (E_1\{Q/Z\})\theta) \), where \( Z \not\in \text{FV}(E_1) \cup \text{FV}(\text{dom}(\theta)) \cup \text{FV}(\text{range}(\theta)) \).

The composition of substitutions can be defined in a similar way as in the first-order case:

Definition 25. Let \( \theta = \{V_1/E_1, \ldots, V_m/E_m\} \) and \( \sigma = \{Q_1/E'_1, \ldots, Q_n/E'_n\} \) be substitutions. Then the composition \( \theta\sigma \) of \( \theta \) and \( \sigma \) is the substitution obtained from the set

\[ \{V_1/E_1\sigma, \ldots, V_m/E_m\sigma, Q_1/E'_1, \ldots, Q_n/E'_n\} \]

by deleting any \( V_i/E_i\sigma \) for which \( V_i = E_i\sigma \) and deleting any \( Q_j/E'_j \) for which \( Q_j \in \{V_1, \ldots, V_m\} \).

The substitution corresponding to the empty set will be called the identity substitution and will be denoted by \( \epsilon \). The following proposition is easy to establish:

Proposition 1. Let \( \theta, \sigma \) and \( \gamma \) be substitutions. Then:

1. \( \epsilon\theta = \epsilon \theta = \theta \).
2. For all positive expressions \( E \), \( (E\theta)\sigma = E(\theta\sigma) \).
3. \( (\theta\sigma)\gamma = \theta(\sigma\gamma) \).

where equality should be understood as \( \alpha \)-congruence, i.e., as syntactic equality subject to a possible renaming of bound variables.

Definition 26. Let \( S \) be a set of terms of \( H \) (i.e., expressions of type \( \iota \)). A substitution \( \theta \) will be called a unifier of the expressions in \( S \) if the set \( S\theta = \{E\theta \mid E \in S\} \) is a singleton. The substitution \( \theta \) will be called a most general unifier of \( S \) (denoted by \( \text{mgu}(S) \)), if for every unifier \( \sigma \) of the expressions in \( S \), there exists a substitution \( \gamma \) such that \( \sigma = \theta\gamma \).
6.3 SLD-Resolution

We now proceed to define the notions of answer and correct answer. Notice that both of these notions rely on basic (i.e., not arbitrary) substitutions:

**Definition 27.** Let $P$ be a program and $G$ a goal. An answer for $P \cup \{G\}$ is a basic substitution for free variables of $G$.

**Definition 28.** Let $P$ be a program, $G \leftarrow E$ a goal clause and $\theta$ an answer for $P \cup \{G\}$. We say that $\theta$ is a correct answer for $P \cup \{G\}$ if for every model $M$ of $P$ and for every state $s$ over $M$, $[[E\theta]]_s(M) = 1$.

**Definition 29.** Let $P$ be a program and let $G \leftarrow A$ and $G' \leftarrow A'$ be goal clauses. Then, we will say that $A'$ is derived in one step from $A$ using the basic substitution $\theta$ (or equivalently that $G'$ is derived in one step from $G$ using $\theta$), and we denote this fact by $A \xrightarrow{\theta} A'$ (respectively, $G \xrightarrow{\theta} G'$) if one of the following conditions applies:

1. $p \cdot E_1 \cdot \cdots \cdot E_n \xrightarrow{\cdot} E_1 \cdot \cdots \cdot E_n$, where $p \leftarrow E$ is a rule in $P$.
2. $Q \cdot E_1 \cdot \cdots \cdot E_n \xrightarrow{\theta} (Q \cdot E_1 \cdot \cdots \cdot E_n)\theta$, where $\theta = \{Q/B\}$ and $B$ is a basic template such that $FV(B) \cap FV\{E_1, \ldots, E_n\} = \emptyset$.
3. $(\lambda X.E) \cdot E_1 \cdot \cdots \cdot E_n \xrightarrow{\cdot} (E[X/E_1]) \cdot E_2 \cdot \cdots \cdot E_n$.
4. $(E' \setminus_x E'') \cdot E_1 \cdot \cdots \cdot E_n \xrightarrow{\cdot} E' \cdot E_1 \cdot \cdots \cdot E_n$.
5. $(E' \setminus_x E'') \cdot E_1 \cdot \cdots \cdot E_n \xrightarrow{\cdot} E'' \cdot E_1 \cdot \cdots \cdot E_n$.
6. $(E' \setminus_x E'') \cdot E_1 \cdot \cdots \cdot E_n \xrightarrow{\cdot} (E' \cdot E_1 \cdot \cdots \cdot E_n) \land (E'' \cdot E_1 \cdot \cdots \cdot E_n)$.
7. $(E_1 \land E_2) \xrightarrow{\theta} (E'_1 \land (E_2\theta))$, if $E_1 \xrightarrow{\theta} E'_1$.
8. $(E_1 \land E_2) \xrightarrow{\theta} ((E_1\theta) \land E'_2)$, if $E_2 \xrightarrow{\theta} E'_2$.
9. $(\square \land E) \xrightarrow{\cdot} E$
10. $(E \land \square) \xrightarrow{\cdot} E$
11. $(E_1 \approx E_2) \xrightarrow{\theta} \square$ where $\theta$ is an mgu of $E_1$ and $E_2$.
12. $(\exists Q.E) \xrightarrow{\cdot} E$.

**Definition 30.** Let $P$ be a program and $G$ a goal. An SLD-derivation of $P \cup \{G\}$ consists of (possibly infinite) sequences $G_0 = G$, $G_1$, $\ldots$ of goals and $\theta_1$, $\theta_2$, $\ldots$ of basic substitutions such that each $G_{i+1}$ is derived in one step from $G_i$ using $\theta_{i+1}$.

**Definition 31.** Let $P$ be a program and $G$ a goal. An SLD-refutation of $P \cup \{G\}$ is a finite SLD-derivation of $P \cup \{G\}$ which has the empty clause $\square$ as the last goal in the derivation. If $G_n = \square$, then we say that the refutation has length $n$.

**Definition 32.** Let $P$ be a program and $G$ a goal. A computed answer $\theta$ for $P \cup \{G\}$ is the basic substitution obtained by restricting the composition $\theta_1 \ldots \theta_n$ to the free variables of $G$, where $\theta_1, \ldots, \theta_n$ is the sequence of the basic substitutions used in an SLD-refutation of $P \cup \{G\}$ of length $n$. 
Example 3. Consider the program of Example 1. We have:

\[
\begin{align*}
\text{closure } Q \ a \ b & \quad \theta_0 = \epsilon \\
(\lambda R.\lambda X.\lambda Y.(R \ X \ Y)) \ Q \ a \ b & \quad \theta_1 = \epsilon \\
Q \ a \ b & \quad \theta_2 = \{Q/(\lambda X.\lambda Y.(X \approx X_0) \land (Y \approx Y_0))\} \\
(\lambda X.\lambda Y.(X \approx X_0) \land (Y \approx Y_0)) \ a \ b & \quad \theta_3 = \epsilon \\
(a \approx X_0) \land (b \approx Y_0) & \quad \theta_4 = \{X_0/a\} \\
\Box \land (b \approx Y_0) & \quad \theta_5 = \epsilon \\
(b \approx Y_0) & \quad \theta_6 = \{Y_0/b\}
\end{align*}
\]

where some simple steps involving \(\lambda\)-abstractions have been omitted. The composition of the above substitutions gives the substitution \(\sigma_1\) below. Similarly we can get \(\sigma_2\), etc.

\[
\sigma_1 = \{Q/\lambda X.\lambda Y.(X \approx a) \land (Y \approx b)\} \\
\sigma_2 = \{Q/(\lambda X.\lambda Y.(X \approx a) \land (Y \approx Z)) \lor (\lambda X.\lambda Y.(X \approx Z) \land (Y \approx b))\} \\
\]

\[\ldots\]

Notice that \(\sigma_1\) above corresponds to the set \{\((a,b)\)\}, \(\sigma_2\) to the set \{\((a,Z),(Z,b)\)\}, for every \(Z\) in the Herbrand universe, and so on.

6.4 Soundness and Completeness of SLD-resolution

The proofs of soundness and completeness of the proposed proof procedure (which due to space limitations will appear in the full version of the paper) follow along similar lines as the corresponding results for the first-order case.

Theorem 3 (Soundness). Let \(P\) be a program and \(G\) a goal. Then, every computed answer for \(P \cup \{G\}\) is a correct answer for \(P \cup \{G\}\).

As in the first-order case, we have various forms of completeness. We start with the analogue of a theorem due to Apt and van Emden (see [5]).

Theorem 4. Let \(P\) be a program and \(G\) a goal and \([G]_s(M_P) = 1\) for all states \(s\). Then, there is a refutation for \(P \cup \{G\}\) using the identity substitution.

The following is a generalization of Hill’s theorem (see [5, Theorem 8.4]) for the higher-order case:

Theorem 5. Let \(P\) be a program, \(G\) a goal and assume that \(P \cup \{G\}\) is unsatisfiable (i.e., it does not have any models). Then, there exists an SLD-refutation of \(P \cup \{G\}\).

Finally, the following is a generalization of Clark’s theorem (see [5, Theorem 8.6]) for the higher-order case:

Theorem 6 (Completeness). Let \(P\) a program and \(G\) a goal. For every correct answer \(\theta\) for \(P \cup \{G\}\), there exists an SLD-refutation for \(P \cup \{G\}\) using the computed answer \(\sigma\) and a basic substitution \(\gamma\) such that \(\theta = \sigma \gamma\).

7 Conclusions

An implementation of the proposed proof procedure has been performed in Haskell.\(^1\) The main difference in comparison to a first-order implementation is that the proof procedure has to generate

\(^1\)The code can be retrieved from \(\text{http://code.haskell.org/hopes}\).
an infinite (yet enumerable) number of basic templates. This affects the search tree, making it in general infinite not only in depth but also in breadth. As a result, the naïve depth first search strategy would in general be unfair with respect to the enumeration of the solutions. In our implementation we use the strategy for interleaving different solutions proposed in [4], which solves the search problem in a satisfactory way.

References


