Hiord: A Type-Free Higher-Order Logic Programming Language with Predicate Abstraction

Daniel Cabeza[1], Manuel Hermenegildo[1,2], and James Lipton[1,3]

[1] Technical University of Madrid, Spain
{dcabeza,herme,jlipton}@fi.upm.es
[2] University of New Mexico, USA
herme@unm.edu
[3] Wesleyan University, USA
jlipton@wesleyan.edu

Abstract. A new formalism, called Hiord, for defining type-free higher-order logic programming languages with predicate abstraction is introduced. A model theory, based on partial combinatory algebras, is presented, with respect to which the formalism is shown sound. A programming language built on a subset of Hiord, and its implementation are discussed. A new proposal for defining modules in this framework is considered, along with several examples.

1 Introduction

This paper presents a new declarative formalism, called Hiord, for logic programming with untyped higher-order logic and predicate abstractions. This is followed by a discussion of various practical restrictions of this logic to make it amenable to speedy translation to WAM-compilable code and static analysis.

A number of proposals have been made over the past two decades to introduce higher-order features into logic programming in a declarative fashion by extending the underlying logic, among them λProlog and Hilog [1–4]. This has proven a very useful way to place on a solid logical ground certain natural steps that, in the original first-order context of pure logic programming, seem to compromise declarative transparency. For example, the simple transformation of code such as the following:

```
all(Prop,[]).
all(Prop,[H|Tl]) :- call(Prop,H), all(Prop,Tl).
```

to

```
all(Prop,[]).
all(Prop,[H|Tl]) :- Prop(H), all(Prop,Tl).
```

or a typed version thereof, turns a Prolog meta-program into a fully declarative program in higher-order logic. This simple example tells only a small part

of the story, of course, as there are other ways to give a declarative semantics to meta-predicates. However, in our view, higher-order logic with predicate abstraction is an excellent choice for bringing metaprogramming within the scope of declarative programming, especially when management of substitutions in the object language is involved, via higher-order abstract syntax. It is also a natural framework for robust declarative treatment of automated deduction, and code and data specification.

It is also our feeling that translation of explicit higher-order notions into a first-order formalism simply places the original specification at a greater distance from the program semantics, and hence, the programmer a greater distance from the aims of declarative programming.

The work discussed in this paper extends the *untyped* classical first-order Horn clauses of core Prolog to untyped classical Horn clauses in higher-order logic, with predicate abstractions allowed. The main rationale for keeping types out of the picture is compatibility with existing Prolog code and Prolog systems, with all their tools for static analysis and program development, and so as to implement higher-order programming as a package loadable from Prolog. We are proposing an extension to the syntax and semantics of core Prolog, not a compilation into it, as a basis for the syntax and semantics of the input code, (irrespective of whether or not the implementation actually does compile the code to Prolog in the end).

A second, important consideration is that we consider powerful applications, including a new proposal for declarative definition of modules, that make use of self-application and head flex variables that is *not* typable in simply typed lambda-calculus, and, in particular, not legal λProlog code. Many applications may, in fact, be typable in a sufficiently strong type discipline. Even so, our eventual interest is in compile-time type inference through static analysis of type-free code.

There is a type free higher order extension of Prolog, namely Hilog, which has been presented [4] with a proof theory and a semantics based on (the well-known) translation of higher-order logic into first-order logic. It lacks predicate abstraction, however, which for us is an essential feature of Hiord. Although our formalism is inspired by the Hilog work, the semantics requires significant reworking to permit abstraction.

The paper is divided into two parts. The first presents a strong formalism that allows higher-order resolution and term-rewriting with unrestricted abstraction of all terms and goals. The aim is to define a framework within which any number of practical restrictions can be studied. In this formalism we continue along the lines of the "anything goes" philosophy of *Hilog*. All terms can have a truth value, compound terms can be functors, a functor can have multiple arities. We define a model theory based on partial combinatory algebras [5] with certain semilattices serving as an object of truth-values, and show that our completely general resolution is sound.

The formalism defined is, in a sense, too strong to be a useful programming language. Since it contains the full untyped lambda-calculus, it permits untyped

higher-order logical-functional programming. Indeed, one could virtually ignore the logic and simply program in the lambda-calculus (which, of course, is not our aim). For this reason we regard the Hiord formalism as more of a blueprint for defining restricted type-free higher order languages, and we have included a second section in which a restriction of the language is discussed along with an implementation, the *Hiord* package included in the latest release (1.11) of the Ciao system.

A serious concern, of course, is that by combining higher-order logic with a type-free function calculus with a fixed point operator one is coming dangerously close to inconsistency. Indeed if one adds unrestricted abstraction to the *full logic* of *Hilog* (as compared to the Horn Clause subset), one has an easy formalization of Curry's paradox, a simple variant of Russell's, by defining a predicate $p = \lambda x.\neg x(x)$. Then we have $p(p) = \neg p(p)$! More subtle paradoxes can be found even in the absence of negation (see e.g. [6, 7]). We steer away from these problems by staying within the Horn fragment of logic with SLD resolution, which is shown sound with respect to the model theory introduced in section 3.

2 The Syntax of Formal Hiord

We initially consider a language with a very liberal syntax, which incorporates the flexibility of Hilog by allowing arity-free functors, and not distinguishing between functors and relators.

A *language* for Hiord is a set $\mathcal{S}$ of non-logical parameters (which will contain all names for constants, function and relation symbols). It is also equipped with a set of variables $\mathcal{V}$, as well as the logical parameters "," (comma), = (equality) and "E" (existence).

Hiord terms and formulas are defined by mutual recursion, as shown in the table below:

Terms:

1. A variable is a term.
2. A nonlogical parameter is a term.
3. If $t, t_1, \ldots, t_n$ are terms, then $t(t_1, \ldots, t_n)$ is a term, called a *simple* term if $t \in \mathcal{S} \cup \mathcal{V}$.
4. If G is a goal and $\boldsymbol{x}$ is a sequence of variables, $\{(\boldsymbol{x})\text{:-}G\}$ is a term, known as an *abstraction*.

Atomic Formulas, Goals and clauses

1. $\top$ is an atomic formula, called *true*.
2. If $t, t_1, \ldots, t_n$ are terms, then $t(t_1, \ldots, t_n)$ is an atomic formula. If $t \in \mathcal{S}$ the formula is called *rigid*, and if $t \in \mathcal{V}$ *flex*.
3. If t_1 and t_2 are terms then $t_1 = t_2$ is an atomic formula.
4. An atomic formula is a goal. If G_1 and G_2 are goals, then the *conjunction* G_1, G_2 is a goal, and for any goal G and variable x, $E(x)G$ is a goal.
5. A clause is a formula of the form $H \leftarrow G$ where H is a rigid atomic formula with no occurrences of equality or abstractions, and G is a goal.

Definition 1. *A* Hiord *logic* **program** *is a finite set of clauses. A* **state** *is a pair* $\langle P|\mathfrak{g}\rangle$ *where* P *is a program, and* $\mathfrak{g}$ *is a sequence of goals. The empty sequence, denoted* $\Box$*, is allowed. Goal sequences are defined by the following grammar:*

$$\mathfrak{g} ::= \Box \mid G \mid \mathfrak{g} \otimes \mathfrak{g}$$

When the program is understood from context it may be omitted. When we write $\langle P, A \leftarrow Tl \mid \mathfrak{g}\rangle$, it is understood that we are distinguishing one of the clauses $A \leftarrow Tl$ of the program P. All program clauses are treated as tacitly closed by standardizing variables in clauses apart from any other variables appearing in a state or a deduction. For this reason, application of a unifying substitution θ to a state $\langle P|\mathfrak{g}\rangle$ results in the new state $\langle P|\mathfrak{g}\theta\rangle$.

Definition 2. *A* **substitution** θ *is a map from variables in* $\mathcal{V}$ *to terms over* $\mathcal{V} \cup \mathcal{S}$*. Such a map lifts to a unique map (also denoted* θ*) from terms or goals to terms or goals, defined as follows:*

$$\begin{aligned} t(t_1, \ldots, t_n)\theta &= t\theta(t_1\theta, \ldots, t_n\theta) \\ \{(\boldsymbol{x}) \texttt{:-} G\}\theta &= \{(\boldsymbol{y}) \texttt{:-} G[\boldsymbol{x} := \boldsymbol{y}]\theta\} \end{aligned}$$

where $\boldsymbol{y}$ *is a sequence of variables (of the same length as* $\boldsymbol{x}$*) which are disjoint from the domain and range of* θ*, and* $G[\boldsymbol{x} := \boldsymbol{y}]$ *is the result of simultaneously replacing, in* G*, every free occurrence of each variable* x_i *in the sequence* $\boldsymbol{x}$ *with the corresponding variable* y_i *of* $\boldsymbol{y}$*. For nonatomic goals* G_1, G_2 *and* $\mathtt{E}(x)G$ *we define*

$$(G_1, G_2)\theta = (G_1\theta, G_2\theta) \text{ and } (\mathtt{E}(x)G)\theta = (\mathtt{E}(y)G[x := y]\theta)$$

with the same conditions as above for x *and* y*.*

Substitutions lift to goal sequences in the obvious way. We now give resolution proof rules for Hiord logic programs.

Definition 3. *A resolution step for* Hiord *is a ternary relation* $\rightsquigarrow$ *on states* $\times$ *subs* $\times$ *states. We write* $\mathfrak{s}_1 \overset{\theta}{\rightsquigarrow} \mathfrak{s}_2$ *instead of* $(\mathfrak{s}_1, \theta, \mathfrak{s}_2) \in \rightsquigarrow$*.*

There are six kinds of resolution steps. In the rules defining them, below, equality may be taken as one of **strict** *equality (the terms must be identical without any reduction taking place),* **alpha***-equivalence, or* **beta-eta** *equivalence.*

1. **Backchain:** $\langle P, A \leftarrow Tl|\mathfrak{g} \otimes A' \otimes \mathfrak{g}'\rangle \overset{\theta}{\rightsquigarrow} \langle P, A \leftarrow Tl|\mathfrak{g}^\theta\rangle \otimes Tl\theta \otimes \mathfrak{g}'\theta$ *where* $\theta A = \theta A'$*.*
2. **Unify:** $\langle P|\mathfrak{g} \otimes t_1 = t_2 \otimes \mathfrak{g}\rangle \overset{\theta}{\rightsquigarrow} \langle P|\mathfrak{g}\theta \otimes \mathfrak{g}'\theta\rangle$ *where* $t_1\theta = t_2\theta$*.*
3. **Reduce:** $\langle P|\mathfrak{g}\rangle \overset{\beta}{\rightsquigarrow} \langle P|\mathfrak{g}'\rangle$*. denoting any* α *or* β *reduction (or conversion, their congruence closure) of a term or subterm in a sequence of goals.*
4. **Conjunction:** $\langle P|\mathfrak{g} \otimes G_1, G_2 \otimes \mathfrak{g}'\rangle \overset{\otimes}{\rightsquigarrow} \langle P|\mathfrak{g} \otimes G_1 \otimes G_2 \otimes \mathfrak{g}'\rangle$
5. **Existence:** $\langle P|\mathfrak{g} \otimes \exists x.G \otimes \mathfrak{g}'\rangle \overset{\exists}{\rightsquigarrow} \langle P|\mathfrak{g} \otimes G \otimes \mathfrak{g}'\rangle$ *where the bound variable* x *is assumed distinct from any other variable occurring in* $\mathfrak{g} \otimes G \otimes \mathfrak{g}'$*.*
6. **True:** *The instantiation of head variable in a flex goal* $X, X(t_1, \ldots, t_n)$ *by* $\top$ *or* $\lambda(x_1, \ldots, x_n).\top$ *is a resolution step,* $\langle P|\mathfrak{g}\rangle \overset{\theta}{\rightsquigarrow} \langle P|\mathfrak{g}\theta\rangle$*, with substitution* $\theta = [X := \top]$ *or* $[X := \lambda(x_1, \ldots, x_n).\top]$ *as the case may be.*

To fix one formalism, we take the last option for equality, $\beta\eta$ equivalence, as the "official equality" of Hiord. This, of course, requires the implementation of potentially non-terminating higher-order unification. In practice, this is one of the areas where restrictions are of interest [8].

The symbols $\exists$, β and $\otimes$ written over the reduction symbol, a notational convenience, are just different names for the identity substitution. A **convention** we will adopt, except where otherwise indicated, in this paper, is that the unifying substitution displayed over a reduction arrow is understood to be restricted to the free variables in its source.

All bound variables in explicit existential quantifications and lambda-abstractions are assumed to be distinct from each other, all free variables present in each state, and all variables occurring in (the domain or range of) substitutions.

Finally, we define a resolution proof of a state $\langle P|\mathfrak{g}\rangle$ to be a sequence of resolution steps ending with the empty sequence $\Box$ of goals.

3 Semantics

We will need the following algebraic notion of an object of truth values, which can be thought of as a limited Boolean or Heyting algebra.

Definition 4. *An LP-algebra* $\mathbf{\Omega} = (\Omega, \top, \leq, \wedge)$ *consists of a meet-semilattice* $(\Omega, \leq, \wedge)$ *with a top element* $\top$.

In the presence of a Hiord structure, defined below, LP-agebras will be required to have certain potentially infinite parametrized suprema.

Definition 5. *A* Hiord **structure**

$$\mathfrak{A} = \langle U, \mathfrak{s}, \mathfrak{k}, \top, \wedge_{\mathfrak{A}}, \exists_{\mathfrak{A}}, \mathsf{eq}_{\mathfrak{A}}, \odot, p, \pi_l, \pi_r, \omega, \Omega\rangle$$

is given by the following data:

1. *A nonempty set* U*, called the carrier, domain or underlying set of* $\mathfrak{A}$ *, and also denoted* $|\,\mathfrak{A}\,|$*, with* $\top, \mathfrak{s}, \mathfrak{k}, \wedge_{\mathfrak{A}}, \exists_{\mathfrak{A}}, p, \pi_l, \pi_r \in U$.
2. $\langle U, \mathfrak{s}, \mathfrak{k}, \odot\rangle$ *is a partial combinatory algebra with pairing operator* p *and projections* π_l, π_r
3. *An LP algebra* $(\Omega, \top, \leq, \wedge)$ *with all (finite and infinite)* U-parametrized *joins required in the following definition of* ω.
4. *A partial map* $\omega : U \rightharpoonup \Omega$.

This structure must also satisfy the following conditions (where juxtaposition denotes left-associative application $\odot$*):*

$$\pi_l(puv) = u \qquad \pi_r(puv) = v$$
$$\mathfrak{s}uvw = (uw)(vw)$$
$$\mathfrak{k}uv = u$$

$$\begin{aligned}
\omega(\top) &= \top_\Omega \\
\omega(\wedge_{\mathfrak{A}} uv) &= \wedge_\Omega(\omega(u), \omega(v)) \\
\omega(\exists_{\mathfrak{A}} u) &= \bigvee\{\omega(ud) : d \in U \textit{ and } ud \textit{ defined in } U\} \\
\omega(\mathsf{eq}_{\mathfrak{A}} u_1 u_2 \cdots u_n) &= \top_\Omega \textit{ iff } u_1 = u_2 = \cdots = u_n
\end{aligned}$$

In a Hiord structure, the (possibly) infinite meets used in the condition for $\omega(\exists_{\mathfrak{A}} u)$ must exist. This condition is considerably weaker than requiring arbitrary suprema to exist, but the reader may take Ω to be complete lattice without significant loss of generality.

In the presence of a pairing operator and projections, we can assume the existence of the following derived notions of *n-tuples* and *n-ary projections* (where Parenthesized superscripts denote iteration):

$$\begin{aligned}
\langle u \rangle &:= u \\
\langle u_1, \ldots, u_n \rangle &:= p u_1 \langle u_2, \ldots, u_n \rangle \qquad (n \geq 2) \\
\pi^n_k &:= \begin{cases} \pi_l & \text{if } k = 1 < n \\ \pi^{n-1}_{k-1} \pi_r & \text{if } 1 < k < n \\ \pi_r^{(n-1)} & \text{if } k = n \end{cases}
\end{aligned}$$

If $u \in \mathfrak{A}$, we write $(u)_i$ for π^n_i when n is clear from context.

With these definitions, we have $\pi^n_k \langle u_1, \ldots, u_n \rangle = u_k$ and if $u = \langle u_1, \ldots, u_n \rangle$ then $(u)_k = u_k$ in $\mathfrak{A}$.

Definition 6. *Let $\mathcal{S}$ be a set of parameters. Let $\mathfrak{A}$ be a* Hiord *structure. An* **$\mathfrak{A}$-assignment** *is a map*

$$I : \mathcal{S} \to U,$$

and an **$\mathfrak{A}$-environment** *is a map*

$$\nu : \mathcal{V} \to U.$$

A structure $\mathfrak{A}$, an assignment I and an $\mathfrak{A}$ -environment ν induce an **interpretation**, that is to say, a map $\nu[\![\_]\!]^{\mathfrak{A},I}$ (abbreviated to $\nu[\![\_]\!]$ when the remaining parameters are clear from context) from terms and formulas to the domain U of $\mathfrak{A}$ as follows:

$$\begin{aligned}
\nu[\![X]\!] &= \nu(X) \text{ defined, for } X \in \mathcal{V} && (1) \\
\nu[\![s]\!] &= I(s) \text{ for } s \in \mathcal{S} && (2) \\
\nu[\![t(t_1, \ldots, t_n)]\!] &= \nu[\![t]\!]\langle \nu[\![t_1]\!], \ldots, \nu[\![t_n]\!] \rangle && (3) \\
\nu[\![t_1 = t_2]\!] &= \mathsf{eq}_{\mathfrak{A}} \nu[\![t_1]\!] \nu[\![t_2]\!] && (4) \\
\nu[\![\{(\boldsymbol{x}) : \text{-} G\}]\!] &= [\mathbf{x}] \nu[\![G[x_i := (\mathbf{x})_i]_{1 \leq i \leq n}]\!] \qquad \text{where } \boldsymbol{x} \equiv (x_1, \ldots, x_n) && (5) \\
\nu[\![G_1, G_2]\!] &= \wedge_{\mathfrak{A}} \nu[\![G_1]\!] \nu[\![G_2]\!] && (6) \\
\nu[\![\exists x G]\!] &= \exists_{\mathfrak{A}} [\mathbf{x}] \nu[\![G[x := \mathbf{x}]]\!] && (7)
\end{aligned}$$

and where each $\mathbf{x_i}$ is a fresh variable not in $\mathcal{V}$ (and hence not in the domain of ν) and $[\mathbf{x_i}]u$ denotes so-called *bracket abstraction* in the model, definable in any partial combinatory algebra, and described below. First, we briefly note that in the setting of a combinatory algebra, currying of terms is automatically enforced, since a sequence of applications, allowed in our syntax, has the semantics $\nu[\![(t_1 \cdots t_n)]\!] = \nu[\![t_1]\!] \cdots \nu[\![t_n]\!]$. However we incorporate the conventional syntax of core prolog by treating multiple arity arguments as vectors in clause 3 of the preceding definition.

3.1 Bracket Abstraction over $\mathfrak{A}$

Syntactic translation of closed λ-terms to variable free combinatory logic is well-known, and has been used extensively in compilation of functional programming languages. Here we have to consider the additional wrinkle of doing it with respect to an ambient model, so we give the details.

In order to define *semantic* bracket abstraction and the interpretation of terms and formulas with bound variables rigorously, we will need to make use of several intermediate notions of term: those built up from elements of the carrier U of a Hiord structure and a fresh set variables, and the collection of Hiord terms and goals built up using two sets of variables. Let $\mathcal{W}$ be a fresh set of variables (i.e. disjoint from $\mathcal{S} \cup \mathcal{V}$), in one-to-one correspondence with $\mathcal{V}$, via the mapping $x \mapsto \mathbf{x}$.

Let $\mathfrak{A}$ be a Hiord structure with carrier U, and let $U[\mathcal{W}]$ be the set of terms freely built from U and $\mathcal{W}$ using application in $\mathfrak{A}$:

- if $\mathbf{x} \in \mathcal{W}$ then $\mathbf{x} \in U[\mathcal{W}]$,
- if $u, v \in U[\mathcal{W}]$ then $uv \in U[\mathcal{W}]$.

We extend $U[\mathcal{W}]$ to the set $\lambda U[\mathcal{W}]$ of bracket-abstracted terms as follows:

- if $\mathbf{x} \in \mathcal{W}$ then $\mathbf{x} \in \lambda U[\mathcal{W}]$,
- if $u, v \in \lambda U[\mathcal{W}]$ then $uv \in \lambda U[\mathcal{W}]$.
- if $u \in \lambda U[\mathcal{W}]$ and $\mathbf{x} \in \mathcal{W}$ then $[\mathbf{x}]u \in \lambda U[\mathcal{W}]$.

Let $\wp[\mathcal{W}]$ be the set of Hiord terms and goals containing occurrences (possibly bound) of variables in $\mathcal{V}$ and only free occurrences of variables in $\mathcal{W}$. Let $\nu : \mathcal{V} \to U$ be a $\mathfrak{A}$-environment. Extend ν to a function $\nu : \mathcal{V} \cup \mathcal{W} \to U[\mathcal{W}]$ by defining it to be the identity function on $\mathcal{W}$, and then to a function $\nu : \wp[\mathcal{W}] \to U[\mathcal{W}]$ in the usual way, i.e. according to the equations (1-7). Then, the result of applying ν to goals using these equations is a bracket abstraction, i.e. a term in $\lambda U[\mathcal{W}]$. These expressions denote members of U according to the following rules:

Definition 7. *We define* **bracket abstraction** *with respect to* $(\mathfrak{A}, \nu)$. *Terms in* $\lambda U[\mathcal{W}]$ *denote the following unique members of* U*:*

$$[\mathbf{x}]\mathbf{x} = \mathfrak{s}\mathfrak{k}\mathfrak{k} \tag{8}$$

$$[\mathbf{x}]u = \mathfrak{k}u \qquad \text{if } \mathbf{x} \text{ does not occur freely in } u, \tag{9}$$

$$[\mathbf{x}]uv = \mathfrak{s}([\mathbf{x}]u)([\mathbf{x}]v) \tag{10}$$

Note that these rules define the denotation of nested abstractions (such as $[\mathbf{x}][\mathbf{y}]u$) by first replacing $[\mathbf{y}]u$ by the member of U it denotes.[4]

3.2 Truth in an Interpretation

A structure together with an assignment $(\mathfrak{A}, I)$ will be called a *model*. It induces, in the presence of an environment ν a mapping from goals to truth values:

$$[\![\_]\!]^{\mathfrak{A}}_{\nu} : Goals \to \Omega$$

given by:

$$[\![G]\!] = \omega(\nu[\![G]\!]) \qquad \text{for all goals } G.$$

The mapping is independent of the environment if G is closed.

We say a clause $Tl \to Hd$ is true in a model if, for every environment ν we have $[\![Tl]\!]_\nu \leq_\Omega [\![Hd]\!]_\nu$. A program is true in a model (or $(\mathfrak{A}, I)$ is a model of a program) if its clauses are.

We can lift interpretations to sequences of goals in the obvious way:

$$[\![G_1 \otimes \cdots \otimes G_n]\!] = [\![G_1]\!] \wedge \cdots \wedge [\![G_n]\!].$$

Theorem 1 (Soundness).

If $\langle P|\mathfrak{g}_1\rangle \overset{\theta}{\rightsquigarrow} \langle P|\mathfrak{g}_2\rangle$ *then in every model of* P *and for any environment* ν, $[\![\mathfrak{g}_2]\!]_\nu \leq [\![\mathfrak{g}_1\theta]\!]_\nu$. *Therefore, in particular, if* $\langle P|\mathfrak{g}\rangle \overset{\theta}{\rightsquigarrow} \square$ *then* $[\![\mathfrak{g}\theta]\!]_\nu = \top_\Omega$.

To prove this theorem, we need several technical lemmas.

Lemma 1 (α and β soundness).

Renaming of bound variables is sound in Hiord *semantics. In particular, suppose* y *is a variable distinct from* x *and not occurring freely in* G. *Then*

$$[\![\lambda x.G]\!]_\nu = [\![\lambda y.G[x := y]]\!]_\nu \qquad y \text{ fresh}$$
$$[\![\exists x.G]\!]_\nu = [\![\exists y.G[x := y]]\!]_\nu \qquad y \text{ fresh}$$

β-reduction is sound. In particular

$$[\![(\lambda x.G)t]\!]_\nu = [\mathbf{x}]\,[\![G[x := \mathbf{x}]]\!]_\nu \cdot [\![t]\!]_\nu$$

[4] The denotation of bracket abstraction can be defined in terms of an evaluation map $()^*$ from $\lambda U[\mathcal{W}]$ to U given by transition rules imitating the equations just given. The extra notational step does not seem to add any clarity to the definition. In either case one must show that for any (closed) bracket abstraction u, there is a unique normal form, i.e. a unique abstraction-free member of U denoted by u. This is left to the reader.

The proof of soundness of α, by induction on the cases of the definition of bracket abstraction, is straightforward and left to the reader.

The proof of the soundness of β, given the combinatory nature of our models, is a straightforward adaptation of Curry's combinatory completeness arguments [9] to structures with environments. We prove one step of contraction is sound for the last two of the three cases of the bracket abstraction definition, by showing $([\mathbf{x}]u)v = u[\mathbf{x} := v]$, the first case amounting to a verification of the fact that $\mathfrak{skk}$ is the identity function in a combinatory algebra.

If $\mathbf{x}$ does not occur freely in u, then $([\mathbf{x}]u)v = \mathfrak{k}uv = u$ which agrees with $u[\mathbf{x} := v]$. If u is u_1u_2, then $([\mathbf{x}]u)v = \mathfrak{s}([\mathbf{x}]u_1)([\mathbf{x}]u_2)v$ which in turn gives $(([\mathbf{x}]u_1)v)(([\mathbf{x}]u_2)v)$. By induction, the result is immediate.

The reader can easily check full β conversion is sound.

Lemma 2 (Substitution lemma). *Let G be a goal (or a sequence of goals), θ a substitution, ν an environment into a structure $\mathfrak{A}$. Let ν_θ be the modified environment induced by θ, that is to say, for each variable x, $\nu_\theta(x) = \nu(\theta(x))$. Then $\nu[\![G\theta]\!] = \nu_\theta[\![G]\!]$, and hence $[\![G\theta]\!]^{\mathfrak{A}}_{\nu} = [\![G]\!]^{\mathfrak{A}}_{\nu_\theta}$.*

We now prove the substitution lemma, first for *terms* t, then for formulas G by structural induction.

Proof. Suppose

t is a parameter in S:

Then $\nu[\![t\theta]\!] = I(s)$ and also $\nu_\theta[\![t]\!] = I(s)$.

t is a variable X:

Then $\nu[\![X\theta]\!] = \nu_\theta[\![X]\!]$ by definition.

t is of the form $u \cdot v$:

A special case is $t = X(t_1 \dots t_n)$ for a variable X or $t = r(t_1 \dots t_n)$ for some parameter r. Then $\nu[\![u \cdot v\theta]\!] = \nu[\![u\theta \cdot v\theta]\!] = \nu[\![u\theta]\!] \cdot \nu[\![v\theta]\!]$. By induction hypothesis, this is equal to $\nu_\theta[\![u]\!] \cdot \nu_\theta[\![v]\!]$ and hence $\nu_\theta[\![u \cdot v]\!]$.

t is $\lambda x.G$:

Then $\nu[\![t\theta]\!]$ is $\nu[\![\lambda y.G[x := y]\theta]\!]$ for any y distinct from x and disjoint from the variables in the domain or range of θ, which gives $[\mathbf{y}]\nu[\![G[x := y]\theta]\!]$. By the induction hypothesis, this is equal to $[\mathbf{y}]\nu_\theta[\![G[x := y]]\!]$. and, by soundness of α conversion, to $[\mathbf{x}]\nu_\theta[\![G]\!]$, which is precisely $\nu_\theta[\![\lambda x.G]\!]$.

The cases $t_1 = t_2$ and G_1, G_2 follow immediately from the induction hypothesis, and the $\exists x.G$ case is similar to abstraction, and left to the reader.

We now prove the soundness theorem.

Proof (Soundness). The result is shown by induction on the length of the given resolution deduction. The length 0 case gives the conclusion trivially. Suppose the claim holds for all deductions length smaller than sone natural number $n > 0$, and that we are given a deduction of length n whose first step is *backchain*:

$$\langle P, A \leftarrow Tl|\mathfrak{g}_0 \otimes A' \otimes \mathfrak{g}_1\rangle \overset{\theta_1}{\rightsquigarrow} \langle P, A \leftarrow Tl|\mathfrak{g}_0\theta_1 \otimes Tl\theta_1 \otimes \mathfrak{g}_1\theta_1\rangle \overset{\theta'_1}{\rightsquigarrow} \cdots \rightsquigarrow \langle P, A \leftarrow Tl|\mathfrak{g}'\rangle$$

where $\theta = \theta_1\theta'_1$.

By the induction hypothesis, in any model of $P, A \leftarrow Tl$ we have

$$[\![\mathfrak{g}']\!]_\nu \leq [\![(\mathfrak{g}_0\theta_1 \otimes Tl\theta_1 \otimes \mathfrak{g}_1\theta_1)\theta']\!]_\nu$$

the latter truth value being equal to $[\![\mathfrak{g}_0\theta]\!]_\nu \wedge [\![Tl\theta]\!]_\nu \wedge [\![\mathfrak{g}_1\theta]\!]_\nu$. Now, since $[\![\,]\!]$ is assumed a model of P, for any environment ν we have $[\![Tl]\!]_\nu \leq [\![A]\!]_\nu$, so, in particular, for any substitution θ we have $[\![Tl]\!]_{\nu_\theta} \leq [\![A]\!]_{\nu_\theta}$. By the substitution lemma $[\![Tl\theta]\!]_\nu \leq [\![A\theta]\!]_\nu$. Since $[\![A\theta]\!]_\nu = [\![A'\theta]\!]_\nu$, we have

$$\begin{aligned}[\![\mathfrak{g}']\!]_\nu &\leq [\![\mathfrak{g}_0\theta]\!]_\nu \wedge [\![Tl\theta]\!]_\nu \wedge [\![\mathfrak{g}_1\theta]\!]_\nu \\ &\leq [\![\mathfrak{g}_0\theta]\!]_\nu \wedge [\![A'\theta]\!]_\nu \wedge [\![\mathfrak{g}_1\theta]\!]_\nu \\ &\leq [\![(\mathfrak{g}_0 \otimes A' \otimes \mathfrak{g}_1)\theta]\!]_\nu\end{aligned}$$

as we wanted to show.

Now we consider the case where the first step in the deduction is an occurrence of the *unify* rule:

$$\langle P | \mathfrak{g}_0 \otimes t_1 = t_2 \otimes \mathfrak{g}_1 \rangle \overset{\theta}{\rightsquigarrow} \langle P | \mathfrak{g}_0\theta \otimes \mathfrak{g}_1\theta \rangle \rightsquigarrow \cdots \rightsquigarrow .$$

where $t_1\theta = t_2\theta$.

It suffices to show that $[\![(\mathfrak{g}_0 \otimes \mathfrak{g}_1)\theta]\!]_\nu \leq [\![(\mathfrak{g}_0 \otimes t_1 = t_2 \otimes \mathfrak{g}_1)\theta]\!]_\nu$ in any model of P. But this requirement is equivalent to

$$\begin{aligned}[\![(\mathfrak{g}_0 \otimes t_1 = t_2 \otimes \mathfrak{g}_1)\theta]\!]_\nu &= [\![\mathfrak{g}_0\theta]\!]_\nu \wedge [\![t_1\theta = t_2\theta]\!]_\nu \wedge [\![\mathfrak{g}_1\theta]\!]_\nu \\ &= \mathsf{eq}_{\mathfrak{A}}\, [\![t_1\theta]\!]_\nu\, [\![t_2\theta]\!]_\nu = \top\end{aligned}$$

which always holds.

Now suppose the first step in the deduction is an occurrence of the *exists* rule:

$$\langle P | \mathfrak{g}_0 \otimes \exists x G \otimes \mathfrak{g}_1 \rangle \overset{\exists}{\rightsquigarrow} \langle P | \mathfrak{g}_0 \otimes G\theta \otimes \mathfrak{g} \rangle \rightsquigarrow \cdots \rightsquigarrow ,$$

with x fresh. It suffices to show that for any environment $[\![G\theta]\!]_\nu \leq [\![\exists x G]\!]_\nu$ which is straightforward, and left to the reader.

4 Restricting the Formalism

We have so far defined a very general formalism intended to capture essentially all the higher-order features of Hilog, together with full-blown abstraction and β rewriting. As mentioned in the introduction, the formalism should be viewed as a framework for defining higher-order declarative languages, by suitably restricting the calculus, imposing type disciplines, and making use of abstract interpretation for type inference and specialization.

The aim of this section is to give examples of the use of the higher order features described, and suggest some interesting restrictions of the formalism.

4.1 The Hiord-1 Language

A language for Hiord-1 is composed by a set $\mathcal{F}$ of names for constants and functions, a set $\mathcal{R}$ of names of relations, and a set $\mathcal{V}$ of variables, such that the three are nonempty and disjoint pairwise.

Data (terms) and predicates are distinguished. Terms are restricted to those formed using parameters in $\mathcal{F}$ at the head, and Atomic goals are restricted to terms formed with relational parameters or variables at the head. The table below summarizes the formal abstract syntax of Hiord-1, a restriction of the Hiord syntax.

Definition of terms:

1. A variable is a term.
2. A name in $\mathcal{F}$ is a term.
3. If $t_1, \ldots, t_n$ are terms, and $s \in \mathcal{F}$, then $s(t_1, \ldots, t_n)$ is a term.
4. If G is a goal and $\overline{x}$ is a sequence of variables, then $\{(\overline{x})\texttt{:-}\ G\}$ is a term, known as an abstraction.

Definitions of atomic formulas, Goals and Clauses:

1. $\top$ is an atomic formula.
2. A name $r \in \mathcal{R}$ is an atomic formula, and if $t_1, \ldots, t_n$ are terms, then $r(t_1, \ldots, t_n)$ is an atomic formula. This kind of atomic formulas are called *rigid*.
3. If X is a variable and $t_1, \ldots, t_n$ are terms, then X and $X(t_1, \ldots, t_n)$ are atomic formulas.
4. If t_1 and t_2 are terms, then $t_1 = t_2$ is an atomic formula.
5. An atomic formula is a goal. If G_1 and G_2 are goals, then $G_1 \& G_2$ is a goal, and if x is a variable, $\mathrm{E}(x)G_1$ is a goal.
6. A clause is a formula of the form $H \leftarrow G$, where H is a rigid atomic formula and G is a goal.

Formally, we take Hiord-1 resolution rules to be a subset of Hiord, using strict first-order equality of terms in unification, which is, of course, a subset of $\beta\eta$-conversion. Thus our model theory and soundness results provide a semantic base for this fragment. In practice, the language is sufficiently restricted to permit some obvious compile-time transformations that produce WAM-ready Prolog code. These are discussed in [10] and in the documentation for its implementation as the Hiord-1 *package* in Ciao [11].

4.2 Concrete Syntax of Higher-Order Data and Examples

We now propose a concrete syntax for higher-order data in Hiord-1. Our proposal aims at syntactically differentiating higher-order data from ordinary terms. Thus, in modules using the `hiord` package, all terms to be considered higher-order data are surrounded by `{}`. The most general syntax for *predicate abstractions* follows the pattern:

`{` *sharedvars* `->` `''(`*absvars*`)` `:-` G `}`

which represents the term in the formalism $\{(\overline{x})$`:-` $\mathrm{E}(y)$ G$\}$ and where *absvars* is a comma-separated sequence of distinct variables representing the vector of abstracted variables listed in $\overline{x}$, and $\langle$*sharedvars*$\rangle$ is a comma-separated sequence listing all *exported* variables, i.e. all variables that are not existentially quantified. These variables are shared with the rest of the clause. Variables not appearing in *sharedvars* or in *absvars* are existentially quantified (i.e., correspond to those in y), i.e., they are local to the predicate abstraction, even if their names happen to coincide with variables outside the predicate abstraction.

When *sharedvars* is empty the arrow is omitted. Also, when *absvars* is empty no surrounding parenthesis are written (i.e., only `''` is used). Finally, when G is `true` the "`:- true`" part can also be omitted. Note that the functor name in the head is the void atom `''`.

Conjunction is written with a comma and disjunction, treated in the theory as a defined symbol, is written with semicolon.

Some simple examples of higher-order predicates and uses of predicate abstractions are:

```
% list(List,Pred) : Pred is true for all elements of List
list([], _).
list([X|Xs], P):- P(X), list(Xs, P).

% map(List1,Rel,List2) : Rel is true for all pairs of elements of
%                        List1 and List2 in the same position
map([], _, []).
map([X|Xs], P, [Y|Ys]) :- P(X,Y), map(Xs, P, Ys).

all_less(L1, L2) :- map(L1, {''(X,Y) :- X < Y}, L2).

% child_of(Person, Mother, Father) : Family database
child_of(tom, mary, john).
...

same_mother(L) :- list(L, {M -> ''(S) :- child_of(S,M,_)}).

same_parents(L) :- list(L, {M,F -> ''(S) :- child_of(S,M,F)}).
```

The decision of marking shared variables instead of existential variables is based on the following considerations:

- This approach makes clear which variables can affect the predicate abstraction "from outside."
- Unique existential variables in the predicate abstraction can be written simply as anonymous variables (`_`).
- Compile-time code transformations are simplified, since new variables introduced by expansions should be existential (there is no need to add them to the head).
- The compilation of the predicate abstraction is also simplified.

An additional syntactic form is provided: *closures*. Closures are "syntactic sugar" for certain predicate abstractions, and can always be written as predicate abstractions instead (but they are more compact). All higher-order data (surrounded by `{}`) not adhering to predicate abstraction syntax is a closure. In a closure, each occurrence of the atom `#` corresponds to a parameter of the predicate abstraction that it represents. All variables in a closure are shared with the rest of the clause (for compatibility with meta-programming). As an example, the following definition of `same_parents/2` using a closure is equivalent to the previous one:

```
same_parents(L) :- list(L, {child_of(#,_M,_F)}).
```

If there are several `#`'s in the closure, they each correspond to a successive element of the sequence of abstracted variables in the corresponding predicate abstraction (i.e., in the same order). For example, the following definition of `all_less/2` is equivalent to the one above:

```
all_less(L1, L2) :- map(L1, {# < #}, L2).
```

Note that closures are simply a compact but limited abbreviation. If a different order is required, then a predicate abstraction should be used instead.

Some Examples and a Comparison of Programming Style with Hilog. We start by showing the higher-order predicate which defines the transitive closure of a given relation:

```
closure(R,X,Y) :- R(X,Y).
closure(R,X,Y) :- R(X,Z), closure(R,Z,Y).
```

Assume now that we have the family database defined in a previous example `child_of(Person, Mother, Father)`. Then, given a list of people, to verify that all have the same father one could do:

```
same_father(L) :- list(L, {F -> ''(S) :- child_of(S,_,F)}).
```

This would be expressed in Hilog as:

```
father(F)(S) :- child_of(S,_,F).
same_father(L) :- list(L, father(_)).
```

Which is more laborious. Admittedly, the explicit definition of a predicate sometimes is more clear, but this can also be done in Hiord-1, of course.

```
father(F, S) :- child_of(S,_,F).
same_father(L) :- list(L, {father(_, #)}).
```

But assume now that we want to define a predicate to enumerate the descendents of someone which may share a Y-chromosome feature (that is, only the father relation is taken into account). In Hiord-1, given the `father/2` relation above one could say:

```
descendent_Y(X,Y) :- closure({father(#,#)},X,Y).
```

and if father/2 were not defined, one would say:

```
descendent_Y(X,Y) :- closure({''(F,S) :- child_of(S,_,F)},X,Y).
```

In Hilog one would think that, as a `father` relation was already defined for `same_father`, it could be used for this new predicate. But note that the `father` relation defined above (Hilog version) is not a binary relation and thus one would need to define another father2/2 relation:

```
father2(F,S) :- child_of(S,_,F).
descendent_Y(X,Y) :- closure(father2, X, Y).
```

That father2 relation is semantically equivalent to the father relation, but their different uses in higher-order predicates forces one to make several versions.

4.3 Formalizing Module Structure in Hiord

We now consider an example that exploits the fact that we are working with a higher-order function calculus that allows explicit recursion in our case, a fragment of the untyped λ-calculus. The reader should note that although flex terms are not explicitly allowed as heads of clauses, such a generalized notion of clause is expressible via bindings to lambda-terms, in an essential way, below.

In our example we define modules using predicates and higher-order variables and viewing a module as a predicate which returns a series of predicate abstractions. The module in question, `list_mod`, defines the `Member`, `List`, and `Reverse` predicates, using some auxiliary predicates.

```
lists_mod(Member, List, Reverse) :-
        Member =  {Member -> ''(X, L) :- L = [X|_],
                                      ;  L = [_|Xs], Member(X, Xs)
                  },
        List =    {List -> ''(L, P) :- L = []
                                    ;  L = [X|Xs], P(X), List(Xs, P)
                  },  %% Higher-Order predicate
        Rev3 =    {Rev3 -> ''(L, R1, R2) :- L = [], R1 = R2
                                         ;  L = [E|Es], Rev3(Es,[E|L],R)
                  },  %% Internal predicate
        Reverse = {Rev3 ->  ''(L, R) :- Rev3(L, [], R) }.
```

Note that the definitions of the predicate abstractions which are recursive (`Member`, `List`, and `Rev3`) involve unifications which do not pass the occur-check. While this may be considered a problem, note that the compiler can easily detect such cases (a variable is unified to a predicate abstraction which uses this variable in a flex goal) and translate them defining an auxiliary higher-order extension of the predicate, as the following code shows:

```
lists_mod(Member, List, Reverse) :-
        Me =      { ''(X, L, R) :- L = [X|_]
                                ;  L = [_|Xs], R(X, Xs, R)
                  },  %% Internal predicate
        Member = { Me ->
                    ''(X,L) :- Me(X, L, Me)
                  },
        Li =      { ''(L, P, R) :- L = []
                                ;  L = [X|Xs], P(X), R(Xs, P, R)
                  },  %% Internal predicate
        List =    { Li ->
                    ''(L,P) :- Li(L, P, Li)
                  },
        ...
```

This module can then be used in another module for example as follows:

```
main(X) :-
    lists_mod(Member,_,_), %% Import Member from lists_mod.
    Member(X,[1,3,5]).     %% Call Member.
```

or simply called from the top level for example as follows:

```
?- lists_mod(Member,_,_), Member(X,[1,3,5]).
```

5 Conclusions and Further Work

This paper studies a framework for defining the syntax and semantics of a type-free higher-order extensions to core Prolog with predicate abstractions. Some of the uses of type-free predicate abstraction and higher-order features are underscored, including ways to capture metaprogramming and to formalize module structure. The formalism (and the various subsets considered) is shown sound with respect to a model theory based on partial combinatory algebras with an object of truth-values. Practical restrictions of this framework are then discussed, along with an implementation included as a package in Ciao-prolog. Examples are given showing the use of the notions introduced to define various higher-order predicates, and databases applications. These include programs to compare programming in this framework with code in other Higher-order formalisms.

The framework proposed gives rise to many questions the authors hope to address in future research. In particular, a rigorous treatment must be developed for comparison with other higher-order formal systems (Hilog, Lambda-Prolog). For example, it is reasonably straightforward to conservatively translate the Higher-order Horn fragment of λProlog into Hiord by erasing types, as the resolution rules are essentially the same (assuming a type-safe higher-order unification procedure).

Clearly, the formalisms presented need a more thoroughgoing semantical analysis –declarative and operational– as well as completeness theorems for various typed and type-free restrictions, and a with abstract interpretation taken

into account. Also, a formal treatment is needed for the new proposal for module definition given in this paper.

References

1. Miller, D., Nadathur, G., Pfenning, F., Scedrov, A.: Uniform proofs as a foundation for logic programming. Annals of Pure and Applied Logic **51** (1991) 125–157
2. Nadathur, G., Miller, D.: Higher-order horn clauses. Journal of the ACM **37** (1990) 777–814
3. Miller, D.: λprolog: an introduction to the language and its logic. Manuscript available at `http://www.cse.psu.edu/~dale` (2002)
4. Chen, W., Kifer, M., Warren, D.S.: Hilog: A Foundation for Higher-Order Logic Programming. Journal of Logic Programming (1989)
5. Beeson, M.: Foundations of Constructive Mathematics. Springer-Verlag (1985)
6. Lawvere, F.W.: Diagonal arguments and cartesian-closed categories. In: Category Theory, Homology Theory and their Applications. Springer (1969)
7. Huwig, H., Poigné, A.: A Note on Inconsistencies Caused by Fixpoints in a Cartesian Closed Categories. Theoretical Computer Science **73** (1990) 101–112
8. Miller, D.: A logic programming language with lambda-abstraction, function variables, and simple unification. Journal of Logic and Computation **1** (1991) 497 – 536
9. Curry, H.B., Feys, R.: Combinatory Logic. North-Holland, Amsterdam (1958)
10. Cabeza, D.: An Extensible, Global Analysis Friendly Logic Programming System. PhD thesis, Dept. of Computer Science, Technical University of Madrid, Spain, Madrid, Spain (2004)
11. Bueno, F., Cabeza, D., Carro, M., Hermenegildo, M., López-García, P., (Eds.), G.P.: The Ciao System. Reference Manual (v1.10). The ciao system documentation series–TR, School of Computer Science, Technical University of Madrid (UPM) (2002) System and on-line version of the manual available at `http://clip.dia.fi.upm.es/Software/Ciao/`.
12. Warren, D.H.: Higher-order extensions to prolog: are they needed? In Hayes, J., Michie, D., Pao, Y.H., eds.: Machine Intelligence 10. Ellis Horwood Ltd., Chicester, England (1982) 441–454
13. Naish, L.: Higher-order logic programming. Technical Report 96/2, Department of Computer Science, University of Melbourne, Melbourne, Australia (1996)
14. McDowell, R., Miller, D.: A logic for reasoning with higher-order abstract syntax. In: Proceedings of the 12th Annual IEEE Symposium on Logic in Computer Science, IEEE Computer Society (1997) 434
15. Miller, D.: Abstract syntax and logic programming. In: Proceedings of the Second Russian Conference on Logic Programming, Springer Verlag (1991)