Extending F-Logic with Finite Domain Variables

Liviu-Virgil Ciortuț

Department of Computer Science
"Al. I. Cuza" University of Iași
6600 Iași, Romania

Abstract

One new emerging direction into the area of Logic Programming is how to define a logical framework for the general paradigm of object-orientation. The purposed aim of this direction is to give an increasing power to logic languages as programming languages. The particular aim of this paper is to extend the object-oriented approach to Logic Programming given by M. Kifer and his colleagues [KLM, 1990], namely F-logic, to variables ranging on finite domains, in the same way as P. van Hentenrijk did for classic Logic Programming [Hen, 1989]. We extend also F-logic syntax to higher-order terms without loosing the first-orderness of semantics, by using the main ideas of HiLog logic [CFW, 1993]. We prove that these extensions preserves both soundness and completeness of the object-oriented resolution principles through a corresponding enhanced unification. This paper is a revised and extended version of our previous work [Cio, 1993].

1. Syntax

In the following, a domain is a finite set of constants. Let \( R \) be a set of domains such that \( e \in R \) for every \( d \in R \), \( e \subseteq d \), and \( e \neq \emptyset \). The alphabet of an object-oriented logic language L with domains variables on R consists of:

- a set of parameter symbols: \( a, b, c, \ldots \)
- an infinite set of simple variables: \( X, Y, Z, \ldots \)
- an infinite set of domain variables: \( X^d, Y^d, Z^d, \ldots \), for all \( d \in R \)
- a "function applying" polymorphic operator \( (,, \ldots ,) \)
- a set of logical quantifiers, connectives, and orthographic symbols: \( \exists, \forall, \neg, \wedge, \vee, \Longleftrightarrow, \Rightarrow, \ldots, (,),[ ], \oplus, \cdot, \ldots \).

A term is either a parameter symbol, a simple variable, a domain variable (we call them simple terms) or an expression \( t(t_1, t_2, \ldots, t_n) \), where \( t, t_1, t_2, \ldots, t_n \) are terms. In the object-oriented approach, a compound term \( t(t_1, t_2, \ldots, t_n) \) could be seen as being obtained by sending the function applying message \( (,, \ldots ,) \) to \( t \) in the context of the \( t_1, t_2, \ldots, t_n \) arguments. Notice that such compound terms determine a higher-order syntax by allowing the use of variables and even terms on those positions currently occupied by functional symbols. Sometimes we will call them HiLog terms to make a clear distinction from the case of first-order terms.
An atom is an expression of the following syntactic form:

\[
\text{id\_term}\text{[method}_1; \text{method}_2; \ldots \text{method}_k]\]

where id\_term is a term (we call it the identification part of the atom), and each method\_i, for 1 \leq i \leq k, has one of the following forms:

- `fun_method @ context -> value`
- `fun_method @ context ->! types`
- `set_method @ context ->! values`
- `set_method @ context ->! types`
- `pred_method @ context`

where fun\_method, set\_method, pred\_method, value, and type are terms, and context, values, and types are lists of terms. Generally speaking, such an atom identifies an object, or a class of objects (or even a set of classes) to which different types of methods apply. The context sequence of terms defines the context in which a (functional, set-valuated, or predicative) method applies. The value/s atoms identify the result of applying functional or set-valuated methods in the specified contexts, while types of these values (or: classes to which these values belong) are given by the types lists of atoms.

The other syntactic definitions (like well-formed formulas, literals, clauses, Horn clauses, etc.) in object-oriented logic languages on finite domains are given in the same way as in the first-order predicate calculus.

For example,

\[
\text{name[first -> string; last -> string]}
\]

\[
\text{person[id -> name; aliases ->! string; father ->! person]}
\]

are two atoms describing the types of some methods for the name, respectively person classes. We could use a transparent syntax convention to write the conjunction of these two atoms as

\[
\text{person[id -> name[first -> string; last -> string]; aliases ->! string; father ->! person]}
\]

The special is-a predicate, denoted by : , is commonly used to denote that an object belongs to a given class, or that a class is contained in another class. For example,

\[
\text{john: person}
\]

says that john belongs to the person class.

Here is an atom describing the object identified john:

\[
\text{john[id -> [first -> "john"; last -> "bruno"; aliases ->! "red-fox","king"; male]}
\]

The values of the id and aliases methods for this object will have to be of those types which are given by the homonym (type) methods in the class person.
By using variables, we could identify coreferences between different sub-expressions in complex F-logic formulas, as in the following description:

\[ \text{person[id = name[last = X]}; \]
\[ \text{father = name[last = X]} \]

which says that each person has the same last name as his/her father.

We could use not only constants but also variables or even compound terms to denote for instance class or method names, as in

\[ \text{X}[P @ Y] \leftarrow P: \text{symmetric}, Y[P @ X] \]

\[ \text{noun_phrase}(X_d)[\text{head} = X_d]; \]
\[ \text{case} = Y \]

where the first (clause) should be interpreted as an intensive definition for symmetric predicates, and the second (atom) uses d as a domain which contains, let's say, the noun, adjective, and pronoun constants, and e the domain made of the nominative-accusative and dative-genitive case constants.

In this approach we extended the F-logic syntax given in [KLW,1990] firstly by using predicative methods within the atom frames, and secondly by allowing HiLog terms (possibly incorporating domain variables) to be used in those positions commonly occupied by functional or predicative symbols. (See [CKW,1992] for HiLog foundations.)

2. Semantic Interpretation

Let R be a set of finite domains and L an object-oriented logic language defined as in the previous section. A semantic structure for L is a tuple

\[ I = \langle U, l_S, l_D, l_F, l_o, l_{\geq}, l_{\gg}, l_{\gg\gg}, l_P \rangle \]

where

- \( U \) - the universe of interpretation - is a non-empty set;
- \( \leq \) is a partial order relation on \( U \);
- \( l_S \) is the parameter symbols interpretation function, with \( l_S(s) = s' \in U \), for each parameter symbols in the L language alphabet;
- \( l_D : R \rightarrow 2^U \), by \( l_D(d) = d' \subseteq U \) for each \( d \in R \), is the domain interpretation function. \( 2^U \) denotes the set of all subsets of \( U \);
- \( l_F : U \rightarrow \Pi_{\text{Total}}(U^n, U), \) with \( n \geq 1 \), is the "function applying" operator interpretation. \( U^n \) is the \( n \)-fold Cartesian product of \( U \), and \( \Pi_{\text{Total}}(A, B) \) is the set of all totally defined functions from \( A \) to \( B \), while \( \Pi_S \) is the Cartesian product of all \( S \) sets. So \( l_F(u) \) is an infinite tuple \(<l_F^{(0)}(u), l_F^{(2)}(u), \ldots, l_F^{(n)}(u), \ldots>\), with \( l_F^{(0)}(u) : U^n \rightarrow U \), for every natural number \( n \geq 1 \);
- \( l_o : U \rightarrow \Pi_{\text{Partial}}(U^{n-1}, U), \) with \( n \geq 0 \), is the function (single-valued) methods interpretation. \( \Pi_{\text{Partial}}(A, B) \) is the set of all partially defined functions from \( A \) to \( B \). If defined, \( l_o^{(m)}(u, u_1, u_2, \ldots, u_n) \) denotes the value of the \( m \) method applied to the object/class \( u \) in the context defined by the \( u_1, u_2, \ldots, u_n \) objects/classes;
- \( l_{\geq} : U \rightarrow \Pi_{\text{PartialAntimonotone}}(U^{n-1}, 2^U), \) with \( n \geq 0 \), is the interpretation of functional method types. \( 2^U \) is the upward-closed set of \( U \) subsets (namely, \( 2^U \) is defined so that if \( A \subseteq U \) belongs to \( 2^U \) and \( B \subseteq U \), \( A \subseteq B \), then \( B \in 2^U \)), and \( \Pi_{\text{PartialAntimonotone}}(A, B) \) is the set of all antimonotone partial functions from \( A \) to \( B \). A function \( f : A \to B \) is antimonotone if, by definition if \( f(x) \) is defined, \( y \in A \) and \( x \leq y \), then \( f(y) \) is also defined, and \( f(x) \geq f(y) \). If defined, \( l_{\geq}^{(m)}(u, u_1, u_2, \ldots, u_n) \) will say that the value
of the $m$ functional method applied to the object/class $u$ in the context defined by the $u_1, u_2, ..., u_n$ objects/classes belongs to each of those classes (types) in the set denoted by $l_{>}(m)(u, u_1, u_2, ..., u_n)$; $l_{>}: U \rightarrow \Pi_{m} \Pi_{n} \Pi_{u}$, such that $m$ is the set-valued methods interpretation. If defined, $l_{>}(m)(u, u_1, u_2, ..., u_n)$ denotes that subset of $U$ which is the value of the set-valued method $m$ applied to the object/class $u$ in the context defined by the $u_1, u_2, ..., u_n$ objects/classes;

$l_{>}: U \rightarrow \Pi_{m} \Pi_{n} \Pi_{u}$, with $n \geq 0$, is the set-valued methods type interpretation. If defined, $l_{>}(m)(u, u_1, u_2, ..., u_n)$ will be used to say that each element in the value of the $m$ set-valued method applied to the object/class $u$ in the context defined by the $u_1, u_2, ..., u_n$ objects/classes belongs to each of those classes (types) in the set denoted by $l_{>}(m)(u, u_1, u_2, ..., u_n)$;

$L_{p}: U \rightarrow \Pi_{m} \Pi_{n} \Pi_{u}$, with $n \geq 0$, is the predicative methods interpretation. So, for each $n \geq 0$, the $l_{p}(m)$ component of $l_{p}(m)$ denotes the set of all $(n+1)$-uples $<u, u_1, u_2, ..., u_n>$ that make $m$ true, when interpreted as a $(n+1)$-ary predicate.

A semantic structure $l$ for an F-logic language $L$ must satisfy the following two well-typing conditions:

i. If $l_{>}(m)(a_1, a_2, ..., a_n)$ is defined if $l_{>}(m)(a_1, a_2, ..., a_n)$ is defined, and

ii. If $q = l_{>}(m)(a_1, a_2, ..., a_n)$, then $q \leq r$ for all $r \in l_{>}(m)(a_1, a_2, ..., a_n)$, and

if $q \in l_{>}(m)(a_1, a_2, ..., a_n)$, then $q \leq r$ for all $r \in l_{>}(m)(a_1, a_2, ..., a_n)$.

An assignment of the $L$ variables in the universe $U$ is a mapping of each simple variable $x$ in $U$ and of each domain variable $x^d$ in $l_{D}(d)$.

We give the following particular interpretations for the special predicates $a \equiv b$ and $a \equiv b$ equality, in the semantic structure $l$:

$a \equiv b$ is true in the semantic structure $l$ if and only if $l_{S}(a) \leq l_{S}(b)$, if $a$ and $b$ are parameter symbols in $L$, but using varying assignments, if $a$ and $b$ are terms, and

$a \equiv b$ is true in $l$ if and only if $l_{S}(a) = l_{S}(b)$, if $a$ and $b$ are parameters, and similarly, but with respect to variable assignments, if $a$ and $b$ are terms.

The truth values of the F-logic well-formed expressions are inductively defined as follows:

An elementary (single method) atom like

$a[a_1, a_2, ..., a_n], v[m] \equiv v[j]$ is said to be true in the semantic structure $l$ and the assignment $a$ as if and only if $l_{>}(m)(a_1, a_2, ..., a_n)$ is defined and it is equal to $v[m]$, where $m, a_1, a_2, ..., a_n$, and $v[m]$ are respectively the $m, a_1, a_2, ..., a_n$, and $v[m]$ images of $l_{F}$ through $a$;

$a[a_1, a_2, ..., a_n] \equiv v[l_1, l_2, ..., l_3]$ is true in the semantic structure $l$ and the assignment $a$ as if and only if $l_{>}(m)(a_1, a_2, ..., a_n)$ is defined and it contains $v[l_1, l_2, ..., l_3]$, where $m, a_1, a_2, ..., a_n$, and $v[l_1, l_2, ..., l_3]$ are respectively the $m, a_1, a_2, ..., a_n$, and $v[l_1, l_2, ..., l_3]$ images of $l_{F}$ through $a$;

$a[a_1, a_2, ..., a_n] \equiv v[m] \equiv v[j]$ is true in the semantic structure $l$ and the assignment $a$ as if and only if $l_{>}(m)(a_1, a_2, ..., a_n)$ is defined and it contains $v[m]$, where $m, a_1, a_2, ..., a_n$, and $v[m]$ are respectively the $m, a_1, a_2, ..., a_n$, and $v[m]$ images of $l_{F}$ through $a$;

$a[a_1, a_2, ..., a_n] \equiv v[l_1, l_2, ..., l_3]$ is true in the semantic structure $l$ and the assignment $a$ as if and only if $l_{>}(m)(a_1, a_2, ..., a_n)$ is defined and it contains $v[l_1, l_2, ..., l_3]$, where $m, a_1, a_2, ..., a_n$, and $v[l_1, l_2, ..., l_3]$ are respectively the $m, a_1, a_2, ..., a_n$, and $v[l_1, l_2, ..., l_3]$ images of $l_{F}$ through $a$.

An elementary atom is true in a semantic structure $l$ if and only if it is true in all variable assignments in $l$, and false otherwise.
An atom

\[ \text{id} \_ \text{term}[\text{method}_1, \text{method}_2, \ldots, \text{method}_k] \]

is true if and only if all its sub-atoms

\[ \text{id} \_ \text{term}[\text{method}_1], \text{id} \_ \text{term}[\text{method}_2], \ldots, \text{id} \_ \text{term}[\text{method}_k] \]

are true, and false otherwise.

The truth values of well-defined formulas in object-oriented logic languages on finite domains are defined using the truth values of their component atoms in a similar way to the first-order predicate calculus.

3. Unification

Let \( L \) be an object-oriented logic language on finite domains. We name substitution in \( L \) a finite set \( \theta = \{v_1/t_1, v_2/t_2, \ldots, v_n/t_n\} \) with \( v_i \) variable and \( t_i \) term, \( t_i \) not containing \( v_i \), and each \( v_i \) being different from any \( v_j \) for all \( i \neq j \); also, if \( v_i \) is a domain variable \( x^{d_i} \) then \( t_i \) could be only either a \( d \) constant (an element from the \( d \) set), or a domain variable \( x^{e_i} \), with \( e_i \subseteq d \).

The notions of substitution composition and instance of a well-formed expression through a given substitution are defined as in the first-order predicate calculus. As usually, we name simple expressions either atoms or terms.

At the term level, the object-oriented unification with variables on finite domains could be obtained by a generalization of the unification algorithm designed by Pascal van Hentenri (for simple expressions in the first-order predicate calculus extended to domain variables), now taking into account the extension to HiLog terms. Instead, we will give an efficient unification algorithm through term equations solving, obtained by extending the algorithm in [CKW,1993], which is in turn an adaptation of the efficient unification algorithm in [MM,1982]. With respect to terms, we will use the terms of unifier (or: unifying substitution) and most general unifier for a set of terms in the same relationship to the substitution notion as they are in the predicate calculus. We will see in the subsequent that it will not be the same in the case of atom unification.

An equation in an object-oriented language using finite domain variables is an expression of the form \( t = s \), where \( t \) and \( s \) are terms in that language. A solution of the equation set \( \{t_1 = s_1, t_2 = s_2, \ldots, t_n = s_n\} \) is a substitution \( \sigma \) such that \( t_i \sigma = s_i \sigma \), \( i \leq n \). The substitution \( \sigma \) is a most general solution for this set of equations if and only if for each other solution \( \tau \) there is a substitution \( \lambda \) such that \( \tau = \sigma \lambda \). An equation set is solvable if it has at least one solution. Notice that a substitution \( \{X_1/t_1, X_2/t_2, \ldots, X_n/t_n\} \) could be naturally seen as an equation \( \{X_1 = t_1, X_2 = t_2, \ldots, X_n = t_n\} \). Conversely, an equation \( \{X_1 = t_1, X_2 = t_2, \ldots, X_n = t_n\} \), where all \( X_i \) are different and each \( t_i \) does not contain \( X_i \), could be seen as a substitution. The algorithm we give computes a most general solution for an equation set by doing successive transformations on it.

**Term equations solving algorithm:**

**Input:** \( S \) - a set of term equations in an object-oriented language \( L \) with finite domain variables.

**Output:** A most general solution for \( S \), reported as an equation set \( \{X_1 = t_1, X_2 = t_2, \ldots, X_n = t_n\} \), if \( S \) is solvable.

**Procedure:**

Step 0.

Choose nondeterministically an equation from \( S \).
Step 1.
Apply on the chosen equation one of the following transformations according to each one of the
following cases:
i. \( t(t_1, t_2, \ldots, t_n) = s(s_1, s_2, \ldots, s_m), (n, m \geq 0) \)
   if \( n = m \)
   Stop, failure;
   else
   replace the equation by \( t = s, t_1 = s_1, t_2 = s_2, \ldots, t_n = s_m \);
ii. \( f = g \), with \( f \) and \( g \) parameter symbols,
   if \( f \neq g \)
   Stop, failure;
   else
   delete the equation from \( S \);
iii. \( X = X \), with \( X \) either simple or domain variable,
   delete the equation from \( S \);
iv. \( t = X \), with \( X \) either simple or domain variable and \( t \) non-variable,
   replace the equation with \( X = t \);
v. \( X = t \), with \( X \) variable and \( t \) term different from \( X \),
   if \( t \) contains \( X \)
   Stop, failure;
   else
   if \( X \) is simple variable
   replace \( X \) by \( t \) in each equation of \( S \);
   else \( X \) is a domain variable \( X_d \)
   if \( t \) is either a constant from \( d \) or a domain variable \( X^e \), with \( c \subseteq d \)
   replace \( X \) by \( t \) in each equation of \( S \);
   else
   if \( t \) is a domain variable \( X^e \), with \( l = \emptyset \cap d \neq \emptyset \),
   replace \( X = t \) by \( X_d = X^l \) and \( X^e = X^l \) in \( S \);
   else
   Stop, failure;
Step 2.
If no more transformations could be applied on \( S \)
   Stop;
otherwise
go to Step 0.

In particular, two is-a expressions \( a_1 : b_1 \) and \( a_2 : b_2 \) are unifiable, and their most general unifier is \( \sigma \) if
and only if the equation set \( \{ a_1 = b_1, a_2 = b_2 \} \) is solvable and its most general solution is \( \sigma \).

The following theorem gives the correctness of the above algorithm:

**Theorem 1:**
Let \( S \) be a finite set of term equations in an object-oriented logic language with domain variables. If \( S \) is
solvable, then the above algorithm returns a most general solution for \( S \), otherwise the algorithm fails.

**Proof:**
We can use the encoding function \( \text{encode} \) used by [CKW,1993] to reduce HiLog terms (containing
finite domain variables) to predicate calculus terms (with such variables). Proving the algorithm for such
terms implies only a simple adaptation of the proof for the Martelli and Montanari algorithm in
[MM,1982]. Here is the definition for \( \text{encode} \).
encode₁ (X) = X, for each (either simple or domain) variable;
encode₂ (s) = s, for each parameter symbol s in the language alphabet;
encode₀ ( ( t₁, t₂, ..., tₙ ) ) = applyₙ⁺₁ ( encode₀ ( t ), encode₀ ( t₁ ), ..., encode₀ ( tₙ ) ), where
applyₙ⁺₁ is a (n+1)-ary function, for each n ≥ 0.

At the atom level, we will use a slightly modified version of the unification algorithm given by
M. Kifer for object-oriented and frame-based languages [KLIW, 1990], by keeping its form but
replacing classical term unification by term unification with domain variables. Now, we define the
"directed" unification of one atom into another: a substitution σ is said to be a unifying substitution of
the atom a into the atom b if and only if the set of all elementary sub-atoms of aσ is contained in the set
of all elementary sub-atoms of bσ.

The most general unifier of one atom into another is not immediately definable. Given two
substitutions α and β, we say that α is more general than β, and we denote this by α ≤ β if there is a
substitution γ such that β = αγ. A unifier σ of the atom A₁ into the atom A₂ is a most general unifier of
them if and only if for every unifier θ of A₁ into A₂, σ ≤ θ implies θ ≤ σ.

A set Σ of most general unifiers of A₁ into A₂ is said to be complete if and only if for every
unifier θ of A₁ into A₂, there is a mgu σ ∈ Σ, such that σ ≤ θ.

Atom unification algorithm:
Input: a and b, atoms in an object-oriented logic language L on finite domains;
Output: Ω - a complete set of most general unifiers of a into b;
Procedure:
Step 1.
if id(a) and id(b) are unifiable,
then let θ be their unifier (computed with the previous algorithm);
else STOP, a is not unifiable into b;
Step 2.
if a[] (a has no methods),
then STOP, θ is the only most general unifier of a into b.
Step 3.
Ω = ∅;
for each mapping λ from the set of the elementary sub-atoms of a to the set of elementary sum-
atom of b
σₐ = θ;
for each α elementary sub-atom of a
{ λ(α);
if σₐ(α) and σₐ(β) are unifiable (as arrays of component terms, using the previous
algorithm), the substitution τ being their most general unifier,
then σₐτ;
else break the inner loop;
}
Ω = Ω ∪ {σₐ};
Stop, return Ω.

The following theorem gives the correctness of the above algorithm:

Theorem 2:
Let A₁ and A₂ be atoms in an object-oriented logic language with finite domain variables. The above
algorithm returns a complete set of most general unifiers of A₁ into A₂.
Proof:
We use again encode to reduce the proof to a natural extension of the corresponding proof in [KGWJ 1990]. In this proof, the m.g.u. for terms in the first-order predicate calculus are replaced by m.g.u. for first-order terms using domain variables. ■

4. Object-Oriented Resolution Rules

As P. van Hentenrick did for the first-order predicate calculus in order to obtain an adequate resolution on finite domains, all we have to do now is to restate the object-oriented resolution rules given by M. Kifer, by using the previous atom unification algorithm with variables on finite domains, instead of the unrestricted unifying algorithm. (In fact, the form of the resolution rules remain the same, only the involved unification procedure changes.)

1. Resolution:
   If \( \neg A \lor C \) and \( B \lor C' \) are clauses in an object-oriented logic language on finite domains, (here and in the following A and B are positive literals, C and C’ are clauses), and \( \theta \) is a most general unifier of A into B, then derive \((C \lor C')\theta\).

2. Factoring:
   If \( A \lor B \lor C \) is a clause, and \( \theta \) is a most general unifier of A into B, then derive \((A \lor C)\theta\).
   If \( \neg A \lor \neg B \lor C \) is a clause, and \( \theta \) is a most general unifier of A into B, then derive \((\neg B \lor C)\theta\).

3. Paramodulation:
   If \( A[T] \lor C \) and \( (T' = T') \lor C' \) are clauses, with \( T \) a term in the atom A, and \( \theta \) is a most general unifier of \( T \) and \( T' \), then derive \((A[T'] \lor C \lor C')\theta\).

4. Is-A Reflexivity:
   For every term X, one can derive X \( \lor X \).

5. Is-A Antisymmetry:
   If \( P \lor Q \) and \( Q' \lor P' \) are clauses, and \( \theta \) is a most general unifier of the \( \langle P, Q \rangle \) and \( \langle P', Q' \rangle \) tuples, then derive \((P \lor Q \lor C \lor C')\theta\).

6. Is-A Transitivity:
   If \( P \lor Q \lor C \) and \( Q' \lor R \lor C' \) are clauses, and \( \theta \) is a most general unifier of the Q and \( Q' \), then derive \((P \lor R \lor C \lor C')\theta\).

7. Well-Typing:
   If \( O[M@Q_1, \ldots, Q_i, \ldots, Q_k \rightarrow V] \lor C \), then derive \( O[M@Q_1, \ldots, Q_i, \ldots, Q_k \rightarrow O] \lor C \). Here \( O \) denotes the empty list of atoms.
   The same for set-valuated methods.
   If \( O[M@Q_1, \ldots, Q_i, \ldots, Q_k \rightarrow V] \lor C \) and \( O'[M'@Q_1', \ldots, Q_i', \ldots, Q_k' \rightarrow R] \lor C' \) are clauses, and \( \theta \) is a most general unifier of the \( \langle O,M,Q_1, \ldots, Q_i, \ldots, Q_k \rangle \) and \( \langle O',M',Q_1', \ldots, Q_i', \ldots, Q_k' \rangle \) tuples, then derive \((V \lor R \lor C \lor C')\theta\).
   The same for set-valuated methods.

8. Type Inheritance:
   If \( O[M@Q_1, \ldots, Q_i, \ldots, Q_k \rightarrow R] \lor C \) and \( O' \lor P \lor C' \) are clauses, and \( \theta \) is a most general unifier of \( O \) and \( O' \), then derive \((P[M@Q_1, \ldots, Q_i, \ldots, Q_k \rightarrow R] \lor C \lor C')\theta\).
   The same for set-valuated methods.

9. Argument Sub-Typing:
   If \( O[M@Q_1, \ldots, Q_i, \ldots, Q_k \rightarrow R] \lor C \) and \( Q_i \lor Q' \lor C' \) are clauses, and \( \theta \) is a most general unifier of \( Q_i \) and \( Q_i' \), then derive \((O[M@Q_1, \ldots, Q_i', \ldots, Q_k \rightarrow R] \lor C \lor C')\theta\).
   The same for set-valuated methods.
10. Range Super-Typing:
   If \( O[M@Q_1, \ldots, Q_k \Rightarrow R] \lor C \) and \( R' : P \lor C' \) are clauses, and \( \theta \) is a most general unifier of \( R \) and \( R' \), then derive \( (O[M@Q_1, \ldots, Q_k \Rightarrow P] \lor C \lor C') \theta \).
   The same for set-valuated methods.

11. Functionality:
   If \( O[M@Q_1, \ldots, Q_k \Rightarrow V] \lor C \) and \( O'[M'@Q'_1, \ldots, Q'_k \Rightarrow W] \lor C' \) are clauses, and \( \theta \) is a most general unifier of the \(<O,M,Q_1, \ldots, Q_k \lor C>\) and \(<O',M',Q'_1, \ldots, Q'_k \lor C'>\) tuples, then derive \((V=W) \lor C \lor C') \theta \).

12. Merging:
   If \( A \lor C \) and \( B \lor C' \) are clauses, and \( \theta \) is a most general unifier of the identification parts of \( A \) and \( B \), then derive \( R \lor (C \lor C') \theta \), were \( R \) is the canonical union of \( \theta(A) \) and \( \theta(B) \). This is the atom having the same identification term as \( \theta(A) \) and \( \theta(B) \), and the set of elementary sub-atoms given by the union of the set of elementary sub-atoms of \( \theta(A) \) and the set of elementary sub-atoms of \( \theta(B) \).

13. Elimination:
   From \( \neg A \lor C \), derive \( C \).

The following two results extend naturally those reported by [KLW, 1990].

\textit{Theorem 3 (Soundness):}

Let \( S \) be a set of clauses in an object-oriented logic language with domain variables, and \( D_1, \ldots, D_n \) a finite sequence of clauses such that \( D_n = C \) and, for \( 1 \leq k \leq n \), \( D_k \in S \) or \( D_k \) is derived from \( D_i \) and (possibly) \( D_j \), \( i, j < k \), by one of the above resolution rules. \( C \) is a logical consequence of \( S \).

\textit{Theorem 4 (Completeness):}

Let \( S \) be an unsatisfiable set of clauses in an object-oriented logic language with variables on finite domains. There is a refutation from \( S \) using the above resolution rules.

\textit{Soundness and completeness proof:}

We extend the encoding function given in the proof of Theorem 1 to atom encoding:

\[ \text{encode}_a(\text{id} \_\text{term}[\text{method}_1, \ldots, \text{method}_n]) = \text{encode}_a(\text{id} \_\text{term}) \land \text{encode}_m(\text{method}_1), \ldots, \text{encode}_m(\text{method}_n), \]

where \( \text{encode}_m(\text{method}_i) \) is obtained from \( \text{method}_i \) by applying \( \text{encode}_d \) to each of its component terms.

Then, \( \text{encode}_a \) is naturally extended to an encoding function for well-formed formulas, like in

\[ \text{encode}_a(A \land B) = \text{encode}_a(A) \land \text{encode}_a(B), \]
\[ \text{encode}_a(A \lor B) = \text{encode}_a(A) \lor \text{encode}_a(B), \]
\[ \text{encode}_a(\exists x(A)) = \exists x(\text{encode}_a(A)), \]
\[ \text{encode}_a(\forall x(A)) = \forall x(\text{encode}_a(A)), \]
\[ \neg \text{encode}_a(A) = \neg \text{encode}_a(A). \]

In fact, this encoding function translates a language \( L \) that uses HiLog terms into a language \( L' \) that has the same alphabet as \( L \) plus the additional symbols \( \text{encode}_d, \text{encode}_a, \text{apply}_n \), and uses only first-order terms. There is also a natural encoding function that does the transformation of each semantic structure \( I \) in \( L \) into a corresponding semantic structure \( I' \) in \( L' \), and that last transformation entails the truth preservation

\[ \models_{L} \phi \text{ if and only if } \models_{L'} \text{encode}_a(\phi), \]

for every variable assignment \( \xi \), and every well-formed formula \( \phi \) in \( L \).
Using these encoding transformations, and taking into account that there is a Skolemization procedure form well-formed formulas in object-oriented logic languages with domain variables, the remaining work is to extend the corresponding proofs in [KGW,1990] to finite domain variables, and that quite natural. We must mention that Skolemization is done here not in the classic way, but using the special form given in [CFW,1993] because of the higher-order syntax of the terms in our approach.

Conclusions and further research

The present paper shows that there is a natural extension of the F-logic defined in [KLW,1990] to variables ranging on finite domains and to higher-order syntax for terms, while maintaining its first-order semantics. This new approach permits us to use constraint satisfaction techniques (like forward checking and looking ahead) in object-oriented logic programming, as [Hen,1989] did for predicate calculus. We hope that, together with the present results, a better approach to procedural semantics will give birth to a valuable tool for solving many practical problems.

We intend to follow the idea used by H. Ait-Kaci in [AKN,1986] to manage inheritance at the unification level, in order to reduce the large number of resolution principles in F-logic by incorporating their effect in a more elaborated unification process. As we are working now on a Prolog interpreter implementation in the object-oriented language C++ (using the theoretical foundations in [MW,1988]), we would like to extend to the present framework. One particular aim is to use such a system to develop an object-oriented natural language processing system.

Bibliography