EasyChair Preprint

№ 15677

(Higher -Order) Equational Unification as Logic Programming

Murat Sïnan Aygün

January 6, 2025

(Higher-Order) Equational Unification as Logic Programming

Murat Sinan Aygün[1]

[1] Kültür Sitesi, C Blok, Kat 3, Daire 7,
Kükürtlü, Bursa, Turkey
sinan_aygun@yahoo.com

Abstract. This paper addresses a serious problem in the practical implementation of (higher-order) equational unification in higher-order logical frameworks. Naive implementations of (higher-order) equational unification in which variables to be solved are directly represented by free logic variables leads to non-decidability. This paper solves this problem and develops a workable solution set. We propose an implementation of (higher-order) equational unification in a logic programming style using lambda prolog. We formally expose the implementation result in an abstract level, which looks similar to standard (higher-order) equational unification rules. The design of the formal exposition and the implementation is such that the mapping between them is transparent. This result gives concrete and uniform framework for (higher-order) equational unification.

Keywords: Logic programming, decidability, (higher-order) equational unification.

1 Introduction

Equational unification algorithms have been developed in [5, 6, 8, 10]. The technique in these papers is to apply the transformation rules repeatedly until a trivial solution is reached. Consider solving an equational unification problem as a query submitted to a logic programming language. In a natural and clear way when we directly represent variables to be solved by free logic variables, non-decidability easily occurs. Free logic variables are infinitely instantiated in a loop by the application of the transformation rules. Consider solving the goal $z + 0 =^{?} succ(z)$ with the rules $\{x + 0 \rightarrow x,\ x + succ(y) \rightarrow succ(x + y)\}$ where z is a free logic variable. Although the goal has no solution values, without any controlling strategy, z is subject to infinite instantiations, which leads to non-termination. In particular, z is substituted by the left-hand side values of the rewrite rules. Even worse than this, z is instantiated during the proof search by recursively defined subterm relation clauses.

In our solution to this problem, we need to check free logic variable occurrences in goal formulas in order to rule out their infinite instantiations by using encoding terms. Suppose $z_1,..,z_n$ are free logic variables in goal formulas and $c_1,..,c_n$ are new arbitrary constants not available in the current signature. Assume that φ is a one to one

mapping from $\{z_1,..,z_n\}$ to $\{c_1,..,c_n\}$, written as $\{z_i \rightarrow c_i\}$ $(1 \leq i \leq n)$. For a given term s, t is the *encoding* of s if and only if $t = s\,\varphi$, in other words t is the resultant term after applying φ to s. We apply the transformation rules to s and its encoding t simultaneously for preserving the mapping at each step as $s, t \rightarrow \ldots \rightarrow s_n, t_n$ such that t_n is the encoding of s_n. Consider the provability of s, t as the goals

$$z + 0 =^? succ(z),\ c_1 + 0 =^? succ(c_1),$$
$$z =^? succ(z),\ c_1 =^? succ(c_1) \text{ (by a paramodulation step with } x + 0 \rightarrow x).$$

In trying to prove $z =^? succ(z)$, $c_1 =^? succ(c_1)$, z is not instantiated by the left hand side values of the rewrite rules. Particularly, a paramodulation step is applied to z together with c_1 and c_1 fails to match any left hand side values. The step therefore fails. On the other hand, z is not instantiated during the proof searches by recursively defined subterm relation clauses. A proof search is applied to z together with c_1 and it fails because c_1 fails to match any term in subterm relation clauses. The proof searches which instantiate free logic variables vacuously are eliminated by using encoding terms and therefore the goal fails. The constants $c_1,..,c_n$ are used as decision parameters to check the occurrences of free logic variables to achieve a decidable solution set.

By using encoding terms, we organize the search for successful derivations, especially cutting infinite branches which do not yield solution. Although our methodology can be applicable to any equational unification strategy, in order to eliminate high non-determinism, we focus on the narrowing strategy and consider the paramodulation steps at non-variable positions. Since our transformations simulate narrowing derivations, properties such as completeness, soundness, decidability of our transformation system are same as the properties of narrowing. The completeness and soundness results are given for general case. In order to prove our arguments, we consider the termination for a special case. Termination of narrowing is guaranteed by imposing restrictions on rewrite rules [15] in which case our transformations are also terminating. All these results are given in the Appendix. The system can be uniformly extended to higher-order equational unification settings. Assuming higher-order unification[1] as a meta-level rule, implementation of higher-order equational unification is as simple as that of first-order equational unification. We do not give the proof for higher-order cases. Because referring to the work in [17], it is same with first-order cases. After a brief introduction to our notation, we consider first-order equational unification in Section 2. We present unification as a set of transformation rules. We later extend the strategy with paramodulation step. We propose an implementation in λProlog. In Section 3, we extend the formulations for higher-order cases. Experiments are reported in the Appendix.

Given two sets V of variables and F of function symbols, the set of (first-order) *terms* $T(V, F)$ is the smallest set containing V such that $f(t_1,\ldots,t_n)$ is in $T(V, F)$ whenever $f \in F$ and each $t_i \in T(V, F)$ for $(1 \leq i \leq n)$. Each $f \in F$ has an arity and f is called *constant* if f is a symbol of arity zero. $Var(t)$ denotes the set of variables in t. A first-order term may be viewed as *finite, ordered, labeled* tree, the leaves of which are labeled with variables and constants. A position within a term t may be represented as a sequence of positive integers describing the path from the root of t to the root of the

[1] Restricted version of higher-order unification is considered. See [13] for background.

subterm at that position, denoted by $t|_p$. A term u has an occurrence in t if $u = t|_p$ for some positions p in t. $t[s]_p$ is used to denote the term t with its subterm $t|_p$ replaced by s. A substitution is a mapping written as $\{x_1 \rightarrow t_1,\ldots, x_n \rightarrow t_n\}$ where $x_i \neq t_i$ for $(1 \leq i \leq n)$. $t\,\sigma$ denotes the term obtained after applying the substitution σ.

2 Specifying First-Order Equational Unification

2.1 Unification by Transformations

Definition 1 Let $\forall x\ equal\ x\ x$ be a binary predicate. Given any two terms s and t, the goal $equal\ s\ t$ is solved with a most general unifier of s and t if and only if there exists a unifier of s and t. Otherwise, the goal fails.

We use new constants that are not in the signature as decision parameters.

Definition 2 Let φ be a one to one substitution (mapping) from a set of free logic variables to a set of new constants (not in the signature). A term t is the *encoding* of a term s if and only if $t = s\,\varphi$. We may call t an encoding term and s an encoded term. We may also call the new constants encoding constants.

Example 1 Given the function symbols $f, g \in F$, the new constants c_1, c_2, free logic variables x and y, $f(c_1, g(c_1, c_2))$ is the encoding of $f(x, g(x, y))$ where $\varphi = \{x \rightarrow c_1, y \rightarrow c_2\}$.■

In the following, we use c to denote any encoding constant. The transformation rules are applied to encoded and encoding terms in the same equality goal.

Definition 3 Let $=^?$ be a 4-ary function symbol. It can be written in a notation $s, s' =^? t, t'$ where t' and s' are the encodings of t and s respectively. $s, s' =^? t, t'$ is symmetric, in other words $s, s' =^? t, t'$ implies $t, t' =^? s, s'$.

Definition 4 We denote an ordered list by $[s_1,\ldots,s_n]$. The symbol [] is used for empty lists. We give the following operations on lists:

- $s_1 :: [s_2,\ldots,s_n, s_{n+1}] = [s_1,\ldots, s_{n+1}]$.
- $[s_1,\ldots, s_n] \partial [t_1,\ldots, t_n] = [s_1,\ldots, s_n, t_1,\ldots, t_n]$.
- $L_1 \partial (t :: L_2) = (L_1 \partial L_2) \cup \{t\}$ for any ordered lists L_1, L_2 possibly empty.

Unification problem can be denoted by $\lambda x_1 \ldots \lambda x_n.[s_1 =^? t_1,.., s_n =^? t_n]$ where s_i, t_i $(1 \leq i \leq n)$ are first-order terms. Let E be a set of equations. We may call $[u_1,.., u_n]$ a *E-unifier* of $\lambda x_1.\lambda x_2.\ldots. \lambda x_n.[s_1 =^? t_1,.., s_n =^? t_n]$ if $s_i\,\sigma =_E t_i\,\sigma$ for all i $(1 \leq i \leq n)$ where $\sigma = \{x_1 \rightarrow u_1,\ldots, x_n \rightarrow u_n\}$. When E is empty, we may call it a *unifier*.

Consider that Definition 2 can be extended for unification problems. For example, $\lambda x.[g(x, a) =^? g(a, c)]$ is the encoding of $\lambda x.[g(x, a) =^? g(a, F)]$ where $\varphi = \{F \rightarrow c\}$ (c is an encoding constant and F is a free logic variable). Lists are used in an unordered fashion by the transformations. We use the notation $L \cup \{t\}$ to denote that t is selected

arbitrarily. Unification by transformations is given in Definition 5. We use the following equality: $((\lambda x.s)\ t)^{\beta} = s\ \{x \rightarrow t\}$.

The transformations in Definition 5 are applied to a 3-ary relation denoted by *unif* A_1 A_2 A_3. A_1 stands for an original unification problem. A_2 stands for its encoding. A_3 stands for a unifier. The symbol $\leftrightarrow$ is used to denote an arbitrary transformation. For a closed unification problem M and a free logic variable S, when the goal *unif* M M S is satisfied, S is instantiated with a unifier. In overcoming any notational confusion in the definitions, it is assumed that predicates are satisfied from left to right in the expressions of conjunction. For instance, for the transformation step 4 in Definition 5, *equal* X Y is satisfied first and the resultant substitution is applied to the goal automatically by the meta-level system before further any transformation step on the goal begins.

Definition 5 Let M denote any unification problem and M' be the encoding of M. Let L denote any ordered list and a, f respectively denote any constant and function symbol. Given that X_1, X_1', Y_1, Y_1',.., X_n, X_n', Y_n, Y_n' are any terms, unification can be defined by the following transformations applied to a 3-ary relation *unif*.

1. *unif* M M' $(Z_1 :: Z_2)$ iff *unif* $(M\ Z_1)^{\beta}$ $(M'\ c)^{\beta}$ Z_2 where Z_1, Z_2 are new free logic variables, c is a new arbitrary constant that is not in the signature.

2. *unif* $[X_1 =^? Y_1,.., X_n =^? Y_n]$ $[X_1' =^? Y_1',.., X_n' =^? Y_n']$ $[]$ iff
   unif $[]$ $[X_1, X_1' = =^? Y_1, Y_1',.., X_n, X_n' = =^? Y_n, Y_n']$ $[]$.

3. *unif* $[]$ $(L \cup \{a, a =^? a, a\})$ $[]$ iff *unif* $[]$ L $[]$.

4. *unif* $[]$ $(L \cup \{X_1, c =^? Y_1, Y_1'\})$ $[]$ iff *equal* X_1 Y_1 and *unif* $[]$ $L\{c \rightarrow Y_1'\}$ $[]$.

5. *unif* $[]$ $(L \cup \{f(X_1,.., X_n), f(X_1',.., X_n') =^? f(Y_1,.., Y_n), f(Y_1',.., Y_n')\})$ $[]$ iff
   unif $[]$ $(L \partial [X_1, X_1' = =^? Y_1, Y_1',.., X_n, X_n' = =^? Y_n, Y_n'])$ $[]$.

6. *unif* $[]$ $[]$ $[]$ iff *True*.

Example 2 Consider solving $\lambda x.\lambda y.\lambda z.[f(x, a) =^? f(z, y)]$. We use N in $\leftrightarrow_N$ to denote the number of the transformation. When the goal

$$unif\ \lambda x.\lambda y.\lambda z.[f(x, a) =^? f(z, y)]\ \lambda x.\lambda y.\lambda z.[f(x, a) =^? f(z, y)]\ S$$

is satisfied, the free logic variable S is instantiated by $[U_3, a, U_3]$.

unif $\lambda x.\lambda y.\lambda z.[f(x, a) =^? f(z, y)]$ $\lambda x.\lambda y.\lambda z.[f(x, a) =^? f(z, y)]$ S $\leftrightarrow_1, \leftrightarrow_1, \leftrightarrow_1$

unif $[f(U_1, a) =^? f(U_3, U_2)]$ $[f(c_1, a) =^? f(c_3, c_2)]$ S_1 $\leftrightarrow_2$
where S is instantiated by $(U_1 :: (U_2 :: (U_3 :: S_1)))$ (c_1, c_2, c_3 are new constants and U_1, U_2, U_3, S_1 are new free logic variables)

unif $[]$ $[f(U_1, a), f(c_1, a) =^? f(U_3, U_2), f(c_3, c_2)]$ $[]$ $\leftrightarrow_5$
where S_1 is instantiated by $[]$.

unif $[]$ $[(U_1, c_1 =^? U_3, c_3), (a, a =^? U_2, c_2)]$ $[]$ $\leftrightarrow_4$

unif $[]$ $[(a, a =^? U_2, c_2)]$ $[]$ $\leftrightarrow_4$
where U_1 is instantiated by U_3.

unif $[]$ $[]$ $[]$ $\leftrightarrow_6$

where U_2 is instantiated by a.

True. ■

In the definitions, constant and function symbols are treated differently for syntactic manners. In the following, the symbols a, f are used to denote any constant and function symbol respectively. The correctness and completeness of the transformations are given in the Appendix.

2.2 Equational Unification by Transformations

We focus on paramodulation steps at non-variable positions. By applying the inductive method in Definition 6, we can compute subterms at non-variable positions.

Given a term T_1 and its encoding T_1', for any non variable position p in T_1, the goal $sub\ S\ T_1\ C\ S'\ T_1'\ C'$ (S, C, S', C' are free logic variables) is satisfied where S, C, S', C' are respectively instantiated with $T_1|_p$, $\lambda u.T_1[u]_p$, $T_1'|_p$ and $\lambda u.T_1'[u]_p$ and u has no occurrence in T_1 and T_1'.

Definition 6 Let *sub* be a 6-ary relation. It is inductively defined on all terms A, A', $X_1 \ldots X_n$, $X_1' \ldots X_n'$, D, D' as follows:

- $sub\ a\ a\ (\lambda u.u)\ a\ a\ (\lambda u.u)$.
- $sub\ (f\ X_1 \ldots X_i \ldots X_n)\ (f\ X_1 \ldots X_i \ldots X_n)\ (\lambda u.u)\ (f\ X_1' \ldots X_i' \ldots X_n')\ (f\ X_1' \ldots X_i' \ldots X_n')\ (\lambda u.u)$.
- $sub\ A\ (f\ X_1 ... X_i .. X_n)\ (\lambda u.(f\ X_1 .. (D\ u)^{\beta} .. X_n))\ A'\ (f\ X_1' .. X_i' ... X_n')\ (\lambda u.(f\ X_1' .. (D'\ u)^{\beta} .. X_n'))$ iff $sub\ A\ X_i\ D\ A'\ X_i'\ D'$.

Lemma 1 Let T_1 be a first order term and T_1' be the encoding of T_1. For any non variable position p in T_1, the goal $sub\ S\ T_1\ C\ S'\ T_1'\ C'$ (S, C, S', C' are free logic variables) is satisfied where S, C, S', C' are respectively instantiated with $T_1|_p$, $\lambda u.T_1[u]_p$, $T_1'|_p$ and $\lambda u.T_1'[u]_p$ and u has no occurrence in T_1 and T_1' (By using a trivial induction on the length of term tree, the proof can be done).

Definition 7 A rewrite rule is denoted by $\lambda x_1 \ldots \lambda x_n (s \rightarrow_R t)$ where s, t are first-order terms.

In paramodulation steps, fresh variables of an equation (or rule) are used. For computing fresh variables, we apply the transformation below denoted by $\rightarrow^{Variant}$ to a closed rewrite rule for replacing all bound variables with new free logic variables. For its encoding, we replace all bound variables with new constants. $\rightarrow^{Variant}{}_*$ stands for iterative applications of the transformation $\rightarrow^{Variant}$ until none is applicable.

Definition 8 Let R denote any rewrite rule and R' be the encoding of R. Let *variant* be a binary relation and its inductive definition is as follows:

- $variant\ (R_1 \rightarrow_R R_2)\ (R_1' \rightarrow_R R_2')$ where $R_1' \rightarrow_R R_2'$ is the encoding of $R_1 \rightarrow_R R_2$.
- $variant\ R\ R'$ iff $variant\ (R\ Z)^{\beta}\ (R'\ c)^{\beta}$ where Z is a new free logic variable, c is a new arbitrary constant that is not in the signature.

We can adopt the narrowing strategy in which the witness pair unifies. For the witness pair list, we can use the equality form $\approx\ \approx^?$ only the unification by

transformation steps is applicable. In Definition 9, term decomposition step is applied to the witness pair list as many as until none is applicable. We thereafter apply the variable elimination method given by the transformation step 8 in Definition 10:

Definition 9 Let $\approx \approx^?$ be a 4-ary function symbol. It can be written in a notation $s, s' \approx \approx^? t, t'$ where t' and s' are the encodings of t and s respectively. $s, s' \approx \approx^? t, t'$ is symmetric, in other words $s, s' \approx \approx^? t, t'$ implies $t, t' \approx \approx^? s, s'$. The symbol $\rightarrow^{\mathrm{Dec}}$ is used for the following transformations:

- $L \cup \{a, a \approx \approx^? a, a\}$ iff L.
- $L \cup \{f(X_1,..,X_n), f(X_1',..,X_n') \approx \approx^? f(Y_1,..,Y_n), f(Y_1',..,Y_n')\}$ iff
  $L \partial [X_1, X_1' \approx \approx^? Y_1, Y_1',.., X_n, X_n' \approx \approx^? Y_n, Y_n']$.

$\rightarrow^{\mathrm{Dec}}{}_*$ stands for iterative applications of the transformation $\rightarrow^{\mathrm{Dec}}$ until none is applicable.

Definition 10 *Equational unification* can be defined by adding the two more to the transformations in Definition 5:

7. *unif* [] ($L \cup \{A, A' =^? B, B'\}$) [] iff
   sub D A C D' A' C' (D, C, D', C' are free logic variables) and
   variant R R $\rightarrow^{\mathrm{Variant}}{}_*$ *variant* ($U \rightarrow_R N$) ($U' \rightarrow_R N'$) for a closed rewrite rule R and
   $[D, D' \approx \approx^? U, U'] \rightarrow^{\mathrm{Dec}}{}_* L_1$ and
   unif [] ((($C\ N$)$^\beta$, ($C'\ N'$)$^\beta$ =$^?$ B, B') :: ($L_1 \partial\ L$)) [].

8. *unif* [] ($L \cup \{X_1, c \approx \approx^? Y_1, Y_1'\}$) [] iff *equal* $X_1\ Y_1$ and *unif* [] $L\{c \rightarrow Y_1'\}$ [].

Example 3 Given $E = \{\lambda x(f(x) \rightarrow_R g(x))\}$, when the goal

$$unif\ \lambda y.\lambda x.[f(y) =^? g(x)]\ \lambda y.\lambda x.[f(y) =^? g(x)]\ S$$

is satisfied, the free logic variable S is instantiated by $[U_2, U_2]$.

unif $\lambda y.\lambda x.[f(y) =^? g(x)]\ \lambda y.\lambda x.[f(y) =^? g(x)]\ S$ $\leftrightarrow_1, \leftrightarrow_1$

unif $[f(U_1) =^? g(U_2)]\ [f(c_1) =^? g(c_2)]\ S_1$ $\leftrightarrow_2$
where the free logic variable S is instantiated by (U_1 :: (U_2 :: S_1)) (c_1, c_2 are new constants and U_1, U_2, S_1 are new free logic variables)

unif [] $[f(U_1), f(c_1) = =^? g(U_2),\ g(c_2)]$ [] $\leftrightarrow_7$
where S_1 is instantiated by [].

unif [] [($U_1, c_1 \approx \approx^? U_3, c_3$), ($g(U_3), g(c_3) = =^? g(U_2), g(c_2)$)] [] $\leftrightarrow_8$
where *variant* ($f(U_3) \rightarrow_R g(U_3)$) ($f(c_3) \rightarrow_R g(c_3)$) holds for c_3 being a new constant and U_3 being a new free logic variable.

equal $U_1\ U_3$ and *unif* [] $[g(U_3), g(c_3) = =^? g(U_2), g(c_2)]$ $\{c_1 \rightarrow c_3\}$ []

unif [] $[g(U_3), g(c_3) = =^? g(U_2), g(c_2)]$ [] $\leftrightarrow_5$
where U_1 is instantiated by U_3 by the proof of *equal* $U_1\ U_3$.

unif [] $[U_3, c_3 = =^? U_2, c_2]$ [] $\leftrightarrow_4$

unif [] [] [] $\leftrightarrow_6$

where U_3 is instantiated by U_2.

True. ■

The correctness, completeness and termination results of the equational unification transformations are given in the Appendix.

2.3 Implementation

We present the concrete implementation of the equational unification in λProlog [14]. Before our formulation, we give a brief introduction to the language. In addition to the logical connectives ∃, ∧, ∨ in goal formulas as used in classical logic programming, implicational and universal goals are also supported. To prove an implication $B \supset C$, assume B as an hypothesis and attempt to prove C. Similarly, to prove a universal quantifier $\forall x.B$ prove a generic instance $B\{x \rightarrow c\}$ where c is a constant that is not in the current signature. The comma (,), semicolon (;), and arrow (⇒) represent conjunction, disjunction and implication respectively while :- denotes the converse of implication. $\lambda x.e$ is written as x \ e. The symbol *pi* represents universal quantification. The symbols *nil* and :: denote the empty list and the list constructor respectively. Types are assigned to terms. The expression *type s* τ denotes that *s* is a term of type τ. A list whose elements are of type τ is given the type *list* τ. An atomic formula is a term of type *o*. λProlog makes use of *curried syntax*: A term of the form *f*(*t*, *s*) can be written as (*f t s*). We assume universal closure over all variables written as tokens with an upper case initial letter.

Implementing Unification by Transformations

```
type   some   (τ → list τ) → list τ.
type   all    (τ → τ) → τ.
type   eq     τ → τ → τ.
type   eqq    τ → τ → τ → τ → τ.
type   eqq_   τ → τ → τ → τ → τ.
type   rule   τ → τ → τ.

type m_a_r   τ → list τ → list τ → o.
type append  list τ → list τ → list τ → o.
type eqtoeqq list τ → list τ → list τ → o.

m_a_r X (X :: L) L.
m_a_r X (Y :: K) (Y :: L) :- m_a_r X K L.

eqtoeqq nil nil nil.
eqtoeqq ((eq X1 Y1) :: L1) ((eq X2 Y2) :: L2)
        ((eqq X1 X2 Y1 Y2)::L3) :- eqtoeqq L1 L2 L3.

append nil K K.
append (X :: XS) YS (X :: ZS) :- append XS YS ZS.
```

some and *all* are used to represent the outmost λ-bound variables in unification problems and rewrite rules respectively. We have *eq*, *eqq*, *eqq_* and *rule* in places of

$=^?$, $= =^?$, $\approx \approx^?$ and $\rightarrow_R$ respectively. We may select an element t form a list as $L \cup \{t\}$ (see Definition 4) by executing *m_a_r* in a non-deterministic way. *eqtoeqq* clause is used for the transformation step 2 in Definition 5 and *append* for list concatenation. We execute the object-level substitution $L\ \{c \rightarrow Y_1'\}$ (see the transformation step 4 in Definition 5) by using *cp* and *subst* clauses.

```
type a              τ.
type f, h           τ → τ.
type g              τ → τ → τ.
type cp             τ → τ → list τ → o.
type subst          list τ → list τ → list τ → o.

cp X X U :- m_a_r X U _.
cp a a _.
cp (f X1)(f Y1) U :- cp X1 Y1 U.
cp (h X1)(h Y1) U :- cp X1 Y1 U.
cp (g X1 X2) (g Y1 Y2) U :- cp X1 Y1 U, cp X2 Y2 U.

subst nil nil _.
subst ((eqq X1 X2 X3 X4) :: L1)((eqq X1 Y2 X3 Y4) :: L2) U :-
                      cp X2 Y2 U, cp X4 Y4 U, subst L1 L2 U.
```

The notation _ is used in λProlog for logic variables appearing once in formulas. Function symbols *a*, *h*, *f*, *g* are used as *a* represents constants, *f* and *h* represent 1-ary functions, *g* represents 2-ary functions. The scope can be extended to n-ary functions in a similar way.

```
type unf            list τ → list τ → list τ → list τ → o.

unf (some X) (some Y) (Z::S) U :-
               pi c\(unf (X Z) (Y c) S (c::U)).
unf (X1::L1) (X2::L2) nil U :-
               eqtoeqq (X1::L1) (X2::L2) L3, unf nil L3 nil U.
unf nil L nil U :- m_a_r (eqq a a a a) L L1, unf nil L1 nil U.
unf nil L nil U :- m_a_r (eqq (f X1) (f X2) (f X3) (f X4)) L L1,
                         unf nil ((eqq X1 X2 X3 X4)::L1) nil U.
unf nil L nil U :- m_a_r (eqq (h X1) (h X2) (h X3) (h X4)) L L1,
                         unf nil ((eqq X1 X2 X3 X4)::L1) nil U.
unf nil L nil U :-
m_a_r (eqq (g X11 X12) (g X21 X22)
           (g X31 X32) (g X41 X42)) L L1,
unf nil ((eqq X11 X21 X31 X41)::
         (eqq X12 X22 X32 X42)::L1) nil U.
unf nil L nil U :- (m_a_r (eqq X X1 X Y1) L L1;
                    m_a_r (eqq X Y1 X X1) L L1), m_a_r X1 U U1,
((cp X1 Y1 _) => subst L1 L2 U1), unf nil L2 nil U.

unf nil nil nil _.
```

Referring to Definition 5, *unify* relation is given as a 3-ary symbol. The first argument is used for an original unification problem. The second argument is used for its encoding. The third is used for a unifier. Technically, we need to check occurrences of encoding constants (see Definition 2) in order to apply the variable elimination method (see the transformation step 4 in Definition 5). As the fourth

argument in *unf* clause (*unf* stands for *unify*), a list whose elements are encoding constants is used. We can check any encoding term if it is an encoding constant by checking if it is in the list. Referring to the transformation step 4 in Definition 5, we can apply the substitution $L\ \{c \rightarrow Y_1'\}$ by the goal $(cp\ c\ Y_1'\ \_) \supset$ subst $L\ L'\ U$ such that when it is proved, the free logic variable L' is instantiated by $L\ \{c \rightarrow Y_1'\}$ given that U is a list of encoding constants not including c. The clause *cp X X U :- m_a_r X U _* is used for copying encoding constants other than c.

Implementing Equational Unification by Transformations

For function symbols *a*, *h*, *f*, *g*, *sub* relation in Definition 6 can be given as follows:

```
type sub     τ → τ → (τ → τ) → τ → τ → (τ → τ) → o.

sub a a (c\c) a a (c\c).
sub (f X1) (f X1) (c\c) (f X2) (f X2) (c\c).
sub (h X1) (h X1) (c\c) (h X2) (h X2) (c\c).
sub (g X1 X2) (g X1 X2) (c\c) (g Y1 Y2) (g Y1 Y2) (c\c).
sub S1 (f X1) (c\(f (X2 c))) S2 (f Y1) (c\(f (Y2 c))) :-
                      sub S1 X1 X2 S2 Y1 Y2.
sub S1 (h X1) (c\(h (X2 c))) S2 (h Y1) (c\(h (Y2 c))) :-
                      sub S1 X1 X2 S2 Y1 Y2.
sub S1 (g X1 X2) (c\(g (X3 c) X2))
    S2 (g Y1 Y2) (c\(g (Y3 c) Y2)) :- sub S1 X1 X3 S2 Y1 Y3.
sub S1 (g X1 X2) (c\(g X1 (X3 c)))
    S2 (g Y1 Y2) (c\(g Y1 (Y3 c))) :- sub S1 X2 X3 S2 Y2 Y3.
```

rwt is used to represent rewrite rules. E contains only the rule $\lambda x\ f(x) \rightarrow_R h(x)$.

```
type rwt τ → o.
rwt (all x\rule (f x) (h x)).

type dec               list τ → list τ → list τ → o.

dec (A::L1) L3 U :- m_a_r (eqq a a a a) (A::L1) L2, dec L2 L3 U.
dec (A::L1) L3 U :-
   m_a_r (eqq (f X1) (f X2) (f X3) (f X4)) (A::L1) L2,
   dec ((eqq X1 X2 X3 X4)::L2) L3 U.
dec (A::L1) L3 U :-
   m_a_r (eqq (h X1) (h X2) (h X3) (h X4)) (A::L1) L2,
   dec ((eqq X1 X2 X3 X4)::L2) L3 U.
dec (A::L1) L3 U:-
   m_a_r (eqq (g X11 X12) (g X21 X22)
              (g X31 X32) (g X41 X42)) (A::L1) L2,
   dec ((eqq X11 X21 X31 X41)::(eqq X12 X22 X32 X42)::L2) L3 U.
dec ((eqq X1 X2 Y1 Y2)::L1) ((eqq_ X1 X2 Y1 Y2)::L2) U :-
   (m_a_r X2 U _ ; m_a_r Y2 U _), dec L1 L2 U.
dec nil nil _.
```

We apply the term decomposition relation $\rightarrow^{Dec}_{*}$ in paramodulation steps by executing *dec*. We apply as many decomposition transformations as possible. As the third argument, we use a list of encoding constants to check their occurrences in

equalities by using *m_a_r* clause. After we extend the *subst* clause to deal with the equalities *eqq_* in object-level substitutions, we give the paramodulation step.

```
subst ((eqq_ X1 X2 X3 X4) :: L1)((eqq_ X1 Y2 X3 Y4):: L2) U :-
                          cp X2 Y2 U, cp X4 Y4 U, subst L1 L2 U.

type variant list τ → τ → τ → list τ → o.

variant L (rule X1 Y1) (rule X2 Y2) U :-
(m_a_r (eqq A1 A2 B1 B2) L L1 ; m_a_r (eqq B1 B2 A1 A2) L L1),
       sub D1 A1 C1 D2 A2 C2,
       dec ((eqq D1 D2 X1 X2) :: nil) L2 U,
       append L1 L2 L3,
       unf nil ((eqq (C1 Y1) (C2 Y2) B1 B2) :: L3) nil U.
variant L (all X1) (all Y1) U :-
                   pi c\(variant L (X1 Z) (Y1 c) (c::U)).

unf nil L nil U :- rwt R, variant L R R U.
unf nil L nil U :-
     (m_a_r (eqq_ X X1 X Y1) L L1; m_a_r (eqq_ X Y1 X X1) L L1),
      m_a_r X1 U U1, ((cp X1 Y1 _) => subst L1 L2 U1),
      unf nil L2 nil U.
```

3 Specifying Higher-Order Equational Unification

3.1 Pattern Unification by Transformations

Unification problem is denoted by $\lambda x_1 \ldots \lambda x_n.[s_1 =^? t_1,.., s_n =^? t_n]$ where each s_i, t_i $(1 \le i \le n)$ are *patterns*[2].

Definition 11 *Pattern unification* can be defined by adding the four more to the transformations in Definition 5.

9. $unif\ M\ M'\ (Z_1 :: Z_2)$ iff $unif\ (M\ Z_1)^\beta\ (M'\ (\lambda c_0 \ldots \lambda c_{n-1}.c_n))^\beta\ Z_2$
   where Z_1, Z_2 are new free logic variables of a suitable type, for M' of the type $(\tau_0 \rightarrow \ldots \rightarrow \tau_n) \rightarrow \delta$ $(1 \le n)$, $c_0, c_1 \ldots, c_n$ are new arbitrary constants not in the signature, and c_i of the type τ_i, $i \in \{0, 1,..,n\}$.

10. $unif\ []\ (L \cup \{N, N' =^? K, K'\})\ []$ iff
    $unif\ []\ (((N\ b)^\beta, (N'\ b)^\beta =^? (K\ b)^\beta, (K'\ b)^\beta) :: L)\ []$
    where b is a new arbitrary function (or constant) of a suitable type.

11. $unif\ []\ (L \cup \{op\ X_1 \ldots X_n, op\ X_1' \ldots X_n' =^? op\ Y_1 \ldots Y_n, op\ Y_1' \ldots Y_n'\})\ []$ iff
    $unif\ []\ L\ \partial\ [X_1, X_1' =^? Y_1, Y_1',.., X_n, X_n' =^? Y_n, Y_n']\ []$.

12. $unif\ []\ (L \cup \{b, b =^? b, b\})\ []$ iff $unif\ []\ L\ []$.

---

[2] See [13] for the definiton and unification of patterns.

For pattern unification, we extend the transformation system given in Definition 5. The transformations that are given in Definition 11 are used for dealing with bound variables. We use any n-ary operator symbol *op* (including application) for bound variables. E^{β} represents the β-normal form of E.

Example 4 Consider the solution of $\lambda F.\lambda G\ [\lambda x.\lambda y.F\ x =^{?} \lambda x.\lambda y.f\ (G\ y\ x)]$. When the goal *unif* $\lambda F.\lambda G\ [\lambda x.\lambda y.F\ x =^{?} \lambda x.\lambda y.f\ (G\ y\ x)]\ \lambda F.\lambda G\ [\lambda x.\lambda y.F\ x =^{?} \lambda x.\lambda y.f\ (G\ y\ x)]\ S$ is proved, the free logic variable S is instantiated by $[\lambda b_1.f(U_3\ b_1), \lambda b_2.\lambda b_1.U_3\ b_1]$ (U_3 is a new free logic variable).

unif $\lambda F.\lambda G\ [\lambda x.\lambda y.F\ x =^{?} \lambda x.\lambda y.f\ (G\ y\ x)]\ \lambda F.\lambda G\ [\lambda x.\lambda y.F\ x =^{?} \lambda x.\lambda y.f\ (G\ y\ x)]\ S$ $\leftrightarrow_9, \leftrightarrow_9$

unif $[\lambda x.\lambda y.U_1\ x =^{?} \lambda x.\lambda y.f(U_2\ y\ x)]$
$[\lambda x.\lambda y.((\lambda c_0.c_1)\ x)^{\beta} =^{?} \lambda x\lambda y.f((\lambda c_2.\lambda c_3.c_4)\ y\ x)^{\beta}]\ S_1$ (β-conversion)
where S is instantiated by $(U_1 :: (U_2 :: S_1))$ (c_0, c_1, c_2, c_3, c_4 are new arbitrary constants (or function symbols) and U_1, U_2, S_1 are new free logic variables)

unif $[\lambda x\lambda y.U_1\ x =^{?} \lambda x\lambda y.f(U_2\ y\ x)]\ [\lambda x.\lambda y.c_1 =^{?} \lambda x.\lambda y.f(c_4)]\ S_1$ $\leftrightarrow_2$

unif $[]\ [\lambda x.\lambda y.U_1\ x, \lambda x\lambda y.c_1 = =^{?} \lambda x.\lambda y.f(U_2\ y\ x), \lambda x.\lambda y.f(c_4)]\ []$ $\leftrightarrow_{10}, \leftrightarrow_{10}$
where S_1 is instantiated by [].

unif $[]\ [(U_1\ b_1), c_1 = =^{?} f(U_2\ b_2\ b_1), f(c_4)]\ []$ $\leftrightarrow_4$
where b_1, b_2 are new arbitrary function symbols (or constants).

equal $(U_1\ b_1)\ f(U_2\ b_2\ b_1)$ and *unif* $[]\ []\ []$ (Pruning and Flexible-Rigid step)

unif $[]\ []\ []$ $\leftrightarrow_6$
where U_2 with $\lambda b_2.\lambda b_1.U_3\ b_1$ and U_1 with $\lambda b_1.f(U_3\ b_1)$ are substituted by the proof of *equal* $(U_1\ b_1)\ f(U_2\ b_2\ b_1)$ (U_3 is a new free logic variable).

True. ■

3.2 Implementing Pattern Unification by Transformations

The capital letter *A* is used in type declarations for higher-order cases. We use the function symbol *fv* of type $A \rightarrow \tau$ to enclose free logic variables in a unifier for convenience that the unifier can contain not only first-order terms but also higher-order terms. *abst* is used to represent bound variables. The predicate *bv* for enclosing bound variables is used as an hypothesis during proof searches. *some1* and *some2* are used to represent solution variables of types $\tau \rightarrow \tau$ and $\tau \rightarrow \tau \rightarrow \tau$ respectively. We use the operator symbol *app* for bound variables taking one argument. The scope can be extended to n-ary operator symbols and n-ary solution variables in a similar way.

```
type fv      A → τ.
type bv      τ → o.
type abst    (τ → τ) → τ.
type app     τ → τ → τ.
type some1   ((τ → τ) → list τ) → list τ.
type some2   ((τ → τ → τ) → list τ) → list τ.
```

We extend *cp* and *unf* clauses (see Section 2.3) to deal with bound variables and abstractions.

```
cp X X _:- bv X.
cp (app X Y1)(app X Y2) U:- cp Y1 Y2 U.
cp (abst X1)(abst Y1) U :- pi c\cp c c _ => cp (X1 c) (Y1 c) U.

unf (some1 X) (some1 Y) ((fv Z)::S) U :-
             pi c\(unf (X Z) (Y c1\c) S (c::U)).
unf (some2 X) (some2 Y) ((fv Z)::S) U :-
             pi c\(unf (X Z) (Y c1\c2\c) S (c::U)).
unf nil L nil U :- m_a_r (eqq X X X X) L L1,
                   bv X, unf nil L1 nil U.
unf nil L nil U :-
   m_a_r (eqq (app X Y1) (app X Y2) (app X Y3) (app X Y4)) L L1,
   unf nil ((eqq Y1 Y2 Y3 Y4)::L1) nil U.
unf nil L nil U :-
   m_a_r (eqq (abst X1) (abst X2) (abst X3) (abst X4)) L L1,
   pi c\bv c =>
       (unf nil ((eqq (X1 c) (X2 c) (X3 c) (X4 c)) ::L1) nil U).
```

3.3 Higher-Order Equational Unification by Transformations

We consider the unification of patterns in the presence of a first-order equational theory.

Definition 12 We add the two more to *sub* relation given in Definition 6 to deal with bound variables and abstractions.

□ $sub\ A\ B\ (\lambda c.\lambda x.(D\ x\ c)^{\beta})\ A'\ B'\ (\lambda c.\lambda x.(D'\ x\ c)^{\beta})$ iff
$sub\ (A\ x)^{\beta}\ (B\ x)^{\beta}\ (D\ x)^{\beta}\ (A'\ x)^{\beta}\ (B'\ x)^{\beta}\ (D'\ x)^{\beta}$
where x is an arbitrary variable not free in $A, B, A', B', \lambda x.(D\ x)^{\beta}, \lambda x.(D'\ x)^{\beta}$.

□ $sub\ A\ (op\ X_1 \ldots X_n)\ (\lambda c.(op\ X_1..(D\ c)^{\beta}..X_n))$
$A'\ (op\ X_1' \ldots X_n')\ (\lambda c.(op\ X_1'..(D'\ c)^{\beta}..X_n'))$ iff $sub\ A\ X_i\ D\ A'\ X_i'\ D'$.

Definition 13 We add one rule to $\rightarrow^{Dec}$ relation given in Definition 9 to deal with abstractions:

□ $L \cup \{A, A' \approx \approx^{?} G, G'\}$ iff $L\ \partial\ [(A\ b)^{\beta}, (A'\ b)^{\beta} \approx \approx^{?} (G\ b)^{\beta}, (G'\ b)^{\beta}]$.
where b is a new arbitrary function symbol (or constant) of a suitable type.

Definition 14 Let *lifting* be a binary relation. $y_1..\ y_n$-*lifting* of a

$$variant\ (R_1 \rightarrow_R R_2)\ (R_1' \rightarrow_R R_2')$$

is of the forms

□ $lifting\ (\lambda y_1..\lambda y_n.R_1 \rightarrow_R \lambda y_1..\lambda y_n.R_2)\delta\ (\lambda y_1..\lambda y_n.R_1' \rightarrow_R \lambda y_1..\lambda y_n.R_2')$ (for $n \geq 0$)
where $\delta = \{F \rightarrow (F\ y_1..y_n) \mid F$ is any free logic variable of R_1 or $R_2\}$,

□ $lifting\ (R_1 \rightarrow_R R_2)\ (R_1' \rightarrow_R R_2')$ (for $n = 0$)

Definition 15 We update the transformation step 7 given in Definition 10 to deal with bound variables.

unif [] ($L \cup \{A, A' =^{?} B, B'\}$) [] iff
sub $D\ A\ C\ D'\ A'\ C'$ (D, C, D', C' are free logic variables) and
variant $R\ R \rightarrow^{\text{Variant}}_{*}$ *variant* ($R_1 \rightarrow_R R_2$) ($R_1' \rightarrow_R R_2'$) for a closed rule R and
lifting ($U \rightarrow_R N$) ($U' \rightarrow_R N'$) is $y_1..\ y_n$-*lifting* of *variant* ($R_1 \rightarrow_R R_2$) ($R_1' \rightarrow_R R_2'$)
for $D = \lambda y_1..\lambda y_n.f(s_1,\ldots,s_m)$ or $\lambda y_1..\lambda y_n.a$ ($n \geq 0$) and
$[D, D' \approx \approx^{?} U, U'] \rightarrow^{\text{Dec}}_{*} L_1$ and
sub $N\ H\ C\ N'\ H'\ C'$ and
unif [] ($(H, H' =^{?} B, B') :: (L_1 \partial L)$) [].

3.4 Implementing Higher-order Equational Unification by Transformations

We present the formulation of lifting as the following clauses.

```
type lift            τ → τ → τ → τ → τ → list τ → o.
type lift1           τ → τ → list τ → o.
type last_and_rest   τ → list τ → list τ → o.
type apply           A → τ → list τ → o.

last_and_rest A (A::nil) nil.
last_and_rest A (B::L1) (B::L2) :- last_and_rest A L1 L2.

apply A A nil.
apply A B L1 :- last_and_rest X L1 L2, apply (A X) B L2.

lift1 (fv X1) X2 U        :- apply X1 X2 U.
lift1 a a _.
lift1 (g X1 X2) (g Y1 Y2) U      :- lift1 X1 Y1 U,lift1 X2 Y2 U.
lift1 (h X1) (h X2) U            :- lift1 X1 X2 U.
lift1 (f X1) (f X2) U            :- lift1 X1 X2 U.

lift A (rule X1 Y1) (rule X2 Y2)(rule X11 Y11) (rule X2 Y2) U :-
          (A = a;A = (f _);A = (h _);A = (g _ _)),
           lift1 X1 X11 U, lift1 Y1 Y11 U.
lift (abst A) R1 R2 (rule (abst X1)(abst Y1))
                    (rule (abst X2)(abst Y2)) U :- pi c\(
lift (A c) R1 R2 (rule (X1 c)(Y1 c))(rule(X2 c)(Y2 c)) (c::U)).
```

We extend the *sub* and *dec* clauses in accordance with Definition 12 and Definition 13.

```
sub S1 (app X1 X2) (c\(app X1 (X3 c)))
    S2 (app Y1 Y2) (c\(app Y1 (Y3 c))) :-sub S1 X2 X3 S2 Y2 Y3.
sub (abst S1) (abst X1) (c\(abst x\(D1 x c)))
    (abst S2) (abst X2) (c\(abst x\(D2 x c))) :-
      pi c\(sub (S1 c) (X1 c) (D1 c) (S2 c) (X2 c) (D2 c)).

dec (A::L1) L3 U :-
 m_a_r (eqq (abst X1) (abst X2) (abst X3) (abst X4)) (A::L1) L2,
  pi c\dec ((eqq (X1 c) (X2 c) (X3 c) (X4 c))::L2) L3 U.
```

We update the paramodulation step given in Section 2.3 in accordance with Definition 15.

```
variant L (rule X1 Y1) (rule X2 Y2) U :-
  (m_a_r (eqq A1 A2 B1 B2) L L1 ;m_a_r (eqq B1 B2 A1 A2) L L1),
   sub D1 A1 C1 D2 A2 C2,
   lift D2 (rule X1 Y1)(rule X2 Y2)
           (rule XX1 YY1)(rule XX2 YY2) nil,
   dec ((eqq D1 D2 XX1 XX2) :: nil) L2 U, append L1 L2 L3,
   sub YY1 AA1 C1 YY2 AA2 C2,
   unf nil ((eqq AA1 AA2 B1 B2) :: L3) nil U.
variant L (all X1) (all Y1) U :-
          pi c\(variant L (X1 (fv Z)) (Y1 c) (c::U)).
```

4 Conclusion

Unification problems in the literature have been considered for *syntactic* and *semantic* manners. As far as generality is concerned, the both issue should be treated in a uniform framework [5, 10]. In this work, we consider a framework unifying the two approaches in which the management of semantic and syntactic variables is precisely made and propose a practical way for solving (higher-order) equational unification problems in higher-order logical frameworks. This eases our understanding of (higher-order) equational unification as much as it gives clearance in practical issues. Similar approach has been used in [1, 4, 9]. But our wok is generic, uniform and easy to understand. Since we tackle the problem directly, it can be verified easily and side effects are avoided. Moreover it is easily extended to higher-order settings in which case pattern unification is considered as a meta-level rule. Finally this work enhances functional logic paradigms and illustrates how λ-terms embedded in logic programming improve the meta-programming capabilities.

Acknowledges

The author thanks to referees for helpful comments on previous drafts.

References

1. S.Antoy, M.Hanus, B.Massey, Frank Steiner. An implementation of narrowing strategies. Third International Conference on Principles and Practice of Declarative Programming PPDP'01, Firenze, Italy, Sept, 2001, pages 207-217.
2. M.S.Aygün. Implementation of Higher-Order Narrowing in a High-Level Meta-Level System, Boğaziçi University, 1998.
3. M.S.Aygün. A Logic Programming Approach to Implementing Higher-Order Narrowing. LFM Proceedings, 1999.
4. P.H. Cheong and L.Fribourg. *Implementation of narrowing*: The Prolog-based approach. In K.R. Apt, J.W.de Bakker, and J.J.M.M. Rutten, editors, Logic programming languages: constraints, functions, and objects, The MIT Press, 1993, 1-20.

5. H.Common, C. Kirchener. Constraint Solving on Terms. H.Common, C.Marche, and R.Treinen (Eds.): CCL'99, LNCS 2002, pp.47-103, 2001.
6. D.J. Dougherty and P.Johann. An Improved General *E*-Unification Method. In 10th International Conference on Automated Deduction, Kaiserslautern, FRG, July 24-27, 1990.
7. A. Felty. A Logic Programming Approach to Implementing Higher-Order Term Rewriting. Proceedings of the 1991 International Workshop on Extensions of Logic Programming, Lars-Henrik Eriksson, Lars Hallnas, and Peter Schroeder-Heister, editors, Springer-verlag Lecture Notes in Artificial Intelligence, 1992.
8. J.H.Gallier and W.Snyder. Complete Sets of Transformations for General *E*-Unification. In *TCS* 67:2,3, pp.203-260, 1989.
9. M.Hanus. The integration of Functions into Logic Programming: From Theory to Practice. The J.Logic programming 1994:19,20:583-628.
10. J.P.Jouannaud and Claude Kirchner. Solving Equations in Abstract Algebras: a Rule-Based Survey of Unification. Computational Logic. Essay in honor of Alan Robinson. The MIT Press, pages 257-321, Cambridge, 1991.
11. C. Liang. Free Variables and Subexpressions in Higher-Order Meta Logic. In Theorem Proving in Higher Order Logics, 11th International Conference, Springer-Verlag LNCS Vol.1479. September 1998.
12. D.Miller. Unification of Simply Typed Lambda-Terms as Logic Programming. In the Proceedings of the 1991 International Conference on Logic Programming, edited by Koichi Furukawa, June 1991.
13. D.Miller. A Logic Programming Language with Lambda-Abstraction, Function Variables, and Simple Unification. Journal of Logic and Computation, Vol. 1, No. 4, 1991.
14. D.Miller. λProlog: An Introduction to the Language and its Logic, 1996.
15. C. Prehofer. On Modularity in Term Rewriting and Narrowing. Proceedings of the First International Conference on Constraints in Computational Logics, volume 845 of Lecture Notes in Computer Science, Springer-Verlag, pages 253-268, Berlin, 1994.
16. C. Prehofer. Higher-Order Narrowing. Proceedings of the ninth Annual IEEE Symposium on Logic in Computer Science, pp.507-516.
17. Z.Qian. Higher-Order Equational Logic Programming. Appeared in the Proceedings of the 21st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.
18. N.Tobias, Z.Qian. Modular Higher-Order E-unification. Proceedings of the 4th International Conference, RTA-91, pp.200-214, Springer-Verlag.

A Appendix: Experimental Reports

Example 5 When the query

unf (*some z\some x\some y\eq* (*g x* (*g z x*)) (*g a y*)::*nil*)
(*some z\some x\some y\eq* (*g x* (*g z x*)) (*g a y*)::*nil*) *L nil*

is solved, the free logic variable *L* is substituted by the list *Z* :: *a* :: *g Z a* :: *nil* (*Z* is a new free logic variable).

Example 6 When the query

unf (*some x\some y*(*eq* (*g a* (*f x*)) (*g a* (*h y*)))::*nil*)
(*some x\some y*(*eq* (*g a* (*f x*)) (*g a* (*h y*)))::*nil*) *L nil*

is solved, *L* is substituted by the list *Z* :: *Z* :: *nil* (*Z* is a new free logic variable). *E* contains the rule rwt (all x\rule (f x) (h x)).

Example 7 The goal in Example 4 can be given by the query

unf (*some1 x\some2 y\eq* (*abst b*(*abst a*(*x b*))) (*abst b*(*abst a\f*(*y a b*)))::*nil*)
(*some1 x\some2 y\eq* (*abst b*(*abst a*(*x b*))) (*abst b*(*abst a\f*(*y a b*)))::*nil*) *L nil*.

When the query is solved, *L* is substituted by the list

(*fv c1\ f* (*S c1*)) :: (*fv c2\ c3\ S c3*) :: *nil*

where *S* is a new free logic variable.

Example 8 When the query

unf (*some2 x\some2 y\eq* (*abst b*(*app b* (*abst a*(*f* (*x b a*)))))
(*abst b*(*app b* (*abst a*(*h* (*y a b*)))))::*nil*)
(*some2 x\some2 y\eq* (*abst b*(*app b* (*abst a*(*f* (*x b a*)))))
(*abst b*(*app b* (*abst a*(*h* (*y a b*)))))::*nil*) *L* nil.

is solved, *L* is substituted by the list

(*fv c1\ c2\ S c2 c1*) :: (*fv c1\ c2\ S c1 c2*) :: *nil*

where *S* is a new free logic variable. *E* contains the rule

rwt (all x\rule (f x) (h x)).

B Appendix: Completeness, Soundness and Termination Results of the Equational Unification Transformation for First-Order Unification Problems

Theorem 1 (Soundness) Let E be an empty set. For a closed unification problem M and a free logic variable S, when the goal *unif M M S* is satisfied where S is instantiated by a list L, then L is a unifier of M.

Proof:
The transformations in Definition 5 are same as the transformations presented for the standard unification transformations in [8]. So it can be trivially shown that a list returned by the query is a unifier. ■

Theorem 2 (Completeness) Let E be an empty set. If L' is a unifier of a closed unification problem M, then the goal *unif M M S* is satisfied where S is instantiated by a list L and $L\ \sigma = L'$ for some substitution σ.

Proof:
The proof can be trivially made since each transformation step 4 in Definition 5 produces a solved form[3]. ■

Theorem 3 (Soundness) Let E be a set of equations. For a closed unification problem M and a free logic variable S, when the goal *unif M M S* is satisfied where S is instantiated by a list L, then L is a E-unifier of M.

Proof:
Based on the results in [8], the proof can be done trivially. ■

Theorem 4 (Completeness) Let E be a set of equations so that $\rightarrow_E$ is confluent and terminating. If L' is a unifier of a closed unification problem M, then the goal *unif M M S* is satisfied where S is instantiated by a list L and $L\ \sigma =_E L'$ for some substitution σ.

Proof:
Following the completeness of narrowing in [9], the proof can be done trivially because the equational unification transformations mimic narrowing derivations. ■

Theorem 5 Let R be a convergent rewrite system in which every left hand side is of the form $f(t_1,\ldots,t_n)$ such that each t_i is either a variable or ground term. The equational unification transformations are terminating.

Proof:
The proof can be done trivially because the equational unification transformations mimic rewriting derivations under this case. ■

[3] See [8] for the definition of solved form.