A Logic Programming Approach to Implementing Higher-Order Narrowing

Murat Sinan AYGUN

Department of Computer Engineering, Bogazici University, Istanbul, Turkey

Abstract *We present the implementation of higher-order narrowing in the framework of λProlog. The work illustrates the strength of the language by presenting how λ-terms and types embeded in λProlog extend the concept of logic programming.*

We specify variables to be solved by variables bound at the meta-level and directly apply the meta-level unification via these meta-level variables. The main contribution of this work is that we at the same time successfully encode operational behavior of these meta-level solvable variables at the object-logic. This approach frees the programmer from implementing most of the computations and as a result significantly simplifies the implementation problem from the programmer's point of view. Our work is the first concrete implementation of higher-order narrowing.

We finally state that the adaptability of our techniques requires a meta-level system that supports λ-abstraction, types, polymorphic typing, implications and universal quantification in goals and the body of clauses.

1 Introduction

Logic plays an important role in specification of formal systems by its simplicity and generality. By the discovery of the unification process and its exploitation in reasoning, new language design strategies based on first-order logic have been proposed. Prolog is a very simple engine whose inference mechanism is based on backchaining. Subsequent efforts have extended these strategies to more powerful meta-level systems such as λProlog [1] and L_λ, a fragment of λProlog [2], which are built over higher-order structures. Computational characteristics of syntactic structures are generalized and direct meta-level support is provided in λProlog and L_λ. As a result, manipulation of these structures is simplified compared to other programming languages such as Pascal, Prolog and ML [2]. For example, implementation of a proof system based on natural deduction is easy in λProlog since quantification logic can be supported naturally by using the higher-order features of λProlog [1]. Certain restrictions are placed on functional variables for ensuring the computability of a most general unifier in L_λ. Compared to interpreters for Horn clauses, L_λ has a more complex computational mechanism [2]. This feature is due to the enhancement of Horn logic by allowing implications and universal quantifiers within goal formulas.

On the other hand, the unification mechanism is exploited in rewriting for the computation of answers as in Prolog. This technique is known as *narrowing*. It has recently been suggested to extend the narrowing strategy over higher-order structures to provide a more powerful meta-level system [3, 4].

The idea to implement equational reasoning tools on high-level meta-level systems is not new [5]. The implementation of higher-order term rewriting in a tactic style theorem proving is considered in [6]. The main difference between our research and previous work is the following ideas, which are applied to the implementation of higher-order narrowing in λProlog:

Operational behavior of solvable variables is specified by meta-level existentially quantified variables. With this idea, instead of implementing our own unification procedure at the object-logic, which is the most important part, we directly apply L_λ's unification mechanism via these meta-level solvable variables. The instances provided by the meta-level system L_λ are the solution values.

As a result of the idea above, the implementation problem is simplified from the programmer's point of view.

- We encode the meta-level solvable variables by a special constructor "*evar*" at the object-level and develop the technicalities around the idea of preserving the encoding mechanism.

By using the idea of encoding, we restrict search space values for the meta-level solvable variables. The reason is to rule out infinite vacuous instances of them, which block the computation of desired results. We therefore improve the inference engine for the application.

According to our knowledge, there is no concrete implementation of higher-order narrowing. Indeed, programming techniques for implementing λProlog can also be used for implementing higher-order narrowing. But this approach highly complicates the problem because of the reasons mentioned above.

The adaptation of higher-order narrowing is described in Section 2. The technical details about the encoding mechanism are presented in Section 3. The complete code for the implementation is given in Section 4 and Appendix.

2 Preliminaries

Definition 2.1 Let *term* and *string* be base types where *string* contains strings and *term* contains the terms called *object-level terms*. Let Σ_o be the signature

app : *term* → *term* → *term*,

cons : *string* → *term*,

la : (*term* → *term*) → *term*,

rule : *term* → *term* → *term*,

some : (*term* → *term*) → *term*,

all : (*term* → *term*) → *term*,

eq : *term* → *term* → *term*.

cons, *app*, *la*, *eq*, *rule* are used to specify respectively constants, application, abstraction, query and rewrite rule. *T*- and *t*-formulas respectively represent higher-order and first-order terms.

C ::= *cons* s | *app C t* | (*C*)

t ::= F | *C* | (*t*)

T ::= F | x | *cons* s | *app T T* | *la* λx.*T* | (*T*)

R ::= *rule t t* | *all* λF.*R*

Q ::= *eq T T* | *some* λF.*Q* .

s denotes strings. In order not to lead to any confusion with meta-level quantified variables, F variables in *Q*- and *R*-formulas are called *object-level quantified variables*. *Object-level existentially quantified variables* specify variables to be solved. It is assumed that arguments of object-level existentially quantified variables can only be bound variables.

Example 2.1 The object-level term

some λF.*eq* (*la* λx.(*app* F x))
(*la* λx.(*app* (*cons* "*f*") x))

represents the query $\lambda \mathbf{x}.(X\ \mathbf{x}) =^? \lambda \mathbf{x}.(f\ \mathbf{x})$ where the free variable X is represented by the object-level existentially quantified variable F.

Note: It is assumed that *u* and *t* denote *T*-formulas.

Definition 2.2 Subterms in *T*-formulas are defined in the following manner:

- $t_{/\varepsilon} = t$,
- $(la\ \lambda x.t)_{/1.p} = t_{/p}$,
- $(app..(app\ (cons\ s)\ t_1)...t_n)_{/i.p} = t_{i/p}$ for $1 \le i \le n$,
- $(app...(app\ x\ t_1)...t_n)_{/i.p} = t_{i/p}$ for $1 \le i \le n$.

Note: $t[u]_p$ is the result of replacing $t_{/p}$ in *t* by *u*.

In Definition 2.2, we intentionally ignore the positions within the formulas

$(app...(app\ F\ x_1)...x_n)$

since we do not apply narrowing steps to $x_1...x_n$.

Definition 2.3 *M*-formulas called *constant terms* are defined in the following manner:

M ::= *cons* s | *app M T*.

T denotes *T*-formulas and s denotes strings and the name of constant terms.

Definition 2.4 and Definition 2.5 are respectively the adaptations of the higher-order lifting and narrowing procedures given in [4].

Qu is used to denote the list of meta-level quantifiers in λProlog [2] and [*Qu*, *P*] is a meta-level predicate stating that the object-level term *P* is in the context *Qu*.

Note: *nil* denotes empty lists.

Definition 2.4 Let $X_1, X_2,...,X_n$ be all the object-level universally quantified variables of a rewrite rule. Let $y_1, y_2,...,y_k$ be distinct bound variables and $H_1, H_2,...,H_n$ be distinct new meta-level existentially quantified variables whose quantifiers are appended to the end of *Qu* (*Qu* is the list of meta-level quantifiers in L_λ). *Higher-order lifting of the rewrite rule over* $y_1,...,y_k$ is the result of replacing all occurrences of X_i by $H_i\, y_1...y_k$.

In the following higher-order narrowing procedure, for a given query, the object-level variables that are used to specify the solvable variables are replaced with meta-level existentially quantified variables in order to apply directly L_λ's unification mechanism. After the query is solved, the instances computed by the inference system are the solution values we expect for these object-level solvable variables.

Note: $\overline{app}\ u\ \bar{t}$ denotes $(app...(app\ u\ t_1)...t_n)$ in Definition 2.5.

Definition 2.5 Let *P* denote a closed *Q*-formula and *P′* be the formula to which higher-order narrowing steps are applied. [*Qu′*, *P′*] is obtained from [*Qu*, *P*] according to the rule below such that *P′* is the result of replacing all the object-level existentially quantified variables in *P* with meta-level existentially quantified variables:

Let Qu_1 be *Qu*. If [Qu_i, *some* λF.*Q*] ($1 \leq i \leq n$) then [Qu_{i+1}, $\omega_i(Q)$] where $Qu_{i+1} = Qu_i \exists Y$ (Y is not bound in Qu_i) and for each subterm $\overline{app}$ F $\bar{x}$ in *Q*, $<\overline{app}\ F\ \bar{x},\ Y\bar{x}> \in \omega_i$ (or for each subterm F in *Q*, $<F, Y> \in \omega_i$).

Note: $\omega_i(Q)$ is obtained by replacing *a* in *Q* with *b* for each $<a, b> \in \omega_i$.

Example 2.2 We will present the solution of the query

some λF.
eq (*la* λx.(*app* (*app* (*cons* "+")(*app* F x)) x))
(*la* λx.(*app* (*cons* "succ") x))

in the presence of the following rules:

- *all* λX.
  rule (*app* (*app* (*cons* "+") (*cons* "zero")) X)
  X.
- *all* λX.*all* λY.*rule*
  (*app* (*app* (*cons* "+") (*app* (*cons* "succ") X)) Y)
  (*app* (*cons* "succ") (*app* (*app* (*cons* "+") X) Y)).

step 1

The object-level existentially quantified variable F is replaced with the meta-level existentially quantified variable Z.

[*nil*, *some* λF.
eq (*la* λx.(*app* (*app* (*cons* "+")(*app* F x)) x))
(*la* λx.(*app* (*cons* "succ") x))]

<*nil*>

leads to

[∃Z, *eq* (*la* λx.(*app* (*app* (*cons* "+")(Z x)) x))
(*la* λx.(*app* (*cons* "succ") x))]

<*Z*>

Note: <....> denotes the ordered list containing instances of solvable variables in the same order their quantifiers appear in *Q*-formulas.

step 2

A narrowing step is applied to the subterm

(*app* (*app* (*cons* "+")(Z x)) x)

with the second rule. The lifting procedure is applied to the second rule over *x*.

[$\exists H_1 \exists H_2$, *rule*
(*app* (*app* (*cons* "+") (*app* (*cons* "succ")(H_1 x)))(H_2 x))
(*app* (*cons* "succ")(*app* (*app* (*cons* "+")(H_1 x))(H_2 x)))]

step 3

The resultant term after the narrowing step:

[$\exists H_1$, *eq*
(*la* λx.(*app* (*cons* "succ")
(*app* (*app* (*cons* "+")(H_1 x)) x)))
(*la* λx.(*app* (*cons* "succ") x))]

<λx.(*app* (*cons* "succ") (H_1 x))>

where Z is substituted by

λx.(*app* (*cons* "succ") (H_1 x)).

step 4

A narrowing step is applied to the subterm

(*app* (*app* (*cons* "+")(H_1 x)) x)

with the first rule. The lifting procedure is applied to the first rule over *x*.

[∃H_3, *rule*
(*app* (*app* (*cons* "+") (*cons* "zero")) (H_3 x))
(H_3 x)]

step 5

The resultant term after the narrowing step:

[*nil*, *eq* (*la* λx.(*app* (*cons* "succ") x))
(*la* λx.(*app* (*cons* "succ") x))]

<λx.(*app* (*cons* "succ") (*cons* "zero"))>

where H_1 is substituted by λx.(*cons* "zero").

step 6

Finally the result

[*nil*, *eq* (*la* λx.(*app* (*cons* "succ") x))
(*la* λx.(*app* (*cons* "succ") x))]

yields the formula [*nil*,**T**].

Note: T denotes tautologies.

Final result:

[*nil*,**T**]
<λx.(*app* (*cons* "succ") (*cons* "zero"))>

In general, a given query

[*nil*, *some* λF_1… *some* λF_n.Q]

<*nil*>

results in

[*nil*,**T**]

<t_1...t_n>

where t_1...t_n are respectively the solution values for F_1...F_n.

3 Implementation details

Below we present several examples of λProlog programs. The symbol ⇒ denotes implication, :- denotes its reverse. \ denotes λ-abstraction and pi along with a λ-abstraction denotes universal quantification. Upper case letters are assumed to be universally quantified. The symbol *o* in type expressions denotes the predicate types.

Example 3.1 The program below specifies the computation of all subterms within closed *T*-formulas.

Note: Closed *T*-formulas denote the *T*-formulas that do not contain any quantified variable.

```
type  fst term → term → term → o.

fst (cons T) (cons T) context.

fst (app T1 T2) (app K1 T2) context :-
    fst T1 K1 context.

fst (app T1 T2) Z (app L1 T2) :-
    fst T1 Z L1,   L1 = (app _ _).

fst (app T1 T2)  Z  (app T1 L2) :- fst T2 Z L2.

fst (la T) (la Z) (la L) :- pi c\(fst (T c) (Z c) (L c)).
```

Lemma 3.1 Let *s* be a closed *T*-formula. The goal ∃*Z*∃*L fst s Z L* produces the results for all the constant terms $s_{/p}$ such that *Z* and *L* are respectively substituted by

la y_1\...*la* y_n\$s_{/p}$ and *s*[*context*]$_p$

where y_1\...y_n\ are all the λ-abstractions in *s* covering p.

The constant *context* of the type *term* is used to denote the position to which replacement mechanism is applied in a higher-order narrowing step. After we apply the lifting procedure (see Definition 2.4) to a rewrite rule over y_1\...y_n\, we unify $s_{/p}$ with the left hand side of the rule. Then, *context* in *s*[*context*]$_p$ will be replaced with the right hand side.

Note: {*x* → *x'*}*s* denotes that all *x* in *s* are replaced with *x'*. The expression $\Sigma\ ;\ \Gamma \vdash_I G$ denotes *intuitionistic provability* of goal *G* from signature Σ and program Γ[2].

The Proof of Lemma 3.1: The proof is by induction.
Basis step:
CASE 1: when *s* is a variable, it should be a bound variable since the assumption. The expression fails.

CASE 2: s is a constant term. If it is of the form *cons c*, the expression is satisfied by backchaining with the first formula. If it is of the form *app* s_1 s_2, the expression is satisfied by backchaining iteratively with the second formula until it is reduced to the form *cons c*. For the both forms, Z and L are respectively substituted with s and *context*.

Inductive step:

CASE 1: s is *la* $x\backslash s_1$. The resultant expression

$$\Sigma \cup \{x': term\} ; \Gamma \vdash_I fst\ (\{x \rightarrow x'\}s_1)\ (Z_1\ x')\ (L_1\ x')$$

is computed by backchaining with the fifth formula where Z and L are respectively substituted with *la* Z_1 and *la* L_1. We apply induction hypothesis and the proof for this case is concluded.

CASE 2: s is *app* s_1 s_2. The resultant expression $\Sigma ; \Gamma \vdash_I fst\ s_2\ Z\ L_1$ is computed by backchaining with the forth formula where L is substituted with *app* s_1 L_1. Backchaining with the forth formula is used to apply *fst* procedure an argument of s. In order to successively apply *fst* procedure to the all arguments of s, backchaining with the third formula is used. We apply induction hypothesis to conclude the case. ■

Example 3.2 We will present a narrowing step applied to the subterm

```
(app (app (cons "+") (cons "zero"))
          x)
```

of the *T*-formula

```
(la λx.(app (app (cons "+")
             (app (app (cons "+") (cons "zero"))
                       x))
        x))
```

in the presence of the following rule:

```
all λX.
  rule (app (app (cons "+") (cons "zero")) X)
       X.
```

step 1

fst procedure is applied to the *T*-formula to compute

```
(la λx.(app (app (cons "+") (cons "zero")) x))
```

and

```
(la λx.(app (app (cons "+") context) x)).
```

step 2

The lifting procedure is applied to the rule over x.

```
[∃H1, rule
     (la λx.(app (app (cons "+") (cons "zero"))
                 (H1 x)))
     (la λx.(H1 x))]
```

step 3

The subterm

```
(la λx.(app (app (cons "+") (cons "zero"))
        x))
```

and the left hand of the rule

```
(la λx.(app (app (cons "+") (cons "zero"))
        (H1 x)))
```

are unified where H_1 is substituted by $\lambda x.x$.

step 4

context in

```
(la λx.(app (app (cons "+") context) x))
```

is replaced with the right hand side of the rule.

resultant term:

```
(la λx.(app (app (cons "+") x) x)).
```

When we deal with meta-level solvable variables at the object-level, we should specify their operational behavior at the object-logic. We do this specification by restricting their solution values. In the rest of this section, we will present a clear account for the special treatment of these meta-level solvable variables and the technicalities around that idea. For example, when s in Example 3.1 contains meta-level existentially quantified variables, they are vacuously unified with object-level terms within the programming formulas during the proof (see Example 3.3).

Example 3.3 In the proof of the expression,

$$\Sigma ; \Gamma \vdash_I fst\ (la\ x\backslash la\ y\backslash(app(app(cons\ \text{"+"})(Z\ y))x))\ K\ L$$

(K, L, K' L', Z are meta-level existentially quantified variables), the following expression is computed:

$$\Sigma \cup \{x': term, y': term\} ; \Gamma \vdash_I fst\ (Z\ y')\ K'\ L'.$$

During the proof of the goal *fst* (Z y′) $K'L'$, the variable Z is vacuously instantiated with the infinite sequence

y′\(*app* (*cons* F_1)F_2),

y′\(*app* (*app*(*cons* F_1) F_2) F_3),

……………………………………………

……………………………………………

y′\(*app*….(*app*(*cons* F_1) F_2)…F_n),

……………………………………………

……………………………………………

where F_1, F_2,…,F_n are meta-level existential variables. Because of the depth first search strategy of λProlog, Z is instantiated with these vacuous values in an infinite loop and other cases are not considered. This is the case that blocks the computation of desired results.

Position tree introduced in Definition 3.1 is a subsidiary structure, which is used to eliminate the case described above by preventing these infinite vacuous instantiations of meta-level solvable variables.

Definition 3.1 Let *s* be an object-level term over the constructors *app*, *la*, *cons*, *rule*, *eq* which may contain meta-level existentially quantified variables. If *s* is converted to a binary tree *t* whose nodes are defined over the constructors *n2*, *n1*, *nil*, and *evar* by an information erasing mapping such that

- The two-argument constructors *app*, *rule*, *eq* are mapped to the node *n2* with degree 2,
- The one-argument constructor *la* along with a λ-abstraction is mapped to the node *n1* with degree 1,
- The constructor *cons* and bound variables are mapped to the node *nil* with degree 0,
- The subterms Z y_1...y_n are mapped to the nodes *evar* Z′, which denote the existence of a meta-level solvable variable in the current position (Z and Z′ are meta-level existentially quantified variables),

t is called *the position tree* or briefly *position* of *s* and denoted by $(s)^*$.

The types are assigned to the constructors *n2*, *n1*, *nil*, *evar* as follows:

n2 : *term* → *term* → *term*,

n1 : *term* → *term*,

nil : *term*,

evar : *A* → *term*.

A denotes type variables (polymorphic types) and is universally quantified with a type quantifier around the type declaration.

Example 3.4

(*n1*(*n1*(*n2*(*n2 nil* (*evar* Z_1)) *nil*)))

is the position tree of

(*la* x*la* y\(*app*(*app*(*cons* "+")(Z y))x)).

The subterm (Z y) here is encoded with the node (*evar* Z_1). We always guarantee the following case for preserving the mapping by using some techniques, which will be given later: when the variable Z is substituted with y*t*, we also substitute the variable Z_1 with $(t)^*$ and eliminate the *evar*.

By using the notion of position tree, we update the program *fst* in the following way.

```
type   fst   term → term → term →
                    term → term → term → o.

fst (cons T) (cons T) context (nil) (nil) context.

fst (app T1 T2) (app K1 T2) context
    (n2 N1 N2) (n2 M1 N2)  context     :-
                 fst T1 K1 context N1 M1 context.

fst (app T1 T2) Z (app L1 T2)
    (n2 N1 N2) M (n2 Q1 N2) :- fst T1 Z L1 N1 M Q1,
                        L1 = (app _ _), Q1 = (n2 _ _).

fst (app T1 T2) Z (app T1 L2)
    (n2 N1 N2) M (n2 N1 Q2) :- fst T2 Z L2 N2 M Q2.

fst (la T) (la Z) (la L) (n1 N) (n1 M) (n1 Q) :-
                         pi c\(fst (T c) (Z c) (L c) N M Q).
```

Lemma 3.2 Let *s* be an object-level term over the constructors *app*, *la*, *cons* which may contain variables existentially quantified at the meta-level.

For all p, (p denotes the positions of the constant terms in *s* and $y_1\backslash...y_n\backslash$ are all the λ-abstractions in *s* covering p), the goal

$$\exists Z \exists L \exists Z' \exists L'\ fst\ s\ Z\ L\ (s)^*\ Z'\ L'$$

is satisfied where *Z*, *L* are respectively substituted by *la* $y_1\backslash ...$*la* $y_n\backslash s_{/p}$, $s[context]_p$ and then the goal terminates.

Note: *Z'* and *L'* are respectively substituted by the position trees of *la* $y_1\backslash ......$*la* $y_n\backslash s_{/p}$ and $s[context]_p$.

The Proof of Lemma 3.2:

The proof is similar to that of Lemma 3.1. For the termination consider the case when *s* is of the form $H\ y_1...y_n$ where *H* is a meta-level existential quantified variable. Assume that it is encoded by (*evar H'*). The goal

$$\exists Z \exists L \exists Z' \exists L'\ fst\ (H\ y_1...y_n)\ Z\ L\ (evar\ H')\ Z'\ L'$$

fails since it can not find any formula to backchain. The termination is guaranteed by the encoding mechanism. ■

Example 3.5 In the proof of the expression,

$\Sigma; \Gamma \vdash_I$ *fst* (*la* x*la* y\\(*app*(*app*(*cons* "+")(Z y))x))
K L
(*n1*(*n1*(*n2*(*n2 nil* (*evar* Z_1)) *nil*)))
M N

(*K*, *L*, *M*, *N*, Z, Z_1, *K'*, *L'*, *M'*, *N'* are meta-level existentially quantified variables), the following expression is computed:

$\Sigma \cup \{x' : term, y' : term\}$; $\Gamma \vdash_I$ *fst* (Z y') *K' L'*
(*evar* Z_1) *M' N'*

The variable *Z* here will not be vacuously instantiated as in Example 3.3 because the expression will be failed by the existence of *evar* in the goal. Here, we should make the point clear. The reason for encoding the meta-level variable Z with *evar* is **not** that we want to specify the computation of non-variable subterms that will be unified with the left side of a rewrite rule. The reason is exactly to eliminate the infinite vacuous instantiations of Z, which were described in Example 3.3.

The correctness of the implementation depends on the preservation of the mapping between original object-level terms and their position trees through unification and replacement processes in higher-order narrowing steps. These processes are applied in parallel to both original object-level terms and their position trees so that the preservation is guaranteed. For example, in *fst* program, we also compute the position trees of *la* $y_1\backslash ...$*la* $y_n\backslash s_{/p}$ and $s[context]_p$ (see Lemma 3.2) in order to apply in parallel the same narrowing steps to $(s)^*$ with the position tree of the same rewrite rule used for the narrowing step of *s*. We use a specialized unification algorithm for positional structures (see the program ***up*** in Appendix). The program *up* unifies positional structures ignoring the "evar". But after the unification, it is the case that some *evar* constructors encode terms different from meta-level solvable variables. See Example 3.6.

Example 3.6 Let *n1*(*nil*) and *n1*(*evar*(*G*)) be respectively the positions of *la* λx.x and

la λx.(F x).

Unification is applied to the pair

<*la* λx.x, *la* λx.(F x)>

by the expression *la* λx.x = *la* λx.(F x) and the specialized unification algorithm is simultaneously applied to the pair <*la* λx.x, *la* λx.(F x)> by the expression

$\Sigma; \Gamma \vdash_I$ *up n1*(*nil*) *n1*(*evar*(G)).

After the expressions are proved, the result is the following forms <*la* λx.x, *la* λx.x> and

<*n1*(*nil*), *n1*(*evar*(*nil*))>.

We should further eliminate the *evar* from *n1*(*evar*(*nil*)) since it no longer encodes a meta-level solvable variable. After the elimination, the mapping between the second elements of these pairs still holds.

We need the following procedure, which is further applied to positional structures for preserving the mapping after their unification.

Definition 3.2 All *evar* constructors whose occurrences in a position tree are of the form *evar t* where *t* is not a meta-level existentially quantified variable are systematically eliminated by replacing *evar t* with *t*. The procedure is called *the mapping preserving* procedure.

In the application (see Section 4), after unification and replacement processes are applied in parallel to both original object-level terms and their position trees, the mapping preserving procedure is further applied to the position trees.

In the rest of this section, we will consider the implementation of the mapping preserving procedure, which is given in the following steps:

a) Replacement of meta-level existential variables with object-level existential variables

b) Elimination of all *evar* constructors

c) Replacement of object-level existential variables with meta-level existential variables

The implementation of the technique given in the step a is based on the iterative applications of the following step where each application replaces one meta-level variable.

- A new object-level variable is added to the argument list of each of the remaining meta-level existential variables. Then, one of them is unified with this new object-level variable.

In the step given above, unification is exploited for the replacement process. A new object-level variable is added to the argument lists for ensuring that the unification be always successful. We will give the specification of the technique in the step a as follows:

```
type rmo, rmo1 term → term → o.
type rmo02 term → term → term → o.

rmo1 (evar V)       V  :- !.
rmo1 (n2 N _)       V  :- rmo1 N V, !.
rmo1 (n1 N)         V  &
rmo1 (evar N)       V  &
rmo1 (n2 _ N)       V :- rmo1 N V.
```

```
rmo02 (n1 N1) V (n1 N2) :- rmo02 N1 V N2.
rmo02 nil _ nil.
rmo02 (evar N)  V  (evar (N V)) :- !.
rmo02 (evar N1) V (evar N2) :-  rmo02 N1 V N2, !.
rmo02 (evar N) _ (evar N).
rmo02 (n2 N1 N2) V (n2 U1 U2) :-
                    rmo02 N1 V U1, rmo02 N2 V U2.

rmo Y (some Q) :- pi c\(rmo02 Y c (Z c),
          rmo1 (Z c) c, rmo (Z c) (Q c)), !.
rmo Y Y.
```

The formula *rmo02* is used for the addition of a new object-level variable to the argument list of each of the remaining meta-level existential variables. The formula *rmo1* is used to replace one meta-level existential variable with this object-level variable by exploiting unification.

In the formula

```
rmo02 (evar N)  V  (evar (N V)) :- !,
```

if the variable N is instantiated with a meta-level existential variable, the formula is satisfied since meta-level existential variables are of the type A (meaning that they are of any type). On the other hand, if the variable N is instantiated with an object-level term, the formula is failed since object-level terms are of the type *term*. By specifying meta-level existential variables being of any type, we place them in a more general category than that the object-level terms belong to.

Example 3.7 We will present an application of the step a to the position tree

```
(n1 (n2
      (n2 nil (evar Z1))
      (n2
         (n2 nil (evar Z1))
         (evar nil)))).
```

We change the notation to the following

```
(n1 (n2
      (n2 nil (evar Z1 : A))
      (n2
         (n2 nil (evar Z1 : A))
         (evar nil : term))))
```

in order to explicitly show how the type expressions have an effect on the resulting computations.

Note: τ in the expression (*evar p* : τ) denotes that *p* is of the type τ. Z_1, Y_1, Y_2, Y_3, U_1, U_2, U_3, U_4, U_5 are meta-level existentially quantified variables.

step 1

$\Sigma; \Gamma \vdash_I$ *rmo* (*n1* (*n2* (*n2 nil* (*evar* Z_1 : A)) (*n2* (*n2 nil* (*evar* Z_1 : A)) (*evar nil* : *term*)))) Y_1 **(I)**

leads to

$\Sigma \cup \{c_1 : term\}; \Gamma \vdash_I$ **(II)**

rmo02 (*n1* (*n2* (*n2 nil* (*evar* Z_1 : A)) (*n2* (*n2 nil* (*evar* Z_1 : A)) (*evar nil* : *term*)))) c_1 (U_1 c_1)

and

$\Sigma \cup \{c_1 : term\}; \Gamma \vdash_I$ *rmo1* (U_1 c_1) c_1 **(III)**

and

$\Sigma \cup \{c_1 : term\}; \Gamma \vdash_I$ *rmo* (U_1 c_1) (Y_2 c_1) **(IV)**

where Y_1 is substituted with *some* Y_2.

step 2

By the proof of **II** U_1 is substituted by

c_1\(*n1* (*n2* (*n2 nil* (*evar* (Z_1 c_1) : B)) (*n2* (*n2 nil* (*evar* (Z_1 c_1) : B)) (*evar nil* : *term*))))

Note: We change the type variable from A to B in order to express the changing type. The proof of **II** induces the two proof expressions

$\Sigma \cup \{c_1 : term\}; \Gamma \vdash_I$ *rmo02* (*evar* Z_1 : A) c_1 U_2 **(V)**

and

$\Sigma \cup \{c_1 : term\}; \Gamma \vdash_I$ *rmo02* (*evar nil* : *term*) c_1 U_3. **(VI)**

Notice that the expression **V** will backchain with

rmo02 (*evar* N) V (*evar* (N V)) :- !.

whereas the expression **VI** will fail to backchain with that formula since *nil* is of the atomic type *term*. The type expressions enhance the computations.

step 3

By the proof of **III** Z_1 is substituted with c_1\c_1 and the value for U_1 is changed to

c_1\(*n1* (*n2* (*n2 nil* (*evar* c_1 : *term*)) (*n2* (*n2 nil* (*evar* c_1 : *term*)) (*evar nil* : *term*))))

Notice that the type is changed to *term*.

Note: The proof of **III** induces the expression

$\Sigma \cup \{c_1 : term\}; \Gamma \vdash_I$ *rmo1* (*evar* (Z_1 c_1) : B) c_1.

The formula

rmo1 (*evar* V) V :- !.

occurs at the beginning of the program in order to guarantee that the expression will first backchain with that formula. Cut (!) is put in order to prevent backchaining with the other formula which leads a vacuous instantiation of Z_1.

step 4

The proof of **IV** induces to

$\Sigma \cup \{c_1 : term\} \cup \{c_2 : term\}; \Gamma \vdash_I$ *rmo02* **(VII)**

n1 (*n2* (*n2 nil* (*evar* c_1 : *term*)) (*n2* (*n2 nil* (*evar* c_1 : *term*)) (*evar nil* : *term*))) c_2 (U_4 c_2)

and

$\Sigma \cup \{c_1 : term\} \cup \{c_2 : term\}; \Gamma \vdash_I$ *rmo1* (U_4 c_2) c_2 **(VIII)**

step 5

By the proof of **VII** U_4 is substituted with

c_2\(*n1* (*n2* (*n2 nil* (*evar* c_1 : *term*)) (*n2* (*n2 nil* (*evar* c_1 : *term*)) (*evar nil* : *term*)))).

Note: The expression

$\Sigma \cup \{c_1 : term\} \cup \{c_2 : term\}; \Gamma \vdash_I$ *rmo02* (*evar* c_1 : *term*) c_2 U_5.

is induced by the proof of **VII**. It will not backchain with

rmo02 (*evar* N) V (*evar* (N V)) :- !.

since c_1 is of the atomic type *term*. By using types, we impose constrains and specify the direction of computations.

Notice that the proof of **VIII** will fail. This is because the expression

```
Σ∪ {c1 : term} ∪ {c2 : term}; Γ ├I rmo1
        (n1 (n2  (n2 nil (evar c1 : term))
                 (n2
                     (n2 nil (evar c1 : term))
                     (evar nil : term))))
        c2
```

does not lead to any expression that successfully backchains with the formula

```
rmo1 (evar V) V   :- !.
```

step 6

The expression **IV** will backchain with the formula

```
rmo Y Y
```

and the proof is completed where Y_2 is substituted by

```
c1\(n1 (n2  (n2 nil (evar c1 : term))
            (n2
                (n2 nil (evar c1 : term))
                (evar nil : term))))
```

and therefore Y_1 is substituted by

```
some c1\(n1 (n2  (n2 nil (evar c1 : term))
                 (n2
                     (n2 nil (evar c1 : term))
                     (evar nil : term)))).  ■
```

Lemma 3.3 For any positional structure p, the goal $\exists H$ *rmo p H* succeeds once and then terminates.

The specification of the technique in the step b is as follows:

```
type el_ev term → term → o.

el_ev  (n1 N1) (n1 N2) :- el_ev N1 N2.
el_ev (some N1) (some N2) :-
                    pi c\( el_ev (N1 c) (N2 c)).
el_ev (n2 N1 N2) (n2 N3 N4) :-
                    el_ev N1 N3, el_ev N2 N4.
el_ev (evar (n2 N1 N2)) T :- el_ev (n2 N1 N2) T, !.
el_ev (evar N) N.
el_ev nil nil.
```

Example 3.8 We will consider an application of the step b to the result produced in the Example 3.7. The proof expression

```
Σ; Γ ├I el_ev
some c1\(n1 (n2  (n2 nil (evar c1))
                 (n2
                     (n2 nil (evar c1))
                     (evar nil ))))
H1
```

induces the formulas

```
Σ∪ {c1 : term} ; Γ ├I el_ev (evar c1) H2        (I)
```

and

```
Σ∪ {c1 : term} ; Γ ├I el_ev  (evar nil) H3      (II)
```

where H_1, H_2 and H_3 are meta-level existentially quantified variables.

H_2 and H_3 are respectively substituted by c_1 and *nil* via the backchaining of **I** and **II** with

```
el_ev (evar N) N.
```

and H_1 is as a result substituted by

```
some c1\(n1 (n2  (n2 nil c1)
                 (n2
                     (n2 nil c1)
                     nil))). ■
```

The specification of the technique in the step c is as follows:

```
type rom term → term → o.

rom (n1 N1) (n1 N2) :- rom N1 N2.
rom nil nil.
rom (some N1) N2 :-
            pi y\ rom y (evar _) ⇒ rom (N1 y) N2.
rom (n2 N1 N2) (n2 N3 N4) :-
        rom N1 N3, rom N2 N4.
```

Example 3.9 We will consider an application of the step c to the result produced in the Example 3.8. The proof expression

```
Σ; Γ ├I rom
some c1\(n1 (n2  (n2 nil c1)
                 (n2
                     (n2 nil c1)
                     nil)))
H1
```

induces

```
Σ∪ {c1 : term} ; Γ∪ {rom c1 (evar Z)} ├I
                                   rom c1 H2
```

where H_1, H_2, and Z are meta-level existentially quantified variables. The goal *rom* c_1 H_2 backchains with the formula *rom* c_1 (*evar Z*).

and H_2 is substituted by (*evar Z*). H_1 is as a result substituted by

```
(n1 (n2  (n2 nil  (evar Z))
         (n2
            (n2 nil (evar Z))
             nil))). ■
```

The specification of the mapping preserving procedure is as follows:

mpp NI NF :- *rmo* NI M, *el_ev* M L, *rom* L NF.

4 Implementation

In this section we will present the implementation of higher-order narrowing. *mpp* and *fst* programs, which were given in the previous section are used in this section. See Appendix for the rest of the programs used in the implementation.

4.1 Specification of higher-order narrowing

For simplicity, we will first consider higher-order steps applied only to the left hand side of a query. We will later consider higher-order steps applied to the both sides.

The list constructor *subs* of the type

term → *term* → *term*

denotes the list containing instances of variables to be solved in the same order their quantifiers appear in *Q*-formulas and *nil* here is used to denote empty lists.

Note: Recall that the symbol <....> in Section 2 denotes the ordered list containing instances of solvable variables. In this section it is denoted by the constructor *subs*.

Example 4.1.1 After the query

some λF. *some* λK. *some* λL.*Q*,

is solved, the list

subs (*evar* N) *subs* (*evar* M) *subs* (*evar* S) *nil* respectively contains the solution values N, M, S of F, K, L.

We assume that after a query is solved, the *subs* list contains only the closed instances.

4.1.1 Higher-order narrowing applied only to the left hand side

```
type query          term → term → o.
type  pre_nar       term → term → term → o.
type  ho_narr_step  term → term → term → term → o.
type  rewrite_rules  term → o.

ho_narr_step T N U Y :-
        fst T Z1 L1 N Z2 L2,
        rewrite_rules R,
        r_o_m_rl  R   (rule LS1 RS1),
        c_pos   R   (n2 LS2 RS2),
        lf1 L2 (rule LS1 RS1) (rule LS3 RS3),
        rev1 Z2  (rule LS3 RS3) (rule LS4 RS4),
        Z1 = LS4,  up Z2 LS2,
        rc L2 L1 RS4 RS2 U Y.

pre_nar (subs X K)  N (subs X L) :- pre_nar K N L.
pre_nar (eq T T) _ nil.
pre_nar (eq T1 T2)  (n2 N1 N2)  nil :-
        ho_narr_step T1 N1 T3 N3,
        mpp (n2 N3 N2)  N4,
        pre_nar (eq T3 T2)  N4  nil.

query Q S :- rep_o_m Q K,
          c_pos Q L,
          pre_nar K L S.
```

Example 4.1.1.1 We will present the solution of the query below in the presence of the following rules:

rules:

```
• rewrite_rules (all  X\
        rule (app (app (cons "+") (cons "zero")) X)
             X).
• rewrite_rules  (all  X\all  Y\rule
    (app (app (cons "+") (app (cons "succ") X)) Y)
    (app (cons "succ") (app (app (cons "+") X) Y))).
```

query:

```
query
(some F\
  eq (la λx.(app (app (cons "+")(app F x)) x))
      (la λx.(app (cons "succ") x)))
U.
```

Note: *U* is a meta-level existentially quantified variable. For notational simplicity we ignore Σ and Γ in the expression

$$\Sigma;\ \Gamma \vdash_{I} G.$$

For the following computations Z_1, G, G_1, G_2, H_1, H_2, H_3, H_4, H_5, H_6 are meta-level existentially quantified variable.

step 1

The object-level existentially quantified variable F is replaced with the meta-level existentially quantified variable Z.

```
|-I
query
(some F\
  eq (la x\(app (app (cons "+")(app F x)) x))
     (la x\(app (cons "succ") x)))
U
```

leads to

```
|-I
pre_nar
  eq (la x\(app (app (cons "+")(Z x)) x))          (I)
     (la x\(app (cons "succ") x))

 n2 (n1 (n2 (n2 nil (evar G)) nil))
     (n1 (n2 nil nil))

 nil
```

where U is substituted by

subs (*evar* Z) *nil*.

step 2

The proof of **I** induces

```
|-I ho_narr_step
     (la x\(app (app (cons "+")(Z x)) x))         (II)
     (n1 (n2 (n2 nil (evar G)) nil))
     H1
     H2
```

and

```
|-I
mpp (n2 H2 (n1 (n2 nil nil))) H3                   (III)
```

and

```
|-I
pre_nar                                            (IV)
  eq H1
     (la x\(app (cons "succ") x))
 H3
 nil.
```

step 3

By the proof of **II**, a narrowing step is applied to the subterm

(*app* (*app* (*cons* "+")(Z x)) x)

with the second rule and H_1, H_2 are respectively substituted by

```
(la x\(app (cons "succ")
          (app (app (cons "+")(Z1 x))x)))
```

and

(*n1* (*n2 nil* (*n2* (*n2 nil* (*evar* G_1)) (*evar nil*)))).

Z is substituted by

x\(*app* (*cons* "succ") (Z_1 x)).

step 4

By the proof of **III**,

```
|-I
mpp (n2 (n1 (n2 nil
                (n2 (n2 nil (evar G1))
                    (evar nil))))
        (n1 (n2 nil nil)))
    H3
```

H_3 is substituted by

```
(n2 (n1 (n2 nil
            (n2 (n2 nil (evar G2))
                nil)))
    (n1 (n2 nil nil))).
```

step 5

The proof of **IV**

```
|-I
pre_nar
  eq (la x\(app (cons "succ")
                (app (app (cons "+")(Z1 x))
                     x)))
     (la x\(app (cons "succ") x))

  (n2 (n1 (n2 nil
              (n2 (n2 nil (evar G2))
                  nil)))
      (n1 (n2 nil nil)))

  nil
```

induces

```
|-I
ho_narr_step                                       (V)
(la x\(app (cons "succ")
          (app (app (cons "+")(Z1 x))
               x)))
(n1 (n2 nil
        (n2 (n2 nil (evar G2))
            nil)))
H4
H5
```

and

```
|-I
mpp (n2 H5 (n1 (n2 nil nil))) H6                   (VI)
```

and

```
|-_l
pre_nar
  eq H4                                   (VII)
     (la x\(app (cons "succ") x))
  H6
 nil.
```

step 6

By the proof of **V**, a narrowing step is applied to the subterm

```
(app (app (cons "+")(Z1 x))
                    x)
```

with the first rule and H_4, H_5 are respectively substituted by

```
(la x\(app (cons "succ") x))
```

and

```
(n1 (n2 nil (evar nil))).
```

Z_1 is substituted by x\(*cons* "succ") and therefore U is substituted by

```
subs (evar x\(app (cons "succ") (cons "zero")))
     nil.
```

By the proof of **VI**, H_6 is substituted by

```
(n2  (n1 (n2 nil nil))
          (n1 (n2 nil nil))).
```

The expression **VII**

```
|-_l
pre_nar
  eq (la x\(app (cons "succ") x))
     (la x\(app (cons "succ") x))
(n2  (n1 (n2 nil nil))
     (n1 (n2 nil nil))).
 nil
```

is satisfied. ■

4.1.2 Higher-order narrowing applied to the both sides

We change the specification of *pre_nar* in 4.1.1 as follows:

```
pre_nar (subs X K)  N (subs X L) :- pre_nar K N L.
pre_nar (eq T T) _ nil.
pre_nar (eq T1 T2)  (n2 N1 N2)  nil :-
        (ho_narr_step T1 N1 T3 N3,
        mpp (n2 N3 N2) N4,
        pre_nar (eq T3 T2)  N4  nil) ;
        (ho_narr_step T2 N2 T3 N3,
        mpp (n2 N1 N3)  N4,
        pre_nar (eq T1 T3)  N4  nil).
```

4.1.3 Conjunction of queries

For simplicity, we have considered the implementation for a single query. By simple modifications we can extend the result for a conjunction of queries:

- *Q*-formulas are redefined as follows:

  A ::= *eq T T* | *and* (*eq T T*) *A*,

  Q ::= *A* | *some* λF.*Q*,

where *and* is of *term → term → term*.

- In Definition 3.1 the two-argument constructor *and* is also mapped to the node *n2* with degree 2.
- Last we change the specification of *pre_nar* in 4.1.1 as follows:

```
pre_nar (subs X K)  N (subs X L) :- pre_nar K N L.
pre_nar (eq T T) _ nil.
pre_nar (and (eq T T) K) (n2 (n2 N1 N2) L) nil :-
      up   N1 N2,
      mpp L P,
      pre_nar K P nil.
pre_nar (eq T1 T2)  (n2 N1 N2)  nil :-
      (ho_narr_step T1 N1 T3 N3,
       mpp (n2 N3 N2)  N4,
       pre_nar (eq T3 T2)  N4  nil) ;
      (ho_narr_step T2 N2 T3 N3,
       mpp (n2 N1 N3)  N4,
       pre_nar (eq T1 T3)  N4  nil).
pre_nar (and (eq T1 T2) K) (n2 (n2 N1 N2) L) nil :-
      (ho_narr_step T1 N1 T3 N3,
       mpp  (n2 (n2 N3 N2) L)  N4,
       pre_nar (and (eq T3 T2) K)  N4 nil);
      (ho_narr_step T2 N2 T3 N3,
       mpp  (n2 (n2 N1 N3) L)  N4,
       pre_nar (and (eq T1 T3) K) N4 nil).
```

4.2 Extending to pattern rules

For simplicity, we have considered the implementation for first-order rules. By simple modifications we can extend the result for pattern rules.

- *R*-formulas are redefined as follows:

  R ::= *rule T T* | *all* λF.*R*.

- The last clause in *fts* program is changed to

```
fst (la T) (la Z) (la L) (n1 N) (n1 M) (n1 Q) :-
       pi c\(fst c c context (nil) (nil) context  =>
                          fst (T c) (Z c) (L c) N M Q).
```

and the following clause is added to the top.

```
fst (la T) (la T) context (n1 N) (n1 N) context.
```

4.3 Improvement of the search

In the implementation, the search for unifiers can go down infinite paths. We will in this section introduce a control mechanism in order to improve the search. We assume that the rules are first-order and terminating.

4.3.1 Apply narrowing if it is needed

Example 4.3.1.1 Even though the query

```
some λF.eq
(la λx.(app (cons "succ")
            (app  (cons "succ")
                  (app (app (cons "+") (app F x))
                       x))))
(la λx.(app (cons "succ") (cons "zero")))
```

is unsolvable, the interpreter does not terminate and infinitely applies narrowing steps.

Assume that the object-level existentially quantified variable F is replaced with the meta-level existentially quantified variable Z. Before a narrowing step being applied to the subterm

```
(app (app (cons "+") (Z x))
     x)
```

it is needed to check whether the top level constant term

```
(app  (cons "succ") (app (app (cons "+")(Z x))
                          x))
```

can be reduced to (*cons* "zero"). Since it can not match the left side of any rule in Example 4.1.1.1, it can not be reduced to (*cons* "zero"). Applying a narrowing step to its subterm

```
(app (app (cons "+") (Z x))
     x)
```

leads to going down the infinite path without a solution.

We can use a control procedure that checks the possibility of a constant term to match the left side of a rule before narrowing steps being applied.

Example 4.3.1.2 For a constant term c, the goal $\exists H$ *is_match* $(c)^{*}c\ H$ succeeds. If c can match the left side of a rule, H is substituted by the object-level term *suc*. Otherwise H is substituted by the object-level term *fail*.

4.3.2 Problematic case

The following is the situation that can not be treated by the procedure given in Section 4.3.1. Assume that t is the constant term to which narrowing is applied and t_p is another constant term that occurs in t at the position p. l_1 and l_2 are left hand sides of two rewrite rules. l_1 and l_2 match t and t_p respectively. Compared to t, instead of t_p, an object-level universally quantified variable is at the position p in l_1.

Example 4.3.2.1 Let the following be a query where the object-level existentially quantified variable F is replaced with the meta-level existentially quantified variable Z.

```
some λF.
eq (la λx.(app (cons "f")
               (app (app (cons "+")(app F x))
                    x)))
   (la λx.(cons "zero")).
```

Let l_1 be the following constant term

```
(app (cons "f") X)
```

where X is an object-level universally quantified variable. Assume that t and t_p are respectively

```
(app (cons "f") (app (app (cons "+")(Z x))
                     x))
```

and

```
(app (app (cons "+")(Z x))
     x).
```

Even if t can not be reduced to

```
(cons "zero"),
```

the procedure can not detect this since t will always match l_1 while going down the infinite path.

Appendix

//* The program *rep_o_m* replaces object-level existentially quantified variables with meta-level existentially quantified variables *//

```
type rep_o_m    term → A → o.

rep_o_m  (cons T1) (cons T1).
rep_o_m  (app T1 T2) (U1 T2) :- rep_o_m T1 U1, !.
rep_o_m  (la T1) (la T2)  :-
        pi y\( rep_o_m y y ⇒ rep_o_m (T1 y) (T2 y)).
rep_o_m  (some T1) (subs (evar F) T2) :-
        pi y\( rep_o_m y F ⇒ rep_o_m (T1 y) T2).
rep_o_m (and T1 T2) (and U1 U2)   &
rep_o_m (app T1 T2) (app U1 U2)   &
rep_o_m (eq T1 T2)   (eq U1 U2)         :-
                        rep_o_m T1 U1, rep_o_m T2 U2.
```

//* The program *c_pos* computes position trees *//

```
type c_pos term → term →  o.

c_pos (cons _) nil.
c_pos nil nil.
c_pos (app T _) (evar U) :- c_pos T (evar U), !.
c_pos (la T1) (n1 T2) :- c_pos (T1 nil) T2.
c_pos (some T1) T2 :-
            pi y\ c_pos y (evar _) ⇒ c_pos (T1 y) T2.
c_pos (all T1) T2 :-
            pi y\ c_pos y (evar _) ⇒ c_pos (T1 y) T2.
c_pos (and T1 T2) (n2 U1 U2)   &
c_pos (app T1 T2) (n2 U1 U2)   &
c_pos (rule T1 T2) (n2 U1 U2)   &
c_pos (eq T1 T2)    (n2 U1 U2)   :-
                        c_pos T1 U1, c_pos T2 U2.
```

//* The program *r_o_m_rl* replaces object-level universally quantified variables in rules with meta-level existentially quantified variables *//

```
type r_o_m_rl   term → term →  o.

r_o_m_rl (cons T) (cons T).
r_o_m_rl (app T1 T2) (evar U1 T2) :-
                          r_o_m_rl T1 (evar U1), !.
r_o_m_rl (la T1) (la T2)  :-
        pi y\(r_o_m_rl  y y ⇒ r_o_m_rl  (T1 y) (T2 y)).
r_o_m_rl (all T1) T2 :-
        pi y\ r_o_m_rl y (evar _) ⇒ r_o_m_rl (T1 y) T2.
r_o_m_rl (app T1 T2) (app U1 U2)   &
r_o_m_rl (rule T1 T2) (rule U1 U2)  :-
                        r_o_m_rl T1 U1, r_o_m_rl T2 U2.
```

//* The program *lf1* takes lifting of a rewrite rule *//

```
type lf1, lf02  term → term → term → o.

lf1 (n1 T) (rule Y1 Y2) (rule (la P1) (la P2)) :-
    pi c\((lf02 Y1 (Z1 c) c), (lf02 Y2 (Z2 c) c),
         (lf1 T (rule (Z1 c) (Z2 c))(rule (P1 c) (P2 c)))).
lf1 (n2 L _) Y P :- lf1  L Y P.
lf1 (n2 _ R) Y P :- lf1  R Y P.
lf1 context  Y Y.

lf02 (evar N) (evar (N V)) V.
lf02 (cons T) (cons T) _.
lf02 (app T1 T2) (app Z1 Z2) V :-
                        lf02 T1 Z1 V, lf02 T2 Z2 V.
lf02 (la T1) (la T2) V :-
        pi y\(lf02  y y _ ⇒ lf02  (T1 y) (T2 y) V) .
```

//* The program *rev1* removes *evar* 's from a rewrite rule *//

```
type rev1 term → term → term → o.

type rev02  term → term → o.

rev1 (n1 T) (rule (la K) (la L)) (rule (la M) (la N)) :-
        pi c\rev1 T  (rule (K c) (L c)) (rule (M c) (N c)).
rev1 (n2 _ _) (rule Y1 Y2) (rule Z1 Z2)       &
rev1 nil        (rule Y1 Y2) (rule Z1 Z2)       :-
                        rev02 Y1 Z1, rev02 Y2 Z2.

rev02 (evar V) V.
rev02 (cons T) (cons T).
rev02 (app Y1 Y2) (app Z1 Z2) :-
                        rev02 Y1 Z1, rev02 Y2 Z2.
```

// The program *up* unifies positional structures //

```
type up term → term → o.

up (n1 N) (n1 M) :- up N M, !.
up (n1 N) M :- up N M.
up (evar N) (evar N).
up (evar (n2 M1 M2)) (n2 M1 M2).
up (n2 M1 M2) (evar (n2 M1 M2)).
up (evar nil) nil.
up nil (evar nil).
up nil  nil.
up (n2 N1 N2) (n2 M1 M2) :-  up N1 M1, up N2 M2.
```

//* The program *rc* replaces *context* *//

```
type rc term → term → term →
              term → term → term → o.

rc (n1 P) (la Y) (la Z) N (la U) (n1 M)   :-
        pi x\(rc P (Y x) (Z x) N (U x) M).
rc (n2 P1 P2) (app T1 T2) Z N
    (app T1 U)  (n2 P1 M) :- rc P2 T2 Z N U M, !.
rc (n2 P1 P2) (app T1 T2) Z N
    (app U T2) (n2 M P2)  :- rc P1 T1 Z N U M.
rc context context Z N Z N.
```

5 Conclusion

We illustrate how λ-terms and types embeded in λProlog extend the concept of logic programming. The adaptability of our techniques requires a meta-level system that supports λ-abstraction, types, polymorphic typing, implications and universal quantification in goals and the body of clauses. By using implications and universal quantification in goals and the body of clauses, we directly reason about λ-terms. A similar way is used in [6]. Moreover, we give polymorphic types to meta-level solvable variables for their special treatment at the object-level. Particularly, we present them in a more general category than that the object-level terms in. By using type expressions we specify the direction of computations.

Selection of a rule and position to which narrowing steps are applied is in the default controlling, that is, the depth first search strategy. The search for unifiers can go down infinite paths. The search can be improved by using control procedures. Using high-level search primitives described in [6] can enhance our implementation.

Acknowledgments I would like to thank Dale Miller for his encouraging comments when I first introduced my ideas. I would thank to Cem Say and referees for their comments on my work.

References

1. Miller, D. and G. Nadathur (1987) "A Logic Programming Approach To Manipulating Formulas and Programs," *IEEE Symposium on Logic Programming*, pp. 379-388, San Francisco, September.
2. Miller, D. (1991) "A Logic Programming Language with Lambda-Abstraction, Function Variables, and Simple Unification," *Journal of Logic and Computation*, Vol. 1, No. 4.
3. Prehofer, C. (1994) "Higher-order Narrowing," *Proceedings of the ninth Annual IEEE Symposium on Logic in Computer Science*, pp. 507-516.
4. Qian, Z. (1994) "Higher-Order Equational Logic Programming," *Proceedings of the 21st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 254-367, January.
5. Nipkow, T. (1989) "Equational Reasoning in Isabelle,“ *Science of Computer Programming* Vol. 12, Number 2, pp.123-149.
6. Felty, A. (1992) "A Logic Programming Approach to Implementing Higher-Order Term Rewriting," *Proceedings of the January 1991 Workshop on Extensions to Logic Programming*, Springer-Verlag LNCS 596.