Computer Science Department Technical Report
University of California
Los Angeles, CA 90024-1596

PROLOG TECHNOLOGY TERM REWRITING: APPLICATIVE SYSTEMS

M. H. M. Cheng
D. S. Parker
M. H. van Emden

December 1993
CSD-930041

Prolog Technology Term Rewriting: Applicative Systems

M.H.M. Cheng[*]
Department of Computer Science
University of Victoria, P.O. Box 3055
Victoria, B.C. V8W 3P6 Canada

D. Stott Parker[†]
Computer Science Department
University of California
Los Angeles, CA 90024-1596 USA

M.H. van Emden[‡]
Department of Computer Science
University of Victoria, P.O. Box 3055
Victoria, B.C. V8W 3P6 Canada

Abstract

We explore *Applicative Term Rewriting Systems*, first-order term-rewriting systems that axiomatize function application. These systems are similar in flavor to Klop's higher order combinatory reduction systems, but involve only first-order terms.

Applicative term-rewriting systems have several aspects that are interesting to us. First, they are simple, and their rules can be integrated directly with declarative, first-order axiomatizations of the equality relation. Second, they are easily implemented with Prolog, thereby making a basis for a Prolog Technology Term Rewriter in the sense of Stickel's "Prolog Technology Theorem Prover", and providing both easy implementation and efficient execution. Third, they permit specification of *computationally useful* fragments of higher order equality theories, by allowing selective omission of equality axioms. Finally, in this approach innermost and outermost reduction, as well as many combinations of these, are obtained by merely rearranging the clauses that express equality axioms.

A number of higher order term rewriting systems proposed recently rely on simplified λ-term unification algorithms, implementing fragments of equality theories for λ-terms. Our approach provides both a

[*]Supported by Natural Sciences and Engineering Research Council of Canada.
[†]Supported by National Science Foundation grant IRI-8917907.
[‡]Supported by Natural Sciences and Engineering Research Council of Canada.

way to implement such schemes, and an alternative equality theory when the fragment required is limited (as is often the case).

1 Introduction

Stimulated by the success of first-order term rewriting systems (TRS), interest turned to the investigation of the potential of higher order rewriting systems [5, 12]. These systems are suggested as vehicles for sophisticated rewriting applications, particularly those involving the notions of bound variables or abstraction. Such applications include systems for manipulating logical formulas with quantifiers and programs with variable declarations. Thus they have been used for specifications of logics, theorem provers, program transformations, and control schemes for rewriting systems (tactics and normalization strategies) [5, 6].

As λ-conversion is nontrivial to implement correctly, there is a benefit in expressiveness and elegance provided by systems that support rewriting with λ terms and include λ-unification [8] as a built-in facility. Also the "combinatory" flexibility of higher order systems makes it possible to develop sophisticated systems with only a few primitives and composition. For example, in the specification of theorem provers a language of rewrite plans can be made available with a small set of higher order functions [5, 14].

However, TRS based on the λ-calculus have serious practical drawbacks. First, it is well-known that systems that manipulate λ-terms directly carry an overhead, and tend to be slow. It is challenging to develop compilers that skirt the inherent complexity caused by λ-terms being equivalence classes, and minimize the runtime overhead of λ-conversion. Second, general λ-unification typically requires intelligent use of types, and is known to be both nondeterministic and undecidable. Overall, the overhead is considerable. For example, while it must be stressed that λProlog 2.7 was not written with speed foremost in mind, it yields under 10 LIPS (successful unifications per second) on the examples provided with the system using SICStus Prolog 0.7 on a Sun ELC — three orders of magnitude slower than the Prolog system itself.

Recognizing these drawbacks, Miller has recently proposed a simplified unification procedure implementing only a pattern-oriented fragment of the equational theory of λ-terms [10] in the simplified L_λ system. This approach has attracted considerable interest [6, 13], since several existing systems employ a slower, more general λ-unification procedure. The sample programs in [10] stress that useful applications of λ-unification can be implemented with a simpler unification problem and greater reliance on universally quantified implication to express useful meta-level goals about substitutions at the object level.

Commenting on the motivations for this proposal, Miller remarks:

> Both the Isabelle theorem prover and λProlog contain simply typed λ-terms, $\beta\eta$-conversion, and quantification of variables at all functional orders. These systems have been used to specify and implement a large number of meta-programming tasks, including theorem proving, type checking, and program transformation, interpretation, and compilation. An examination of the structure of those specifications and implementations revealed two interesting facts. First, free or "logic" variables of functional type were often applied only to distinct bound λ-variables. For eample, the free functional variable M may appear in the following context:
>
> $$\lambda x \ldots \lambda y \ldots (M\, y x) \ldots.$$
>
> When such free variables are instantiated, the only new β-redexes that arise are those involving distinct λ-bound variables. For example, if M above is instantiated with a λ-term, say $\lambda u \lambda v \cdot t$, the only new β-redex formed is $((\lambda u \lambda v \cdot t) y x)$. This is reduced to normal form simply by renaming in t the variables u and v to y and x — a very simple computation. Second, in the cases where free variables of functional type were applied to general terms, meta-level β-reduction was invoked simply to perform object-level substitution. For example, an object-level universal quantifier can be specified using the symbol all of second-order type $(term \rightarrow formula) \rightarrow formula$. The

binary predicate that relates a universally quantified formula to the result of instantiating it with some term can be coded simply by the following meta-level axiom

$$\forall B\ \forall T\ (instan\,(all\,B)(B\,T))$$

where B and T are typed as $term \rightarrow formula$ and $term$, respectively. At the object-level, this predicate relates the formulas $\forall x \cdot A$ and $[x \mapsto T]B$: object-level substitution is expressed at the meta-level using β-conversion.

The logic L_λ is designed to permit the first kind of β-redex but not the second. As a result, implementations of this logic can make use of a very simple kind of unification. Although object-level substitution is not automatically available, it can be specified naturally as an L_λ program. ... Thus, L_λ requires that some of the functionality of β-conversion be moved from the term level to the logic level. The result can be more complex logic programs but with simpler unification problems. This seems like a trade-off worth investigating.

— Dale Miller, section 2.1 of [10].

The enormous power of λ-unification is sometimes unused even by the applications proposed for higher order rewriting. As Miller points out, its main use to date seems to be in pattern matching and β-normalization, both simple operations. A trace of the programs distributed with λProlog 2.7 showed that general λ-term unification, with flexible-flexible pairs, arose in only a few programs (type inference and program transformation). For example, the program that implements theorem-proving tacticals (an application proposed by Felty for higher order rewriting [4, 5]) generated no flexible-flexible pairs, and one flexible-rigid pair.

A natural question is therefore: *can we provide only that part of the equality theory needed by these higher order applications?* Doing so will presumably speed up such applications considerably, and should have the beneficial side-effect that the unifier (or whatever other interface to the equality theory we choose) will be easier to understand and predict.

We argue that it is possible to provide fragments of equational theories in an elegant and relatively efficient way, using a first-order approach that is implementable as a Prolog Technology Term Rewriter.

2 A Prolog Technology Term Rewriter

In this paper, we describe an approach to term rewriting in the spirit of Stickel's Prolog Technology Theorem Prover [17]: exploiting the combination of high level and efficient implementation of the Prolog programming language. Similarly, we show that the logical basis of Prolog makes it possible to implement term rewriting in a way that provides a remarkable combination of effortlessness and run-time efficiency. Moreover, our method is not mere Prolog hacking: it is based on an uncovering of the logical foundations of term rewriting, an activity that is of interest independently of any practical motivation.

Although reduction, the basic step in term rewriting, is usually defined as a primitive operation, it can also be obtained by logical inference from an equation representing the rule and one of the axioms of equality. In this paper we adopt this approach for the practical reason that it allows us to avoid duplicating the equivalent of certain aspects of Prolog implementation. It also allows us to obtain different behaviours of term rewriters by choosing among logically equivalent ways of expressing the axioms of logic. In this way it is easy to see the semantic relations between the different versions.

2.1 The Role of Equality in Logic Programming

To obtain the above advantages, we use logic programming as the framework: the language of clauses and resolution as inference rule. Specifically, we assume a logic program consisting of positive Horn clauses to be activated by a query in the form of a negative clause. Inference is restricted to SLD resolution: resolution of the most recently obtained negative clause (and initially the query) with a positive clause, yielding the next negative clause. Resolution is the only inference step, and syntactic unification is the only form of equality used. Of the axioms of equality, only substitutivity and reflexivity are embodied in unification.

The programmer is of course free to include other equality axioms. At a superficial level this is facilitated by the fact that in their usual formulation, the axioms of equality are substantially in the form of positive Horn clauses. But it is rarely wise to do so, as the search space for SLD resolution is rendered intractable by the inclusion of, for example, transitivity. Thus, logic programming has only succeeded because programmers have learned to avoid the use of the equality relation: for example by saying `sum(X,Y,Z)` instead of `X+Y=Z`.

2.2 The Axioms of Equality

As we will often be referring to axioms of equality, it is important to list them for future reference. We use the clausal form of logic, in which all variables are implicitly universally quantified. The following set Eq of clauses define equality.

Reflexivity	$x = x.$
Symmetry	$x = y \leftarrow y = x.$
Transitivity	$x = z \leftarrow x = y,\ y = z.$
Function substitutivity	$f(x_1, \ldots, x_i, \ldots, x_n) = f(x_1, \ldots, y_i, \ldots, x_n) \leftarrow x_i = y_i.$

The substitutivity axioms range over all argument positions of all n-ary function symbols f.

The method we follow is to rewrite terms by proving the initial term equal to the final term in the reduction sequence. If we base the proof on the standard axioms given above, the required inference is a complex task. Instead we will formulate mini-theories of equality that are justified by being a logical consequence of Eq. These mini-theories define subsets of the equality relation (regarding these relations as sets of pairs). The subsets are large enough to contain the desired results of rewriting and are defined by axioms of a form that qualifies as a logic program. As a result the efficient Prolog theorem prover, being based on SLD resolution, can obtain the result of rewriting at a cost in time and memory that makes this method of practical interest.

To obtain our minitheories, we introduce of different predicate symbols eq_i for $i = 1, 2, \ldots$ that, though different, all have as same meaning the equality symbol used above. Thus we add to Eq the clauses $\mathrm{eq}_i(x, y) \Leftrightarrow (x = y)$ for whatever values of i we need.

We then collect theorems about the eq_i from the axioms in Eq. These theorems can then be used as sole definition of eq_i, that is, not using the axiom $\mathrm{eq}_i(x, y) \Leftrightarrow (x = y)$. In this way the theorems typically define a subrelation of equality; that is, they constitute a mini-theory as referred to above.

2.3 Implementation of Prolog Technology Term Rewriting

The discussion above explains how the single rewrite step and its transitive closure can be represented as clauses suitable for execution as a Prolog program. According to this method (let us call this the "Basic Method"), we translate any set of rewrite rules to a logic program containing the following clauses:

- The *rule reference clause*, which says that the two sides of a rewrite rule are in the eq_1 relation.
- For each function symbol of positive arity, the corresponding clause expressing substitutivity.

- The clause defining eq_2 as the reflexive transitive closure of eq_1.

This is the Basic Method. It does not specify the order of the clauses.

For example, we can define the rules

```
twice : F : X  =>  F : (F : X).
succ : X       =>  s(X).
```

and then specialize our equality implementation for the binary function symbol : to obtain:

```
eq(X,Y) :- eq2(X,Y).

eq2(X,Z) :- eq1(X,Y), eq2(Y,Z).         % transitivity
eq2(X,X).                               % reflexivity

eq1((X1:X2),(Y1:X2)) :- eq1(X1,Y1).     % substitutivity for ':'
eq1((X1:X2),(X1:Y2)) :- eq1(X2,Y2).     % substitutivity for ':'
eq1(X,Y) :- (X => Y).                   % rule-reference clause
```

The query

```
?- eq( twice:twice:twice:succ:0, X ).
```

yields the answer

```
X = s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(0)))))))))))))))).
```

This example in fact shows what we are mainly interested in: applicative definitions.

Performance results for applicative definitions like these, showing the surprising result that even naive Prolog implementations such as this run faster than interpreted Lisp programs using the equivalent **LAMBDA** definitions, are presented in [2, 3].

2.4 Implementing Fragments of Equality

In the Basic Method we have not considered how clauses should be ordered. This has determines whether the rewriting is done innermost or outermost, an important choice. Innermost has the advantage of speed (even on small examples a speed-up of two orders of magnitude can sometimes be observed). It has the disadvantage that a rewriting possible under outermost does not terminate.

A substitutivity clause pertains to a certain function symbol and to a certain argument place. If such a clause occurs before (after)

```
eq1(X,Y) :- (X => Y).                   % rule-reference clause
```

then the entire program will cause innermost (outermost) rewriting with respect to that function symbol and that argument place. This possibility of mixing outermost and innermost rewriting may be novel and is certainly worth pursuing. However, in this paper we will say no more about it as it does not seem to affect the higher order aspects of term-rewriting.

Often the substitutivity axioms are not useful, particularly when we are interested in outermost (lazy) rewriting. For example, with the definition of `plus`

```
plus(X,0)      =>  X.
plus(X,s(Y))   =>  s(plus(X,Y)).
```

substitutivity on the first argument will *never* help in successful unification. We can see this easily, since the rules are *linear* (i.e., no variable appears more than once in the left of any rule) and the first argument of plus is always a variable. Use of substitutivity on this argument in any proof will have the effect of evaluating the argument, even though the value of the argument may not be needed later.

A more striking example comes from considering the standard rules

```
if(true, X,Y)  =>  X.
if(false,X,Y)  =>  Y.
```

in which it is clear that evaluation of the second or third argument is not wanted. When the leftmost innermost strategy is selected for reasons of speed, and exception is wisely made for if-then-else, avoiding evaluation of the arguments in this case.

For instance, with the example above we obtain the following equational theory:

```
eq2(X,Z) :- eq1(X,Y), eq2(Y,Z).                       % transitivity
eq2(X,X).                                             % reflexivity

eq1(X,Y) :- (X => Y).                                 % subst. outermost
eq1(plus(X1,X2),plus(X1,Y2))    :- eq1(X2,Y2).  % subst. 2nd argument of plus
eq1(if(X1,X2,X3),if(Y1,X2,X3)) :- eq1(X1,Y1).  % subst. 1st argument of if
```

2.5 Implementation method

Prolog Technology Term Rewriting is implemented in three stages:

- Choice of a computationally useful fragment of equality, as explained above. The logic program consists of rewrite rules plus selected consequences of the equality axioms. This logic program is executable under Prolog and gives the same results as term rewriting. It is, however, susceptible to optimization as described in the next two stages.

- In the first optimization stage, the modified axioms of equality are optimized by partial evaluation. That is, the clauses defining the equality relation shown above are "unfolded" [16] using the rules provided.

- The results of this stage are then "relationalized" by converting the applicative equational definitions to predicate form. The definition apply(x, y, z) — eq($x : y, z$) is introduced and then partially evaluated. The resulting "apply" predicate executes more rapidly with most Prolog systems as it requires fewer terms to be constructed.

  Actually, this relationalization step is the result of a "fold" transformation in our partial evaluator: partial evaluation of the specialized query eq((X:Y),Z) generates a three-argument apply(X,Y,Z) predicate, and recursive calls of the form eq((U:V),W) produced by this partial evaluation are folded to apply(U,V,W). This folding technique is described in [16].

Such an implementation can be easily extended, as Stickel shows [17], to include complete proof search (by iterative deepening), unification with the occur check, and a complete form of negation. Using Prolog does not imply incompleteness.

For example, with the following equality theory, theorems of the form eq(X,Y,L) are provable only when there is a rewriting ¿from X to Y of length at most L, i.e., with at most L applications of "=>" rules. Every proof of eq1 ultimately results in the application of exactly one "=>" rule.

```
eq(X,Y) :- eq(X,Y,_).

eq(X,Y,L) :- peano_number(L), eq2(X,Y,L,_).  % search bound is L

eq2(X,Z,s(L1),L) :- eq1(X,Y), eq2(Y,Z,L1,L). % decremented bound
eq2(X,X,L,L).

eq1((X1:X2),(Y1:X2)) :- eq1(X1,Y1).      % substitutivity for ':'
eq1((X1:X2),(X1:Y2)) :- eq1(X2,Y2).      % substitutivity for ':'
eq1(X,Y) :- (X => Y).                    % rule-reference clause

peano_number(0).
peano_number(s(N)) :- peano_number(N).
```

The length values L are represented here as Peano numbers both for elegance, and also for speed (!), since Prolog implementations are optimized for such unification. Since peano_number generates all possible Peano numbers in ascending order, this equality theory performs depth-first iterative deepening search, a complete proof search strategy.

3 Applicative Term Rewriting

Applicative Term Rewriting Systems permit rules of the form $f(v_1,\ldots,v_n) : x \quad = \quad E$ where E is a first-order term whose only function symbols are the application operator ":" and function symbols of other rules, and whose only free variables are in the set $\{x, v_1,\ldots,v_n\}$. By "capturing" all these free variables, the term $f(v_1,\ldots,v_n)$ becomes a kind of combinator — a λ-term with no free variables — but treats the free variables $v_1,\ldots,v_n$ as parameters. This term serves the role of a *supercombinator* [15], and we will call it such here. Combinators can be expressed as curried supercombinators. For example, the familiar combinators S, K, and I are easily expressible:

```
i : X        =>  X.
k : X        =>  k1(X).
k1(X) : Y    =>  X.
s : X        =>  s1(X).
s1(X) : Y    =>  s2(X,Y).
s2(X,Y) : Z  =>  (X:Z) : (Y:Z).
```

Applicative Term Rewriting Systems have been studied in the larger setting of combinatory logic, and are subsumed by Klop's *combinatory reduction systems* (CRS) [9]. CRS permit use of λ-terms in both the left- and right-hand sides of rewrite rules. These λ-terms can be used as higher order patterns, which we will discuss below.

Of course, there is a direct relationship between λ-terms and combinators. The relationship is provided by the λ-*lifting* algorithm used in translating functional programs to applicative form (combinator form) [15]. Expressions containing free variables are translated to supercombinators by reintroducing abstractions to bind the free variables. Here we use the first-order term representation to represent these reintroduced abstractions that capture free variables.

Any λ-term can be converted to a supercombinator applicative form with the following recursive algorithm, a λ-lifting algorithm used in compiling functional programs into supercombinators:

$$\begin{array}{lcl} \mathit{lift}(x) & = & x \quad \text{if } x \text{ is a variable (or constant)} \\ \mathit{lift}(MN) & = & \mathit{lift}(M) : \mathit{lift}(N) \\ \mathit{lift}(\lambda x.E) & = & g(v_1, \ldots, v_n) \\ & & \text{if } v_1, \ldots, v_n \text{ are the free variables of } E, \text{ and} \\ & & g \text{ is a new function symbol defined by } g(v_1, \ldots, v_n) : x \; = \; \mathit{lift}(E). \end{array}$$

This translation introduces new supercombinator definitions, ultimately yielding an expression that is conditioned on a set of equations.

Note that a consequence of this algorithm is that

$$\mathit{lift}(\lambda x_1 \ldots \lambda x_m.E) \; = \; g_0(v_1, \ldots, v_n)$$

where we obtain as side effects the (curried) applicative definitions:

$$\begin{array}{rcl} g_0(v_1, \ldots, v_n) : x_1 & = & g_1(v_1, \ldots, v_n, x_1) \\ g_1(v_1, \ldots, v_n, x_1) : x_2 & = & g_2(v_1, \ldots, v_n, x_1, x_2) \\ \vdots & \vdots & \vdots \\ g_{m-1}(v_1, \ldots, v_n, x_1, \ldots, x_{m-1}) : x_m & = & \mathit{lift}(E). \end{array}$$

These definitions introduce new supercombinators $g_j(v_1, \ldots, v_n, x_1, \ldots, x_j)$, for $0 \leq j \leq m-1$. The SKI-combinator rules above are an example of this translation.

Thus there is a direct translation from λ-terms to applicative supercombinator expressions. However, the approach rests firmly in first-order logic. We axiomatize only the application operator ":", and that part of equality that is needed. For higher order term rewriting applications that do little more than β-normalization and argument pattern matching, the applicative term rewriting approach described here is probably sufficient.

4 Sample Application: Higher Order Pattern Matching

We feel the foregoing approach is useful in addressing new problems. To illustrate this we will address its application to a higher order problem. We quickly review the definition of higher order patterns as in [13] and [5], and describe a unification procedure for (both untyped and simply typed) λ-terms with these patterns [12, 13]. We then explore a fragment of this kind of pattern matching that appears to be useful: monadic patterns. We show how unification for these patterns can be implemented using the Prolog Technology Term Rewriter approach.

Throughout this section, by a λ-term (denoted r, s, t) we mean a term constructed from λ-bound variables (denoted x, y, z); constants (denoted c, d, e, f, g); atoms — variables or constants — (denoted a or b); free variables (denoted F, G, H); applications $(s : t)$; and abstractions $(\lambda x.t)$. The normal convention of left associativity of application (so $r : s : t = (r : s) : t$) is followed. We will omit discussion of types here.

4.1 Higher Order Pattern Matching

A *(higher order) pattern* is a λ-term in β-normal form in which every occurrence, if any, of a free variable H appears in a subterm $(H : x_1 : \ldots : x_n)$ $(n \geq 0)$, where $x_1, \ldots, x_n$ are (η-convertible to) distinct λ-bound variables. Examples of higher order patterns include $\lambda x.x$, H, $\lambda x.(H : x)$, $\lambda x.(H : (\lambda x.(y : x)))$, and $\lambda x.\lambda y.(H : y : x)$. On the other hand, terms such as $\lambda x.(H : x) : x$ and $\lambda x.(H : (H : x))$ are not patterns.

Uses for higher order patterns have been described by Miller [10], following the λProlog work described earlier, and in work on higher order term rewriting systems such as the CRS of Klop mentioned above [9], and more recent work by Felty [5] and by Nipkow [12]. For example, a *higher order rewrite rule* is a pair $l \Rightarrow r$ such that l and r are λ-terms of the same primitive type. l is a pattern but not a free variable, and all free variables in l also occur in r. A *higher order rewrite system* is a finite set of these rewrite rules. The rewrite step relation $s \rightarrow t$ for a higher order rewrite system may be defined for terms s and t if there exist terms u, L, R such that: $s =_{\beta\eta} (u : L)$, $t =_{\beta\eta} (u : R)$; $L = \lambda x_1 \ldots \lambda x_n.(l\theta)$, $R = \lambda x_1 \ldots \lambda x_n.(r\theta)$; there is a rule $l \Rightarrow r$; θ is a substitution that maps free variables of l to terms whose only free variables are $x_1, \ldots, x_n$, and $x_1, \ldots, x_n$ do not occur as free variables in any of s, t, l, r.

Nipkow [13] gives a clear presentation of a λ-unification algorithm for λ-terms with higher order patterns, using transformations. Let $\overline{v_m}$ denote the sequence $v_1, \ldots, v_m$, x, y, z denote variables, and s, t denote terms. Then the unification process repeatedly transforms a list of disagreement pairs and substitution into a new list of disagreement pairs and substitution. The leftmost disagreement pair $s =^? t$ is extracted from the input list and replaced by a list (and substitution) obtained from the appropriate case among the following:

$$\begin{array}{rcl}
\lambda x.s =^? \lambda x.t & \rightarrow & \langle [s =^? t], \{\} \rangle \\
a(\overline{s_n}) =^? a(\overline{t_n}) & \rightarrow & \langle [s_1 =^? t_1, \ldots, s_n =^? t_n], \{\} \rangle \\
F(\overline{x_m}) =^? a(\overline{t_n}) & \rightarrow & \langle [H_1(\overline{x_m}) =^? t_1, \ldots, H_n(\overline{x_m}) =^? t_n], \{F \mapsto \lambda \overline{x_m}.a(\overline{H_n(\overline{x_m})})\} \rangle \\
& & \text{where } a \text{ is a constant or a variable in } \overline{x_m} \\
& & F \text{ is not a free variable in } \overline{s_n}, \\
& & \text{and } a(\overline{H_n(\overline{x_m})}) = a(H_1(\overline{x_m}), \ldots, H_n(\overline{x_m})). \\
F(\overline{x_m}) =^? F(\overline{y_n}) & \rightarrow & \langle [], \{F \mapsto \lambda \overline{x_m}.a(H(\overline{z_p}))\} \rangle \\
& & \text{where } \{\overline{z_p}\} = \{x_i \mid x_i = y_i\}. \\
F(\overline{x_m}) =^? G(\overline{y_n}) & \rightarrow & \langle [], \{F \mapsto \lambda \overline{x_m}.H(\overline{z_p}), G \mapsto \lambda \overline{y_n}.H(\overline{z_p})\} \rangle \\
& & \text{where } F \neq G \text{ and } \{\overline{z_p}\} = \{\overline{x_m}\} \cap \{\overline{y_n}\}.
\end{array}$$

Miller demonstrated in [10] that it is decidable whether two patterns are unifiable, and if they are, then there is a most general unifier.

4.2 A Useful Fragment: Monadic Higher Order Patterns

An interesting special case is to consider *monadic* higher order patterns, i.e., patterns of a single bound λ-variable, of the form $(F : x)$ where F is the free variable. Monadic patterns have several attractive features.

First, monadic patterns are useful in practice. Many programs appear to be naturally expressed with only this limited amount of higher order power. For example, in [10] Miller notes that for the commonly-encountered logic program

$$\begin{array}{ll} \forall x \forall l \forall k \forall m & (append{:}l{:}k{:}m \supset append{:}(cons{:}x{:}l){:}k{:}(cons{:}x{:}m)) \\ \forall k & (append{:}nil{:}k{:}k) \end{array}$$

the result of the query

$$\forall y(append{:}(cons{:}a{:}(cons{:}b{:}nil)){:}y{:}(H{:}y))$$

with the monadic higher order pattern H is that H is bound to the value $\lambda w\ .\ (cons\ a\ (cons\ b\ w))$.

Second, monadic patterns are simple and easy to reason about in a way that polyadic patterns are not. If one believes that unification should be an intuitive operation that is useful in constructing and reasoning about programs, such simplicity is important.

Third, monadic higher order pattern unification is efficient in a way that polyadic pattern unification is not. The implementation in ML provided by Nipkow [13] demonstrates the problem clearly, as the pattern subterm

$$F(\overline{x_m}) \equiv ((\ldots((F x_1) : x_2) : \ldots) : x_m)$$

must be isolated within a term being unified, and matched with the corresponding subterm $a(\overline{t_n})$. Dealing with the polyadic case introduces complexity that slows down the unifier.

4.3 Prolog Technology Term Rewriting Implementation

Although the monadic case of higher order pattern matching is an intriguing and promising special case, further investigation is needed to firmly establish the rewards and penalties of being restricted to this special case. Pending the results of such research, we have stayed on the safe side and implemented higher order patterns in their full generality.

We first implemented a unifier for higher order monadic patterns without regard for any particular method. This version was basically the result of Prolog hacking. In the next version the method of this paper was partially followed. The version displayed in the appendix differs in being a more complete application of the method and by lifting the restriction to monadic patterns.

In both the second and third versions we found that our method provided worthwhile simplifications in the code. The third version shows that the generalization to polyadic patterns was achieved at a mild penalty in performance. This suggests that the first and second versions did not fully utilize the performance potential of the restriction to the monadic case. How to obtain a fuller utilization and how restrictive in practice monadic patterns are should be clarified by further research.

5 Conclusions

We have developed Applicative Term Rewriting Systems, a simple kind of TRS that has potential in practical higher order term rewriting applications. A benefit of our approach is that it can build on Prolog technology, permitting efficient implementation with a minimum of implementation effort. Furthermore, a variety of rewriting strategies (bottom-up, leftmost-outermost, lazy) are easily arranged within the framework of SLD resolution, and hence also easily implemented with Prolog.

Applicative TRS use supercombinators, which have been exploited in architectures for functional programming. Combinator-based implementations reduce the problem of evaluating expressions to the evaluating purely applicative forms, with reduced overhead in handling the variable bindings in λ-conversion. The result executes quickly with Prolog, comparing favorably with standard LISP systems [2, 3].

Finally, the Prolog Technology approach described here permits selective implementation of fragments of the equality theory. Although using λ-calculus-based approaches [4, 5, 9, 12] for higher order term rewriting has undeniable benefits -- they are well-defined, logically sound, and extremely expressive — the price in peformance is considerable. The approach here is consistent with the growing body of research in identifying useful fragments of λ-term equality, and provides the speed of Prolog technology. Our results suggest a design method where a suitable program, such as a verifier or a code optimizer, is first specified in higher order logic. In this way maximum expressiveness is obtained. At the same time such a specification can be run; though slow, this is often a good way to debug the specification. When one decides to go ahead with implementation, a succession of steps is available. First the specification is rewritten, if necessary, to contain patterns only. Our experience suggests that this gives already a respectable performance. As a next stage of reformulation it may be not too difficult to restrict to monadic patterns. Further optimization beyond Prolog, if necessary, is less risky when a compact and running Prolog program can be used as specification.

References

[1] H.P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, 1984.

[2] M.H.M. Cheng, M.H. van Emden, B.E. Richards, "On Warren's Method for Functional Programming in Logic", *Proc. 7th International Conf. on Logic Programming*, Jerusalem, MIT Press, 546–560, 1990.

[3] M.H.M. Cheng, K. Yukawa, "AP: An Assertional Programming System", in *Advances in Logic Programming and Automated Reasoning*, volume 1, R.W. Wilkerson (ed.), Ablex Pub., pp. 120-134, 1992.

[4] A. Felty, D. Miller, "Specifying Theorem Provers in a Higher Order Logic Programming Language", *Proc. CADE-9*, 61–80, Springer-Verlag LNCS #310, 1988.

[5] A. Felty, "A Logic Programming Approach to Implementing Higher Order Term Rewriting", in: *Extensions of Logic Programming, Proceedings of the 2nd International Workshop ELP'91*, L.H. Eriksson, L. Hallnäs, P. Schroeder-Heister (eds.), Springer-Verlag, LNAI #596, 135–161, 1991.

[6] J. Hannan, "Implementing λ-Calculus Reduction Strategies in Extended Logic Programming Languages", in: *Extensions of Logic Programming, Proceedings of the 2nd International Workshop ELP'91*, L.H. Eriksson, L. Hallnäs, P. Schroeder-Heister (eds.), Springer-Verlag, LNAI #596, 194–219, 1991.

[7] P. Henderson, *Functional Programming: Application and Implementation*, Prentice-Hall, 1980.

[8] G. Huet, "A unification algorithm for typed λ-calculus", *Theoretical Computer Science* **1**, 27–57, 1975.

[9] J.W. Klop, *Combinatory Reduction Systems*, Amsterdam: Mathematical Centre tract #127, 1980.

[10] D. Miller, "A Logic Programming Language with Lambda-Abstraction, Function Variables, and Simple Unification", *J. Logic and Computation* **1**:4, 497–536, September 1991.

[11] G. Nadathur, D. Miller, "Higher Order Horn Clauses", *J. ACM* **37**:4, 777–814, October 1990.

[12] T. Nipkow, "Higher Order Critical Pairs", *Proc. 6th IEEE Symp. on Logic in Computer Science*, Amsterdam, 342–349, July 1991.

[13] T. Nipkow, "Functional Unification of Higher Order Patterns", *Proc. LICS '93*, Montreal, 1993.

[14] L.C. Paulson, *Logic and Computation*, Chapter 9: Rewriting and Simplification, Cambridge University Press, 1987.

[15] S.L. Peyton Jones, *The Implementation of Functional Programming Languages*, Prentice-Hall, 1987.

[16] D. Sahlin, "An Automatic Partial Evaluator for Full Prolog", Ph.D. dissertation, Dept. of Telecommunications and Computer Systems, Royal Institute of Technology (KTH), S-100 44 Stockholm Sweden. Also available from: Swedish Institute of Computer Science, Box 1263, S-164 28 Kista, Sweden.

[17] M. Stickel, "A Prolog technology theorem prover: a new exposition and implementation in Prolog", *Theoretical Computer Science* **104**, 109–128, 1992.

6 Appendix: Higher Order Pattern Unification in Prolog

```
%--------------------------------------------------------------------------
%
%  Higher-Order Pattern Unification
%
%  This equational theory solves equations of the form S=T (if this is
%  possible), where both S and T can be higher-order patterns. Such
%  patterns are lambda-terms that can contain free variables.
%  A higher-order pattern is a (generalized) lambda-term in beta-normal form
%  such that every occurrence of a free variable F is in a subterm
%                         (F x_1 ... x_n)
%  where (x_1 ... x_n) is eta-equivalent to a list of distinct bound
%  variables. See reference [13].
%
%  Lambda-terms here are represented by logical ground terms as follows:
%  the bound variable (x) is represented by the atom x [one of u,v,w,x,y,z];
%  the constant (c) is represented by the atom c; the free variable (F) is
%  represented by the logical variable  (F); (M N) is represented by (S:T)
%  where S,T are the representations of M,N; (lambda x.M) is represented by
%  (x\T) where T is the representation of M.
%
%  The choice of representing free variables by logical variables makes
%  this implementation more useful in implementing real systems with Prolog
%  since the distinction between free and bound variables is a natural one,
%  and this approach permits us to avoid dealing with substitutions directly.
%  However, it complicates the implementation slightly, as it mixes the
%  meta-level and object-level concept of variable.
%
%  We assume throughout that the only use of logical variables in terms
%  arises in flexible terms (F:x1:...:xN), and that the lambda-terms are
%  well-formed, so that in any term (S\T), S will be a bound variable.
%
%-----------------------------------------------------------------------------
%
:- op(1150,xfx,(=>)).
:- op(1000,xfy,(&)).
:- op(600,yfx,(:)).
:- op(500,xfy,(\)).

hop_unification(X,Y) :- eq((X=Y), true).

eq(X,Y)  :- eq2(X,Y).

eq2(X,Z) :- eq1_(X,Y), !, eq2(Y,Z).
eq2(X,X).

eq1(X,Y) :- (X => Y).
eq1((X1&Y),(X2&Y)) :- eq1_(X1,X2).
eq1((X1=Y),(X2=Y)) :- eq1_(X1,X2).
eq1((X=Y1),(X=Y2)) :- eq1_(Y1,Y2).
eq1((X\Y1),(X\Y2)) :- eq1_(Y1,Y2).
eq1((X1:Y),(X2:Y)) :- eq1_(X1,X2).
eq1((X:Y1),(X:Y2)) :- eq1_(Y1,Y2).
% Lazy unification omits this rule: it matches only the weak head normal
% form of the two arguments being unified.
% Commenting this rule out increases the speed of the unifier somewhat:
% see the timings at the end of this file.

eq1_(X,Y) :- nonfreevar(X), eq1(X,Y).

(S\T)     =>  R  :-  eta_reduct((S\T),R).
(S:T)     =>  R  :-  beta_reduct((S:T),R).
(S=T)     =>  R  :-  term_type(S,St), term_type(T,Tt), match(St,Tt,S,T,R).
(V->T)    =>  R  :-  variable_binding(V,T,R).

variable_binding(V,T,true ) :- freevar(V), !, beta_eta_normal_form(T,V).
variable_binding(V,T,(V=S)) :- beta_eta_normal_form(T,S).

(true&R)  =>  R.

match(flexible, flexible, S,T, R)  :-  match_flex_flex(S,T,R).
match(flexible, rigid,    S,T, R)  :-  match_rigid_flex(T,S,R).
match(rigid,    flexible, S,T, R)  :-  match_rigid_flex(S,T,R).
match(rigid,    rigid,    S,T, R)  :-  match_rigid_rigid(S,T,R).

match(lambda,   lambda,   S,T, R)  :-  alpha_conv((S=T),R).
match(lambda,   flexible, S,T, R)  :-  eta_expand((S=T),R).
match(lambda,   rigid,    S,T, R)  :-  eta_expand((S=T),R).
match(flexible, lambda,   S,T, R)  :-  eta_expand((T=S),R).
match(rigid,    lambda,   S,T, R)  :-  eta_expand((T=S),R).

match_flex_flex( T,     F,     true) :- freevar(F), !, match_flex_freevar(T,F).
match_flex_flex( F,     T,     true) :- freevar(F), !, match_flex_freevar(T,F).
match_flex_flex( S,     T,     R   ) :- bind_flex_flex(S,T,R).

match_flex_freevar(T,F) :- freevar(T), !, F=T.
match_flex_freevar(T,F) :- term_without_freevar(T,F), F=T.


match_rigid_flex( T,  F,     (F->T) ) :- freevar(F), !.
match_rigid_flex( X, (F:X), true   ) :- freevar(F), !, F=(X\X).
match_rigid_flex( Y, (F:X), true   ) :- constant(Y), freevar(F), !, F=(X\Y).
match_rigid_flex( S,  T,     R      ) :- bind_rigid_flex(S,T,R).


match_rigid_rigid( S,       S,        true       ) :- atomic(S), !.
match_rigid_rigid((S1:T1),(S2:T2), (R & T1=T2)) :- match_rigid_rigid(S1,S2,R).

%--------------------------------------------------------------------------
%       Lambda conversion and reduction
%--------------------------------------------------------------------------

alpha_conv((X\S=X\T),S=T)    :- !.
alpha_conv((X\S=Y\T),SV=TV) :-
      newboundvar(V), subst(S,SV,X,V), subst(T,TV,Y,V).

eta_expand((X\S=T),(S=T:X))  :- term_without_loose_var(T,X), !.
eta_expand((X\S=T),(SV=T:V)) :- newboundvar(V), subst(S,SV,X,V).

eta_reduct(X\SX,S)  :- nonfreevar(SX), SX=(S:X), term_without_loose_var(S,X).

beta_reduct(XS:Y,T) :- nonfreevar(XS), XS=(X\S), subst(S,T,X,Y).

subst( S,       S,      X,Y) :- X == Y, !.
subst( S,       Y,      X,Y) :- S == X, !.
subst( F,       F,      _,_) :- freevar(F), !.
subst((Z\SX), (Z\SX), X,_) :- X == Z, !.
subst((Z\SX), (Z\SY), X,Y) :- !, subst(SX,SY,X,Y).
subst((SX:TX),(SY:TY),X,Y) :- !, subst(SX,SY,X,Y), subst(TX,TY,X,Y).
subst( S,       S,      _,_).

beta_eta_normal_form(X,Z) :- beta_eta1_(X,Y), !, beta_eta_normal_form(Y,Z).
beta_eta_normal_form(X,X).

beta_eta1_(X,Y) :- nonfreevar(X), beta_eta1(X,Y).

beta_eta1(S,T) :- beta_reduct(S,T).
beta_eta1(S,T) :- eta_reduct(S,T).
beta_eta1(X1:Y,X2:Y) :- beta_eta1_(X1,X2).
beta_eta1(X:Y1,X:Y2) :- beta_eta1_(Y1,Y2).
beta_eta1(X\Y1,X\Y2) :- beta_eta1_(Y1,Y2).

beta_normal_form(X,Z) :- beta1_(X,Y), !, beta_normal_form(Y,Z).
beta_normal_form(X,X).
```

```
beta1_(X,Y) :- nonfreevar(X), beta1(X,Y).

beta1(S,T) :- beta_reduct(S,T).
beta1(X1:Y,X2:Y) :- beta1_(X1,X2).
beta1(X:Y1,X:Y2) :- beta1_(Y1,Y2).
beta1(X\Y1,X\Y2) :- beta1_(Y1,Y2).

weak_head_normal_form(X,Z) :-
      left_beta1_(X,Y), !, weak_head_normal_form(Y,Z).
weak_head_normal_form(X,X).

left_beta1_(X,Y) :- nonfreevar(X), left_beta1(X,Y).

left_beta1(S,T) :- beta_reduct(S,T).
left_beta1(X1:Y,X2:Y) :- left_beta1_(X1,X2).    % beta reduce rator only

%-----------------------------------------------------------------------------
%       General flexible term binding for HOP unification
%-----------------------------------------------------------------------------

% bind_rigid_flex(S,T,R) :-
%       if  S = a:S1:S2:...:SN  and  T = F:x1:...:xN  then
%           R is the binding F->(x1\x2\...\xN\AHsXs)
%           where AHsXs = a:(H1:x1:...:xN):(H2:x1:...:xN):...:(HN:x1:...:xN)
%           provided that F does not appear in S,
%           and that a is either a constant or one of the xi.
%
bind_rigid_flex(S,T,(ArgUnifications & (F->B))) :-
        flexible_binding_term(T,F,AHsXs,B),
        term_without_freevar(S,F),
        rigid_head(S,A),
        (constant(A) ; member_boundvar(A,T)),
        !,
        bind_rigid_flex(S,T,F,A,AHsXs,ArgUnifications).

bind_rigid_flex(S,T,F,AHsXs0,(AHsXs:HiXs),(R & HiXs=Si)) :-
        S=(Ss:Si), !,
        subst(T,HiXs,F,_Hi),
        bind_rigid_flex(Ss,T,F,AHsXs0,AHsXs,R).
bind_rigid_flex(_A,_,_,AHsXs,AHsXs,true).

% bind_flex_flex(S,T,R) :-
%    if  S = F:x1:x2:...:xN  and  T = F:y1:...:yN  then
%        R is the binding F->(x1\x2\...\xN\HZs)
%        where z1:...:zP = ordered intersect of x1:...:xN and y1:...:yN
%        and HZs = H:z1:...:zP.
%    if  S = F:x1:x2:...:xM  and  T = G:y1:...:yN  then
%        R is the binding F->(x1\x2\...\xM\HZs), G->(y1\x2\...\yN\HZs)
%        where z1:...:zP = unordered intersect of x1:...:xM and y1:...:yN
%        and HZs = H:z1:...:zP.
%
bind_flex_flex(S,T,R) :-
       flexible_binding_term(S,F,HZs,FB),
       flexible_binding_term(T,G,HZs,GB),
        bind_flex_flex(S,T,HZs,F,FB,G,GB, R).

bind_flex_flex(S,T,HZs,F,B,G,_,(F->B)) :-
        F==G, !,
        flex_arity(S,N),
        flex_arity(T,N),
        ordered_boundvar_term(S,T,_H,HZs).
bind_flex_flex(S,T,HZs,F,FB,G,GB,((F->FB) & (G->GB))) :-
        common_boundvar_term(S,T,_H,HZs).

% flexible_binding_term(T,F,H,B) :- T = F:x1:x2:...:xN   is a flexible term
%                                   and B = x1\x2\...\xN\H   (N can be zero).
%
flexible_binding_term(F,F,H,H) :- freevar(F), !.
flexible_binding_term((T:X),F,H,B) :- flexible_binding_term(T,F,(X\H),B).

% common_boundvar_term(S,T,H,HVs)  :-  HVs = H:x:y:...:z
%       where x,y,...,z are the bound variables common to S and T.
%
common_boundvar_term(F,_,H,H) :-
        freevar(F), !.
common_boundvar_term((S:X),T,H,HVs) :-
        member_boundvar(X,T), !,
        common_boundvar_term(S,T,H:X,HVs).
common_boundvar_term((S:_),T,H,HVs) :- common_boundvar_term(S,T,H,HVs).

member_boundvar(X,T) :- nonfreevar(T), T=(_:X), !.
member_boundvar(X,T) :- nonfreevar(T), T=(T1:_), member_boundvar(X,T1).

% ordered_boundvar_term(S,T,H,HVs) :-
%     HVs = H:x:y:...:z
%     where x,y,...,z is the sequence of bound variables common to S and T.
%
ordered_boundvar_term(F,_,H,H) :-
        freevar(F), !.
ordered_boundvar_term((S:X),(T:Y),H,HVs) :-
        X==Y, !,
        ordered_boundvar_term(S,T,H:X,HVs).
ordered_boundvar_term((S:_),(T:_),H,HVs) :-
        ordered_boundvar_term(S,T,H,HVs).

%--------------------------------------------------------------------------
%       Term representation and inspection
%--------------------------------------------------------------------------

term_without_freevar(G,  F) :- freevar(G),  !, F\==G.
term_without_freevar(X,  _) :- atomic(X), !.
term_without_freevar(_\T,F) :- term_without_freevar(T,F).
term_without_freevar(S:T,F) :- term_without_freevar(S,F),
                               term_without_freevar(T,F).

term_without_loose_var(S,X) :- boundvar(X), subst(S,T,X,anything), S==T.

term_type(T, rigid   ) :- rigid(T), !.
term_type(T, flexible) :- flexible(T), !.
term_type(T, lambda  ) :- lambda(T), !.


lambda(T) :- nonvar(T), T = (_\_),  \+ eta_reduct(T,_).

rigid(T) :- atomic(T), !.
rigid(T) :- nonvar(T), T=(T1:_), rigid(T1).

% rigid_head(T,A) succeeds if:
%       T = A:x1:x2:...:xN   (N can be zero)
rigid_head(A,A) :- atomic(A), !.
rigid_head((T:_),A) :- rigid_head(T,A).

flexible(F) :- var(F), !.
flexible(T) :- nonvar(T), T=(T1:X), flexible(T1), boundvar(X).
% each argument X can be either a bound variable or a generic constant

flex_arity(F,0) :- var(F), !.
flex_arity((F:_),s(N)) :- flex_arity(F,N).

nonfreevar(X) :- nonvar(X).

freevar(X) :- var(X).

% atomic(X) :- X is a constant or a bound variable.

nonconstant(X) :- \+ constant(X).

constant(X) :- atomic(X), non_existential_boundvar(X).

nonboundvar(X) :- \+ boundvar(X).

boundvar(X) :- nonvar(X), existential_boundvar(X), !.
boundvar(X) :- nonvar(X), universal_boundvar(X), !.
% Note: boundvar(X) :- X is already eta-converted to an atomic boundvar.

non_existential_boundvar(X) :- \+ existential_boundvar(X).

:- dynamic existential_boundvar/1.
existential_boundvar(u).  existential_boundvar(v).  existential_boundvar(w).
existential_boundvar(x).  existential_boundvar(y).  existential_boundvar(z).

newboundvar(X) :- gensym(x,X), assert(existential_boundvar(X)).

:- dynamic universal_boundvar/1.

newgeneric(C) :- gensym(c,C), assert(universal_boundvar(C)).

%-------------------------------------------------------------------------
%       New symbol generation
%-------------------------------------------------------------------------

:- dynamic recordedSymbolIndex/2.

gensym(Prefix,Symbol) :-
```

```
        name(Prefix,S0),
        symbolIndex(Prefix,Index),
        name(Index,S1),
        [U] = "_",
        appendString(S0,[U|S1],S01),
        name(Symbol,S01).

symbolIndex(Prefix,Index) :-
        retract( recordedSymbolIndex(Prefix,Index) ),
        !,
        NewIndex is Index+1,
        assert( recordedSymbolIndex(Prefix,NewIndex) ).
symbolIndex(Prefix,Index) :-
        Index = 1,
        NewIndex is Index+1,
        assert( recordedSymbolIndex(Prefix,NewIndex) ).


appendString([],L,L).
appendString([X|L1],L2,[X|L3]) :- appendString(L1,L2,L3).

%
%------------------------------------------------------------------------
%       Simple benchmarks
%------------------------------------------------------------------------

test :-
        benchmark(0),   % lambda-term unification not using patterns
        benchmark(1),   % a simple append benchmark adapted from Miller
        benchmark(2),   % essentially a beta-reduction benchmark
        benchmark(3),   % a simple L_{\lambda} interpreter
        benchmark(4).   % a more involved L_{\lambda} interpreter

benchmark(0) :-

                S = x\y\z\(x:z:(y:z)),
                K = x\y\x,
                I = x\x,
        write(' S = x\y\z\(x:z:(y:z))'),nl,
        write(' K = x\y\x'),nl,
        write(' I = x\x '),nl,
        write(' ?- hop_unification( S:K:K:x , I:x ).'),nl,
        goal_time((
                hop_unification( S:K:K:x , I:x )
        )),nl,!,

%               Zero  = s\x\x,                  % Church numeral
                One   = s\x\(s:x),              % Church numeral
                Two   = s\x\(s:(s:x)),          % Church numeral
                Four  = s\x\(s:(s:(s:(s:x)))),  % Church numeral
                Plus  = u\v\s\x\((u:s):(v:s:x)),
%               Succ  = u\s\x\(u:s:(s:x)),
                Times = u\v\s\x\(u:(v:s):x),

        write(' One   = s\x\(s:x)               % Church numeral'),nl,
        write(' Two   = s\x\(s:(s:x))           % Church numeral'),nl,
        write(' Plus  = n\m\s\x\((n:s):(m:s:x))'),nl,
        write(' ?- hop_unification( Plus:One:One , Two ).'),nl,
        goal_time((
                hop_unification( Plus:One:One , Two )
        )),nl,!,

        write(' Two   = s\x\(s:(s:x))           % Church numeral'),nl,
        write(' Four  = s\x\(s:(s:(s:(s:x))))   % Church numeral'),nl,
        write(' Times  = n\m\s\x\(n:(m:s):x)'),nl,
        write(' ?- hop_unification( Times:Two:Two , Four ).'),nl,
        goal_time((
                hop_unification( Times:Two:Two , Four )
        )),nl,!.

%------------------------------------------------------------------------------
%  1. A lambda-unification benchmark adapted from reference [10].
%
%    append [] K K.
%    append [X|L] K [X|M] :- append L K M.
%
%    benchmark(1) :-
%      statistics,
%      pi Y\ (append
%         [a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z] Y (H Y)),
%      statistics.
%
%------------------------------------------------------------------------------

:-op(950,fy,(pi)).

hop_append( L1, L2, L3 ) :-
        hop_unification(L1,nil),
        hop_unification(L2,L3).

hop_append( L1, L2, L3 ) :-
        hop_unification(L1, (cons:U:X)),
        hop_append( X, L2, L4 ),
        hop_unification(L3, (cons:U:L4)).

benchmark(1) :-
  nl,
write('   ?- pi y \ hop_append('),nl,
write('         (cons:1:  (cons:2:  (cons:3:  (cons:4:  (cons:5:'),nl,
write('         (cons:6:  (cons:7:  (cons:8:  (cons:9:  (cons:10:'),nl,
write('         (cons:11: (cons:12: (cons:13: (cons:14: (cons:15:'),nl,
write('         (cons:16: (cons:17: (cons:18: (cons:19: (cons:20:'),nl,
write('         (cons:21: (cons:22: (cons:23: (cons:24: (cons:25:'),nl,
write('         (cons:26: nil)))))))))))))))))))))))))),'),nl,
write('        y, (H:y)).'),nl,nl,
  goal_time((
    pi y \ hop_append(
                (cons:1:  (cons:2:  (cons:3:  (cons:4:  (cons:5:
                (cons:6:  (cons:7:  (cons:8:  (cons:9:  (cons:10:
                (cons:11: (cons:12: (cons:13: (cons:14: (cons:15:
```

```
                (cons:16: (cons:17: (cons:18: (cons:19: (cons:20:
                (cons:21: (cons:22: (cons:23: (cons:24: (cons:25:
                (cons:26: nil))))))))))))))))))))))))),
         y, (H:y)),
     write('Solution obtained: H = '),write(H),nl
   )),nl,!.


%------------------------------------------------------------------------
%  2. A beta-reduction benchmark
%     Difference lists are encoded as lambda-terms.
%     The "fast" difference list algorithm for reverse becomes a
%     lambda-unification/beta-reduction test.
%     Note this approach to implementing reverse is not really invertible...
%
% Lambda-Prolog benchmark for comparison:
%
%   reverse L (Rlambda [])  :-  rev_lambda L Rlambda.
%
%   rev_lambda [] (Z \ Z).
%   rev_lambda [X|L] R :-
%         rev_lambda L R1,
%         append_lambda R1 (Z\[X|Z]) R.
%
%   append_lambda L1 L2 (Z\(L1 (L2 Z))).
%
%   benchmark(2) :-
%         statistics,
%         (reverse [a,b,c,d,e,f] X),
%         statistics.
%
%------------------------------------------------------------------------

reverse( L, R ) :-
        rev_lambda(L,Rlambda),
        hop_unification( (Rlambda:nil), R ).

rev_lambda( L, R ) :-
        hop_unification( L, nil ),
        hop_unification( R, z\z ).
rev_lambda( L, R ) :-
        hop_unification( L, (cons:X:L1) ),
        rev_lambda( L1, R1 ),
        append_lambda( R1, z\(cons:X:z), R ).

append_lambda( L1, L2, L3 ) :-
        hop_unification( v\(L1:(L2:v)), L3 ).

benchmark(2) :-
  nl,write(
  '?- reverse((cons:a:(cons:b:(cons:c:(cons:d:(cons:e:(cons:f:nil)))))), X).'
  ),nl,nl,
  goal_time((
        reverse(
                (cons:a:(cons:b:(cons:c:(cons:d:(cons:e:(cons:f:nil)))))),
                X),
        write('Solution obtained: X = '),write(X),nl
  )),nl,!.


%
%-------------------------------------------------------------------------
%  3. Simple L_{\lambda} program, from reference [10].
%
%  copyterm a a.
%  copyterm (f X) (f U) :- copyterm X U.
%  copyterm (g X Y) (g U V) :- copyterm X U, copyterm Y V.
%
%  copyform (p X) (p U) :- copyterm X U.
%  copyform (q X Y) (q U V) :- copyterm X U, copyterm Y V.
%  copyform (and X Y) (and U V) :- copyterm X U, copyterm Y V.
%  copyform (imp X Y) (imp U V) :- copyterm X U, copyterm Y V.
%  copyform (all X) (all U) :-
%       pi y\( pi z\(copyterm y z => copyterm (X y) (U z))).
%  copyform (some X) (some U) :-
%       pi y\( pi z\(copyterm y z => copyterm (X y) (U z))).
%
%  term a.
%  term (f X) :- term X.
%  term (g X Y) :- term X, term Y.
%
%  atom (p X) :- term X.
%  atom (q X Y) :- term X, term Y.
%
%---------------------------------------------------------------------------

:- op(900,xfy,(==>)).   % logical implication

:- dynamic term/1.              % for use by the l_lambda interpreter below
:- dynamic atom_/1.             % for use by the l_lambda interpreter below
:- dynamic copyterm/2.          % for use by the l_lambda interpreter below
:- dynamic copyform/2.          % for use by the l_lambda interpreter below

term(a).
term((f:X)) :- term(X).
term((g:X:Y)) :- term(X), term(Y).

atom_((p:X)) :- term(X).
atom_((q:X:Y)) :- term(X), term(Y).

copyterm(a,a).
copyterm((f:X),(f:U)) :- copyterm(X,U).
copyterm((g:X:Y),(g:U:V)) :- copyterm(X,U), copyterm(Y,V).

copyform((p:X),(p:U)):- copyterm(X,U).
copyform((q:X:Y),(q:U:V)):- copyterm(X,U), copyterm(Y,V).
copyform((and:X:Y),(and:U:V)):- copyform(X,U), copyform(Y,V).
copyform((imp:X:Y),(imp:U:V)):- copyform(X,U), copyform(Y,V).
copyform((all:X),(all:U)):-
                    pi y\(pi z\(copyterm(y,z)==>copyform((X:y),(U:z)))).
```

```
copyform((some:X),(some:U)):-
                     pi y\( pi z\(copyterm(y,z) ==> copyform((X:y),(U:z)))).


benchmark(3) :-
  nl,write(
'?- copyform((
imp:(all:x\(and:(p:x):(and:(all:y\(q:x:y)):(p:(f:x))))):(p:a)),X)'
          ),nl,nl,
  goal_time((
        l_lambda_prove((
          copyform(
           (imp:(all:x\(and:(p:x):(and:(all:y\(q:x:y)):(p:(f:x))))):(p:a)),
            CopiedForm
                   )
                      )),
        beta_eta_normal_form(CopiedForm,NiceCopiedForm),
        write('Solution obtained: X = '),write(NiceCopiedForm),nl
            )),nl,!.

%--------------------------------------------------------------------------
%  4. A more challenging L_{\lambda} program, from reference [10].
%
%  prenex B B :- atom B.
%  prenex (and B C) D :- prenex B U, prenex C V, merge (and U V) D.
%  prenex (imp B C) D :- prenex B U, prenex C V, merge (imp U V) D.
%  prenex (all B) (all D) :- pi x\(term x => prenex (B x) (D x)).
%  prenex (some B) (some D) :- pi x\(term x => prenex (B x) (D x)).
%
%  merge (and (all B) (all C)) (all D) :-
%       pi x\(term x => merge (and (B x) (C x)) (D x)).
%  merge (and (all B) C) (all D) :-
%       pi x\(term x => merge (and (B x) C) (D x)).
%  merge (and (B (all C)) (all D) :-
%       pi x\(term x => merge (and B (C x)) (D x)).
%  merge (and (some B) C) (some D) :-
%       pi x\(term x => merge (and (B x) C) (D x)).
%  merge (and (B (some C)) (some D) :-
%       pi x\(term x => merge (and B (C x)) (D x)).
%
%  merge (imp (all B) (some C)) (some D) :-
%       pi x\(term x => merge (imp (B x) (C x)) (D x)).
%  merge (imp (all B) C) (some D) :-
%       pi x\(term x => merge (imp (B x) C) (D x)).
%  merge (imp (B (some C)) (some D) :-
%       pi x\(term x => merge (imp B (C x)) (D x)).
%  merge (imp (some B) C) (all D) :-
%       pi x\(term x => merge (imp (B x) C) (D x)).
%  merge (imp (B (all C)) (all D) :-
%       pi x\(term x => merge (imp B (C x)) (D x)).
%
%  merge B B :- quant_free B.
%
%--------------------------------------------------------------------------
```

```
:- dynamic prenex/2.          % for use by the l_lambda interpreter below
:- dynamic merge/2.           % for use by the l_lambda interpreter below
:- dynamic quant_free/1.       % for use by the l_lambda interpreter below

prenex(B,B) :- atom_(B).
prenex((and:B:C),D) :- prenex(B,U), prenex(C,V), merge((and:U:V),D).
prenex((imp:B:C),D) :- prenex(B,U), prenex(C,V), merge((imp:U:V),D).
prenex((all:B),(all:D)) :- pi x\(term(x) ==> prenex((B:x),(D:x))).
prenex((some:B),(some:D)) :- pi x\(term(x) ==> prenex((B:x),(D:x))).

merge((and:(all:B):(all:C)),(all:D)) :-
                        pi x\(term(x) ==> merge((and:(B:x):(C:x)),(D:x))).
merge((and:(all:B):C),(all:D)) :-
                        pi x\(term(x) ==> merge((and:(B:x):C),(D:x))).
merge((and:B:(all:C)),(all:D)) :-
                        pi x\(term(x) ==> merge((and:B:(C:x)),(D:x))).
merge((and:(some:B):C),(some:D)) :-
                        pi x\(term(x) ==> merge((and:(B:x):C),(D:x))).
merge((and:B:(some:C)),(some:D)) :-
                        pi x\(term(x) ==> merge((and:B:(C:x)),(D:x))).
merge((imp:(all:B):(some:C)),(some:D)) :-
                        pi x\(term(x) ==> merge((imp:(B:x):(C:x)),(D:x))).
merge((imp:(all:B):C),(some:D)) :-
                        pi x\(term(x) ==> merge((imp:(B:x):C),(D:x))).
merge((imp:B:(some:C)),(some:D)) :-
                        pi x\(term(x) ==> merge((imp:B:(C:x)),(D:x))).
merge((imp:(some:B):C),(all:D)) :-
                        pi x\(term(x) ==> merge((imp:(B:x):C),(D:x))).
merge((imp:B:(all:C)),(all:D)) :-
                        pi x\(term(x) ==> merge((imp:B:(C:x)),(D:x))).
merge(B,B) :- quant_free(B).

quant_free(A) :- atom_(A).
quant_free((and:B:C)) :- quant_free(B), quant_free(C).
quant_free((imp:B:C)) :- quant_free(B), quant_free(C).


benchmark(4) :-
  nl,write(
  '?-prenex((imp:(all:x\(and:(p:x):(and:(all:y\(q:x:y)):(p:(f:x))))):(p:a)),X)'
          ),nl,nl,
  goal_time((
      l_lambda_prove((
      prenex((imp:(all:x\(and:(p:x):(and:(all:y\(q:x:y)):(p:(f:x))))):(p:a)),
              PrenexForm )
                     )),
      beta_eta_normal_form(PrenexForm,NicePrenexForm),
      write('Solution obtained: X = '),write(NicePrenexForm),nl
           )),nl,!.

%--------------------------------------------------------------------------
%    A simple L_{\lambda} Interpreter using all of the previous material
%--------------------------------------------------------------------------

pi (X\GX) :- newgeneric(C), (X\GX):C.
```

```
((X\GX):Y) :- goal_subst(GX,GY,X,Y), call(GY).

assume(Atom) :- assert(Atom,Ref), undo(erase(Ref)).


l_lambda_prove(true)      :- !.
l_lambda_prove((A,B))     :- !, l_lambda_prove(A), l_lambda_prove(B).
l_lambda_prove((pi G))    :- !, pi z\(l_lambda_prove((G:z))).
l_lambda_prove((X\G):Y)   :- !, goal_subst(G,GY,X,Y), l_lambda_prove(GY).
l_lambda_prove((A==>B))   :- !, assume(A), l_lambda_prove(B).
l_lambda_prove(G)         :- l_lambda_clause(G,B), l_lambda_prove(B).


l_lambda_clause(G,B) :- functor(G,P,N), functor(H,P,N), clause(H,B),
                unifyGoalArgs(0,N,G,H).

unifyGoalArgs(N,N,_,_) :- !.
unifyGoalArgs(I0,N,G,H) :-
        I is I0+1,
        arg(I,G,Gi),
        arg(I,H,Hi),
        hop_unification(Gi,Hi),
        unifyGoalArgs(I,N,G,H).


goal_subst(G,G,_,_) :- var(G), !.
goal_subst(X,Y,X,Y) :- !.
goal_subst(X\G,X\G,X,_) :- !.
goal_subst(GX,GY,X,Y) :-
        functor(GX,P,N),
        functor(GY,P,N),
        goalSubstArgs(0,N,GX,GY,X,Y).

goalSubstArgs(N,N,_,_,_,_) :- !.
goalSubstArgs(I0,N,GX,GY,X,Y) :-
        I is I0+1,
        arg(I,GX,GXi),
        arg(I,GY,GYi),
        goal_subst(GXi,GYi,X,Y),
        goalSubstArgs(I,N,GX,GY,X,Y).

%------------------------------------------------------------------------
%       Output of the benchmarks above on an 8MB Sun ELC
%------------------------------------------------------------------------
% SICStus 0.7 #1: Fri Oct 25 09:12:35 PDT 1991
% {compiling /u/kr/papers/cufe/uvic/impl/test_hop.xpl...}
% {/u/kr/papers/cufe/uvic/impl/test_hop.xpl compiled, 3880 msec 60216 bytes}
% {compiling /u/kr/papers/cufe/uvic/impl/goal_time.pl...}
% {/u/kr/papers/cufe/uvic/impl/goal_time.pl compiled, 50 msec 646 bytes}
%  S = x\y\z\(x:z:(y:z))
%  K = x\y\x
%  I = x\x
%  ?- hop_unification( S:K:K:x , I:x ).
%
```

```
% 0 msec
%
%  One   = s\x\(s:x)            % Church numeral
%  Two   = s\x\(s:(s:x))               % Church numeral
%  Plus  = n\m\s\x\((n:s):(m:s:x))
%  ?- hop_unification( Plus:One:One , Two ).
%
% 10 msec
%
%  Two   = s\x\(s:(s:x))               % Church numeral
%  Four  = s\x\(s:(s:(s:(s:x))))       % Church numeral
%  Times  = n\m\s\x\(n:(m:s):x)
%  ?- hop_unification( Times:Two:Two , Four ).
%
% 10 msec
%
%
%    ?- pi y \ hop_append(
%               (cons:1:  (cons:2:  (cons:3:  (cons:4:  (cons:5:
%               (cons:6:  (cons:7:  (cons:8:  (cons:9:  (cons:10:
%               (cons:11: (cons:12: (cons:13: (cons:14: (cons:15:
%               (cons:16: (cons:17: (cons:18: (cons:19: (cons:20:
%               (cons:21: (cons:22: (cons:23: (cons:24: (cons:25:
%               (cons:26: nil))))))))))))))))))))))))))),
%         y, (H:y)).
%
% Solution obtained: H = c_1\(cons:1:(cons:2:(cons:3:(cons:4:(cons:5:(cons:6
% :(cons:7:(cons:8:(cons:9:(cons:10:(cons:11:(cons:12:(cons:13:(cons:14
% :(cons:15:(cons:16:(cons:17:(cons:18:(cons:19:(cons:20:(cons:21:(cons:22
% :(cons:23:(cons:24:(cons:25:(cons:26:c_1))))))))))))))))))))))))))
%
% 500 msec
%
%
% ?- reverse((cons:a:(cons:b:(cons:c:(cons:d:(cons:e:(cons:f:nil)))))), X).
%
% Solution obtained:
%             X = cons:f:(cons:e:(cons:d:(cons:c:(cons:b:(cons:a:nil)))))
%
% 110 msec
%
%
% ?- copyform((
%       imp:(all:x\(and:(p:x):(and:(all:y\(q:x:y)):(p:(f:x))))):(p:a)),X)
%
% Solution obtained:
%    X = imp:(all:c_3\(and:(p:c_3):(and:(all:(q:c_3)):(p:(f:c_3))))):(p:a)
%
% 130 msec
%
%
% ?-prenex( (imp:(all:x\(and:(p:x):(and:(all:y\(q:x:y)):(p:(f:x))))):(p:a)),X)
%
% Solution obtained:
%    X = some:c_10\(some:c_11\(imp:
```

```
%                    (and:(p:c_10):(and:(q:c_10:c_11):(p:(f:c_10)))):(p:a)))
%
% 540 msec
%
%---------------------------------------------------------------------------
%       Output of the benchmarks above with lazy unification
%       (the second substitutivity axiom for application is omitted)
%---------------------------------------------------------------------------
% SICStus 0.7 #1: Fri Oct 25 09:12:35 PDT 1991
% {compiling /u/kr/papers/cufe/uvic/impl/test_hop.xpl...}
% {/u/kr/papers/cufe/uvic/impl/test_hop.xpl compiled, 3710 msec 60070 bytes}
% {compiling /u/kr/papers/cufe/uvic/impl/goal_time.pl...}
% {/u/kr/papers/cufe/uvic/impl/goal_time.pl compiled, 40 msec 646 bytes}
%  S = x\y\z\(x:z:(y:z))
%  K = x\y\x
%  I = x\x
%  ?- hop_unification( S:K:K:x , I:x ).
%
% 0 msec
%
%  One   = s\x\(s:x)            % Church numeral
%  Two   = s\x\(s:(s:x))                % Church numeral
%  Plus  = n\m\s\x\((n:s):(m:s:x))
%  ?- hop_unification( Plus:One:One , Two ).
%
% 10 msec
%
%  Two   = s\x\(s:(s:x))                % Church numeral
%  Four  = s\x\(s:(s:(s:(s:x))))        % Church numeral
%  Times  = n\m\s\x\(n:(m:s):x)
%  ?- hop_unification( Times:Two:Two , Four ).
%
% 0 msec
%
%
%    ?- pi y \ hop_append(
%               (cons:1:  (cons:2:  (cons:3:  (cons:4:  (cons:5:
%               (cons:6:  (cons:7:  (cons:8:  (cons:9:  (cons:10:
%               (cons:11: (cons:12: (cons:13: (cons:14: (cons:15:
%               (cons:16: (cons:17: (cons:18: (cons:19: (cons:20:
%               (cons:21: (cons:22: (cons:23: (cons:24: (cons:25:
%               (cons:26: nil)))))))))))))))))))))))))),
%        y, (H:y)).
%
% Solution obtained: H = c_1\(cons:1:(cons:2:(cons:3:(cons:4:(cons:5:(cons:6
% :(cons:7:(cons:8:(cons:9:(cons:10:(cons:11:(cons:12:(cons:13:(cons:14
% :(cons:15:(cons:16:(cons:17:(cons:18:(cons:19:(cons:20:(cons:21:(cons:22
% :(cons:23:(cons:24:(cons:25:(cons:26:c_1))))))))))))))))))))))))))
%
% 430 msec
%
%
% ?- reverse( (cons:a:(cons:b:(cons:c:(cons:d:(cons:e:(cons:f:nil)))))), X).
%
% Solution obtained:
```

```
%           X = cons:f:(cons:e:(cons:d:(cons:c:(cons:b:(cons:a:nil)))))
%
% 110 msec
%
% ?- copyform((
%         imp:(all:x\(and:(p:x):(and:(all:y\(q:x:y)):(p:(f:x))))):(p:a)),X)
%
% Solution obtained:
%     X = imp:(all:c_3\(and:(p:c_3):(and:(all:(q:c_3)):(p:(f:c_3))))):(p:a)
%
% 120 msec
%
%
% ?-prenex((imp:(all:x\(and:(p:x):(and:(all:y\(q:x:y)):(p:(f:x))))):(p:a)),X)
%
% Solution obtained:
%    X = some:c_10\(some:c_11\(imp:
%                   (and:(p:c_10):(and:(q:c_10:c_11):(p:(f:c_10)))):(p:a)))
%
% 440 msec
```