

Naïve Reverse can be Linear

Pascal Brisset — Olivier Ridoux

IRISA

Campus Universitaire de Beaulieu,

35042 RENNES Cedex, FRANCE

{brisset,ridoux}@irisa.fr

Abstract

We propose a new implementation of logic programming with higher-order terms. In order to illustrate the properties of our implementation, we apply the coding of lists as functions to the context of logic programming. As a side-effect, we show that higher-order unification is a good tool for manipulating the function-lists. It appears that the efficiency of the program thus obtained relies critically upon the implementation of higher-order operations (unification and reduction). In particular, we show that a good choice for data-structures and reduction strategy yields a linear naïve reverse.

1 Introduction

The extension of Prolog to higher-order terms has been proposed by Miller and Nadathur[6]. The prototypal system is λ Prolog, of which we know two implementations: a Prolog based implementation and a Lisp based one. The first one was intended for experimental use and is very inefficient. We shall only refer to the Lisp based implementation.

We propose another implementation in which we apply two techniques that we have previously developed for Prolog. We translate λ Prolog programs into an imperative language (C), and we use MALI[2] as a term oriented abstract memory.

In this paper, we explain how primitive higher-order operations (unification and reduction) are critical for an efficient usage of λ Prolog. We choose as an experimentation domain the manipulation of lists coded as functions. Beside showing the specific capabilities of the language, we insist on the operations that are critical for the efficiency of the manipulation. The most striking result is that the natural transposition of the naïve reversal predicate is linear. We analyse what makes it linear.

We recall elementary notions on λ -terms and their unification in section 2. Then we describe several implementations of lists in Prolog; among them the function-lists (section 3). Time measurements are given in section 4 and compared with the Lisp-based implementation of λ Prolog.

2 Higher-order terms and Prolog

We briefly present the extension of Prolog to *higher-order terms*. The extension of the computation domain of Prolog requires to specify what is *unification* for the new terms. The restriction to Church's simply typed λ -terms allows for a usable definition of unification. Extensive presentation of higher-order unification can be found in [3] from where we take almost all of the vocabulary.

2.1 Simply typed λ -terms

Simple types, \mathcal{T} , are variable free first-order terms built from a collection of type constants and one dedicated binary type constant, \rightarrow . Constant \rightarrow is given an infix notation and is supposed to be left associative.

Simply typed λ -terms, Λ , are built from a collection of constants, \mathcal{C} , a collection of variables, \mathcal{V} , a collection of unknowns, \mathcal{U} , and two construction rules, *abstraction* ($\lambda x \cdot E$) and *application* ($(E F)$). There is a typing function τ from Λ to \mathcal{T} that verifies rules $\tau(\lambda x \cdot E) = \tau(x) \rightarrow \tau(E)$ and $(\exists \alpha. \tau(E) = \alpha \rightarrow \beta, \tau(F) = \alpha) \Leftrightarrow \tau((E F)) = \beta$.

Application is supposed to be right associative. Abstraction gives rise to the usual notion of free and bound variables.

In [3], *substitutions* are defined to operate on free variables of the terms they are applied to. In Prolog, substitutions are applied to clauses, and variables that are free in clauses are logical variables. Then, in the context of the integration of λ -terms in Prolog, we name *variables* the λ -variables, and *unknowns* the logical variables. Variables are all bound in an abstraction. Unknowns are all bound in an implicit universal quantification. This distinction is of the same nature than Barendregt's *variable convention*[1]. It appears that unknowns and variables deserve very different implementations. Unknowns pertain to the Prolog technology, whereas variables pertain to the functional technology.

Three equivalence relations are defined on Λ . α -*equivalence* defines consistent renaming of variables. β -*equivalence* defines the consistent reduction of an application ($\lambda x \cdot E F$) (called a β -*redex*) into $[x \leftarrow F]E$. η -*equivalence* formalizes extensionality of λ -defined functions. The union of all three is λ -*equivalence*.

2.2 Higher-order unification

To unify two simply typed λ -terms t_1 and t_2 is to find a substitution σ such that $\sigma t_1 = \sigma t_2$. The problem is semi-decidable. There may be several most general solutions, however most general solutions can be enumerated (may be infinitely long). For instance, the two terms $(F 1)$ and 1 , where $F \in \mathcal{U}$, are unified by both $\sigma_1 = F \leftarrow \lambda x \cdot x$ and $\sigma_2 = F \leftarrow \lambda x \cdot 1$. But neither σ_1 nor σ_2 is more general than the other.

2.2.1 Search procedure

Huet's algorithm is a search procedure in an OR-tree in which every node is a unification problem and every arc is an elementary substitution. The invariant of the tree is that, for every arc, the compositions of the elementary substitution with the solutions to the sibling node are solutions to the father node. Terminal nodes are *success nodes* (an empty unification problem) and *failure nodes* (an unsolvable unification problem). The composition of all the elementary substitutions on the path from the root to a success node is a solution to the root unification problem.

In our implementation the search procedure traverses (expands) the search-tree in a depth-first way. As for Prolog, efficiency is paid by the loss of completeness.

Terms are supposed to be in *head normal form*: $\lambda x_1 \dots x_n \cdot (@ t_1 \dots t_p)$ where @ is a constant, an unknown or a variable. @ is called the *head*, $\lambda x_1 \dots x_n \cdot @$ is called the *heading*. A term is called *flexible* if the head is an unknown, *rigid* if not.

Unification problems are supposed to be in *simplified* form (a set of pairs of head normal form terms which are not both rigid). Simplified form is obtained by a recursive descent in the structure of the two terms. Trivial failure is detected during simplification when the heads are both rigid and are different. All this is done by a procedure called SIMPL. It works like the first-order unification procedure on *rigid-rigid* pairs, but it remembers the abstraction context.

The expansion of a non-terminal node is done by a procedure called MATCH of which we do not say much because we will never use it in the following. It operates on nodes that contain at least one flexible-rigid pair, and invents substitutions according to two rules, *imitation* and *projection*. The preconditions of the two rules are not exclusive, hence the non-determinism.

This search procedure calls for some remarks.

1. A pair $\langle X, t \rangle$ (X does not occur in t) has a most general unifier $X \leftarrow t$ which is computed expensively by MATCH. SIMPL calls a procedure, named TRIV in [7], that handles cheaply as many as possible similar cases.
2. A flexible-flexible pair is not solved but delayed as a *constraint*. The constraint will be tested for satisfiability as soon as the pair becomes more rigid.
3. Types are essential because (1) in a non typed λ -calculus some terms have no normal form, (2) types make the unification problem well defined, and (3) types are used in the projection operation of MATCH.

3 Representation of lists

In this section, we present three representations for lists in Prolog: the Prolog list, the difference-list and the function-list. See [10] for an exposition of the difference-list technique and its generalisation, the incomplete structure technique. See also [4] for an exposition of the function-list technique.

3.1 Prolog lists

When lists are represented by Prolog lists, a *cons* is represented by the binary functor `./2` (noted `[|]`) and the empty list is represented by `nil` (noted `[]`). For instance, `cons(1, cons(2, cons(3, nil)))` is represented by `1.2.3.nil` (noted `[1,2,3]`).

Prolog list representation is used in the classical list manipulation predicates: “concatenate”, “naïve reverse” and “concatenate 3 lists”:

```
conc([], Y, Y).
conc([A|X], Y, [A|Z]) :- conc(X, Y, Z).
```

```
nrev([], []).
nrev([A|X], Y) :- nrev(X, RX), conc(RX, [A], Y).
```

```
conc3(A, B, C, ABC) :- conc(B, C, BC), conc(A, BC, ABC).
```

A *mode* is a specification of a particular input convention. A mode expression is a literal in which terms are replaced by `+` (always instantiated), `-` (never instantiated) and `?` (do not know). Among the three predicates above, `conc` is the only one which can operate in every mode. `nrev` and `conc3` enter a loop for modes `nrev(-, +)` and `conc3(-, -, -, +)`. So `conc` can be used to split a Prolog list, whereas `conc3` cannot. The order in which lists `A`, `B` and `C` are composed in `conc3` is arbitrary. However, for every composition order there is a mode which does not work.

Note that predicate `conc` can concatenate a list to itself.

```
twice(L, LL) :- conc(L, L, LL).
```

3.2 From Prolog lists to difference-lists

In logic programming, the programming with incomplete structures is a well-known technique. It is even a part where the logic programming paradigm is at its best. One example of this technique is the difference-list (noted `List-SubList`). A list is represented by the difference between a Prolog list and one of its sublists (the *tail* of the list) that must be an unknown. For instance, the empty list is represented by `X-X` and `cons(1, cons(2, cons(3, nil)))` is represented by `[1,2,3|X]-X`.

Because the tail is an unknown, two lists can be concatenated with only one unification which is a binding of the tail. Then, a list can be a left concatenand only once in its life.

```
dconc(A-ZA, ZA-ZB, A-ZB).
```

```
dconc3(A-ZA, ZA-ZB, ZB-ZC, A-ZC).
```

Note that, unlike predicate `conc`, `dconc` cannot be used to split a list. Nothing in Prolog enforces the constraint that the tail is a sublist of the list. So Prolog will bind `ZA` to something which is neither a sublist of `A` nor a superlist of `ZB`. Verifying the constraint can only result from a programming discipline. Another difficulty is that testing the empty list requires to perform an occurrence-check because otherwise Prolog is always willing to unify `[...|Z]-Z` and `X-X`. Similarly, lack of occurrence-check makes an attempt to concatenate a difference-list to itself succeed and produce an infinite term.

For all these reasons, the following predicates are bogus.

```
dtwice(L, LL) :- dconc(L, L, LL).
```

```
common_prefix(P, L1, L2) :- dconc(P, _, L1), dconc(P, _, L2).
```

However, transformation of the representation of lists, followed by some partial evaluation[10], generally leads to more efficient programs. `nrev` can be transformed into the following predicates:

```
revd1([], Y, Y).
```

```
revd1([A|L], Y, Z) :- revd1(L, [A|Y], Z).
```

```
revd(L, RL) :- revd1(L, [], RL).
```

Transforming difference-lists into Prolog lists is trivial (if it is allowed to be destructive as `dconc` is), but the way back needs to use predicate `conc`.

```
dlist2list(L-nil, L).
```

```
list2dlist(L, AL-ZL) :- conc(L, ZL, AL).
```

3.3 From difference-lists to function-lists

Function-lists can yield the same improvement as difference-lists while allowing a list to be a left concatenand more than once.

3.3.1 Function-lists

Since higher-order unification is well defined on typed terms only, we have to type the constants. Type declarations may contain unknowns. Their scope is exactly the declaration in which they occur.

The Prolog list constructors have the following types:

```
kind list type -> type.
```

```
type '.' A -> (list A) -> (list A).
```

```
type nil list _.
```

A list is represented by the function that left-concatenates the Prolog representation of the list to its argument. For instance, the empty list is represented by `z\z` and `cons(1, cons(2, cons(3, nil)))` is represented by `z\1,2,3|z`. The function-list representation is unique up to λ -equivalence. A function-list has type `(list A) -> (list A)` if its elements have type `A`. It is abbreviated to `(flist A)` throughout this paper.

Terms are curried and the structure of clauses is as with Edinburgh syntax. The backslash (`\`) represents λ -abstraction, unknowns have the standard syntax of Prolog variable (they begin with a `_` or a capital letter), and variables have the syntax of identifiers.

The concatenation predicates are:

```
type fconc (flist A) -> (flist A) -> (flist A) -> o.
fconc L R z\ (L (R z)).
```

```
type fconc3 (flist A) -> (flist A) -> (flist A)
           -> (flist A) -> o.
fconc3 L M R z\ (L (M (R z))).
```

Higher-order unification cannot capture variables, but the end of the right concatenand (`R`) must be related some way to the end of the total list. The solution is to substitute to the variable that represents the end of `R` a variable that is already captured as the end of the total list. This is done via an application.

Unlike predicates `dconc`, `conc3` and `dconc3`, predicates `fconc` and `fconc3` operate in all modes because equality of function-lists is completely encompassed by higher-order unification. In mode `fconc3(-,-,-,+)` the backtracking implementation of unification will enumerate the different possible splits of the fourth list. `fconc` and `fconc3` are not destructive because of the semantics of β -reduction. So, the following predicates work as expected.

```
type ftwice (flist A) -> (flist A) -> o.
ftwice L LL :- fconc L L LL.
```

```
type common_prefix (flist A) -> (flist A) -> (flist A) -> o.
common_prefix P L1 L2 :- fconc P _ L1, fconc P _ L2.
```

The naïve reverse predicate is:

```
type fnrev (flist A) -> (flist A) -> o.
fnrev z\z z\z.
fnrev z\[A|(L z)] z\ (RL [A|z]) :- fnrev L RL.
```

The first clause is obvious since `z\z` represents the empty list. The second clause uses higher-order unification to split a list and construct another.

As we did with difference-lists, function-lists can be used to produce an inversion predicate that operates on Prolog lists but uses function-lists internally.

```

type revf1 (list A) -> (flist A) -> o.
revf1 [] z\z.
revf1 [A|L] z\(RL [A|z]) :- revf1 L RL.

```

```

type revf (list A) -> (list A) -> o.
revf L (RL []) :- revf1 L RL.

```

3.3.2 Function-lists and universal quantification

The transformation from function-lists into Prolog lists and difference-lists is trivial and is not destructive.

```

type flist2list (flist A) -> (list A) -> o.
flist2list L (L []).

```

In spite of (because of) higher-order unification, these predicates are not totally symmetrical. They cannot be used to transform a Prolog list into a function-list because there are other solutions than the intended one, and there is no way to “separate the wheat from the chaff”. For instance, let `[[1]]` be a Prolog list the only element of which is a list. The solutions to `[[1]] = (L [])` are `L = z\[1]`, `L = z\[1|z]`, `L = z\[1|z]` and `L = z\[1|z|z]`. The intended solution is the second, others are “chaff”. Non-intended solutions are produced by an excessive use of the imitation rule (first solution) and by a confusion between different occurrences of the same subterm (`[]` in the third and fourth solutions). We have to design a special purpose predicate to do correctly the job.

```

type list2flist (list A) -> (flist A) -> o.
/* First solution */
list2flist L FL :- pi nil\(conc L nil (FL nil)).

/* Second solution */
list2flist [] z\z.
list2flist [A|L] z\[A|(FL z)] :- list2flist L FL.

```

The second solution uses cautiously the structure of function-lists. The first solution is more interesting in that it makes a critical use of a logical quantifier. `pi` and `sigma` are the universal and existential quantifier of λ Prolog. They can be presented independently from the higher-order terms, but it is an experimental fact that the quantifications (especially universal) and the higher-order terms are tightly related. In the example above, `pi` introduces the universally quantified symbol `nil`. It cannot be mistaken for the occurrences of `[]` and it cannot be captured by bare imitation. So it solves the imitation and confusion problems.

3.3.3 Function-lists and combinators

A concatenation combinator can be written and used autonomously. It is the function composition combinator, $l1 \setminus l2 \setminus z \setminus (l1 (l2 z))$. In the same vein, the nil combinator is $z \setminus z$.

Statements about the properties of lists and concatenation can be expressed in the language itself, and the nil and concatenation combinators can be generated automatically.

```
type monoid ((flist A) -> (flist A) -> (flist A))
            -> (flist A) -> o.
monoid CONC NIL :-
    pi l1 \ ( pi l2 \ ( pi l3 \
        ( CONC (CONC l1 l2) l3 = CONC l1 (CONC l2 l3) ) )),
    pi l \ ( CONC l NIL = l ),
    pi l \ ( CONC NIL l = l ),
    pi e \ ( pi l \
        ( CONC z \ [e | (l z)] _ = z \ [e | ((CONC l _) z)] ) ).
```

In the declarative reading of the above clause, `pi` actually reads “for all”. The current state of the art of λ Prolog implementation makes this more a curiosity than a tool. In mode `monoid + +`, the predicate `monoid` can indeed verify that two combinators have the required properties. In mode `monoid - -`, the same predicate finds solution `CONC = x \ y \ z \ (x (y z))`, `NIL = z \ z`. The fourth literal is required to describe the relation between `CONC` and `cons`. If the fourth literal is absent, another solution where `x` and `y` are permuted is possible.

Note that the resolution of `monoid CONC NIL` makes use of unification delay. The first literal produces a flexible-flexible pair which becomes a constraint. The other literals awake the constraint every time a binding is found for `CONC`.

4 Performances of function-lists

As we have repeatedly alluded to, the function-list technique makes an intensive use of higher-order unification and β -reduction. In this section, we look closer to the implementation of higher-order logic programming in order to show the cost of its elementary operations.

We give the results of some experiments with function-lists. Then we describe the internal representation of higher-order terms and critical parts of the β -reduction and unification procedures.

4.1 Execution times and memory consumption

Table 1 shows the run-times in seconds of some of the predicates that we have presented in the previous section, and of a built-in predicate, `normal_form`, which normalizes terms.

Length	32	64	128	256	512	1024	2048	4096	8192
nrev	0.19	0.7	2.6	10	41	167	685	2940	13733
revd	0.01	0.02	0.04	0.08	0.17	0.34	0.67	1.3	2.7
revf	0.07	0.13	0.26	0.52	1	2.1	4.1	8.2	17
(n) fnrev	0.46	0.91	1.8	3.6	7.2	14.5	29	59	119
(nn) fnrev	0.47	0.94	1.9	3.7	7.5	15.5	33	77	322
(1) list2flist	0.1	0.2	0.35	0.7	1.4	2.7	5.5	11	22
(2) list2flist	0.05	0.09	0.17	0.35	0.69	1.4	2.7	5.5	11
normal_form	0.03	0.06	0.12	0.23	0.46	0.91	1.8	3.6	13

Table 1: Run times (s on Sun3/60 with 3 Mb)

Normalisation must be included in the measured times. The different non-normal forms that a function-list is likely to assume are presented in subsection 4.2.2. The outputs of the two versions of `list2flist` are not only λ -equivalent as expected, they are identical. Furthermore, the times for normalizing the outputs of either version of `list2flist` and for `fnrev` are the same. The entry for `normal_form` gives the times for the complete reduction of the output of `fnrev`. The entries giving the times for `fnrev` are tagged with (n) when its input is normalised, and (nn) when it is the output of `list2flist`. The entries tagged (1) and (2) give the times for the first (with the `pi`) and second versions of `list2flist`.

It can be seen that `normal_form` and the two versions of `list2flist` are linear. This is desirable, but should be noted because the computations involved in these predicates are not generally linear. Finally, `fnrev` appears to be linear (even normalization included).

Times for large lists show a deviation which is explained because memory is almost exhausted and the garbage collector is called more frequently and has more to do.

Predicate	<code>normal_form</code>	<code>(1) list2flist</code>	<code>(2) list2flist</code>	<code>(nn)fnrev</code>	<code>(n) fnrev</code>
Length	9000	9000	9000	4000	8000

Table 2: Maximum lengths with 3 Mb

The programs measured in table 1 do not use the output of the tested predicate after its execution (input is used). Terms can be discarded as soon as they are produced. This prevents the memory limitation from perturbing the time measurement. But it gives no indication on the capacity of the

system. Table 2 shows for some predicates the size of the largest list they can operate on while using both their input and output after their execution. This corresponds to the most pessimistic memory requirements. Knowing that the measurements have been done with 3 mega-bytes of memory, it indicates the capacity of the system.

4.2 β -reduction

β -reduction is an important procedure since it is used to put terms in head-normal form before unification.

4.2.1 The higher-order terms

The higher-order terms are represented by reversibly mutable graphs. Graph means that we intend to do graph reduction[9]. Mutable means that it is possible to physically replace a redex by its reduced form in the graph. This provides sharing of the reduction effort. Reversibly means that mutations (reductions) must be undone when backtracking. This is the result of inserting graph reduction in a Prolog context.

4.2.2 Lazy reduction

The first thing to note is that β -reduction is done only when required, i.e. before unification, to allow comparison of terms. Even when unification requires β -reduction, it does not require to achieve a complete normalization. What is required is to put the terms to be unified in head-normal form, so as to exhibit the headings of the terms. A second thing is that β -reduction applies the outer-most strategy. Note that outer-most reduction is not necessary to have a converging normalization of simply typed λ -terms. In this context, it is only used for its connection with lazyness.

A consequence is that a common form of function-list is $z\backslash[e1|(L z)]$ (e.g. $z\backslash[1|(z\backslash[2|(z\backslash[3|z] z]) z])$) rather than $z\backslash[e1,\dots,eN|z]$ (e.g. $z\backslash[1,2,3|z]$). The two versions of predicate `list2flist` produce lists which have the first form. In both forms, the first element of the list and the end of the list (the z) are accessible in constant time. A third form is $z\backslash(L [eN|z])$ (e.g. $z\backslash(z\backslash(z\backslash(z\backslashz [1|z]) [2|z]) [3|z])$). Predicate `fnrev` produces a list with this form. With this representation, the end of the list is accessible in constant time, but an access to the first element requires to reduce all the redexes. It is linear because of the outer-most strategy. Note that after one access to its first element, a list is normalized. So, further accesses will be immediate.

4.2.3 Saving copies during reduction: combinators recognition

With graph reduction, β -reduction must duplicate the left part of the β -redex because it may have other occurrences in another context.

Duplication is useless for subterms that contain no free occurrences of the variable to be substituted. It would be too heavy to record for all terms all the variables that occur free in them. It is easier, if less precise, to record the terms in which no variable has a free occurrence. These terms are usually called combinators.

A lot of terms are combinators. Every instance of a combinator is itself a combinator because higher-order unification forbids variable capture. So, it is effective to record which source terms are combinators. Every binding value is a combinator by definition of higher-order unification. So, binding values are tagged as combinators as soon as they are created. Then, combinator detection incurs no dynamic cost.

Our experience is that the recognition of *all* source combinators, and the tagging as combinators of *all* binding values is crucial. The mere exception causes a visible slow down. Note that a lot of terms are recognized to be combinators because variables and unknowns are distinguished. Every literal argument, every unknown is a combinator.

The effect is three-fold. Less time is spent by β -reduction. Less memory is consumed, hence less time is spent garbage collecting. More sharing is achieved, hence unification and β -reduction costs are better factorized.

4.3 Optimization of unification: TRIV

In many cases, a unifying substitution can be straightforwardly computed. Two difficulties arise: the need for an occurrence-check and the fact that an unknown may be hidden in an abstraction by η -expansion.

The most trivial case is a unification problem with the form $\langle X, t \rangle$. A solution is the substitution $X \leftarrow t$ if X and t satisfy an occurrence-check. The notion of occurrence-check for higher-order unification is not as simple as for first-order unification. An occurrence of X in t may be discarded by further reduction. Sufficient criteria for performing the trivial unifications are presented by Huet. A failure of TRIV does not imply a failure of unification. It only means that the general procedure must be applied. The permutation version of TRIV[7] is not implemented in the measured system, but it can be, and probably will be in a near future. As for first-order Prolog, the occurrence-check is not implemented.

η -expansion causes an unknown X to be replaced by $\lambda u \cdot (X u)$. An efficient TRIV procedure must recognise similar cases. When **fnrev** is called in mode **fnrev(+,-)**, its second argument is an unknown but is η -expanded. The procedure TRIV recognises it and does the substitution.

4.4 Application to the predicate fnrev

We apply all the mechanisms we have described to the resolution of **fnrev** $z \setminus [1 \setminus (z \setminus [2 \setminus (z \setminus [3 \setminus z] z]) z]) \text{ LOut}$.

The unification problem associated with the first parameter of **fnrev** is $z \setminus [1 \setminus (z \setminus [2 \setminus (z \setminus [3 \setminus z] z]) z]) = z \setminus [A \setminus (L z)]$. After simplification,

it becomes $z\backslash 1 = z\backslash A$, $z\backslash(z\backslash[2|(z\backslash[3|z] z)] z) = z\backslash(L z)$. The first pair yields substitution $A \leftarrow 1$ by imitation. In the second pair, $z\backslash(L z)$ is recognised to be η -equivalent to L . So the second pair yields the trivial substitution $L \leftarrow z\backslash(z\backslash[2|(z\backslash[3|z] z)] z)$. No occurrence-check is required because it is the first occurrence of L .

The unification problem associated with the second parameter of `fnrev` is $L\text{Out} = z\backslash(\text{RL } [A|z])$ (if it is treated after the first parameter, it is $L\text{Out} = z\backslash(\text{RL } [1|z])$). It produces the trivial substitution $L\text{Out} \leftarrow z\backslash(\text{RL } [1|z])$. Again, no occurrence-check is required because it is the first occurrence of $L\text{Out}$.

So, one resolution step has been done in constant time and produces the derived goal `fnrev` $z\backslash(z\backslash[2|(z\backslash[3|z] z)] z) L\text{Out}$. After a β -reduction of the outer-most redex of its first parameter, the derived goal has the same profile as the father goal; the next resolution step will take the same time. And so on. So the resolution of a goal `fnrev` $L\text{In } L\text{Out}$ with mode `fnrev + -` will be time-linear with the length of L .

When the input list is completely β -reduced, `fnrev` is linear for similar reasons. When the list is in the third form (see 4.2.2), the access to its first element causes its reduction; then the previous analysis applies.

4.5 Other implementations

We compare our implementation with eLP (version 0.15), the Lisp based implementation of λ Prolog (Ergo Project at Carnegie Mellon University). Table 3 gives times for eLP. To be fair, one must say that eLP is interpreted. However, complexity has nothing to do with the implementation technology, and `fnrev` is quadratic on eLP. The two versions of `list2flist` are probably linear, but are the victims of an intrusive garbage-collector. Note also that no predicate can operate on lists longer than 4096, in spite of the garbage-collector of Lisp and the nearly 5 times bigger memory space. The sign \perp means that the computation aborted.

Length	32	64	128	256	512	1024	2048	4096
<code>nrev</code>	8.6	48.5	\perp					
<code>fnrev</code>	6.2	20.6	79	338	1600	\perp		
(1) <code>list2flist</code>	.9	1.7	3.8	8.2	20	53	152	\perp
(2) <code>list2flist</code>	.5	1	2.2	4.9	10.7	27	89	\perp

Table 3: eLP times (s on sun4 with 14 Mb)

Another implementation of λ Prolog is under study[8]. It will be WAM inspired, will use internally the de Bruijn's nameless representation of λ -terms,

and will be environment-based. This is a radical difference with our implementation but it is difficult to foresee its consequences. Our optimizations will likely not apply to this scheme. For instance, combinators recognition seems irrelevant. Performance bottlenecks (if any) must be located, and new optimizations designed to fix them.

4.6 Memory management with MALI

MALI[2] is a logic programming oriented virtual memory. It has been implemented in software and hardware. All the times given in this paper have been measured with a software implementation written in C. MALI is not dedicated to any particular logic programming dialect. For instance, it is not specialised for the representation of λ -terms. But it provides general purpose data-structures for which an efficient memory management is implemented[5]. The theory of MALI memory management is based on the temporal relations between unknown creation, unknown binding and choice-point creation. We call usefulness-logic the specification of which data-structures are useful. The main result is that the usefulness-logic of logic programming is not reducible to the one of functional programming. In concrete terms, this means that to base Prolog memory management on Lisp memory management is not the best solution.

5 Conclusion and further work

We have used list manipulation as a pretext to expose efficiency issues. Primitive operations such as unification and reduction must be carefully designed so as to have a robust system. List manipulation can be more than a pretext and can give a standard to measure speed of higher-order Prolog just like `nrev30` for first-order Prolog. The first version of `list2flist` or `fnrev` or a combination of both seem to be a good candidate to become a standard. We propose that such a standard involves large memory requirement, i.e. to choose `fnrev1000` rather than `fnrev30`.

We have implemented the higher-order primitive operations on a graph-reduction basis. The technical result of this study is a set of optimizations which are the necessary companions of a graph-reduction oriented implementation of λ Prolog. These optimizations are: a lazy outer-most β -reduction, the recognition of combinators, and a TRIV that knows about η -equivalence. The study also confirms the need for an efficient memory management such as the one that MALI offers.

Although our implementation enjoys nice complexity properties, it is rather slow when it is compared with the current state of the art for first-order Prolog. We wish to improve the speed of our system. In its present state the control of resolution is compiled but unification is not. Our current implementation task is to devise a compilation scheme for unification and

indexation so as to bring the performance level of the first-order part closer to the current state of the art.

References

- [1] H. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. Volume 103 of *Studies in logic and the foundations of mathematics*, North-Holland, 1981.
- [2] Y. Bekkers, B. Canet, O. Ridoux, and L. Ungaro. MALI: a memory with a real-time garbage collector for implementing logic programming languages. In *3rd Symp. Logic Programming*, IEEE, Salt Lake City, UT, USA, 1986.
- [3] G. Huet. A unification algorithm for typed λ -calculus. *Theoretical Computer Science*, (1):27–57, 1975.
- [4] R.J.M. Hughes. A novel representation of lists and its application to the function “reverse”. *Information Processing Letters*, (22):141–144, 1986.
- [5] S. Le Huitouze. A new data structure for implementing extensions to Prolog. In P. Deransart and J. Małuszynski, editors, *2nd Int. Work. Programming Languages Implementation and Logic Programming*, pages 136–150, Springer-Verlag, 1990. LNCS 456.
- [6] D.A. Miller and G. Nadathur. Higher-order logic programming. In E. Shapiro, editor, *3rd Int. Conf. Logic Programming*, pages 448–462, Springer-Verlag, London, UK, 1986. LNCS 225.
- [7] G. Nadathur. *A Higher-Order Logic as the Basis for Logic Programming*. Ph.D. Thesis, University of Pennsylvania, 1987.
- [8] G. Nadathur and B. Jayaraman. Towards a WAM model for λ Prolog. In E.L. Lusk and R.A. Overbeek, editors, *1st North American Conf. Logic Programming*, pages 1180–1198, MIT Press, Cleveland, OH, 1989.
- [9] S.L. Peyton Jones. *The Implementation of Functional Programming Languages*. *Int. Series in Computer Science*, Prentice-Hall, 1986.
- [10] L. Sterling and E. Shapiro. *The Art of Prolog*. MIT Press, 1986.