Higher-Order Programming in an OR-intensive Style

Paul Tarau
Dépt. d'Informatique
Université de Moncton
tarau@info.umoncton.ca

Bart Demoen
Dept. of Computer Science
Katholieke Universiteit Leuven
bimbart@cs.kuleuven.ac.be

Abstract

Because implementations tend to favour a recursive deterministic AND-intensive programming style, it seems counter intuitive that a backtracking `findall`-based paradigm can be more efficient. This is explained by a source level technique which is called *goal precomputation*, by a heap-based copy-once findall and optimized meta predicates. A precomputed goal containing a future (reusable) computation is backtracked over instead of being constructed repeatedly. These techniques are enough, in terms of efficiency, to rehabilitate a generator based OR-intensive programming style.

On the other side, we will try to find out how to improve the traditional AND-intensive implementation of higher-order primitives and study the impact of meta-programming constructs like `call/N`.

We introduce a new 'universal' higher-order predicate implemented also with *goal precomputation* which combines on the fly selected answers from a generator. By using it for the executable specification of key higher-order Prolog constructs, we point out the (unsuspected) expressive power of a pure OR-intensive programming style.

Keywords: design and implementation of language facilities for higher-order programming, optimization of logic programming engines, WAM, continuation passing binary logic programs.

1 Introduction

Higher-order Prolog programming is considered inefficient and textbooks in general suggest writing problem specific first order versions for common programming tasks. This inefficiency has been overcome by macro expansion [3] and, more generally, by partial evaluation [6]. Macro-expansion is even a standard feature in systems like Eclipse and partial-evaluators for full-Prolog like Mixtus are in current use.

However, efficient generic higher-order primitives are still needed when enough compile-time information is not available, (for instance when in `findall(X,Goal,Xs)` `Goal` is unknown and cannot be inferred) or when code size explosion and excessive processing time can make partial evaluation and macro-expansion prohibitive.

In the past `assert` based implementations of higher-order built-ins like `findall` unduly overtaxed their useful work by high constant factors and redundant copying. This makes them almost useless if efficiency is of any concern to the application programmer.

We will analyze the impact of a heap-based copy-once `findall` [8, 13] and a sharing preserving and term compressing `copy_term` built-in [13] to eliminate the overhead in the implementation of generic higher-order programming built-ins, based on a new *OR-intensive* style implementation using precomputed goals.

We will describe the implementation of the proposed techniques in the BinProlog system [1] and what optimizations can be done in native code Prolog implementations like ProLog by BIM `4.0` and Sicstus `2.1_8` for an efficient implementation of generic higher-order predicates.

On the application side, we will explore the opportunities opened by these techniques for more efficient and also more elegant higher-order programming. In particular we will rehabilitate a somewhat neglected *OR-intensive* programming style and show its unsuspected expressive power in combination with the appropriate higher-order primitives.

Note that the work described in this paper is complementary to [4] where techniques to avoid using `findall/3` through the OR-to-AND transformations of [14, 15] are given.

The paper is organized as follows. In section 2, maplist is used to illustrate the concept of precomputed goal and the usual implementation of maplist is contrasted with a findall based backtracking implementation; (surprising) performance figures are presented for different implementations and a preliminary explanation of these figures is given. In section 3, the concepts of OR-intensive and AND-intensive programming are introduced and contrasted. In section 4, the implementation of cooperation between OR-branches is discussed. Section 5 revises the implementation of higher-order primitives. Section 6 attempts to give a balanced view on the choice between the two approaches. Section 7 extends our techniques to competition between OR-branches, describes a goal precomputation implementation (code in Appendix 1) of `bestof/3` and a novel higher-order predicate: combine/4, which is universal in the sense that it covers both cooperation and competition between OR-branches (code in Appendix 2). Section 8 and 9 discuss briefly related work and give a conclusion.

2 Two implementations of `maplist`

Let's start with an example which will help in the sequel to stress the main point of the paper.

2.1 A straightforward `maplist`

Maplist/3 is usually defined as:

```
straightforward_maplist(FXs,Is,Os) :- maplist1(Is,FXs,Os).

% The list argument is first for indexing.
maplist1([],_,[]).
maplist1([I|Is],Closure,[O|Os]) :-
      copy_term(Closure,NewClosure),
      add_args(NewClosure,I,O,NewGoal),
      call(NewGoal),
      maplist1(Is,Closure,Os).

add_args(Closure,I,O,Goal) :-
      Closure=..L1,
      concat(L1,[I,O],L2),
      Goal=..L2.
```

Note that we have taken care to create a *new* Closure [2] at each step to deal with the general case when Closure is not known to be ground.

[1] Available by ftp from clement.info.umoncton.ca

[2] A closure is a record of information needed for some future computation. By applying it to some arguments we obtain an evaluable object, i.e. a function or predicate, depending on the

Compiler/Bmark	straightforward	with findall
Quintus 3.1.4 (asm-emulated)	16417	14500
Sicstus 2.1_8 (native)	3710	4200
ProLog by BIM 4.0 (native)	1960	740
BinProlog 2.38 (C-emulated)	2510	670

Figure 1: Two maplist/3 implementations

2.2 `Maplist` with `findall`

Let us contrast this with a findall based implementation. We will implement `maplist` by 'precomputing' the goal used inside of `findall` only once. A closure is extended to a Prolog goal constructed in advance and backtracked over instead of creating and calling it inside a forward recursive loop. This technique obtains the computed answers in minimal space thus has the amortized benefit of reducing/avoiding garbage collection.

We obtain the following program for maplist/3.

```
% maplist with findall
maplist_with_findall(Closure,Is,Os) :-
      add_args(Closure,I,O,Goal),
      findall(O,for_one(Goal,I,Is),Os).

for_one(Goal,I,Is) :- member(I,Is), call(Goal).
```

This can be used as follows:

```
    ?- maplist(plus(1),[10,20,30],T).
    => T=[11,21,31]
```

after defining:

```
     plus(X,Y,Z) :- Z is X + Y.
```

2.3 A performance paradox

The timings for Quintus and Sicstus, Prolog by BIM and BinProlog on a Sparcenter 1000 with no garbage collection and enough memory to have everything in-core are shown in figure 1, in milliseconds. The benchmark compares the performance of the `findall`-based maplist with the straightforward recursive implementation by applying the closure plus(1) to a list of 50000 integers.

It turns out that meta-calling and backtracking on a goal constructed in advance is faster than constructing it over and over in a loop[3]. As the Quintus and, to some extent, the Sicstus figures show, the real performance is usually distorted by the fact that the `findall`, `call` and `univ` built-ins tend to overtax goals doing useful computation, often by an order of magnitude.

BinProlog's good performance is explained by the composite effect of the following factors:

programming paradigm. For instance in Prolog something like `plus(1)` applied to arguments `2` and `R` gives `plus(1,2,R)`, an executable predicate.

[3]This is, after all, what one should expect as a result of factoring out invariant computation from inside a loop.

- fast copy-once `findall`
- fast *call* built-in
- faster choice point handling in the BinWAM[4],
- the relative slowness of =.. in some implementations

On the other hand, in a native code system like ProLog by BIM 4.0 with an efficient (not assert-based) `findall` and `univ` implemented in C, performance follows the BinProlog patterns, the double copying[5] in `findall` being the main reason for the differences.

3 OR-intensive vs. AND-intensive programming

We can characterize the `findall` based `maplist` program as a typical *OR-intensive* logic program. It uses the logic engine to implicitly iterate through its answer-space. The programmer can think *elementwise*, focus on the *individual* properties of the objects and ignore implementation of their *aggregation* (for instance their accumulation on a list). By contrast, we can characterize the conventional implementation of maplist as *AND-intensive*. Its main feature is that it takes the pain of accumulating each answer explicitly.

The simplicity and the unusually good performance of the `findall` based `maplist` suggests that it is useful to optimize an implementation for an OR-intensive programming style. The advent of constraint programming extensions built on top of logic programming systems reinforces the original search-engine function of Prolog and adds (yet) another reason for such optimizations.

3.1 Early construction vs. late construction

Let us start with an engine-level comparison of two execution models. We refer the reader to [12] for the binarization transformation BinProlog is based upon, to [2] for pointing out the possible differences in complexity between a program and its binary equivalent and [13] for a quantitative analysis of the resource consumption of the latest version of BinProlog's engine[6].

In procedural and functional languages featuring only deterministic calls it makes sense to avoid eager early structure creation. The WAM [17] follows this trend based on the argument that most logic programs are deterministic and therefore calls and structure creation in logic programming languages should follow this model. A more careful analysis suggests that the choice between

- late and repeated construction (standard WAMs with AND-stack)
- eager early construction (once) and reuse on demand as in BinWAM

will favor different programming styles. Let's note at this point that the WAM's assumptions are subject to the following paradox:

- If a program is mostly deterministic then it will tend to fail only in the guards (shallow backtracking). In this case, when a predicate succeeds, all structures

[4] We refer the reader to [7, 9, 10] for the details of BinProlog optimizations

[5] BinProlog's findall manages to copy its computed anser only once, using a heap splitting technique [8], while other Prologs usually copy to a dedicated area first and then back.

[6] Called the *BinWAM* i.e. a simplified WAM-like abstract machine specialized for the execution of binary logic programs.

specified in the body of a selected clause will eventually get created. By postponing this, the WAM will be only as good as doing it eagerly upon entering the clause (as in BinWAM).

- If the program is mostly nondeterministic then late and repeated construction (WAM) is not better than eager early creation (BinWAM) which is done only once, because it implies more work on backtracking.

This explains in part why (against all obvious intuitions), a standard AND-stack based WAM is not necessarily faster than a properly implemented BinWAM, as the BinProlog performance analysis of [10] shows.

Note that in the case of early *failure* in the context of deep backtracking conventional WAMs have a clear advantage although this will again compete with the BinWAM's smaller and unlinked choicepoints to the point that, for instance, the CHAT80 parser will turn out to be faster in BinProlog than in a good C-emulated WAMs like Sicstus 2.1 as shown in [10].

In practice, different programming styles will benefit differently from the various tradeoffs implied by one of these basic choices. In any case, system-level knowledge on what's really happening inside the engine should be abstracted in higher-order constructs to the advantage of application-level users.

3.2 High-level information exchange between OR-branches

We can see direct state manipulation in Prolog (either through the traditional assert/retract based approach or through BinProlog's blackboard[7]) as a disguised (and low-level) form of information exchange between different OR-branches. In practice, direct use of state-manipulation does not fit well neither with the declarative semantics of the programs neither with the operational predictability of their backtracking behaviour.

We will informally classify higher-order primitives as *cooperating* or *competing* according to whether or not the final result of their computation requires information from more than one OR-branch of their *generator* [8].

Cooperation (in contrast with *competition*) between OR-branches implies that computed answers from different branches of a generator will not be used to exclude each other but gathered in a data-structure that can be further processed by a unique resolution branch.

`Findall` is a typical example of a cooperative higher-order primitive which collects the computed answers from *all* the OR-branches of a generator with no additional processing.

On the other hand, getting the *best* answer with respect to an order relation which applies to pairs of answers of a generator is a typical example of *competition* between OR-branches.

4 Efficient *cooperation* between OR-branches

To make implementation of cooperation between OR-branches efficient, the overhead of *term copying* and *answer collecting* should be made as small as possible. As it was the case with `maplist`, defining cooperative higher-order predicates can be done efficiently in terms of `findall` and appropriate generators.

[7]Basically a permanent data and dynamic code store which gives explicit control over 'copying', 'indexing' and 'survival on failure' [11]

[8]We call *generator* a nondeterministic predicate which gives one-by-one elements of a (usually) finite domain. A list, a range of integers, the leaves of a binary trees are examples of data easily represented as generators with predicates like `member/2`.

The following features (present in BinProlog 2.38) make this technique possible:

- a fast `copy_term` primitive, based on a Cheney-style algorithm using a form of generalized CDR-coding (term-compression)
- a heap-oriented *copy-once* `findall`.

We refer to [13] for the details of the implementations of `copy_term` with term compression based on *last argument overlapping* and an unconventional *tag-on-data* low-level representation. `Findall` is implemented using a technique called *heap lifting* [8], which manages to copy the computed answers collected by `findall` only once [9].

4.1 Optimizing OR-intensive higher-order primitives

In this section we will discuss some other techniques that can further speed-up the findall based implementation of maplist and other similar predicates.

4.1.1 Fast implementation of basic generators

Note that a unique basic generator like `member/2` covers all the `findall`-based higher-order predicates working over lists and a few others can cover various finite domains.

Fast implementation of the most frequently used basic generators like `member/2` is actually a special case of using iterative techniques for simple recursive procedures [5]. In particular, choice-point reuse (destructive assignment on part of an existing choice-point instead of destruction-creation) can benefit for this purpose. Note that such built-in generators are *reusable* by any findall based higher-order predicate.

Our benchmarks with the ProLog by BIM compiler which uses a similar choice point reuse optimization (depending on a compiler switch), give a speed-up of 15% on member/2 with positive effect on generator-based higher-order primitives when the overhead of findall is factored out.

4.1.2 Using partial evaluation

Clearly `member/2` and in general *known* generators can be partially evaluated with respect to the execution of their precomputed goals as in:

```
maplist(Closure,Is,Os) :-
      add_args(Closure,I,O,Goal),
      findall(O,member_call(Goal,I,Is),Os).

member_call(Goal,I,[I|_]) :- call(Goal).
member_call(Goal,I,[_|Is]) :- member_call(Goal,I,Is).
```

Clearly a BinWAM-based system like BinProlog profits the most from this partial evaluation mostly due to the BinWAM's faster deep backtracking. (figure 2).

Overall, a partial evaluation of findall itself with respect to a known generator with choicepoint reuse can further improve performance, if such a technique is adopted in a BinWAM system.

Note that we have not considered here partial evaluation with respect to a known operation which shall collapse a `maplist` construct to its first order equivalent.

[9] Basically, the heap is split in a small lower half and a larger upper half. The goal is executed in the upper half of the heap. We push a copy of the answer to an open ended list located in the lower half starting from the initial position of the top of the heap H. The list grows with each new answer. After finding an answer we force backtracking. At the end, `H` is set to point just after the last answer on the heap. An internal stack ensures correctness for embedded uses of `findall`.

Compiler/Bmark	member + call	p. evaluated member
Quintus 3.1.4 (asm-emulated)	14500	14533
Sicstus 2.1_8 (native)	4200	4160
ProLog by BIM 4.0 (native)	720	719
BinProlog 2.38 (C-emulated)	670	590

Figure 2: Maplist with partially evaluated member

Compiler/Bmark	copy and `univ`	`univ`	library(call)
Quintus 3.1.4 (asm-emulated)	33267	10516	24700
Sicstus 2.1_8 (native)	2660	2539	-
ProLog by BIM 4.0 (native)	1780	1210	-
BinProlog 2.38 (C-emulated)	2310	1900	-

Figure 3: AND-intensive maplist with emulated call/3

5 The implementation of AND-intensive higher-order primitives revisited

Although standard library versions of maplist or text-book versions of maplist were all slower in terms of absolute performance than our version based on goal precomputation in the context of a fast copy-once findall, the following question still remains unanswered:

> How does *the best possible* conventional maplist implementation compare with a findall based maplist, and more generally, which of them will perform better on frequently occurring programming idioms?

In the AND-intensive implementation of maplist, basically two primitives are used: the meta-call (i.e. `call/1`) and the `univ` operator (`=../2`). About `univ`, one can say little more than that it should be implemented as an inline predicate and specialized whenever it is possible.

Some implementations recognize the conjunction Goal =.. SomeList , call(Goal) and optimize it accordingly, especially if Goal doesn't occur elsewhere in the clause: indeed, the left hand side of the `univ` operator doesn't need to be created in that case. Performance is is shown in figure 3

The story about the meta-call is longer: there is a difference between `call/1` in ordinary implementations and in BinProlog. In most Prolog systems (Aquarius is another notable exception) the argument of call/1 is meta-interpreted to find out if it is a control construct (like conjunction, if-then-else ...) or a user defined predicate. It means that the argument of call/1 is usually given to a (one level) meta interpreter. On the other hand, in BinProlog, this is avoided and a metainterpreter is called only upon an exception like an undefined predicate. Control constructs are treated as ordinary predicate definitions so that the argument of the meta-call is always known as a being a callable predicate. As the goal constructed from a closure can be restricted to be a callable predicate, the efficiency of the BinProlog `call/1` can be recovered through hidden primitives such as `$$call/1` (ProLog by BIM) or

Compiler/Bmark	w. call/1	w. fast call/1	w. built-in call/3
ProLog by BIM 4.0 (native)	1270	1060	280
Sicstus 2.1_8 (native)	2539	900	-

Figure 4: AND-intensive maplist with 'accelerated' call/1 and call/3

`prolog:$nodebug_call` (SICStus). For the latter two, the overhead of `call/1` compared to these hidden primitives can be measured by redoing the maplist benchmarks with `call/1` replaced by them (see figure 4).

As an alternative to the repeated use of `univ`, one can introduce the predicate `call/3` [10] which works as if defined by:

```
call(Closure,I,O) :-
       Closure =..L, append(L,[I,O],R) ,
       Goal=..R, call(Goal).
```

Of course, one gains nothing by putting this definition in a library as such. Implementing it as a variant of `$$call/1` in ProLog by BIM proved to have a major impact on the AND-intensive maplist (see figure 4)

This shows, that a fast implementation of the meta-call and the extended meta-call predicates, is essential for an AND-intensive higher-order programming style. The OR-intensive style benifits less from the extended predicates, because the extended goal is constructed only once.

6 A final comparison of the two approaches

Performance is subject to various and often subtile factors and omission of any of them can lead to wrong or incomplete conclusions. When fixing the benchmarking context, it is important to ensure that it simulates the most frequent uses in real-life programs and that factors which influence the measurements can be singled out and studied separately. The following issues will be considered: the answer size, the answer/computation ratio, the differences in operational semantics and internal sharing.

6.1 Answer-size related factors

We have tested AND- versus OR-intensive maplist implementations in BinProlog with `nrev/2` together with the following predicates

```
id(X,X).
l2dl(L,Head-Tail):-append(L,Tail,Head).
const(_,1).
```

working on a list contining multiple instances of a (shared) list of length 100.

The relative performance of the two techniques w.r.t. the answer-size ratio is depicted in figure 5. Thicker lines represent computionally more intensive operations. Note that the complexity of AND- vs. OR- maplist is different only in the case of `id`.

[10] The *extended call/n* with $n > 1$ is known in Prolog folklore and in the libraries of various Prolog systems.

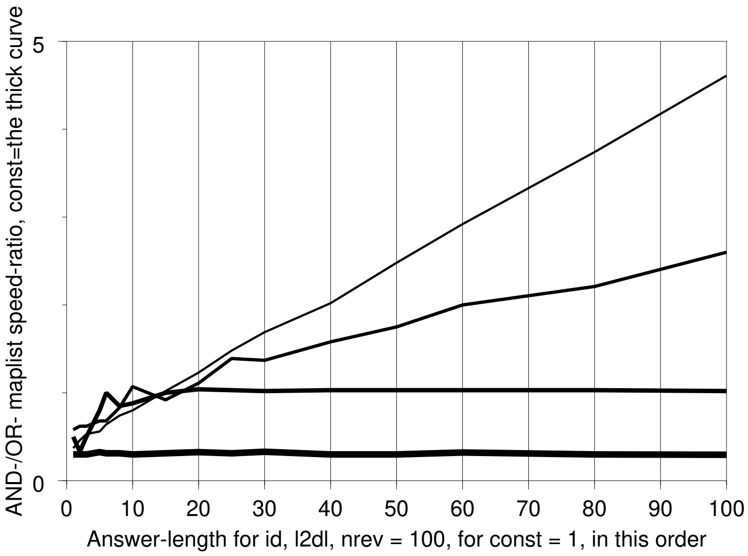

Figure 5: AND- vs. OR- speed w.r.t answer size

It can be seen that smaller answers (which are the frequent case) tend to favor OR-intensive style. In the case of a computationally intensive closure like `nrev`, obviously, differences vanish.

Clearly, the ratio between the size of the computed answer and the size of the computations is also an important factor which is neglected in this comparison.

If the computed answer is smaller, then the OR-intensive approach will benefit because copying it (once or twice) becomes relatively cheaper. Moreover, amortized garbage collection time should be accounted for in the case of the AND-intensive approach (and this becomes important when the size of the computation is much greater than the size of its computed answer), while an OR-intensive `maplist` will execute always in minimal space with implicit (generational!) garbage collection for free.

6.2 Differences in the operational semantics

The AND-intensive maplist and other similar predicates are sensitive to failure and cannot be used readily to select only a subset of the answers generated during the computation, unless extra tests are added. On the other hand, an OR-intensive solution will collect naturally only answers coming from success branches. This extra generality can be used to filter solutions while generating them instead of having to do an extra iterative step later.

6.3 Sharing in the context of AND- and OR-intensive computation.

An important issue is the sharing of common structures. In an AND-intensive implementation the programmer has control over the sharing of objects existing before the call to the higher-order predicate and even for parts of the objects created during the iteration.

Suppose maplist is called as follows:

```
?- maplist( = , [<big term>] , L ) .
```

The AND-version of maplist runs in constant time, while due to the copying operation, the OR-version has a complexity linear in the size of the `<big term>`. Similarly, after execution of

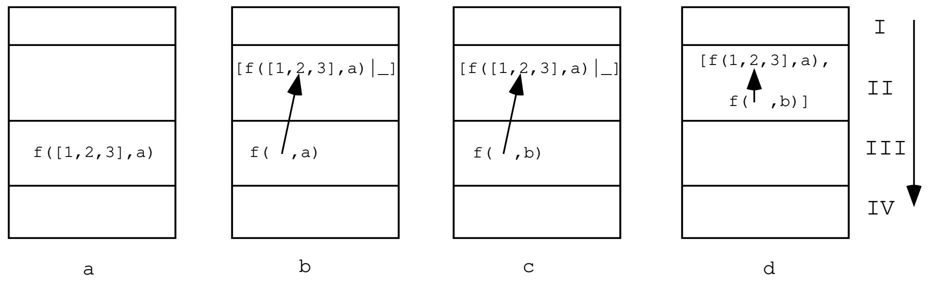

Figure 6: Term-sharing in copy-once findall

```
?- maplist(append([4]), [[1,2,3],O1], [O1,O2]) .
```

the objects O1 and O2 exhibit sharing in the AND-version, but not in the OR-version. In [4], it was pointed out that such internal sharing in the solution list, can be a major reason for transforming a backtracking program using findall/3, into an AND-intensive program which accumulates solutions without the use of findall/3. We will show how some of the internal sharing that exists between solutions of different OR-branches, can be preserved even in the findall approach. *To stress the point: internal sharing in one solution can of course be preserved by* `findall/3`*; all one needs is a copy operator that preserves sharing.* Both the implementation of `findall/3` in BinProlog and ProLog by BIM have this feature. Also, it is clear that compile-time ground terms can be allocated once and shared during the whole program execution.

The issue however, is how to preserve sharing between solutions produced in different OR-branches. We illustrate the point with an example:

```
g(f(A,B)) :- A = [1,2,3] , (B = a ; B = b) .

?- findall(X,g(X),L) .
L = [f([1,2,3],a), f([1,2,3], b)]
```

What one would like is that the list `[1,2,3]` occurs only once on the heap, as if it were constructed by the query

```
?- L123 = [1,2,3] , L = [f(L123,a), f(L123,b)]
```

We will describe how this can be achieved within the split heap implementation of `findall/3` as in BinProlog, but it should be clear that a double copying implementation can achieve the same effect, albeit with a bit more trouble.

When the first solution of `g/1` above is computed, just before copying the solution, the heap looks as in figure 6 a)[11].

The list `[1,2,3]` is in the upper half of the heap (III), and before the choicepoint created by the disjunction in the body of `g/1` (IV). After copying the solution, it could look as in figure 6 b).

The copy algorithm has left a pointer from the original term to the copied term: unification and backtracking can still be performed as usual. When the second solution is computed and just before copying it to the findall heap space (II), the heap looks as in figure 6 c).

Now the copying algorithm can recognize - from the value of the pointers - that some structures have been copied already, so that sharing between the OR-branches can be preserved. Finally, figure 6 d) shows the heap after `findall` has finished.

[11]Note that the heap grows downward in this figure.

The above example shows the sharing of a term created *after* findall was called: it is clear that also terms created *before* the call to findall/3 can be shared between solutions. As an example one can redo the pictures above for the example:

```
?- Y = f([1,2,3],_), findall(Y,g(Y),L) .
```

The above informal description needs to be completed by a specification of the conditions under which terms can be shared, or to put it differently, under which the copy algorithm can replace a term by a pointer to its copy. A sufficient condition is that a term can be shared if it is atomic, or if it is not a variable and it is not trailed since the call to `findall/3` and all of its substructures can be shared.

This applies to structures of the lower and upper heap equally. The condition about trailing, seems prohibitive for an efficient implementation, as it suggests that every term should be checked against the current trail. However, it is possible to first mark the terms that were trailed (since the call to `findall/3`) before starting the copy operation. The condition which excludes variables from being shared has several aspects: firstly, variables cannot be shared because this would change the operational semantics of `findall/3`. Secondly, an AND-style maplist can produce an output list that shares variables, but in [4] it was also noted that one of the conditions for the transformation from OR to AND is groundness of the solutions, or that non-ground solutions should be copied iteratively. The implementation of the above algorithm is part of future research, as well as the investigation of how the conditions can be weakened. Also, the close relationship with the conditions for transforming OR to AND must be investigated.

7 Competition between OR-branches with precomputed goals

Goal precomputation can also improve performace of predicates that retain only a small (possibly singleton) subset of the set of answers, as it is the case of *competing OR-branches*. *Competition* between OR-branches means here that computed answers from alternate OR-branches may exclude some other previous or future computed answers.

7.1 Implementing Bestof/3 with goal precomputation

BinProlog's `bestof/3` works like `findall/3`, but instead of accumulating alternative solutions, it selects successively the *best* one with respect to an arbitrary *total order* relation. If the test succeeds the new answer replaces the previous one. At the end, either the goal has no answers, case in which `bestof` fails, or an answer is found that is better than every other answer with respect to the order relation. The syntax is:

```
?-bestof(X, TotalOrder, Goal)
```

At the end, X is instantiated to the *best* answer. For example, the maximum of a list of integers can be defined simply as:

```
max(Xs,X) :- bestof(X, >, member(X,Xs)).
```

Note that an error condition is raised if the first argument of bestof is not free, therefore the overloading of X is safe.

`Appendix 1` contains an implementation of a precomputed goal based `bestof/3`. BinProlog's blackboard operations [11] used in the implementation are given (for portability) in a few lines which do not depend on the semantics of the underlying assert/retract (see the end of `Appendix 1`).

Reduction of some higher-order extensions to equivalent first-order constructs has been pioneered by Warren's seminal paper [16]. The Hilog system [1, 18] features a higher-order syntax with first order semantics.

The first-order semantics of `bestof` is surprisingly simple, at least in the case of definite programs. More precisely, if P and G are a *definite* program and a definite goal both possibly containing occurrences of `bestof/3` and θ is a computed answer of $P \cup \{G\}$,then θ is also a computed answer of $P' \cup \{G'\}$ where P' and G' are obtained by replacing each occurrence of the form `bestof(X,Order,Generator)` by `Generator`.

Note that the first argument of bestof should be a free (universally quantified) variable. If the generator has at least one answer bestof will behave exactly as if the OR-tree of its generator had only one successful branch, the one which actually contains the best answer with respect to the order relation.

This means that computed answers in a program containing `bestof/3` are also computed answers of the 'first-order' program P'.

As P' is obtained from P by a syntactic deletion operation we call this kind of first order semantics a *deletion* semantics. Definite programs using only higher-order primitives with a deletion semantics will have a well-defined meaning which is a subset of the meaning of their first order counterparts.

A nice consequence is that we can see `bestof/3` as a *pragma* that is guiding the *search* for a proof without interfering with the proof itself. Obviously this leads to a more declarative programming style than directly using *assert* and *retract* to obtain similar effects.

A heap-based efficient version of `bestof` can be implemented like BinProlog's heap based `findall`, ensuring that embedded calls are dealt with properly. Note that if `findall` ensures efficient *cooperation between alternative OR-branches*, `bestof` ensures efficient *competition*, without explicit use of assert and retract. They cover together the basic information exchanges between alternative OR-branches of Prolog's search tree.

The use of `bestof` not only saves space compared with a recursive loop but (as all higher-order predicates do) frees the programmer from the burden of *explicit iteration*.

7.2 Combine/4: a 'universal' higher-order primitive

As the final step of our argumentation for an alternate view that rehabilitates a 'generator-based' OR-intensive programming style which has been neglected for the common deterministic functional-style in the past, we will show how to derive some key higher-order programming constructs from an universal *goal precomputation* based primitive, `combine/4`.

Combine/4 will be used to express quite naturally both *cooperating* and *competing* higher-order predicates.

`Combine(Closure,I,Generator,Final)` combines the selected answers `I` of `Generator` using `Closure` and accumulates in `Final` the result (see the code in `Appendix 2`).

A typical query is the following:

```
?-combine(plus, X, member(X,[1,2,3,4]), S).
```

where `plus(X,Y,Z) :- Z is X+Y`, giving S=10.

The operational semantics of `combine/4` is quite simple: `combine/4` accumulates answers in result produced by a applying a precomputed goal to the result sofar and a new answer of the *generator*.

The reader can recognize here a well-known functional programming idiom. The major difference is that one of logic programming's major strength (non-determinism) is used. Note that Combine/4 works on an arbitrary generator instead of explicitly iterating through a particular, 'reified' data-structure.

Using a precomputed goal means also that little 'interpretative' effort is needed inside the 'loop' and that we properly use a kind of 'frame-axiom': from one answer to the other only minimal changes occur in the computation and most of the work is spent on them instead of being spent on what is known as unchanged. At engine-level, this work consists basically in restoring registers, un-trailing variables and having them reinstantiated efficiently by compiled code. Due to backtracking this is performed in space proportional with the size of the final answer with positive impact on locality of reference.

Next, to show the 'universality' of combine/4 we will express both findall/3 and bestof/3 in terms of it. As a consequence, both *cooperative* higher-order predicates like maplist/3 and *competing* ones built on top of bestof can be specified in terms of combine/4.

This somewhat contrived find_all/3 uses the list construction operation cons/3 as the Closure which combines the answers.

```
find_all(X,G,Xs) :- combine(cons, Y, (Y=[];Y=X,call(G)), Ys), reverse(Ys,Xs).

cons(Xs,X,[X|Xs]).
```

Note that this should be seen as an executable specification because the complexity of find_all is obviously not the same as that of its built-in counterpart.

The emulation of bestof/3 is done by constructing a max/3-like goal which combines two answers into the best one with respect to a given *total order* relation.

```
% X is the best answer of G with respect to TotalOrder (a closure)
bestof(X,TotalOrder,Generator) :-
      add_args(TotalOrder,[X,_],Test),
      combine(compare_closure(Test),X,Generator,X).

compare_closure(Test,X,Y,R) :-
      arg(2,Test,X),
      call(Test), !, R=Y.
compare_closure(_,X,_,X).
```

Clearly, predicates like maplist/3 can be easily emulated in terms of find_all/3 as we have already shown at the beginning of the paper,

8 Related work

A recent findall-related paper by Mariën and Demoen [4] explores reverse transformations with the objective of avoiding to use findall and to transform an OR-intensive program to and AND-intensive equivalent. As often advantages of a given technique are program-dependent, this paper should be considered complementary to [4]. Macro-expansion [3] is still the fastest way to deal with special cases of higher-order predicates when the arguments are known and size increase in code does not matter. Partial evaluators of full Prolog [6] can also be extended incorporate reduction of some higher-order predicates to first-order equivalents.

9 Conclusion

We have shown overall that the obvious deterministic forward oriented recursion based implementation of higher-order predicates is in a typical case slower than backtracking based implementation based on early structure creation and in particular on precomputation of goals.

This proves that AND-intensive, functional style is not always the best choice for logic programming, both on for reasons of efficiency and conceptual parsimony and that no-overtaxing meta-predicates should be provided to fully take advantage of the expressiveness of logic programming languages.

Moreover, our analysis has pointed out what can be done to push to their limit the implementation of higher-order primitives both in AND-intensive and OR-intensive programming styles.

Overall, our results show that higher-order programming can be done with execution speed comparable to first-order programming with adequate support from the underlying WAM and careful implementation of some key built-ins.

The optimizations that have been added to the BinProlog system to obtain the performance gains reported in the paper can be ported with minimal adaptations to other implementations.

10 Acknowledgements

People on USENET's comp.lang.prolog made valuable comments on our earlier ideas. The first author thanks for support from K.U.Leuven through Fellowship F/93/36, from NSERC (grant OGP0107411) and from the FESR of the Université de Moncton. The second author thanks the Belgian Ministry (DPWB) for support through project IT/IF/4. We also thank the anonymous referees for their useful comments.

References

[1] W. Chen and D. S. Warren. Compilation of predicate abstractions in higher-order logic programming. In J. Maluszyński and M. Wirsing, editors, *Proceedings of the 3rd Int. Symposium on Programming Language Implementation and Logic Programming, PLILP91, Passau, Germany*, number 528 in Lecture Notes in Computer Science, pages 287–298. Springer Verlag, Aug. 1991.

[2] B. Demoen and A. Mariën. Implementation of Prolog as binary definite Programs. In A. Voronkov, editor, *Logic Programming, RCLP Proceedings*, number 592 in Lecture Notes in Artificial Intelligence, pages 165–176, Berlin, Heidelberg, 1992. Springer-Verlag.

[3] S.-i. Kondoh and T. Chikayama. Macro processing in Prolog. In R. A. Kowalski and K. A. Bowen, editors, *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 466–480. The MIT Press, 1988.

[4] A. Mariën and B. Demoen. Findall without Findall/3. In D. S. Warren, editor, *Proceedings of the Tenth International Conference on Logic Programming*, pages 408–423, Budapest, Hungary, 1993. The MIT Press.

[5] M. Meier. Recursion vs. iteration in Prolog. In K. Furukawa, editor, *Proceedings of the Eighth International Conference on Logic Programming*, pages 157–169, Paris, France, 1991. The MIT Press.

[6] D. Sahlin. The Mixtus approach to automatic Partial Evaluation of full Prolog. In S. Debray and M. Hermenegildo, editors, *Proceedings of the 1990 North American Conference on Logic Programming*, pages 377–398, Cambridge, Massachusetts London, England, 1990. MIT Press.

[7] P. Tarau. A Simplified Abstract Machine for the Execution of Binary Metaprograms. In *Proceedings of the Logic Programming Conference'91*, pages 119–128. ICOT, Tokyo, 7 1991.

[8] P. Tarau. Ecological Memory Management in a Continuation Passing Prolog Engine. In Y. Bekkers and J. Cohen, editors, *Memory Management International Workshop IWMM 92 Proceedings*, number 637 in Lecture Notes in Computer Science, pages 344–356. Springer, Sept. 1992.

[9] P. Tarau. Program Transformations and WAM-support for the Compilation of Definite Metaprograms. In A. Voronkov, editor, *Logic Programming, RCLP Proceedings*, number 592 in Lecture Notes in Artificial Intelligence, pages 462–473, Berlin, Heidelberg, 1992. Springer-Verlag.

[10] P. Tarau. WAM-optimizations in BinProlog: Towards a Realistic Continuation Passing Prolog Engine. Technical Report 92-3, Dept. d'Informatique, Université de Moncton, July 1992. available by ftp from clement.info.umoncton.ca.

[11] P. Tarau. Language Issues and Programming Techniques in BinProlog. In D. Sacca, editor, *Proceeding of the GULP'93 Conference*, Gizzeria Lido, Italy, June 1993.

[12] P. Tarau and M. Boyer. Elementary Logic Programs. In P. Deransart and J. Małuszyński, editors, *Proceedings of Programming Language Implementation and Logic Programming*, number 456 in Lecture Notes in Computer Science, pages 159–173. Springer, Aug. 1990.

[13] P. Tarau and U. Neumerkel. A Novel Term Compression Scheme and Data Representation in the BinWAM. In J. P. M. Hermeneguildo, editor, *Proceedings of Programming Language Implementation and Logic Programming*, Lecture Notes in Computer Science 844, pages 73–87. Springer, Sept. 1994.

[14] K. Ueda. Making exhaustive search programs deterministic. In E. Shapiro, editor, *Proceedings of the Third International Conference on Logic Programming*, Lecture Notes in Computer Science, pages 270–282, London, 1986. Springer-Verlag.

[15] K. Ueda. Making exhaustive search programs deterministic : Part II. In J.-L. Lassez, editor, *Proceedings of the Fourth International Conference on Logic Programming*, MIT Press Series in Logic Programming, pages 356–375, Cambridge, Massachusetts London, England, 1987. MIT Press.

[16] D. H. D. Warren. Higher-order extensions to Prolog – are they needed? In D. Michie, J. Hayes, and Y. H. Pao, editors, *Machine Intelligence 10*. Ellis Horwood, 1981.

[17] D. H. D. Warren. An Abstract Prolog Instruction Set. Technical Note 309, SRI International, Oct. 1983.

[18] D. S. Warren. The XOLDT System. Technical report, SUNY Stony Brook, electronic document: ftp sbcs.sunysb.edu, 1992.

Appendix 1: bestof

```
% true if X is an answer of Generator such that
% X Rel Y for every other answer Y of Generator
bestof(X,Closure,Generator) :- var(X),!,
```

```
      add_args(Closure,[X,Y],Test),
      bestof0(X,Y,Generator,Test).
bestof(X,_,_) :- errmes('free variable expected',X).

bestof0(X,Y,Generator,Test) :-
      inc_level(bestof,Level),
      call(Generator),
      update_bestof(Level,X,Y,Test),
      fail.
bestof0(X,_,_,_) :-
      dec_level(bestof,Level),
      bb_val(bestof,Level,X),
      bb_rm(bestof,Level).

% uses Rel to compare New with so far the best answer
update_bestof(Level,New,Old,Test) :-
      bb_val(bestof,Level,Old),!,
      call(Test),
      bb_set(bestof,Level,New).
update_bestof(Level,New,_,_) :-
      bb_def(bestof,Level,New).

% ensure correct implementation of embedded calls
inc_level(Obj,X1) :-
      bb_val(Obj,Obj,X), !, X1 is X+1,
      bb_set(Obj,Obj,X1).
inc_level(Obj,1) :-
      bb_def(Obj,Obj,1).

dec_level(Obj,X) :-
      bb_val(Obj,Obj,X),
      X>0, X1 is X-1,
      bb_set(Obj,Obj,X1).

% extends a Closure to a Goal
add_args(T,Xs,TXs) :- T=..L1, append(L1,Xs,L2), TXs=..L2.

% portable emulated blackboard operations
bb_def(K1,K2,V) :- T=..[K1,K2,X], \+ on_bb(T), X=V, assert(T).
bb_set(K1,K2,V) :- T=..[K1,K2,X], \+ \+ retract(T), X=V, assert(T).
bb_val(K1,K2,V) :- T=..[K1,K2,V], on_bb(T).
bb_rm(K1,K2) :- T=..[K1,K2,_], retract(T), !.
on_bb(T) :- clause(T,true), !.
```

Appendix 2: combine

```
% combine/4: combines the selected answers I of Generator using Closure;
% accumulates in Final the result.

combine(Closure,I,Generator,Final) :-
      add_args(Closure,[SoFar,I,O],Selector),
      combine0(SoFar,I,O,Generator,Selector,Final).

combine0(SoFar,I,O,Generator,Selector,_) :-
      inc_level(find,Level),
      call(Generator),
      select_or_init(Selector,Level,SoFar,I,O),
```

```
      fail.
combine0(_,_,_,_,_,Final) :-
      dec_level(find,Level),
      bb_val(find,Level,F),
      bb_rm(find,Level),
      Final=F.

select_or_init(Selector,Level,SoFar,_,0) :-
      bb_val(find,Level,SoFar),!,
      call(Selector),
      bb_set(find,Level,0).
select_or_init(_,Level,_,I,_) :-
      bb_def(find,Level,I).
```