# Some Reflexions on Implementation Issues of PROLOG

M.Bruynooghe

Departement Computerwetenschappen
Katholieke Universiteit Leuven

Celestijnenlaan, 200 A
B 3030 HEVERLEE

## INTRODUCTION

Current interest in PROLOG is high. This papers aims at opening a discussion on implementation related issues which, in our opinion, can have a great impact on the acceptance of PROLOG as a valuable programming language. Our focus is on issues concerning users of todays PROLOG, not on implementation issues in current research on logic programming (parallellism, intelligent backtracking, special architecture, control).

The issues are:

- the bad influence of cut on programming style but its necessity, in current implementations, to obtain efficient (time and space) execution of programs.

- Can/will everyone develop good (efficient) PROLOG programs or is this an art for a small club of skillful experts.

- an observed desire for standardisation.

## 1 The influence of an implementation on programming style.

The PROLOG community has gone a long way from the first PROLOG interpreters to current compilers promising to allow efficient execution without running out of space, even for infinite queries if they are determinate.

In the first interpreters, we could distinguish two major work areas:

1 a dictionary of clauses, spoiled but slowly because "retract" does not free the space.

2 an environmentstack ("trail" included or separate) which grows until backtracking liberates it.

This resulted in "dirty" programstyle, exemplified by the following:

When you are affraid of running out of core, then:

- assert your useful results.
- fail and backtrack.
- restart, picking up your useful results and retract their assertion.

It has been learned to separate the global/copystack from the environmentstack, to keep the environmentstack small by exploiting determinism and to apply garbage collection on the copystack. Also compilation techniques have been developped to obtain more efficiency.

Does the combination of all these features provide the paradise for the purists among logic programmers willing to apply such an advanced programming technique as the usage of abstract data types ?

Let us look at a simple example:

A procedure Partition(x, l, l1, l2) which separates a list l into a list l1 of elements less than or equal to x, and a list l2 of elements greater than x.

The datastructure can be implemented as follows:

Empty(l) : a test to see whether a list l is empty, also to initialize an empty list.

A possible realisation is Empty(Nil) <- (another one could be with difference lists: Empty(d(x,x)) <- )

Select(l, x, tail): a list l is separated into its first element x and its remainder (tail) (Fails for an empty list)

Realisation: Select(x.l, x, l) <-

Construct(x, tail, l): a list l is constructed with first element x and remainder tail

Realisation: Construct(x, l, x, l) <-

Now Partition can be defined:

```
Partition(x, l, l1, l2) <- Empty(l), Empty(l1), Empty(l2)

Partition(x, l, l1, l2) <- Select(l, e, l'), e <= x,
                           Construct(e, l1', l1), Partition(x, l', l1', l2)

Partition(x, l, l1, l2) <- Select(l, e, l'), e > x,
                          Construct(e, l2', l2), Partition(x, l', l1, l2')
```

To my knowledge, the best of all existing PROLOG systems cannot prevent that a heavy price is paid for this programming style:

- Efficiency: the recursive calls contain 4 calls instead of 2, the number of logical inferences required to obtain a solution is doubled.

- Space: current implementations are unable to recognize the determinism of the above program. Half of the calls will be considered as nondeterministic, backtrackpoints will be created and will stay on the environmentstack. Completion of a partition call will not free the environmentstack. As a consequence, references to the global/copystack are not removed and the potential for garbage collection is severely reduced.

Preprocessing the calls to Empty, Select and Construct can improve the situation. A technique as "partial evaluation" seems promising to automate this. Doing the partial evaluation by hand, we arrive at:

```
Partition(x, Nil, Nil, Nil) <-
Partition(x, e.l', e.l1', l2) <- e <= x, Partition(x, l', l1', l2)
Partition(x, e.l', l1, e.l2') <- e > x, Partition(x, l', l1, l2')
```

This solves the efficiency problem but not the space problem (to recognize the determinism of the base case (empty list), indexing on the second argument is necessary)

To recognize determinism, a cut in the second clause is needed. At the same time, an experienced programmer will drop the condition in the third clause. We obtain:

```
Partition(x, Nil, Nil, Nil)<-
Partition(x, e.l', e.l1', l2') <- e <= x, /, Partition(x, l', l1', l2)
Partition(x, e.l', l1, e.l2') <- Partition(x, l', l1, l2')
```

This cut, an ugly feature to purists and beginners, changes the whole nature of the program. Now, the program Partition(x, l, l1, l2) cannot be used to obtain all possible merges l of l1 and l2. This restriction on the use of Partition is not declared. As in most cases, it is the purpose of the cut to inform the execution mechanism about determinism, about the opportunity to optimise the execution of the program. At the same time, the possible use of the program is severely restricted. It is a sad fact that this guidance and its accompagnying restriction are not given on a more elegant and more explicit way. It is the obscurity of the cut which restricts the use of PROLOG, beyond toy examples, to skillful expert programmers having a good understanding of the underlying implementation.

Actually, we can state two questions about the above program, (I am affraid that studying the manual of a particular implementation will not answer them):

1 Is a cut needed in the first clause to recognize the determinism of the base case (Nil)? Not recognizing the base case as deterministic has a dramatic effect on memory usage, frames are locked on the stack!

2 Will tail recussion optimisation be obtained when the second clause is selected ? It cannot be applied at the time of unification with the heading because the call is nondeterministic (the third clause provides an alternative), it becomes deterministic only after execution of the cut; at that moment, the opportunity exists to collapse two stack frames into one.

## 2 Applicability of PROLOG

The application of PROLOG is rather limited. It is only used by skillfull experts, mainly in the field of artificial intelligence. Can it be applied to more conventional problem areas, where Fortran, Cobol, Basic or Pascal are used, i.e business applications ? Recently, we conducted a few experiments. Our tools: a slow PROLOG interpreter written in Pascal (about 400 logical inferences per second on a VAX 750) , the vendors Basic and Pascal compiler.

Experiment 1

A parser was developed by rather unexperienced programmer using a compiler generator system semantic actions hand-written. The result: 2276 lines of Pascal. A parser which is roughly functionnally equivalent was written in PROLOG by an expert PROLOG programmer: 246 lines of PROLOG (factor 9). Execution time (parsing the same file): 52 s (seconds) with Pascal, 296 s with PROLOG (factor 5 - 6).

Experiment 2

A program with complex data structures. An unexperienced Pascal programmer: 2371 lines of code, a skillful PROLOG programmer: 136 lines of PROLOG (factor 17). Both programs have roughly the same functional equivalence. Execution times: Pascal: 43 s, PROLOG: 119 s. (factor 3).

Experiment 3

A complex retrieval task involving 4 files on the vendors file system (RMS). A Basic program of 170 lines required an execution time of 12.5 sec. A PROLOG program with exactly the same functionality required 70 lines (factor 12.5) and an execution time of 191 s. (factor .17). (Due to the experimental nature of the interface PROLOG-RMS, each call to the file system required a lot of additional logical inferences, also the program did more file accesses.). A, for PROLOG, simple optimisation (bringing a small part of 2 files in core) reduced the execution time to 47 s. (factor 4).

Taking into account the slowness of the interpreter, an improvement of factor 5 to 10 (2000 - 4000 LIPS) seems not difficult to obtain, using compilation techniques, further improvements are possible,this suggests that PROLOG becomes a competitive language to be used on a large scale when:

- complex data structures are to be manipulated (making use of the full power of unification)
- a substancial amount of time is spend on file accesses.

However, this requires:

- A robust interpreter, not running out of space while executing large programs and sufficiently efficient.

- Easily extendible set of evaluable predicates, allowing to develop specific predicates for specific applications. e.g.:

  * connection with a particular file system or database
  * screen management functions
  * allowing to implement components with insufficient efficiency at a lower level.

- An environment allowing the development of good PROLOG programs by mediocre programmers, this probably requires:

  * A cut free variant of PROLOG, making less an art of the writing of space efficient programs (see reflection 1)

  * An automatic optimiser based on partial evaluation techniques (see reflection 1)

  * More compile time verification (types, restrictions on the usage of procedures), (Preferably incremental and integrated in a syntax oriented editor)

- A lot of programming is involved with side effects which should be in a particular order. e.g. interaction with a terminal, producing a report. We need a well choosen set of metalevel predicates to control such side-effects. Some possibilites:

  * For_each (<u>cond</u>, <u>action</u>), e.g.:
    For_each (employee(<u>x</u>), compute_salary(<u>x</u>))

    (Definition in Prolog :
     For_each(<u>cond</u>,<u>action</u>) <- <u>cond</u>, <u>action</u>, fail
     For_each(<u>cond</u>,<u>action</u>) <-

  * Repeat_until_exit(<u>command</u>), e.g.:
    Repeat_until_exit(Read_and_process(<u>x</u>))
    The execution backtracks, reading commands until a special "exit" call is executed and the infinite backtracking is destroyed.

Although Prolog allows every expert to implement his own set of such metapredicates, some standardisation is desirable and integration of them in compiletime verification tools is needed.

## 3 Portability of Prolog programs

Currently, Prolog is not only a subject for research, but becomes accepted as a suitable language for implementing diverse applications (e.g. its role in the Japanese Fifth Generation Project and in the European ESPRIT project). This creates the problem of exchanging programs, of portability of programs. Although Edinburgh's DEC10 Prolog tends to be a de facto standard, different implementations exist and many more are likely to appear. Porting Prolog programs from one Prolog system to another is problematic due to :

- Differences in syntax. As far as the syntax is syntactical sugar for Horn clauses, automatic conversion seems not difficult. However, conversion to Horn clauses poses a problem when the syntax allows for alternatives (e.g. (P;Q) ) inside a clause, in cases where such an alternative contains different calls and one of them is a cut (scope of the cut).
- Differences in evaluable predicates. Although the core of Prolog is extremely simple, manuals are becoming wieldy, they look like christmas trees, full of evaluable predicates. Some have to do with high level control of the system, but others are extensively used inside programs.

The observable desire to use Prolog as a programming language for large projects creates a need for standardisation of syntax and evaluable predicates. Is it possible to define a minimal set of evaluable predicates and to define all extensions as Prolog procedures? For reasons of efficiency, an implementor can provide these extensions at a lower level. Taking Edinburgh's Prolog as the de facto standard is probably not optimal, some reflection on the choice seems preferable.