A CLASSICAL IMPLEMENTATION FOR PROLOG-II

Patrice Boizumault (*) (**)

The aim of this paper is to reconcile the particular needs of Prolog-II with the "classical" three stacks architecture, thereby regaining an optimal memory management. To achieve this goal, we introduce a fourth stack (stack of frozen goals) specially to manage the coroutines. And we show that all classical mechanisms used for optimally recovering space can be directly applied.

(*)Institut de Mathématiques Appliquées
3, Place André Leroy
49005 Angers Cedex

(**)LITP
CNRS LA 248
75251 Paris Cedex 05

It is a remarkable fact that virtually all implementations of Prolog use an architecture with three pushdown stacks [WARR77,83], [BRUY76,80], [MELL80], [McCAB81], [CLOC85]. The outstanding exception is Prolog-II, of Marseilles, as described in Van Caneghem's Thesis [VnCAN84]. It may be shown that the three stacks technique is a way to achieve an optimal memory management [BOIZ85], so that Prolog-II is less efficient in this respect.

The aim of this paper is to reconcile the particular needs of Prolog-II with the "classical" three stacks architecture, thereby regaining an optimal memory management.

To achieve this goal, we propose to introduce a fourth stack (stack of frozen goals) specially to manage the coroutines. And we show that all classical mechanisms used for optimally recovering space can be directly applied.

1 HOW PROLOG-II DIFFERS FROM A CLASSICAL INTERPRETER :

Prolog-II [COLMS2,83], [KAN082], [VnCAN82], has introduced for Prolog the new concepts of coroutines and delayed evaluation (the famous predicates *dif* and *freeze*). But its architecture [VnCAN84] differs from those developed by Anglo-Saxon schools [WARR77,83], [MELL80], [McCAB81], [CLOC85].

Consequently, the space saving techniques in addition to backtrack, which are used in a classical interpreter, cannot be applied. So, Prolog-II memory management is less efficient in this respect.

Firstly, we will recall the motivations which conducted the conception of a classical interpreter. Then, we will describe Prolog-II architecture showing how a particular mode of implementation of the two predicates *dif* and *freeze*, has influenced its design. Finally, we will argue that Prolog-II memory management is less efficient than a classical one.

1.1 The three stacks architecture :

The various studies carried out for ten years now, through out the world [WARR77,83] [McCAB81] [BRUY76,80] [MELL80] have led to a

division of the working area into three parts :

- . A local stack.
- . A global stack or copy stack.
- . A trail.

This organisation is independant of a particular way of representing constructed terms, whether using structure sharing [WARR77], or whether copying [MELL80], [McCAB81], [BRUY76,80].

The activation records are allocated on the local stack. Each one has a control part managing continuation and backtrack, and an environment part (vector) describing variable values.

The second stack is used for representing complex terms. The distinction can be static (by structure-sharing, each environment is divided in two parts : a local and a global one [WARR77]), or made at run-time (by copying, new constructed terms are allocated in a particular copy stack [MELL80], [BRUY76,80], [McCAB81]).

The trail is used to store the addresses of variable cells which are bound to delays (such a variable is created before the last choice point and bound later). On backtracking, this stack restore the exact state of environments by resetting these variables to their initial values.

Splitting environments in half avoids dangling references on the local stack. So, space can be recovered on it in addition to backtrack, first by reclaiming activation records as soon as determinism can be detected, then by optimizing tail end calls [WARR80], [McCAB81], [BOIZ83,85].

This organisation is controled [BOIZ85] by the will of managing memory both simply (stack mechanism) and optimally (quickest space recovery). The more recently developped compilers also use the same organisation [BOWE83], [WARR83], [CLOC85].

1.2 The architecture of Prolog-II :

The working area of Prolog-II is different from those used by classical interpreters. In fact, environment parts and control parts

are completely disassociated. This choice is needless to say largely influenced by a particular method of implementing coroutines and delayed evaluation.

1.2.1 The working area :

Prolog-II uses a structure sharing representation of terms. Its working area is divided into five stacks :

- . A goal stack managing continuations.
- . A choice stack associated to the backtrack.
- . An environment stack.
- . A trail.
- . An equation stack.

We will not discuss here the equation stack which allows infinite trees unification [COLM82,83]. If we meet the indispensable trail, on the other hand, that which corresponded to the local and global stacks of Anglo-Saxon architectures, is now separated into three areas. There are two main reasons for this : the putting aside and the realising of the frozen goals.

1.2.2 The waiting goals representation :

Freezing a goal consists of binding the corresponding variable to a molecule (pair of pointers). The first pointer indicates the goal, and the other its execution environment. But a same variable may be bound to more than one frozen goal.

Prolog-II [VnCAN84] represents this association by a list stored in the environment stack. Let x be a frozen variable, and l its already existent list of waiting goals : every new goal b put aside, is added to the head of l . This insertion is done by allocating two records at the top of the environment stack. The first describes the goal b and its execution environment, the other allows to have access to l .

So, the environment stack is not only used to store the bindings like in an usual implementation, but also to build the different lists of frozen goals.

1.2.3 Frozen goals set off :

When a unification succeeds, the interpreter refers to the list of the goals that it must now activate. This list is constituted by pairs of pointers indicating the goals to execute and their environments. This phenomenon disturbs the usual mechanism of depth-first proof, where the next goal to activate was up to there described by the caller's activation record.

The solution proposed by M Van Caneghem [VnCAN84] consists in splitting the control parts into the goal stack managing continuations, and the choice stack associated to backtrack. Each record allocated on the goal stack has two fields : the next goal to execute, and its environment. Like that, when the body of a clause is activated, each tail literal is pushed on the goal stack with its corresponding environment.

So the control management of Prolog-II is different of a classical one which stores determinate records and choice records both in the local stack.

1.3 Prolog-II memory management is less efficient :

Prolog-II memory management is less efficient than a classical one. In fact, no space recovery in addition to backtrack is henceforward allowed on the environment stack, on account of the appearance of dangling references (Prolog-II does not distinguish global variables from local ones). This fact is furthermore strengthened by the choice of storing the lists of waiting goals in the environment stack.

The absence of environments space recovery leads determinate programs to overflow this area. It was this fact which motivated the distinction between local and global stacks in a classical interpreter.

2 A CLASSICAL IMPLEMENTATION FOR PROLOG-II :

We propose a solution which integrates dif and freeze into a classical interpreter, and at the same time, allows us to benefit directly from the initial space saving techniques. We use a

structure-sharing representation of terms.

The main idea which guided our study was to keep, in spite of the putting aside of frozen goals, the properties allowing space recovery in addition to backtrack. This leads us to add a fourth area (the stack of frozen goals). This manages the setting aside of goals and lets the interpreter behaves the same way in the absence of dif and freeze.

Firstly, we will describe the architecture of the working area. Then, we will implement the delayed evaluation mechanism. Lastly, we will demonstrate that all initial space saving techniques, not only remains, but are directly applicable.

2.1 The working area :

We introduce a fourth stack in the classical architecture :

- . A local stack.
- . A global stack.
- . A stack of frozen goals.
- . A trail.

Activation records are always allocated on the local stack. Each one owns a control part and an environment part. We add to every choice point a further control field indicating the top of the frozen goals stack before its building. On backtracking, this extra field is used to restore the initial state of the frozen goals stack.

The global stack is always used to store global parts of environments. We extend Warren's definition of a global variable [WARR77] in order that each variable appearing as an argument of a predicate freeze will be global.

Example :

```
p(x1,x2,x3) -> q(x2) freeze(x1,s(x1,x3));
  x1 and x3 are classically global and x2 local.
```

```
p(x1,x2,x3) -> q(x2) freeze(x1,s(x2,x3));
  x2 and x3 are classically global, but x1 becomes global
  even though it doesn't appear in a complex term.
```

So, a global variable is from now either a Warren's one, or the first argument of a freeze predicate. The aim of this extension is both to avoid references from the local stack to the frozen goals stack (a frozen variable will be always a global one), and to make strictly global the execution environment of a frozen goal (all variables appearing in a frozen goal are global).

The frozen goals stack represents the associations between frozen variables and waiting goals. It is a push-down list allocated by the binding algorithm, and reclaimed on backtracking.

The trail preserves its first use, i.e. resetting to their old values, variables which are bound to delays. But each record allocated on the trail is henceforward a molecule of two pointers : the first one always indicates the address of the cell representing the variable, and the other describes its initial value. In fact, we have two sorts of unbound variables : the "true" associated to undef, and the frozen associated to a list of waiting goals.

2.2 Implementation :

With the introduction of the frozen stack, it becomes easy to take into account the specific requirements of the setting aside and the setting off of the frozen goals.

After having presented the modifications to be carried out in the initial binding algorithm, we will describe the methods of freezing a goal, and the management of the frozen goals stack.

2.2.1 The binding algorithm :

The binding algorithm is of prime necessity to validate the space saving techniques in addition to backtrack.

Its behaviour is exactly the same as Warren's algorithm [WARR77] for all bindings without frozen variables. On the other hand, we must adapt it to manage bindings with such variables.

Let x be a frozen variable, l its waiting goal list, and y the

term we want to unify x :

```

Begin
  Let t the dereferenced value of y [WARR77] :
  If t is an atom or a structure
    Then x is bound to t
      l1 is setting off
  Else (* t is an unbound variable *)
    if t is a "true" free variable
      Then t is bound to x
    Else (* t is a frozen variable *)
      let l2 the waiting goals list of t
      bindf1(x,l2,l1)
      t is bound to x
End

```

The frozen variable x has the same behaviour as an other global variable when t is not also a frozen variable [WARR77], [BOIZ85]. On the other hand, unifying two frozen variables x and t leads to link them together (t is bound to x) and to join the two lists of waiting goals l1 and l2, binding the result to x by the procedure bindf1.

2.2.2 The evaluable predicate freeze :

The evaluable predicate freeze(x,b) postpones the execution of the goal b until x is bound.

```

Begin
  Let t the dereferenced value of x [WARR77] :
  If t is an atom or a structure
    Then b is activated
  Else (* t is a global unbound variable *)
    If t is a "true" unbound variable
      Then bindf0(t,b,nil)
    Else (* t is a frozen variable *)
      Let l the waiting goals list of t
      bindf0(t,b,l)
End

```

As in the binding algorithm, we distinguish t truly free, from t frozen variable. In the first case, we create a new waiting goals list for b, in the other one, we insert b at the head of the existent list l.

2.2.3 The management of the frozen goals stack :

Three fields records are pushed at the top of the frozen goals stack, either during the execution of the predicate freeze (bindf0), or when unifying two frozen variables (bindf1). On backtrack, the corresponding field of the last choice point is used to restore the initial state of this stack.

2.2.3.1 Procedure bindf0(x,b,l) :

The procedure bindf0(x,b,l) first allocates a new record at the top of the frozen goals stack. This record is as follows :

- . A pointer to the waiting goal b.
- . A pointer to the execution environment of b.
- . A pointer to the waiting list l.

Then the procedure links x (which is compulsorily global), by creating in the second part of the cell representing x in the (global) environment, a pointer to this new record, while the first part is always associated to undef like another free variable.

2.2.3.2 Procedure bindf1(x,l1,l2) :

The procedure bindf1(x,l1,l2) first allocates a new record at the top of the frozen goals stack :

- . A pointer to the waiting list l1.
- . unused (l1 is already a frozen goals record).
- . A pointer to the waiting list l2.

Then x is linked to this new record as in the previous procedure.

So, the first field of a frozen goals stack record references the code area (bindf0) or the frozen goals stack (bindf1). The second one is a pointer to the global stack. Finally, the third is always a pointer to the frozen goals stack. So, the frozen goals stack never references the local stack.

2.3 Example :

```

test -> p(x) q(x);
p(x1) -> freeze(x1,s(x1)) freeze(x1,t(x1));
q(alpha) ->
s(alpha) ->
t(alpha) ->

```

Let see the binding state after execution of the first predicate freeze : (x is a local variable while x1 is a global one).

When the predicate freeze(x1,s(x1)) is activated, bindf0 allocates a first record on the frozen goals stack. Then x1 is linked to this record while the first part of its cell in the global environment is always undef.

The execution of the second freeze predicate builds a new record at the top of the frozen goals stack. Then x1 is linked to its new value. When q(x) is proved, x1 gets the alpha non-variable value. So the waiting goals list is activated: t(x1) and s(x1) are successively proved.

2.4 Space recovery in addition to backtrack can be directly applied :

All mechanisms allowing an optimal memory management (reclaiming activation records as soon as determinism can be detected, optimizing tail end calls) can be directly applied. This is due to both introducing a fourth stack, and extending Warren's global variable definition.

The local stack behaviour is exactly the same as a classical one. First, the distinction between local and global variables avoids dangling references on the local stack. Then, as unification algorithm privileges binding in the sense local-global, we are insured against references from global stack to local stack.

More, records allocated on the frozen goals stack neither use a local stack information. All variables appearing in a predicate freeze are global, so the execution environment of a waiting goal will be always global. This prevents from references from the frozen goals stack to the local stack.

Finally, the local stack never references the frozen goals stack (A frozen variable is always global).

Space recovery in addition to backtrack is from now allowed on the local stack, as in a classical interpreter. More, all these mechanisms can be directly applied without any modification, as the local stack preserves an identical behaviour.

Our solution contrary to Prolog-II, reconciles a classical architecture with the particular needs of coroutines and delayed evaluation, thereby regaining an optimal memory management. All these improvements have been added to our Prolog interpreter Vlog (Prolog using Vlist [GREU77,82]). Prolog programs using dif and freeze can now execute without any restraint due to a possible environment stack overflow.

Acknowledgement :

We are grateful to J.F. Perrot for having inspired and commented this work.

References

- Boizumault P., Sur la transformation de l'appel terminal en itération dans un interprète Prolog, 2ndes journées Prolog du CNET, M. Dincbas éditeur, (1983).
- Boizumault P., Etude de l'interprétation de Prolog Réalisation en Lisp Thèse de 3ème cycle, Univ. Paris VI, (1985).
- Bowen D.L., Byrd L.M., Clocksin W.F., A portable Prolog compiler, Logic Programming Workshop'83, (1983).
- Bruynooghe M., An interpreter for predicate logic programs, Report CW10, Katholieke Universiteit Leuven, (1976).
- Bruynooghe M., The memory management of Prolog implementations, Proc. Logic Programming Workshop, Debrecen, (1980).
- Clocksin W.F., Design and simulation of a Prolog sequential machine, New Generation Computing, vol 3, n°1, (1985).
- Colmerauer A., Prolog-II: Manuel de référence et modèle théorique, G.I.A., Univ. Aix-Marseille, (1982).
- Colmerauer A., Kanoui H., Van Caneghem M., Prolog bases théoriques et développements actuels, TSI vol 2 n°4, (1983).
- Greussay P., Contribution à la définition interprétative et à l'implémentation des Lambda-langages, Thèse, Univ. Paris VII, (1977).
- Greussay P., Le système Vlistp. Documentation en ligne, Greco de Programmation, (1982).
- Kanoui H., Manuel d'exemples de Prolog-II, G.I.A., Univ. Aix-Marseille (1982).
- Mac Cabe F.G., Micro-Prolog programmer's reference manual, Logic Programming Associates Ltd, (1981).
- Mellish C.S., An alternative to structure-sharing in the implementation of a prolog interpreter, Proc. Logic Programming Workshop, (1980).
- Van Caneghem M., Manuel d'utilisation de Prolog-II, G.I.A., Université Aix-Marseille, (1982).
- Van Caneghem M., L'anatomie de Prolog-II, Thèse, Univ. Aix-Marseille, (1984).
- Warren D.H.D., Implementing Prolog, D.A.I. research reports 39/40, Univ. Edinburgh, (1977).
- Warren D.H.D., An improved Prolog implementation which optimizes tail recursion, Proc. Logic Programming Workshop, Debrecen, (1980).
- Warren D.H.D., An abstract Prolog instruction set, Technical note 309, S.R.I. International, (1983).