

Partial Evaluation in Prolog: Some Improvements about Cuts and Control

M. Bugliesi F. Russo

ENIDATA S.p.A *

Abstract

Two main aspects of Partial Evaluation for Prolog programs are considered: treatment of cuts and control of recursion. The analysis about cut is exhaustive: we consider occurrences of cut within both conjunctions and disjunctions. We show which restrictions are necessary to safely deal with cut in Partial Evaluation and which transformations are allowed. We define a set of conditions for compile-time execution and removal of cuts. The safety of these conditions is formally proved. Some interesting results about the impact of mode inference techniques within this framework are finally drawn.

As for control issues, we address a new approach to the problem of detecting infinite derivation paths. It is based on a theoretical characterization of non-termination and provides a general technique whose effectiveness does not depend on the structure of program domains.

1 Introduction

The technique of partial evaluation (PE) was first developed within the functional programming paradigm and was then introduced in logic programming.

The basic idea was to develop an automatic tool for program transformation and optimization. PE was first conceived as a *source-to-source* transformation technique, based on a compile-time execution, which does not change the semantical behaviour of programs and reduces the number of computational steps needed during run-time execution.

Further applications have shown the relevant role of PE in two distinct fields of logic programming, *data base query optimization* [VD87] and *metainterpreter specialization* [?].

The basic transformations on which such techniques rely are well-known and consist of :

- unfolding (in-line substitution) of procedure calls
- forward and backward propagation of data structures
- evaluation of built-in predicates whenever possible

Special evaluation schemes, such as lazy evaluation, are needed to provide propagation of partially instantiated data structures in functional languages; in logic languages unification automatically supports this feature.

PE for *pure* Prolog is in fact nearly trivial since two of the basic mechanisms, unfolding and forward-backward data structure propagation, are already supported by the underlying interpreter. Within this framework PE can be understood as follows.

Given a logic program P, a goal G and a set of *stopping conditions* S, partially evaluating G wrt P consists of expanding each derivation path in the proof tree for P and G till a success or a failure or a stopping condition arises along the path.

The set of stopping conditions S is to be understood as a set of rules that prevent further expansions when entering an infinite derivation path. Therefore the PE of P wrt G can be viewed as a *partial* SLD-tree of the form :

*This work was partially supported by the ESPRIT project P973



where each G_i is either the empty clause or a conjunction of stopped literals (according to the rules in S).
The tree corresponds to the transformed program:

$$\begin{array}{l}
 G\lambda_1 : -G_1 \\
 \\
 G\lambda_2 : -G_2 \\
 \\
 \dots \\
 \\
 G\lambda_n : -G_n.
 \end{array}$$

where, in turn, the i -th clause corresponds to the i -th derivation path in the SLD-tree for P and G and λ_i is the corresponding substitution.

In case of pure Prolog the above transformation scheme can be simply defined in terms of unfolding: when a stopping condition arises for an atom in a conjunction, it is sufficient to stop unfolding that atom and go ahead with the rest of the conjunction.

The crucial problem is the definition of the set of stopping conditions in case of recursive programs. In fact, since the problem is undecidable, finding a good approximation of the optimal solution is a most relevant issue for the development of refined PEs.

Furthermore this simple scheme is not feasible for practical implementations: real Prolog programs, with all Prolog's extra-logical features need in fact more refined schemes in order to get *correct* and *complete* [LS87b] PEs.

Many authors in the literature [VD87] [O'K85] [LS87a] [?] [?] [FLZ88] have addressed these issues and proposed partial solutions.

The aim of this paper is to give a further contribution in both these directions and focus the attention especially on :

- treatment of cut and
- control issues

Our analysis about cut is exhaustive: we consider occurrences of cut within both conjunctions and disjunctions; we show which restrictions are necessary to safely deal with cut in PE and, on the other side, which transformations are allowed. We define a set of conditions for compile-time execution and removal of cuts and prove their safety. The role of mode-inference techniques within this context is also discussed and some interesting results about safe backward propagation are derived.

Our approach to control issues is fairly new and it is based on a theoretical characterization of non-termination for logic programs which is a suited refinement of the ideas presented in [VP86]. The relevance of the approach is that it provides a general method to define an appropriate set of stopping conditions for any given source program and does not rely on any *meta-knowledge* on the structure of the program domain.

The rest of the paper is organized as follows: in section 2 we focus on control issues; in section 3 we address the problem of cut and define a new PE scheme based on unfold/fold transformations. We then derive the main results. Finally in section 4 we discuss the role of mode inference within this framework.

2 Control Issues

The detection of infinite derivation paths within proof trees is one of the most relevant aspects of PE for recursive programs. Since the problem is undecidable, defining sets of stopping conditions for unfolding that provide good approximations of the optimal solution is crucial for more refined PEs. Various approaches to this problem have been explored in the literature but none of them seems to be completely satisfactory.

Some researchers [FLZ88] have concentrated in finding exact solutions for specific sub-classes of recursive programs. This approach, however, has only succeeded for very restricted classes of programs and furthermore these solutions seem to be hardly generalizable to more interesting cases.

The most common approach found in the literature (see [?] for example), is to define a set of approximate conditions for termination that are safe in the general case. The idea is to find some well-founded ordering in the sequence of recursive calls and to allow the unfolding of such calls as long as they are appropriately ordered.

The main problem with this method is the trade-off between conservativeness and safety. Again it seems difficult to define a safe ordering which does not prove to be too conservative in most cases. Choosing a general ordering is hard since ordering relations depend on program domains and different programs may work on completely different domains. Consider for example programs and metaprograms. There is no common structure between the domain of a meta-program — which is the Herbrand Base of the corresponding object program — and that of a *standard* program which is, in turn, the Herbrand Universe over its function and constant symbols. The use of the same relation for a metaprogram and a standard program would then probably bear no meaning for one of the two cases, even if safe for both.

Some degree of metaknowledge over the program domain seems therefore necessary for this method to work satisfactorily.

The main motivation of our approach comes from the idea that PE should be an automated, general-purpose technique and therefore should not rely on user provided metaknowledge. The solution we propose here is derived as a suited refinement of the results on a theoretical characterization of nontermination in Logic Programming which is described in [VP86].

Given a program P and a goal G , the idea is to compute the set $Term_n(P, G)$ of the queries for P which are raised by G and terminate in (less than) n steps.

$Term_n(P, G)$ is defined so that, for each n , it serves as a safe approximation of the limit set $Term(P, G)$ of the terminating queries for P and G . An appropriate degree of approximation can be chosen in each specific case of partial evaluation, since it doesn't need to be wired into the method itself. Furthermore, since the computation of $Term_n(P, G)$ makes no assumption on the structure of the program domain, this method can serve as the basis for the definition of a proper set of stopping conditions for *any* given program. We will in fact use $Term_n(P, G)$ to check nontermination for recursive calls during unfolding. More precisely we will say that a recursive call is safely unfolded if the corresponding atom belongs to $Term_n(P, G)$.

In [VP86] the authors take a more general approach: they give a constructive definition of $Term(P)$, the set of *all* terminating queries for a given program P . Such set is iteratively computed via successive approximations from the sets $Term_n(P)$ of the queries that terminate in n steps.

Given a program P , they define :

- Λ_P to be the set of all the constant, function and predicate symbols in P , plus an enumerable set of distinct variable symbols;
- A_P to be the *universe of the atoms* of P over Λ_P , i.e. the set of all the possible atoms formable from the alphabet Λ_P such that the atoms are unique up to variable renaming;
- for any set S over Λ_P , the set \overline{S} to be the *substitution closure* of S , i.e. the set of all the substitution instances (over A_P) of S .

The set $Term_n(P)$ is computed as the complement, inside A_P , of the set $N_n(P)$ of the queries whose solution takes more than n steps. Formally :

$$Term_n = A_P \setminus \overline{N_n}$$

The set N_n , in turn, is iteratively computed starting from A_P . At each step N_n contains all the atoms q in A_P such that there exists a clause in the program whose head unifies with q , whose body contains a possibly nonterminating call, and each literal on the left-hand side of that call succeeds in $n - 1$ steps.

- $N_0 = A_P$,
- $N_n = \{q \in A_P : \exists (q : -r_1, \dots, r_n) \in \overline{P}, \exists i \text{ such that } r_i\theta \in N_{n-1} \wedge r_1, \dots, r_{i-1} \vdash_{n-1}^\theta \square\}$.

Here the relation \vdash_n^θ denotes a step-by-step SLD-derivability relation. Namely $Q \vdash_n^\theta Q'$ if $Q'\theta$ is derived from Q after n steps.

Even though formally correct, this formalization rises some doubts about its effectiveness. In fact, when the alphabet Λ_P contains function symbols, the sets A_P and \overline{P} are not finite and must be safely approximated. The definition of safe approximations of these sets seems to be the crucial point of the whole construction.

What we can do, instead, is to have a refined method that directly computes the set of terminating queries starting from the empty set. In a PE system, we can do even more: instead of having a *flat* computation of all terminating queries, we can restrict such computation to the set $Adm(Q)$ of those queries which are *admissible* with respect to the given initial goal Q .

An admissible query for a program P and a goal G is an atomic goal selected for resolution during the evaluation of G . Formally we say that

- $q \in Adm(G)$ iff $\exists n$ such that $G \vdash_n^\theta (r_1, \dots, r_i)$ and $q = r_1\theta$.

Any approximation $Adm_l(Q)$ of $Adm(G)$ is now directly obtained by choosing a bound l over the number of steps in the computation of the relation \vdash_n^θ

Once such bound has been fixed, we can use $Adm_l(Q)$ as a safe representative of the set of *possibly* terminating queries and then iteratively compute $Term_n(P, Q)$ choosing the candidates for termination in $Adm_l(Q)$.

What we get is then a sort of *goal-driven* computation of $Term(P, Q)$ which starts from the empty set and incrementally constructs the sets $Term_n(P, Q)$:

- $Term_0(P, Q) = \emptyset$
- $Term_n(P, Q) = \{r \in Adm_l(Q) : \forall (r : -b_1, \dots, b_s) \in \overline{P} \text{ such that } (i) \vee (ii)\}$ where
 - (i) $\exists k \leq s : b_1, \dots, b_s \in FAIL_{n-1}$
 - (ii) $\forall i \leq s \forall j \leq i \ b_1, \dots, b_i \vdash_{n-1}^\theta \square \wedge b_{j+1}\theta \in Term_{n-1}(P, Q)$

Here the set $FAIL_n$ denotes the set of queries which have a finite-failure proof tree of depth $\leq n$. The two conditions (i) and (ii), in turn, simply define the conditions for termination. In fact they state that the body of a clause terminates in n steps iff

- either it fails in (less than) n steps, or if
- for each *initial segment* of the body which succeeds, the rest of the body terminates in (less than) $n - 1$ steps.

The construction of $Term_n(P, Q)$ is now effective; we first compute the approximated set of admissible queries and then iteratively select the terminating ones within it.

A final remark concerns the extension of $Term_n(P, Q)$ to all the queries to non-recursive predicates. This is most meaningful in our method since we only use $Term_n(P, Q)$ to control unfolding during PE. Non-recursive queries can in fact become nonterminating only if they call recursive predicates. To our purposes it is then sufficient to capture the nontermination of recursive queries only. Consider for instance the program

$$\begin{aligned} p(X) & :- r(X). \\ r(X) & :- q(X). \\ q(s(X)) & :- q(X). \end{aligned}$$

$p(X)$, $r(X)$ and $q(X)$ are all nonterminating queries; yet only $q(X)$ will be classified as nonterminating since it is the only query for a recursive predicate. It is worth noticing that $q(X)$ is the call that causes the nontermination of $p(X)$ and of $r(X)$. No stopping condition will then be imposed on $p(X)$ and $r(X)$ in partially evaluating this program.

3 Cut

Operationally, the effect of a cut is to discard certain backtrack points, so that execution can never backtrack to them [?]. The behaviour of *cut* is not universally agreed upon in all contexts. However, cuts are most frequently encountered in one of two static contexts: as part of a conjunction, or within a disjunction in the body of a clause, i.e. in one of the two following contexts :

$ \begin{array}{l} p :- \dots \\ \quad \vdots \\ p :- \dots ! \dots \\ \quad \vdots \\ p :- \dots \end{array} $	$ \begin{array}{l} p :- \dots \\ \quad \vdots \\ p :- \dots (\dots ! \dots ; \dots) \dots \\ \quad \vdots \\ P :- \dots \end{array} $
---	---

In both cases the backward points discarded will consist of all those set up by the literals on the left hand side of the cut and those relative to the remaining alternatives for *p*.

Preserving such a semantical behaviour during PE is all but trivial since it may affect the soundness and completeness of the very basic mechanisms of PE. In fact, maintaining the equivalence between the source and the transformed program involves relevant restrictions on both unfolding and data structure propagation when cuts are considered. More refined transformation schemes are actually necessary in place of the simple unfolding-based one.

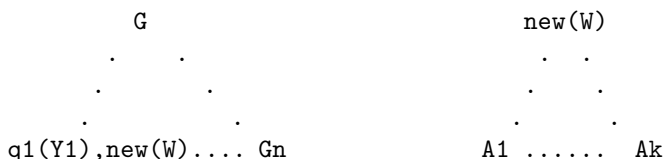
One such scheme can be constructively defined in terms of two mutually recursive functions *p_eval* and *unfold* :

- *p_eval*(*P*, *G*) raises the unfolding of each clause of *P* whose head unifies with the goal *G*,
- *unfold*(*Clause*) replaces each literal in the body of *Clause*, by the the appropriately instantiated body of each clause which defines it, till a failure or a success or a stopping condition is reached.

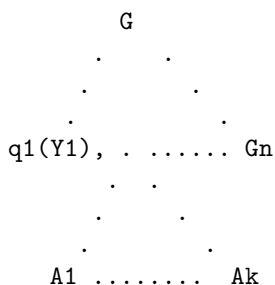
When a stopping condition occurs for a literal in the clause, the rest of the conjunction in the body is *folded* into a new atom and a new partial evaluation is launched for such atom. Namely: given a goal *G* and a clause

$$G' :- q1(Y1), \dots, qn(Yn)$$

suppose a stopping condition arises on *q1*(*Y1*). Then the conjunction *q2*(*Y2*), ..., *qn*(*Yn*) is folded by a new atom *new*(*W*) where *W* is the vector of the variables in *q2*(*Y2*), ..., *qn*(*Yn*) which occur also in { *G'*, *q1*(*Y1*) }. A new partial tree is then built for *new*(*W*). At the end of PE we will get two partial trees of the form :



The tree for the folded atom *new*(*W*) can be then *unfolded back*, i.e. linked to that for *G*, thus yielding the expected partial tree



3.1 Restrictions

Relevant restrictions are necessary to preserve the safeness of the above scheme when cuts are to be considered and they involve both unfolding and unfolding-back.

The first issue concerns stopping conditions: new restrictions are needed to get safe unfolding in presence of cuts since unfolding an atom may alter the scope of the cuts that possibly occur in its defining clauses [LS87a]. More precisely:

- **[R1]** unfolding must be inhibited for those atoms whose defining clauses contain one or more occurrences of cuts. Such atoms are to be partially evaluated by themselves.

Further problems arise when unfolding atoms that occur on the right-hand side of a cut in a conjunction. In this case in fact unfold turns out to be unsound if it yields more than one derivation path [?]. A transformation of the form

$$\begin{array}{l}
 p(X) :- !, q(X). \\
 q(T1) :- Body1. \\
 \cdot \\
 \cdot \\
 q(Tn) :- Bodyn
 \end{array}
 \quad \text{====>} \quad
 \begin{array}{l}
 p(T1) :- !, Body1. \\
 \cdot \\
 \cdot \\
 p(Tn) :- !, Bodyn.
 \end{array}$$

is unsound since the alternatives for $q(X)$ are mutually exclusive in the transformed program, due to the cut occurrence, whereas they weren't in the original program. This is actually a general issue :

- **[R2]** if a conjunction occurs on the right hand side of a cut in the body of a clause, then it must be folded and
- **[R3]** unfolding-back is allowed provided that it does not yield more than one alternative for the folded conjunction

Further restrictions are necessary to prevent unsafe backward data structure propagation, when unfolding-back folded conjunctions.

It turns out in fact that if a variable has its value committed by a cut, then it cannot be safely backward instantiated. Take for example the following program P and the goal $:-p(X)$:

$$\begin{array}{l}
 P :: \quad p(X) :- a(X), !, b(X), c(X). \\
 \quad \quad p(val). \\
 \quad \quad a(val1) :- !. \\
 \quad \quad a(X). \\
 \quad \quad b(val2). \\
 \quad \quad c(val2).
 \end{array}
 \quad \text{====>} \quad
 \begin{array}{l}
 P' :: p(val2) :- a(val2), !. \\
 \quad \quad p(val). \\
 \quad \quad a(val1) :- !. \\
 \quad \quad a(X).
 \end{array}$$

$a(X)$ is not safely unfoldable; accordingly it is stopped and its clauses are separately unfolded.

$b(X)$ is not safely unfoldable too since unfolding $b(X)$, involves backward instantiating the variable X of $a(X)$ over the cut. Notice in fact, that the two programs are not equivalent since $:-p(X)$ yields no answer for P and the answer $X = val2$ for P'.

Furthermore backward instantiation can be unsafe even in more general situations as shown below. Take for example the program

$$\begin{array}{l}
 P :: \quad p(X) :- a(X), b(X), c(X). \\
 \quad \quad a(val1) :- !. \\
 \quad \quad a(X). \\
 \quad \quad b(val2). \\
 \quad \quad c(val2).
 \end{array}$$

No cut occurs in the body of the clause for p. Backward instantiation is still unsafe, though. In fact, given the query $p(X)$, the corresponding transformed program P'

```

P' :: p(val2) :- a(val2).
      a(val1) :- !.
      a(X).

```

derives the answer $X = \text{val2}$ which is not in the success set of P .

In both cases then, the conjunction $b(X), c(X)$ must be folded and unfolding-back must be inhibited to prevent backward instantiation on $a(X)$. A final restriction over the unfold/fold/unfold-back can be stated as follows:

- [R4] unfolding back a folded conjunction is safe provided that it does not yield backward instantiation both over a cut and on stopped literals.

3.2 Compile-time of Pruning The Proof Tree

We have so far outlined the main drawbacks of the use of cut in Prolog as far as PE is concerned. It seems that a wide use of cut in programs makes partially evaluating such programs almost useless. In fact almost no form of backward propagation is safe and unfolding too must be performed very carefully.

Several solutions to these problems can be found in the current literature. Yet they all approach the problem indirectly. Some authors [O'K85] [TF85] address the issue of preprocessing the source program to eliminate cuts in favour of more structured control mechanisms such as if-then-else. Others [VD87] propose to introduce explicit mark-points to rule backtracking in the transformed program.

Our, more direct, approach is based on the idea of exploiting the control mechanism provided by cut at compile-time, i.e. to allow for a compile-time pruning of the proof tree according to the semantics of cut. This idea is also addressed in [?]. What is new here is the definition of the conditions under which this transformation is safe. Safety is in fact, once more, the crucial requirement : the set of the answer substitutions must be kept unchanged in the original program and the transformed one. To this purpose we now define the conditions for a safe *executability* and *removal* of cuts during partial evaluation.

Proposition 1

Let P be the following program :

```

(1)    p(T1) :- B1.
        :
(n)    p(Tn) :- Bn.

```

and let $\text{:- } p(T)$ be a given query.

Let i be the first index such that $B_i = \text{cond}, !, \text{rest}$ and conditions [A] and [B] below hold.

- [A] T and T_i are unifiable and $\theta = \text{mgu}(T, T_i)$ does not instantiate any variable in T (i.e. $T\theta = T$).
- [B] the resolution of cond succeeds without instantiating any variable of T , i.e. $\text{cond}\theta \vdash_{\lambda} \square$ where λ is such that $T\lambda = T$

Then the subtree of the SLD-tree for P and $p(T)$ corresponding to the clauses $(i+1) \dots (n)$ can be safely pruned at PE time.

Proof

We have only to prove that for all substitutions σ the tree for P and $p(T)\sigma$ does not contain any derivation path corresponding to the clauses $(i+1) \dots (n)$.

This is trivially true if σ is the empty substitution. Furthermore, given any substitution δ ,

- from [A] it follows that $T\delta$ and T_i are unifiable with $\text{mgu } \theta\delta$;
- from [B] it follows that $\text{cond}\theta\delta$ is also provable with substitution λ , since the proof of $\text{cond}\theta$ does not depend on any variable in T .

Then it follows that no backtracking will occur over clauses $(i+1) \dots (n)$ due to the success of cond and the cut which follows cond in clause (i) ; therefore the corresponding subtree will be accordingly pruned.

□



figure 2

3.3 Enhancing forward data-structure propagation

Executing and removing cuts is especially relevant when performing new PEs on those atoms whose unfolding has been inhibited according to restriction **R1**. If unfolding $q1(X1)$ in the clause

$$(C) \quad p(X) :- q1(X1), q2(X2), \dots, qn(Xn).$$

involves lifting cuts into the clause for p , then, see [**R1**], a new PE is launched for $q1(X1)$. If the cut that occur in the definition of $q1(X1)$ becomes finally executable during this phase, then its proof tree can be safely pruned. The resulting clauses can be exploited to propagate any computed binding for $X1$ to the rest of the conjunction.

More precisely, if the clauses produced for $q1$ are unit clauses, then we can restart unfolding (C) taking these new clauses as the actual definition for $q1$. No folding occurs for the rest of the conjunction since re-unfolding $q1(X1)$ yields no residual.

Otherwise $q1$ is stopped and the rest of the conjunction $q2(X2), \dots, qn(Xn)$ is folded into a new predicate $new(W)$. However, also in this case, the clauses which have been produced for $q1$ can serve to compute the bindings for $X1$. Let

$$\begin{array}{l}
q1(X1)s1 :- B1. \\
\vdots \\
q1(X1)sn :- Bn.
\end{array}$$

be the new clauses for $q1$ and $\{s1, \dots, sn\}$ the corresponding substitutions. A further condition must hold for $\{s1, \dots, sn\}$ to be safely propagable to $new(W)$. Namely, they must be *disjoint*, i.e.

- $\forall i, j \text{ mgu}(new(W)si, new(W)sj) = \emptyset$

If this is the case we can transform (C) into the schema:

$$\begin{array}{l}
p(X) :- q1(X1), new(W). \\
\\
new(W)s1 :- (q2(X2), \dots, qn(Xn))s1. \\
\vdots \\
new(W)sn :- (q2(X2), \dots, qn(Xn))sn.
\end{array}$$

and partially evaluate $new(W)$ wrt the newly introduced clauses. This way, we also provide forward propagation from $q1$ to $new(W)$.

Note that condition • is necessary. As a counter-example, consider the following clause:

$$(c) \quad p(x) :- q(x), rest(x, z).$$

. and let $\{x=f(a)\}$ and $\{x=f(y)\}$ be the two bindings returned by partially evaluating $q(x)$. Let finally $rest(x, z)$ be solvable with empty substitution for x and with $\{z=t\}$. Then the scheme

$$\begin{array}{l}
p(X) :- q(x), new(x). \\
\\
new(f(a)) :- rest(f(a), z). \\
new(f(y)) :- rest(f(y), z).
\end{array}$$

is not equivalent to (c) since the binding $x=f(a)$ is returned three times here and only once in (c).

What we can do in this case is to select in $\{s_1, \dots, s_n\}$, the (subsets of) substitutions which are not-disjoint and replace them with (the set of) their minimal generalization [LS87a]. When we finally get a (possibly singleton) set of substitutions that satisfy condition \bullet , we can then propagate them forward.

[LS87a] also addresses the idea of forward propagation of the minimal generalization of a set of bindings; yet it does not take into account the above condition over $\{s_1, \dots, s_n\}$. It results from some sample programs, that there is a real enhancement in performance wrt the solution proposed in [LS87a]; forward propagating data structures through conjunctions allows in fact to detect a greater deal of failure derivation paths which can be safely pruned.

This technique has been implemented in our partial evaluator as a two passes transformation, the first one performing separate PEs, the second computing the bindings to be forward propagated.

3.4 Handling disjunctions

The method proposed in section 3.2 can be further extended to handle disjunctions. The extension is indeed quite trivial if the disjunct occurs as the first component of the body of a clause. In fact, since in this case, a definition of the form

$$(*) \quad p :- (q_1, \dots ; q_2 \dots), r, s, \dots$$

is equivalent to the two following definitions

$$(**) \quad \begin{aligned} p &:- q_1, \dots, r, s, \dots \\ p &:- q_2, \dots, r, s, \dots \end{aligned}$$

we can simply split (*) into (**) and directly apply **proposition 1** to (**).

Such a simple split is no longer feasible if one or more literals occur on the left-hand side of the disjunct in the body. Consider for example the clause :

$$[1] \quad p(X) :- a(X), (b_1(X), ! ; b_2(X)), c(X).$$

and let $a(X)$ be a stopped literal. In this case the occurrence of the cut can be eventually executed to *solve* the disjunction, i.e. to discard backtrack points within the disjunction ; yet it can't be removed since its scope actually involves the whole body. Then, what we can do is :

- apply the folding rule to the conjunction $(b_1(X), ! ; b_2(X)), c(X)$ to get the new definition

$$\text{new}(X) :- (b_1(X), ! ; b_2(X)), c(X)$$

or, equivalently the two clauses :

$$\begin{aligned} \text{new}(X) &:- b_1(X), !, c(X). \\ \text{new}(X) &:- b_2(X), c(X). \end{aligned}$$

- apply **proposition 1** to eventually discard alternatives for **new**.
- unfold-back the clauses for **new** so that the scope of the cut is kept unchanged.

Note that we are forced to apply the unfold-back rule in order to keep the scope of the cut unaltered. Again this is a general rule to which one must resort in cases other than that of disjunction:

- Whenever a folded conjunction contains one or more occurrences of cut, folding must be followed by unfolding back to lift cuts into the original body.

Any form of unsafe backward propagation must be prevented as well during this forced unfold-back phase; this can be achieved through the use of explicit equalities as proposed in [VD87].

4 The impact of mode-inference techniques

Coming back to **proposition 1**, one may argue that conditions [A] and [B] can hardly be satisfied by most queries. This actually depends on the *degree of specialization* of the goal which is given as the input query to the PE system. Namely, the more this query is instantiated, the larger is the number of cases where conditions [A] and [B] are satisfied.

A significant enhancement to partial evaluation can be obtained if extra *mode* information is available to the system. This allows in fact to select the input and output segments within the tuple of parameters of the query. Consequently conditions [A] and [B] can be safely imposed only on the input segment of the queries.

Proposition 1 can in fact be restated as follows :

Proposition 1'

Let P be the following program :

- ```
(1) p(T1) :- B1.
 :
 :
(n) p(Tn) :- Bn.
```

and let  $p(T)$  be a given query. Let  $T\text{-in}, T\text{-out}$  be declared as the input and output segments of the tuple of parameters T respectively. Let  $i$  be the first index such that  $B_i = \text{cond}, !, \text{rest}$  and conditions [A] and [B] below hold.

- [A] T and  $T_i$  are unifiable and  $\theta = mgu(T, T_i)$  is such that  $T\text{-in}\theta = T\text{-in}$ .
- [B]  $\text{cond}\theta \vdash_{\lambda} \square$  where  $\lambda$  is such that  $T\text{-in}\lambda = T\text{-in}$

i.e. the head unification and the resolution of `cond` succeeds without instantiating any input variable of T.

Then the subtree of the SLD\_tree for P and  $p(T)$  corresponding to the clauses  $(i+1) \dots (n)$  can be safely pruned at PE time.

**Proof**

The proof trivially follows from proposition 1. In fact, since  $T\text{-out}$  is the output segment of T, any instance  $T\theta$  of T, which is consistent with the mode declaration for T, has the same output segment. (i.e. for any such instances  $T\theta$ , it holds that  $T\text{-out}\theta = T\text{-out}$ . As a consequence [A] and [B] hold for the complete tuple T.

□

Having mode declarations available is especially useful in partial evaluation of metaprograms. Metainterpreters provide in fact a typical example in which cuts are mostly used to get mutual exclusion between clauses over the procedure call parameter, which is in fact an input parameter. Extra parameters, if any, of the metainterpreter are usually output parameters. The treatment of cuts can therefore be most efficient in such situations.

Take for example the naive sorting program and, correspondingly, a corouter :

| Sorting Program          | Corouter                  |
|--------------------------|---------------------------|
| perm([], []).            | coroute(true, true) :- !. |
| perm([X Y], [U V]) :-    | coroute(G1, G2) :-        |
| del(U, [X Y], W),        | solve(G1, NewG1),         |
| perm(W, V).              | solve(G2, NewG2),         |
|                          | coroute(NewG1, NewG2).    |
| del(X, [X Y], Y).        | solve(true, true) :- !.   |
| del(X, [U Y], [U V]) :-  | solve(G, NG) :-           |
| del(X, Y, V).            | clause(G, B),             |
|                          | expand(B, NG).            |
| ord([]).                 |                           |
| ord([Y S]) :- ord(Y, S). | expand((G1, G2), E) :- !, |
|                          | call(G1),                 |
| ord(_, []).              | expand(G2, E).            |

```

ord(Y, [Z|S]) :- Y =< Z,
 ord(Z,S)
 expand(G, G).

```

Here `perm(L, L1)` generates a permutation `L1` of the input list `L` and `ord(L)` checks whether the list is sorted or not. Having the object-program execution guided by the metainterpreter, we get a polynomial algorithm in which a sorting check occurs as soon as a new element is added to the permuted (sub)list. The whole system suffers though all the overhead due to metainterpretation.

An equivalent, more efficient, result can be obtained by partially evaluating the metainterpreter with respect to the sorting program. Notice that for both `solve/2` and `expand/2` the second parameter is an output one. This information allow to execute and eliminate all cuts in the clauses for the metainterpreter. The corresponding *specialized* metainterpreter is the following program:

```

cor(perm([], []), ord([])).

cor(perm([A|B], [A|C]), ord([A|C])) :-
 cor(perm(B, C), ord(A, C)).

cor(perm([A|B], [C|D]), ord([C|D])) :-
 del(C, B, E),
 cor(perm([A|E], D), ord(C, D)).

cor(perm([], []), ord(A, [])).

cor(perm([A|B], [A|C]), ord(D, [A|C])) :-
 D =< A,
 cor(perm(B, C), ord(A, C)).

cor(perm([A|B], [C|D]), ord(E, [C|D])) :-
 del(C, B, F),
 E =< C,
 cor(perm([A|F], D), ord(C, D)).

del(A, [A|B], B).
del(A, [B|C], [B|D]) :-
 del(A, C, D).

```

There is no cut occurring in the clauses of the transformed program. The coroutinging policy is now completely hardwired in the resulting code which is in fact polynomial. Some run tests show that, given an input list of ten elements, this program runs 10 times faster then the metainterpreter-plus-sorting program system.

## 4.1 About safe backward propagation

A further significant use of mode analysis concerns relaxing some restrictions on backward propagation. In section 3.1 we showed that backward propagation is not safe in general if cuts are considered. We noticed in fact, that it is intrinsically unsafe to backward instantiate values over a cut since this may cause a failure to occur on the left-hand side of the cut whereas it should have occurred on the right-hand side.

Furthermore, as shown in the last example listed there, it turns out to be also unsound to backward instantiate those variables which should be unbound when the atoms in which they occur are selected for resolution.

Yet, some form of backward propagation would be desirable for the development of more refined PEs. In fact, having bindings backward propagated right to left within a conjunction, would allow for faster checks about the global success of the bindings which are incrementally computed during the left-to-right evaluation of the conjunction itself.

The relevance of mode analysis in this framework derives from its capability to (partially) determine the status of the program variables at each instant of run-time execution. In fact, as shown below, backward

propagation turns out to be safe if it only occurs on variables which are bound to be associated to ground terms during run-time execution.

**Proposition 2**

Let C be the clause :

$$(C) \quad p(T) :- \text{susp}(T'), \text{todo}(x).$$

where T and T' denote two tuples of terms and x denotes a variable. Let  $\text{susp}(T')$  be splittable into two sub-conjunctions  $\text{nongr}_x, \text{gr}_x$  such that  $\text{gr}_x$  does not contain cuts and  $\text{nongr}_x$  is bound to return a ground binding for x during run time execution.

Let's also suppose that unfolding  $\text{todo}(x)$  yields a residual Res and a substitution  $\lambda$ . Finally let t be the term that  $\lambda$  associates to x, i.e.  $\lambda \equiv \{x \leftarrow t\}$ .

Then we can safely propagate this binding for x over  $\text{gr}_x$  back to  $\text{nongr}_x$ . In other words (C) is equivalent to the following clause :

$$(C1) \quad p(T) :- \text{nongr}_x, x = t, \text{gr}_t, \text{Res}.$$

where  $\text{gr}_t$  denotes the instance of  $\text{gr}_x$  under  $\lambda$  and the explicit equality  $x=t$  is used to prevent further backward propagation on  $\text{nongr}_x$  (as addressed in [VD87]).

**Proof**

First notice that the transformation would be trivially safe if we prevented any form of backward propagation when unfolding  $\text{todo}(x)$ . In that case we would get the clause:

$$(C2) \quad p(T) :- \text{nongr}_x, \text{gr}_x, x = t, \text{Res}.$$

Therefore we can proof the equivalence of (C) and (C1) by showing that (C1) and (C2) are equivalent; what we do. More precisely, we show that either (C1) and (C2) both fail, or they yield the same binding for x.

Assume that  $\sigma$  is the substitution returned by evaluating  $\text{nongr}_x$  and let  $t_1$  the term that  $\sigma$  associates to x. Then, after evaluating  $\text{nongr}_x$ , we get

- $(x = t, \text{gr}_t, \text{Res})\sigma \equiv (t_1 = t, \text{gr}_t\sigma, \text{Res}\sigma)$  in (C1);
- $(\text{gr}_x, x = t, \text{Res})\sigma \equiv (\text{gr}_{t_1}\sigma, t_1 = t, \text{Res}\sigma)$  in (C2).

Now the equivalence between (C1) and (C2) trivially follows from the hypothesis on  $t_1$ . In fact, if  $t_1 = t$  is going to fail in (C1), then it will fail in (C2) as well.

Otherwise, let  $\gamma$  be the mgu of  $t_1$  and t. Notice that, since  $t_1$  is ground, in (C1) we get  $\text{gr}_t\sigma\gamma = \text{gr}_{t_1}\sigma$ .

Therefore, the only difference between (C1) and (C2) concerns the propagation of the binding  $\gamma$  to the rest of the conjunction. Namely, it occurs before the evaluation of  $\text{gr}_{t_1}\sigma$  in (C1) and after that in (C2). However this is clearly irrelevant as far as the equivalence of (C1) and (C2) is concerned.

□

## 4.2 Concluding Remarks

In this paper we have dealt with two main issues of partial evaluation in Prolog.

Our approach to the recursion problem is fairly new and interesting . The application of Vasak and Potter's ideas provide in fact a good method to check non-termination during PE. Our solution still needs some more development and experimentation in order to estimate its actual performance for practical cases of Partial Evaluations. Some interesting and stable results have been also reported for the treatment of cut and more efficient PE's. The relevance of mode inference techniques has been finally addressed within this context.

## References

- [FLZ88] L. Fanti, G. Levi, and S. Zanobetti. Correttezza e completezza della valutazione parziale di programmi logici. In *G.U.L.P.*, 1988.
- [LS87a] G. Levi. and G. Sardu. Partial evaluation of metaprograms in a multiple world's logic language. In *Workshop on Partial Evaluatio and Mixed Computation*, 1987.

- [LS87b] J. W. Lloyd and J. C. Shepherdson. Partial evaluation in logic programming. Technical Report CS-87-09, Dept. of Comp. Sc. Univ. of Bristol, Dept. of Math., Univ. Walk Bristol, December 1987.
- [O'K85] R. A O'Keef. On the treatment of cuts in prolog source-level tools. In *Proc. Symp. on Logic Programming*, 1985.
- [TF85] A. Takeuchi and K. Furukawa. Partial evaluation of prolog programs and its application to meta programming. In *Logic Progr. Conf (Tokyo)*, 1985.
- [VD87] R. Venken and B. Demoen. A partial evaluation system for prolog: Some practical considerations. In *Workshop on Partial Evaluation and Mixed Computation*, 1987.
- [VP86] T. Vasak and J. Potter. Characterisation of terminating logic programs. In *Proc. Symp. on Logic Programming*, 1986.