Towards Efficient Partial Evaluation in Logic Programming

David A. Fuller, Sacha A. Bocic, Leo E. Bertossi

Computer Science Department
Pontificia Universidad Católica de Chile
PO Box 306
Santiago 22
Chile

Tel. +(56-2) 552 2375 x. 4440
Fax +(56-2) 552 4054

E-mail (internet): {dfuller, sacha, bertossi}@lascar.puc.cl

ABSTRACT: Partial evaluation is a symbolic manipulation technique used to produce efficient algorithms when part of the input to the algorithm is known. Other applications of partial evaluators such as universal compilation and compiler generation are also known to be possible. A partial evaluator receives as input a program and partially known input to that program, and outputs a residual program which runs at least as efficient as the input program with restricted input.

In this paper we study the case where both the input and residual programs are logic programs, being the partial evaluator itself a logic program. Up to now, partial evaluators have failed to produce large "non-toy" examples. Here we present extensions to the partial evaluation operations which will allow us to produce more efficient residual programs using less computing resources during partial evaluation. Examples are given.

KEYWORDS: Partial evaluation, logic programming, symbolic manipulation.

1. INTRODUCTION

Partial evaluation has been known as a program optimization technique since the early seventies [8, 3]. Only in the last few years, however, its use has been made possible. In this paper we will define extensions to partial evaluator which will made possible to optimize large programs when part of its input is known at compile time (i.e. partial evaluation time). We will also present some examples showing the power of this optimization technique. One of the examples will also describe a partial evaluator working as a universal compiler, producing efficient target programs.

Partial evaluation of a subject program wrt known values of some of its input parameters will result in a residual program. Running the residual program on any remaining input values will yield the same result as running the original subject program on all of the same input values. A residual program, therefore, is a specialization of a subject program with respect to known values of some of its parameters.

In expert systems and logic programming, this technique is now known as a way of eliminating the extra layers of interpretation produced by meta-interpreters [16, 15, 10].

In this section we will describe the equations defining partial evaluation and the meaning they have in the optimization of algorithms, automatic production of compilers and compiler generators. Let P be a programming language with a semantic function $\mathcal{P}$: `D → (D → D)`, where D is a set whose elements may represent programs in various languages, as well as their input and output. Note that $\mathcal{P}$ is a partial function, since programs may have infinite loops. A residual program r is defined by:

$$\mathcal{P}\texttt{(p) (<d1,d2>)} = \mathcal{P}\texttt{(r) (<d2>)}$$

where d1 is the known parameter, d2 the unknown, and p and r are valid programs in language P. Let *mix* and int be two valid programs in P, where *mix* is a partial evaluator, int is an interpreter for programs in language L, and l a valid program in L. The description of the three residual programs is given in the following equations:

$$\text{target} = \mathcal{P}(mix)(\langle \text{int}, \text{l} \rangle) \quad (1.1)$$

$$\text{comp} = \mathcal{P}(mix)(\langle mix, \text{int} \rangle) \quad (1.2)$$

$$\text{cocom} = \mathcal{P}(mix)(\langle mix, mix \rangle) \quad (1.3)$$

The residual program target is generated by running the partial evaluator *mix* with input parameters int and l, and corresponds to the compiled program l into P. It should be noted that this is exactly how partial evaluation is being used in meta-interpretation. This equation also shows the way of using a partial evaluator as a program optimizer. In this case, program int represents the input program to be optimized, l corresponds to the partially known values to int, and target is the optimized program int wrt l.

The second equation shows the production of a compiler comp by running *mix* with the inputs *mix* and int. In other words, we are specializing the interpreter int for language L into a compiler for language L. A *mix* partial evaluator with the ability to apply to itself will be called a *mix* self-applicable partial evaluator. As explained in a latter section, not every partial evaluator has this ability.

The third equation describes how to produce the residual program cocom, a compiler-compiler or compiler generator, by running a *mix* self-applicable partial evaluator with its two inputs being copies of *mix* . Thus, when running cocom it will produce a compiler for a language L when an interpreter for language L is given.

We have briefly summarized the theory of partial evaluation and self-application. The first *mix* self-applicable partial evaluator was described in [9], using a first order subset of LISP. In the case of logic programming, a *mix* self-applicable partial evaluator for PROLOG programs was first described in [4, 5].

In this paper, we present an extended partial evaluator capable of undertaking powerful partial evaluation of logic programs. First, we present basic operations in a partial evaluator such as expansion and suspension of predicates during the process of building the proof tree. The following two sections introduce new operations to partial evaluators. Subsequently, a treatment of negated predicates during partial evaluation is presented. Then, local optimizations are introduced in order to optimize the residual program, and practical experience is presented. Finally, conclusions are drawn and further research is outlined.

2. BASIC OPERATIONS

In terms of logic programming, a partial evaluator *mix* attempts to build a proof tree of its first input program wrt its second one. For example, from Equation (1.1), *mix* will attempt to produce a proof tree for int running l. The proof tree thus produced will correspond to the residual target program. There are basic operations to build the proof tree, which will be discussed in this section.

2.1. Expansion

Expansion of a predicate `p` corresponds to building a partial proof tree for `p` [11]. The expansion of the user's goal will then produce the residual program.

This can be implemented easily in PROLOG with the help of a predicate `pe/2` which receives as first argument a predicate and returns in its second argument the proof tree for the predicate as a list. This is shown in Fig. 1.

Note that this implementation does not handle built-in predicates. In fact, if the variable `Goal` is instantiated to a built-in predicate, the first definition of `pe/2` will fail because `clause(Goal,true)` fails to find a solution for all predicates that are not facts in the PROLOG database. The second definition of `pe/2` also fails because `Goal` cannot unify with a term like `(Goal,Goals)`. The third definition fails because there is no definition for `Goal` in the database (since it is a built-in predicate that the user cannot redefine). However, extending the definition of `pe/2` to handle built-in predicates is trivial. This will be shown later. The implementation uses the PROLOG unification mechanism to unify variables of predicates that are being partially evaluated.

```
:- mode pe(+,-).
pe(Goal,[Goal]):- clause(Goal,true).[1]

pe((Goal,Goals),Tree):- pe(Goal,Tree1),
                        pe(Goals,Tree2),
                        append(Tree1,Tree2,Tree).

pe(Goal,Tree):- clause(Goal, Body),
                pe(Body,Tree1),
                append([(Goal:-Body)],Tree1,Tree).[2]
```

Figure 1. Expansion in partial evaluation.

1. `clause(H,B)` is a built-in predicate which is true if there is a clause `H:-B` in the PROLOG database. If `B` is the constant "true" then `H` is a fact.
2. `append/3` is true if its third argument is syntactically equal to the concatenation of the first two arguments.

An important fact is that the implementation only finds the first solution. Other solutions have to be found using backtracking or the built-in predicate `bagof`. At partial evaluation time we have less information than at execution time. This presents problems for the expansion of the proof tree, especially if the search space is infinite. This, nonetheless, is not an uncommon situation in partial evaluation. Uninstantiated variables can lead us to infinite solutions or infinite depth proof trees. Ways of handling this problem will be discussed in the following sections.

2.2. Suspension

The process of constructing the proof tree for a predicate may not terminate. It seems, therefore, necessary to determine the predicates that are not safe to expand during this process. These predicates have to be suspended, i.e. expansion has to be avoided. We will define a meta-predicate `unsafe/1` to be true if its predicate argument is not safe to expand, i.e. it may lead to a non-termination situation or to incorrect evaluations due to the lack of appropriate information (such as the treatment of negated predicates discussed in a latter section). If the expansion of a predicate `p` is suspended during the construction of its proof tree, `p` will become a leaf in the proof tree. Thus, the expansion of a predicate `p` is suspended if `p` is considered *unsafe*.

The problem at this point is how to decide which predicates are unsafe. To solve this problem there are several approaches. A simple and inexpensive way (during partial evaluation time) is to keep a depth counter, and suspend the expansion of a predicate when depth level h is reached. We call this procedure *depth suspension control* (DSC). This method has not yet been used in partial evaluation. In this case, the condition for expansion is checked at partial evaluation time.

The implementation of a DSC partial evaluator is shown in Fig. 2. We add two new parameters to the meta-predicate `pe/2`, which will be used to handle the DSC. The first, has to be initialized to zero The second parameter is a constant corresponding to the depth level *h*.

```
:- mode pe(+,-,+,+).
pe(Goal,[Goal],H,H):- !.

pe(Goal,[Goal],N,H):- clause(Goal,true).

pe((Goal,Others),Tree,N,H):-
                    pe(Goal,SubTree1,N,H),
                    pe(Others,SubTree2,N,H),
                    append(SubTree1,SubTree2,Tree).

pe(Goal,Tree,N,H):- clause(Goal,Body),
                    N1 is N +1,
                    pe(Body,SubTree,N1,H),
                    append([(Goal:-Body)],SubTree,Tree).
```

Figure 2. Partial evaluation with depth suspension control.

A second technique to suspend predicates at partial evaluation time is based on predicate subsumption [5]. However, the less expensive solution is to specify the unsafe predicates before the partial evaluation process. A manual method would allow the user to specify the predicates to be suspended through a process of user annotations, using a meta-predicate `unsafe/1` to be true if its argument is a predicate defined as unsafe. The implementation of meta-predicate `pe/2` in Fig. 1 only needs to add, as first definition of `pe/2`, the clause:

```
pe(Goal,[Goal]):- unsafe(Goal).
```

With this extension to `pe/2`, the unsafe predicates will be "captured" and will not be expanded. Consider, for example, a predicate `length/2` as unsafe if its first argument is an unin-

stantiated variable. In this case, it is possible to define `unsafe/1` for predicate `length` as follows.

```
unsafe(length(X,N)):- \+ ground(X).
```
[3]

Note that a predicate might be unsafe to expand under certain conditions, but it could be safe under others. For this reason, it is important to specify those conditions. The definition of unsafe predicates depends on the context in which a predicate is used. In some cases, it is possible to specify only a subset of all the conditions that make a predicate unsafe.

It should be noted that this annotation process involves the users in a programming problem. There is work on automatizing the predicate annotations as a process done before partial evaluation time [6] based on abstract interpretation of logic programs. However, this is still matter of on-going research.

3. NEW POWERFUL OPERATIONS

3.1. Freezing and Melting

An uninstantiated variable could trigger the suspension of a predicate. Given PROLOG's resolution mechanism, it is possible that a variable, not instantiated during the process of partial evaluation of a predicate, could instantiate later during the partial evaluation of other predicates. Let us consider the example shown in Fig. 3. In this example, `length/2` is considered unsafe (because of an uninstantiated variable). The partial evaluation of this predicate, therefore, is suspended.

```
foo(N):- length(X,N),
         X = ['this','is','an','example'].

unsafe(length(X,_)):- \+ ground(X).
```

Figure 3. Freezing a predicate.

During the expansion of the second predicate in the body of `foo/1`, variable X is instantiated, therefore, `length/2`, at this point, is no longer unsafe. This example proposes the necessity of temporarily suspending the expansion of an unsafe predicate, and trying its expansion at a later time.

In logic programming, these techniques are known as *freezing* and *melting* predicates. They, however, have never been used in partial evaluation. A necessary condition for freezing the expansion of a predicate is that there should be the possibility of melting it. In other words, the uninstantiated variables which made us consider the predicate as unsafe, appear in other predicates with a possibility of instantiation. Now we will give a more formal treatment of freezing and melting in partial evaluation.

Given a program P and an atom `p(X)`, let:

```
p(X)θ1 :- P1.
...
p(X)θn :- Pn.
```
(3.1)

be the residual program corresponding to a partial evaluation of `p(X)` wrt P. `P1`, ..., `Pn` are subgoals and θ1, ..., θn are substitutions [11]. Let `r` be an atom and `rS` a subgoal in an OR proof tree for `p(X)` wrt P.

3. `\+ ground(X)` is true if it is not the case that `X` is instantiated to a ground term.

Figure 4. Fragment of the expansion of `p(X)`.

Then, at this stage we have for a substitution ρ, the clause:

```
p(X)ρ :- rS.
```
(3.2)

Let us assume that `r` becomes frozen. `S` is a subgoal, i.e. a sequence of literals. For simplicity, we will only consider the case where `S` is an atom. We could further extend `S`. Let us assume that,

```
Sσ1 :- S1.
...
Sσm :- Sm.
```
(3.3)

is a residual program corresponding to the partial evaluation of `S` wrt P, with substitutions σ1, ..., σm. Graphically,

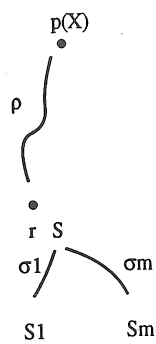

Figure 5. Expansion of `S`.

At this stage we could think of melting `r`. Notice that at the nodes `S1`, ..., `Sm`, the variables originally appearing in `S` (and also in `r`) are partially instantiated. Then, in the melting process for `r` we should further expand this partially instantiated `r`. That is, for each substitution σi, we have a residual program corresponding to the partial evaluation of `r`σi, the instantiation of `r` according to σi.

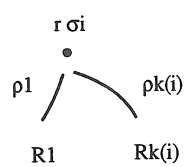

Figure 6. Expansion of `r`σi.

```
rσiρ1 :- R1.
...
rσiρk_i :- Rk_i.
```
(3.4)

In order to obtain a partial evaluation of `p(X)`, i.e. the original goal, that represents this freezing-melting process at this stage, we have to combine (3.2), (3.3) and (3.4). Thus, the residual program of `p(X)` contains the new clauses:

```
p(X)ρσ1ρi :- RiS1.
...
p(X)ρσmρi :- RiSm.
```
(3.5)

for i = 1, ..., k(i). To these clauses we should add the clauses corresponding to the other branches in the OR proof tree for a partial evaluation of `p(X)`. In this proof tree one of the subgoals is `S`. This subgoal was further expanded by first freezing `r` and then melting it. If `rS` is `P1` in (3.1), then the residual program for a partial evaluation of `p(X)` is given by (3.1) with the first clause replaced by (3.5).

From Fig. 5, we can see that the expansion of `S` after freezing `r` can avoid the generation of failing branches for `r`, thus producing an improvement during the execution of the residual program. Notice that the instantiations for `S` constrain the non-failing instantiations for `r` since we have to satisfy the whole subgoal `rS`.

In order to explore necessary conditions to characterize a predicate as frozen, we need to consider PROLOG´s resolution strategy (such as left-to-right and top-down). Also, we need to use variable modes, either by user annotations (many PROLOG systems allow the user to declare parameter modes) or by an automatic process through static analysis [12]. If during the partial evaluation of an input program, a predicate `p` (which has been frozen) instantiates its arguments so that it is no longer considered unsafe, we say that `p` can be *melted.*

We now require an algorithm to handle frozen predicates. The predicates considered here are those predicates `p` which are unsafe to expand, but have necessary conditions to be melted. In this case, one can take one of two actions. If the predicate gets enough instantiated, one could reflect it (as explained in next section). Otherwise, expand `p` according to rules (3.5).

3.2. Reflection

Operationally, a partial evaluator translates a program in one meta-level to a lower meta-level. Sometimes, it is possible to directly execute operations which are in a higher meta-level in lower levels. This is called reflection, and will be introduced to partial evaluation as an extension.

The implementations of partial evaluators, shown in this paper, use the PROLOG unification mechanism to unify terms belonging to the predicates that are being partially evaluated. In other words, the implementations we propose are using lower meta-level features (the PROLOG interpreter) to handle predicates of an upper meta-level.

We can use this concept in a more general context. For instance, we can take an operation of a level k, send it to a level j (j < k) execute it, and return the result to level k. To implement reflection in our partial evaluator, we only have to extend the definition of predicate `pe/2` of Fig. 1, as shown in Fig. 7.

In this definition, we consider as *reflectable* a predicate if it satisfies a certain "reflectability criterion". This is defined using the predicate `reflectable/1`. It is clear that the evaluation of higher meta-level built-in predicates can be reflected, to the PROLOG meta-level, if all their input mode parameters are instantiated. The reflection of those built-in predicates whose reflection might produce side-effects, should be avoided. Finally, no other necessary conditions to define a predicate as reflectable, have been identified.

```
pe(Goal,[]):- reflectable(Goal),
              call(Goal)
```

Figure 7. Extending partial evaluation with reflection.

4. OPERATIONALIZATION

The technique of operationalization is "borrowed" from the theory of explanation based generalization (known as EBG) [14]. This is a deductive learning technique, which has never before been used in partial evaluation. In this perspective, the operationalization of a predicate is related to its reformulation in terms of other predicates, which are easier to calculate, called "operational predicates".

The main idea is to expand the proof tree of a predicate until it reaches the operational predicates, obtaining a reformulation in terms of these predicates. This reformulation is the conjunction of the leaves in the proof tree. In order to apply this technique, it is necessary to define an operational criterion capable of identifying the operational predicates during the construction of the proof tree. Obviously, suspended predicates will be defined as operational. Also, we could decide to keep some predicates in the reformulated clauses, e.g. to handle incomplete programs. For this, we use a meta-predicate `operational/1` to define the predicates in its argument as operational.

In Fig. 8, we present the implementation of a partial evaluator extended with the operational capacity (OPE). Note that the satisfaction of the operational criterion is implemented by the meta-predicate `operational/1`, which is defined in terms of the predicate `member/2`. In order to be consistent with the previous example, `son/2` and `parents/2` are considered operational.

```
:- mode pe(+,-).
pe(Goal,GoalOut):- operational(Goal),
                   copy(Goal,GoalOut)[4],
                   call(Goal),!.

pe((Goal,Goals),(Tree1,Tree2)):- pe(Goal,Tree1),
                                 pe(Goals,Tree2).

pe(Goal,Tree):- clause(Goal,Body),
                pe(Body,Tree).

operational(Goal):- member(Goal,[son(_,_),parents(_,_)]).
```

Figure 8. Implementation of an OPE.

During the construction of a proof tree in partial evaluation, it is possible to face the problem of not having enough information to decide which node to expand (in or-nodes). We can either try to expand all possible branches (with an imminent combinatorial explosion) or to abort the expansion (meaning that we operationalize the node).

5. TREATMENT OF NEGATED PREDICATES

Negation will be treated based on techniques shown in previous sections, i.e. we will consider expansion, reflection and freezing of negated predicates.

---

4. `copy(Goal,GoalOut)` is true if `GoalOut` is a copy of `Goal` with fresh variables.

5.1 Expansion of Negated Predicates

Expansion of negated predicates cannot be treated in the same way as the expansion of any other predicate, and the problem arises when the predicate is insuficiently instantiated. With the purpose of partially evaluating the negation of a predicate `p(X)`, i.e. `not p(X)`, where not stands for negation as failure, we will introduce new clauses to the residual program. These clauses have `not p(X)` in their heads. Obviously, since negation as failure is not allowed in the heads of clauses, we will replace `not p(X)` by a new atom `not_p(X)`. Expansion of a negated predicate is not equivalent to the negation of the expansion of the predicate, when uninstantiated arguments are involved. Thus, if we want an answer to the query `not p(X)`θ, we evaluate the query `not_p(X)`θ with respect to the residual program corresponding to the partial evaluation of `not p(X)`. In this section we will explain how to obtain this residual program.

As an example, let us analyze the partial evaluation of `not p(X)` in the context of Fig. 9. The partial evaluation of `p(X)` produces `p(1)` after elimination of tautologies (which will be shown later). We might be tempted to evaluate `not p(X)` wrt the partial evaluation of `p(X)`. If we do this, we obtain the non-intuitively expected answer false. In consequence, if we evaluate `not p(xo)` wrt this partial evaluation, we will always obtain the answer false. Notice that we would expect the answer *yes* for `xo=2`. An explanation for this phenomenon is that `X` was instantiated and then eliminated, losing solutions.

Intuitively, the negation of `p(X)` is true if `X` does not unify with "`1`". It is clear then, that the problem arises because variable `X` is being uninstantiated during partial evaluation.

```
p(X):- q(X),r(X).
q(1).
r(1).
```

Figure 9. Example of problems for P.E. with Negation.

Now we will describe a general procedure to construct a residual program for the partial evaluation of `not p(X)`. Let,

```
p(X)θ1 :- G1.
...
p(X)θn :- Gn.
```

be the new clauses that we replace for the definition of `p(X)` in the original program P in order to obtain the partial evaluation of the program wrt `p(X)` [11]. `G1`, ..., `Gn` are subgoals in the non-failing branches in a proof tree for `p(X)` wrt program P. This residual program will be used in the partial evaluation of `not p(X)`. Then,

```
not p(X) succeeds   ⇔   p(X) fails
                    ⇔   X ≠ Xθ1 ∧ ... ∧ X ≠ Xθn,
                        ∨
                        X = Xθ1 ∧ G1 fails
                        ∨
                        ...
                        ∨
                        X = Xθn ∧ Gn fails
```

According to this, the partial evaluation of `not p(X)` wrt P can be defined as the partial evaluation of `p(X)`, plus the following rules:

```
not_p(X) :- X =/= Xθ1, ..., X =/= Xθn.
not_p(X) :- X == Xθ1, not G1.
...
not_p(X) :- X == Xθn, not Gn.
```

The first clause corresponds to the case where the construction of the proof tree passes through a failing branch, without getting to any of the G1,..., Gn.

5.2. Freezing Negated Predicates

Freezing can be used successfully for the treatment of negated predicates. It is possible to freeze negated predicates considered unsafe if variable instantiations during partial evaluation will revert the unsafe situation, allowing predicates to melt. Here the same considerations regarding local and global freezing are valid.

5.3. Reflection in Negation

Reflection can be applied to the treatment of negated predicates, expanding the proof tree as much as possible, in order to obtain better residual programs. Let us consider the definition of a predicate p:

$$p{:-}\ L_1, L_2, \ldots, L_n.$$

where L_i $(1 \leq i \leq n)$ is a negated predicate such as not q. If the proof of q produces an empty set[5], it is possible to express p as follows:

$$p{:-}\ L_1, \ldots, L_{i-1}, L_{i+1}, \ldots, L_n.$$

Note that this is possible since predicate q failed, and therefore, no instantiation of variables is done during the process of partial evaluation. Here, reflection is used to execute a predicate of a higher meta-level in the lower PROLOG meta-level. One of the major advantages of this technique is that it is possible to handle part of the negated predicates considered unsafe.

If the solution set of q is partitioned into $S_1 \cup S_2 \cup \ldots \cup S_n$ where S_i (S_i not empty) represents the solution i we will have:

$$\text{not } q = \text{not } S_1 \wedge \text{not } S_2 \wedge \ldots \wedge \text{not } S_n$$

allowing us to reformulate predicate p as follows:

$$p{:-}\ L_1 \wedge \ldots \wedge L_{i-1} \wedge \text{not } S_1 \wedge \ldots \wedge \text{not } S_n \wedge L_{i+1} \wedge \ldots \wedge L_n.$$

Obviously, this method works for n finite. Unfortunately, we do not yet know how to detect the unsafe cases. As an example, let us consider an example from [13] shown in Fig. 10. Even though fat and weight are finite relations, eats has a breadth-infinite solution space. A simple solution to this problem is to consider a *breadth counter control mechanism* (BCC).

```
eats(X,Y):- fat(X).
fat(X):- weight(X,Y), Y > 100.
```

Figure 10. Predicate with breadth-infinite solutions.

5. An empty solution set means that q is always false.

6. EXPERIMENTS

We wrote an interpreter of an imperative language, which we called NORMA3, to be used as a basis for our experiments. This interpreter has registers, conditional jumps, goto´s, assignments, arithmetic operations over registers, handling integer numbers, chars, strings and lists. The interpreter is written in PROLOG. We also wrote a program in NORMA3 to sort a list, based on the insertion sort algorithm.

According to Equation (1.1), the NORMA3 interpreter represents int, and the sort program is given by l. Some experiments were done, partially evaluating l wrt int, instantiating the input list to the sorting algorithm to a completely instantiated value and to a list with 3, 4 and 5 elements, obtaining residual programs ri, r3, r4 and r5, resp. These residual programs correspond to the compiled version of the imperative sorting program, i.e. we translated the sorting program from NORMA3 to PROLOG.

Three different techniques were applied to generate these residual programs, which we labeled with letters A, B and C. The first one corresponds to pure expansion (A), the second is expansion with suspension by subsumption (B), and the third is expansion, reflection, operationalization, suspension by user annotations, and tautology eliminations (C). We used a DIGITAL Decstation 3100 with Ultrix operating system and 24 MBytes of RAM, running the SICTUS PROLOG interpreter. Table 1 shows the values obtained during partial evaluation time, in CPU time units.

case	A	B	C
ri	2,331	3,316	79
r3	-	*	14,389
r4	-	*	64,054
r5	-	*	155

Table 1. Partial evaluation time.

The "-" symbol denotes that the experiment cannot be done. In the case of the previous table, this is due to infinite branches, since technique A does not have loop detectors. The "*" symbol denotes that the experiment was aborted after having generated a 6 MByte residual program. Note that the time for r5 with technique C is smaller than the ones for r3 or r4 since in the former, suspension had to be applied closer to the root, avoiding a disjunctive explosion. Table 2 shows the size of the residual programs, in number of clauses and predicates.

case	A	B	C
ri	132/422	81/244	1/1
r3	-	*	6/39
r4	-	*	24/238
r5	-	*	47/114

Table 2. Clauses/predicates in residual programs.

We can also show the execution time for the residual programs and for the sort program on top of the NORMA3 interpreter. Since the values depend on the order of the list to be sorted, we show an average time obtained doing 30 experiments each time, where the order of the elements in the list had a uniform distribution. Table 3 shows those times.

case	A	B	C	NORMA3
ri	144	128	0	50
r3	-	-	7	50
r4	-	-	35	82
r5	-	-	113.74	114.25

Table 3. Execution times.

From Tables 1 and 2 we see that the partial evaluation process with the extensions presented in this paper (C) is much more efficient than traditional partial evaluation (A, B). Also, as seen from Table 3, our extensions produce a better execution time for residual programs than the ones obtained using traditional techniques. In the case of r5 with technique C, we had to suspend the expansion quite close to the root to avoid the combinatorial explosion, and hence it was not possible to get a significant improvement in performance, as the case of ri, r3 and r4.

Note that if not enough additional information is provided to a program being partially evaluated, it will not be possible to build a deep proof tree, having to suspend the predicates near the root to avoid a disjunctive explosion due to the generation of all possible cases allowed in an algorithm. In the case of disjunctive explosion, one would be transforming the input program to a number of cases, simplifying the algorithm but loading the PROLOG interpreter with top-down search of clauses.

8. CONCLUSIONS

The extensions proposed in this paper obey the need of extending not only the functionality of current logic programming partial evaluators (with only two operations, expansion and suspension of predicates using the subsumption criterion), but also allow obtaining a better performance from the partial evaluator and the residual program.

For example, depth suspension control (DSC) increases the performance of the partial evaluator with respect to subsumption. DSC also gives the user the flexibility to obtain larger or smaller residual programs, depending on his needs, and guarantees termination of the process. The latter does not apply to the subsumption criterion.

The operations of freezing and melting predicates produce more instantiated residual programs and, therefore, more efficient programs. Reflection allows the partial evaluator to run operations from higher meta-levels into PROLOG's meta-level, and then, return their values. This is obviously a gain in partial evaluation time efficiency.

We also included operationalization of predicates, an operation "borrowed" from explanation-based generalization learning techniques, which proved to be very powerful in the construction of residual programs when it is combined with suspension, since in the reformulation one only leaves the root and the operational predicates.

The treatment of negated predicates in the input programs was also introduced. This will give the partial evaluator the capacity to analyze real programs, a feature lacking in current partial evaluators. An important development is proposed wrt expansion of non-instantiated negated predicates. Finally, a number of local optimizations were proposed in order to produce a more efficient residual program.

We have implemented a PROLOG partial evaluator which includes most of the operations here defined, and were able to process large examples of the kind described by Equation (1.1), i.e. as a program optimizer and as a universal compiler.

We are currently in the process of extending the partial evaluator to the problem of mix self-application as described by Equations (1.2) and (1.3).

ACKNOWLEDGEMENTS

This work was partially supported by the Chilean National Fund for Science and Technology (FONDECYT), grant 92-0812, and grants DIUC 91/21, 91/04E and 90/8 from the Pontificia Universidad Católica de Chile.

REFERENCES

1. S. Abramsky, C. Hankin, (eds.), 1987, Abstract Interpretation of Declarative Languages, Wiley.
2. D. Chan, M. Wallace, 1989, A treatment of negation during partial evaluation in "Meta-Programming in Logic Programming", H. Abramson and M. Rogers, eds., Chap. 16, MIT Press, London.
3. A. P. Ershov, 1978, "On the Essence of Compilation" in "Formal Description of Programming Concepts", E. J. Neuhold ed., North-Holland, pp. 391-418.
4. D. Fuller, S. Abramsky, 1988, "Mixed computation of prolog programs", New Generation Computing, Vol. 6, Springer Verlag, Tokyo.
5. D. Fuller, 1989, Partial Evaluation and Mix Computation in Logic Programming, PhD Thesis, Dept of Computing, Imperial College of Science and Technology, London, U.K.
6. D. Fuller, 1992, "Replacing the loop detection scheme in partial evaluation of logic programs", in preparation.
7. D. Fuller, S. Bocic, 1992, "Extending Partial Evaluation in Logic Programming", in Computer Science: Research and Applications, Plenum Press, NY.
8. Y. Futamura, 1971, "Partial evaluation of computation process - an approach to a compiler-compiler", *Systems, Computers, Control,* Vol. 2, No. 5, pp. 41-67.
9. N. Jones, P. Sestoft, H. Søndergaard, 1985, "An experiment in partial evaluation: The generation of a compiler generator" in "Rewriting techniques and applications", J. P. Jouannaud ed., *Lecture Notes in Computer Science*, No. 202, Springer Verlag, pp. 124-140.
10. G. Levi, G. Sardu, 1988, "Partial evaluation of metaprograms in a "Multiple Worlds" Logic Language", *New Generation Computing*, Vol. 6, Springer Verlag, Tokyo .
11. J. W. Lloyd, J. C. Shepherdson, 1989, Partial Evaluation in Logic Programming.
12. C. Mellish, 1987, "Abstract interpretation of PROLOG programs", in [1], pp. 181-198, Wiley.
13. A. Mendelzon, 1988, Logic and Databases. VIII Conference of the Chilean Computer Science Society, Santiago.
14. T.M. Mitchell, R.M. Keller and S.T. Kedar-Cabelli, 1986, "Explanation-based generalization: a unifying view" in Machine Learning Vol I.
15. L. Sterling, R. Beer, 1986, "Incremental flavor-mixing of meta-interpreters for expert system construction", Tech. Rep. TR 103-86, Center for Automation and Intelligent Systems Research, Case Western Reserve University.
16. A. Takeuchi, K. Furukawa, 1986, "Partial evaluation of prolog programs and its application to meta programming", Information Processing 86 (ed. H. Kugler), Proc. IFIP 86 Conference, North-Holland.