New Generation Computing, 14 (1996) 237-259
OHMSHA, LTD. and Springer-Verlag

Towards Efficient Partial Evaluation in Logic Programming

David A. FULLER, Sacha A. BOCIC, and Leopoldo E. BERTOSSI
Computer Science Department,
Pontificia Universidad Católica de Chile,
PO Box 306, Santiago 22, Chile.

Received 15 December 1993
Revised manuscript received 27 June 1994

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 should run 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 process large "non-toy" examples. Here we present extensions to partial evaluators which will allow us to produce more efficient residual programs using less computing resources during partial evaluation.

First, the introduced extensions allow the processing of large examples, which is not possible with the previous techniques. This is now possible since the extensions use less CPU time and memory consumption during the partial evaluation process. Second, the extended partial evaluator produces smaller residual programs, producing important CPU time optimizing effects. With the standard techniques, a partial evaluator will most probably act as a pessimizer, not as an optimizer. Examples are given.

Keywords: Partial Evaluation, Logic Programming, Symbolic Manipulation, Program Optimization.

§1 Introduction

Partial evaluation has been known as a program optimization technique since the early seventies.[10,5] Only in the last few years, however, its use has been made possible. In this paper we will define extensions to partial evaluators which will made possible to process large programs when part of its input is known at compile time (i.e. partial evaluation time), as well as producing optimized output programs. Standard partial evaluation techniques in logic programming cannot handle large examples, and when they produce an output program, it will often use more computer resources than the input program. 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 wrt 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.[22,20,15]

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:

$$\mathrm{P}\colon \mathsf{D} \rightarrow (\mathsf{D} \rightarrow \mathsf{D})$$

where D is a set whose elements may represent programs in various languages, as well as their input and output. Note that P is a partial function, since programs may have infinite loops. A residual program r is defined by:

$$\mathrm{P}(\mathsf{p})\ (<\mathsf{d1},\ \mathsf{d2}>) = \mathrm{P}(\mathsf{r})\ (<\mathsf{d2}>)$$

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:

$$\text{target} = \mathrm{P}(\mathit{mix})(<\text{int},\ \text{l}>) \tag{1.1}$$
$$\text{comp} = \mathrm{P}(\mathit{mix})(<\mathit{mix},\ \text{int}>) \tag{1.2}$$
$$\text{cocom} = \mathrm{P}(\mathit{mix})(<\mathit{mix},\ \mathit{mix}>) \tag{1.3}$$

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. Note that not every partial evaluator has this ability.[3,14)]

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 basic concepts of partial evaluation and self-application. The first *mix* self-applicable partial evaluator was described in Ref. 12), 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 Refs. 6) and 7).

In this paper, we present an extended partial evaluator capable of undertaking powerful partial evaluation of logic programs. This means that the extended partial evaluator will use less memory and CPU time during the partial evaluation process, and the residual program will in fact be an optimized program. Note that a partial evaluator should produce a residual program which runs "at least as efficient" as the input program, which is not the case of standard partial evaluators. For this, we first 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.[16)] 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 easy, 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).*

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).**
```

Fig. 1 Expansion in partial evaluation.

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 baqof. As an example, let us consider the definition for the predicate member/2 in Fig. 2. This predicate receives, as first argument, an element and as a second argument, a list of elements. The predicate is true if the element belongs to the list; otherwise it is false.

```
member(X, [X|_]).
member(X, [Y|Z]) :- X \== Y, member(X, Z).
```

Fig. 2 Implementation of member/2.

To partially evaluate the predicate member/2 with its first parameter instantiated to the constant "3" and its second, to a list with three elements, we use the predicate pe/2 previously defined as follows:

```
?- pe(member(3, [_, _, _]), Tree).
```

* clause(H, B) is a built-in predicate which is true if there is a clause H′ :- B′ in the PROLOG database, such that H and H′ unify and B and B′ unify. If B is the constant "true" then H is a fact.

** append/3 is true if its third argument is syntactically equal to the concatenation of the first two arguments.

to obtain the instantiation of variable Tree = member(3, [3, Y, Z]). Using PROLOG's backtracking feature, we obtain the instantiations of Tree shown in Fig. 3.

```
Tree = member(3, [X, 3, Z]) :- 3 \== X, member(3, [3, Z]).
       member(3, [3, Z]).

Tree = member(3, [X, Y, 3]) :- 3 \== X, member(3, [Y, 3]).
       member(3, [Y, 3]) :- 3 \== Y, member(3, [3]).
       member(3, [3]).
```

Fig. 3 Backtracking the partial evaluation of member/2.

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.

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). Although this method is known in PROLOG program execution, it 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 parial evaluator is shown in Fig. 4. We add two new parameters to the meta-predicate pe/2, which will be used to handle the DSC mechanism. The first, has to be initialized to zero. The second parameter is a constant corresponding to the desired depth level.

A second technique to suspend predicates at partial evaluation time is based on predicate subsumption. It consists of keeping a record of the predicates

```
:- 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).
```

Fig. 4 Partial evaluation with depth suspension control.

that have been expanded. This prevents the expansion of a predicate that can be subsumed to one predicate in the list of already expanded predicates.[7,9] This technique is expensive in terms of time and memory usage. Each predicate has to be checked, at partial evaluation time, from the list being produced.

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

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

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[8,13] based on abstract interpretation of logic programs. However, this is still matter of on-going research.

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

§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. 5. 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).
```

Fig. 5 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 and are well known in PROLOG interpreters. They, however, have never been used in partial evaluation. Our contribution is the definition of the semantics of such operations in the context of partial evaluation and to introduce them as extensions to efficient 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 logic program P and an atom p(X), let:

$$\begin{array}{l} p(X)\ \theta 1 :\text{-}\ P1. \\ \dots \\ p(X)\ \theta n :\text{-}\ Pn. \end{array} \qquad (3.1)$$

be the residual program corresponding to a partial evaluation of p(X) wrt P. P1, ..., Pn are subgoals and $\theta 1$, ..., θn are substitutions.[16] Let r be an atom, S is a subgoal, i.e. a sequence of literals, and rS a subgoal in an OR proof tree for p(X) wrt P, as shown in Fig. 6.
Then, at this stage we have for a substitution ρ, the clause:

$$p(X)\ \rho :\text{-}\ rS. \qquad (3.2)$$

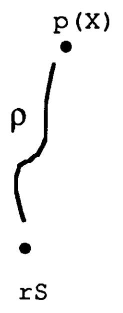


Fig. 6 Fragment of the expansion of p(X).

Let us assume that r becomes frozen. For simplicity, we will only consider the case where S is an atom. We could further extend S. Let us assume that,

$$\begin{aligned} &S\sigma 1 \text{ :- } S1. \\ &\dots \\ &S\sigma m \text{ :- } Sm. \end{aligned} \tag{3.3}$$

is a residual program corresponding to the partial evaluation of S wrt P, with substitutions $\sigma 1$, ..., σm. Figure 7 shows the expansion of S.

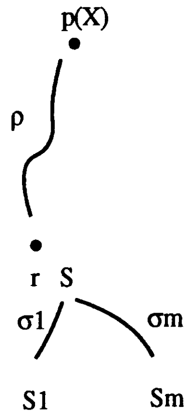


Fig. 7 Expansion of S.

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 $r\sigma i$, the instantiation of r according to σi, as shown in Fig. 8.

The residual program obtained by the substitution σi is:

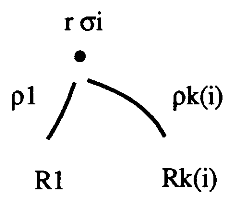


Fig. 8 Expansion of $r\sigma i$.

$$r\sigma i\rho l \text{ :- } Rl. \quad \dots \quad r\sigma i\rho k_{(i)} \text{ :- } Rk_{(i)}. \tag{3.4}$$

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:

$$p(X)\ \rho\sigma i\rho l \text{ :- } RlSi. \quad \dots \quad p(X)\ \rho\sigma i\rho k_{(i)} \text{ :- } Rk_{(i)}Si. \tag{3.5}$$

for i = 1, ..., m. To these clauses we should add the clauses corresponding to the other branches in the OR proof tree for a parial 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 Pl 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. 7, 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.[17]

Definition 3.1.1

A predicate p is a *descendant* of predicate q iff there is a clause q :- pl, p2, ..., pn such that p unifies with any predicate pi; or p is a descendant of any predicate pi in that body. □

Definition 3.1.2

A predicate p is a *right-ancestor* of a predicate q iff q is a descendant of a predicate r; so that r and p belong to the body of any rule, and p is at the right of r in that body. □

Thus, we can formulate the necessary conditions for freezing and melting a predicate, as follows.

Definition 3.1.3
A predicate p belonging to the body of a rule R is *freezable*, if for every variable X in the set of variables that make it unsafe (a set denoted by X_u) it is true that:

(1) X appears as an output variable in a predicate at the right of p in rule r, or;
(2) X is instantiated to an output variable in a right-ancestor of p. □

Definition 3.1.4
If all the variables belonging to X_u of a predicate p satisfy the first condition of Definition 3.1.3, we say that p can be *locally frozen*. □

Definition 3.1.5
If a predicate p has been frozen, and at least one of the variables that belong to X_u satisfy the second condition of Definition 3.1.3, we say that p has been *globally frozen*. □

A condition for melting a predicate p is when the set X_u of p has been instantiated, in which case the predicate is not longer unsafe for expansion. Note that this condition is independent of the manner in which a predicate has been frozen.

Remark 3.1.6
Predicate p will *never be melted* if it has been locally frozen and if the partial evaluation of the body of the rule to which it belongs to has no effect on the state of p (it is still unsafe). □

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 is also a well known operation in PROLOG interpreters. However, this is new as an operation in partial evaluation, which we will define and introduce as an extension in efficient partial evaluators.

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. 9.

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

Fig. 9 Extending partial evaluation with reflection.

This clause produces an empty residual program since the effect on the expansion of the predicate Goal is shown as an instantiation of its arguments only. In this definition, we consider as *reflectable* a predicate if it satisfies a certain "reflectability criterion", which is defined using the user defined 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.

§4 Operationalization

The technique of operationalization is "borrowed" from the theory of explanation based generalization (known as EBG).[19] EBG is a deductive learning technique, and its similarities with partial evaluation were first reported in Ref. 23) and later in Ref. 11). In Ref. 23), the authors established certain similarities between partial evaluation and EBG, but comparisons were done based on specific cases of such techniques. On the other hand, U. Hoppe presents a much deeper analysis of EBG but its relation to partial evaluation is poorly developed.[11]

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". Operationalization is not yet reported as an operation in partial evaluation. In this section, we will define it and include it as a powerful extension to partial evaluators in logic programming.

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. As an example, let us consider the set of rules of Fig. 10 and consider as operational the predicates son/2 and parents/2.

```
relative(X, Y) :- cousin(X, Y).
cousin(X, Y) :- son(X, Z), son(Y, W), brothers(Z, W).
brothers(X, Y) :- parents(X, W), parents(Y, W).
```

Fig. 10 Set of rules for operationalization.

Using the EBG algorithm, the reformulation of relative/2 is:

```
relative(X, Y) :- son(X, Z), son(Y, W), parents(Z, V), parents(W, V).
```

Notice that the reformulation contains the root of the proof tree and the operational predicates only, eliminating internal nodes of the proof tree. In partial evaluation, we can use this technique to both operationally defined predicates and to suspended predicates.

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. 11, 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),*
                     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(_, _)]).**
```

Fig. 11 Implementation of 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).

At present we do not know how to predict the number of reformulations

* copy(Goal, GoalOut) is true if GoalOut is a copy of Goal with fresh variables.
** member/2 as defined in Fig. 2.

obtained from the expansion of a certain predicate. If the operationalization process concludes with m (m < n) reformulations, it is possible to operationalize the "offending predicate" in order to obtain a shorter residual program.

§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 in partial evaluation of logic programs.

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 insufficiently instantiated.[4] 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). The expansion of a negated predicate is not equivalent to the negation of the expansion of the predicate, when uninstantiated arguments are involved. 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).

As an example, let us analyze the partial evaluation of not p(X) in the context of Fig. 12. The partial evaluation of p(X) produces p(1) after elimination of true formulas (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) (for a constant value 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).
```

Fig. 12 Example of problems for partial evaluation 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).[16] 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,

$$\begin{aligned} \text{not } p(X) \text{ succeeds} &\Leftrightarrow p(X) \text{ fails} \\ &\Leftrightarrow X \neq X\theta 1 \wedge \ldots \wedge X \neq X\theta n, \\ &\quad \vee \\ &\quad X = X\theta 1 \wedge G1 \text{ fails} \\ &\quad \vee \\ &\quad \ldots \\ &\quad \vee \\ &\quad X = X\theta n \wedge Gn \text{ fails} \end{aligned}$$

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.

Let us use the example of Fig. 12 to illustrate the new procedure. The partial evaluation of p(X) is:

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

In this case, the substitution θ has the value {X/1}, and G1 stands for the subgoal q(1), r(1). Then, the partial evaluation of not p(X) is given by the program:

```
not_p(X) :- X =/= 1.
not_p(X) :- X == 1, not(q(1), r(1)).
p(1) :- q(1), r(1).
q(1).
r(1).
```

Notice that the second clause should be split into the two clauses:

```
not_p(X) :- X == 1, not q(1).
not_p(X) :- X == 1, not r(1).
```

In this case, if we evaluate not_p(2) we will obtain the intuitively expected value *yes*.

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:

$$p :- L_1, L_2, \dots, L_n.$$

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,* it is possible to express p as follows:

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

Note that this is possible since predicate q failed, and therefore, no instantiation of variables is done during the process of partial evaluation.

To determine if the solution set of a predicate q is empty, we define the meta-predicate noSolution/1 to be true if the proof of the predicate in its argument produces a truth value false.

Here, reflection is used to execute a predicate of a higher meta-level in the lower PROLOG meta-level. A PROLOG implementation is shown in Fig. 13. One of the major advantages of this technique is that it is possible to handle part of the negated predicates considered unsafe.

```
noSolution(q) :- call(q), !, fail.
noSolution(_).
```

Fig. 13 Implementation of meta-predicate noSolution/1.

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

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

allowing us to reformulate predicate p as follows:

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

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

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 Ref.18) shown in Fig. 14. 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.
```

Fig. 14 Predicate with breadth-infinite solutions.

§6 Local Optimizations

Local optimization techniques can be used in order to reduce the complexity of the residual program.

6.1 Elimination of True Formulas

This technique consists on the elimination of formulas which are true in the residual program. There are two kind of formulas which can be eliminated, facts and clauses.

First, we can eliminate those predicates in the body of a clause of the residual program that are true independently of input variables. As an example, consider the set of rules shown in Fig. 15.

```
p(a, Y) :- q(a), r(a, Y), s(Y).
q(a).
r(a, Y) :- ...
s(Y)    :- ...
```

Fig. 15 Eliminating true facts.

It is possible to eliminate q(a) from the body of the first rule without changing the semantics of the residual program. Built-in operators such as "X =< X" also represent true facts which can be eliminated from the body of a clause.

An implementation for this technique is simple. We define a meta-predicate unique/1 to be true if the predicate in its argument has only one solution. Please note that it is not necessary to determine if such a predicate has all its arguments instantiated. Also, note that unique/1 needs to take care of unsafe predicates.

A second way of eliminating true formulas is when the body of a clause contains a fact which is false independently of its parameters, such as "X>X". This makes the clause a true clause, being able to eliminate it from the residual program. The early detection of true formulas during partial evaluation can avoid expanding further subtrees to the right of such predicate, improving efficiency of the partial evaluator.

6.2 Predicate Specialization

This process is expensive in terms of time. It consists of reducing the arity of a predicate, incorporating ground terms to the name of the predicate, thus reducing the number of unifications performed by the residual program at run-time. It is performed for each clause:

$$A :- B_1, \ldots, B_m \quad (m > 0)$$

in the residual program. For each B_i with at least one ground term in it, it is necessary to search for all the clauses in the residual program with instances of B_i. These clauses may be of two types, namely:

$$B'_i :- C_1, \ldots, C_j \quad (j \geq 0)$$

and

$$A'' :- B''_1, \ldots, B''_k \quad (k > 0)$$

such that B_i matches B'_i and B''_i. If there is at least one clause of the form:

$$B'_i :- C_1, \ldots, C_j.$$

the predicates B_i, B'_i and B''_i can be specialized to their ground values.

This is accomplished by first determine the common ground term values to all the predicates to be specialized. For example, the common ground terms of the predicates p(a, Y, f(c)) and p(a, b, f(c)) are the first and the third. Then it is possible to eliminate such values from the predicates and append to the predicate symbols a unique symbol. In our example, the two predicates would become p_it(Y) and p_it(b) ; respectively. The predicate symbol p_it is unique in the residual program. This will create a new residual program with the specialized clauses for B_i.

This process has also to be carried out for the user's goal. Observe that if the total number of predicate calls in the residual program is n, the time-complexity of the process of predicate specialization (as described) is $O(n^2)$.

A different approach for the predicate specialization based on abstract interpretation[2] which will lead to a much faster algorithm, is now being investigated.

§7 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 1. Some experiments were done, partially evaluating 1 wrt int, instantiating the input list of 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, and true formulas elimination (C). In the latter case, we defined via annotations, conditions for the partial evaluator to suspend, reflect, and operationalize predicates of the NORMA3 interpreter. The conditions to suspend NORMA3 predicates were defined using predicate unsafe/1 as specified in Section 2.2. Similarly, conditions to reflect or operationalize NORMA3 predicates were defined using predicates reflectable/1 and operational/1, as specified previously.

We used a DIGITAL Decstation 3100 with Ultrix operating system and 24 MBytes of RAM, running the SICTUS PROLOG interpreter. Table 7.1 shows the values obtained during partial evaluation time, in CPU time units.

Table 7.1 Partial evaluation time.

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

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 7. 2 shows the size of the residual programs, in number of clauses and predicates.

Table 7.2 Clauses/predicates in residual programs.

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

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 7.3 shows those times.

Table 7.3 Execution times.

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

From Tables 7.1 and 7.2 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 7.3, our extensions produce a better execution time for residual programs that 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.

It is possible to adopt the measurements used in Ref. 21) to estimate the optimizing effect of a partial evaluator. The CPU time (or memory size) optimizing effect is the ratio of CPU time (memory) of the unoptimized program to that of the optimized residual program. He distinguishes three qualitative levels of the optimizing effect. The effect is invisible if a program is improved by less that 1.2 with respect to CPU time and by less than 1.1 with respect to memory size. It is visible if the improvement is up 1.2 to 2 in CPU time and up to 1.3 in memory size. It is essential if the improvement is more than 2 in CPU time and more than 1.3 in memory. Table 7.4 shows the CPU time optimizing effect, based on the results shown on Table 7.3.

Table 7.4 CPU time optimizing effect.

case	A	B	C
ri	0.347	0.391	∞
r3	—	—	7.143
r4	—	—	2.343
r5	—	—	1.004

From this table it is clear that techniques A and B cannot be considered as optimizations. As a matter of fact, these techniques work as program pessimizers. Note that P. Abrahams warned about this danger as early as 1970.[1] However, technique C shows an impressive optimization effect for the cases ri, r3 and r4, and no pessimization effect can be achieved. Within the previous classification, these cases are considered to be essential.

§8 Conclusions

In this paper we present extensions to the technique known as partial evaluation in logic programming, obtaining important results. First, the introduced extensions allow the processing of large examples, which is not possible with the previous techniques. This is now possible since the extensions use less CPU time and memory consumption during the partial evaluation process. With the traditional partial evaluation techniques, most of the examples did not even produce a residual program.

Second, the extended partial evaluator produces smaller residual programs, producing essential CPU time optimizing effects. Note that with the standard techniques, a partial evaluator will most probably act as a pessimizer, not as an optimizer.

The extensions proposed here 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 this, these extensions are the needed to partially evaluate large logic programming examples.

For example, depth suspension control (DSC) increases the performance of the parial 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 include 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 is also introduced. This will give the partial evaluator the capacity to analyze real programs, a lacking feature 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. The processed examples did not require the partial evaluation of negated predicates. It was also not necessary to use the freezing and melting operations, although our partial evaluator implementation considers them. We are currently in the process of experimenting with even larger applications, where we expect that such operations will be of much help in producing efficient residual programs.

We are also 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

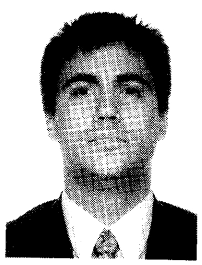
This work was partially supported by the Chilean National Fund for Science and Technology (FONDECYT), grants 1920812, 1950880 and 1930554, and by Oracle Chile.

References

1) Abrahams, P., "Compiler Pessimization," *Datamation*, *17*, *7*, pp. 32-33, 1971.
2) Abramsky, S. and Hankin, C. (eds.), *Abstract Interpretation of Declarative Languages*, Wiley, 1987.
3) Bulyonkov, M. A., "From Partial Evaluation to Mixed Computation," *Theoretical Computer Science*, *90*, Elsevier, pp. 47-60, 1991.
4) Chan, D. and Wallace, M., "A Treatment of Negation During Partial Evaluation," in *Meta-Programming in Logic Programming* (H. Abramson and M. Rogers, eds.), MIT Press, London, chapter 16, 1989.
5) Ershov, A. P., "On the Essence of Compilation," in *Formal Description of Programming Concepts* (E. J. Neuhold, ed.), North-Holland, pp. 391-418, 1978.
6) Fuller, D. and Abramsky, S., "Mixed Computation of Prolog Programs," *New Generation Computing*, *6*, Ohmsha/Springer-Verlag, Tokyo, 1988.
7) Fuller, D., "Partial Evaluation and Mix Computation in Logic Programming," *Ph.D thesis*, Department of Computing, Imperial College of Science and Technology, London, U.K., 1989.
8) Fuller, D., "Replacing the Loop Detection Scheme in Partial Evaluation of Logic Programs," in *Technical Report CS-93/14*, Computer Science Dept., Pontificia Universidad Católica de Chile, 1993.
9) Fuller, D. and Bocic, S., "Extending Partial Evaluation in Logic Programming," in *Computer Science: Research and Applications* (R. Baeza-Yates and U. Manber, eds.), Plenum Press, NY, pp. 95-107, 1992.
10) Futamura, Y., "Partial Evaluation of Computation Process—An Approach to a Compiler-Compiler," *Systems*, *Computers*, *Control*, *2*, *5*, pp. 41-67, 1971.
11) Hoppe, U., "An Analysis of EBG and Its Relation to Partial Evaluation: Lessons Learned," *Technical Report*, *No. 572*, GMD-IPSI, Darmstadt, Germany, 1991.

12) Jones, N., Sestoft, P., and Søndergaard, H., "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, 1985.
13) Jones, N. D., "Static Semantics, Types, and Binding Time Analysis," *Theoretical Computer Science, 90*, Elsevier, pp. 95-118, 1991.
14) Jones, N. D., Sestoft, P., and Søndergaard, H., "MIX: A Self-Applicable Partial Evaluator for Experiments in Compiler Generator," *J. LISP Symbolic Computation*, 1992.
15) Levi, G. and Sardu, G., "Partial Evaluation of Metaprograms in a "Multiple Worlds" Logic Language," *New Generation Computing, 6*, Springer-Verlag, Tokyo, 1988.
16) Lloyd, J. W. and Shepherdson, J. C., "Partial Evaluation in Logic Programming," *Technical Report*, Department of Computing and Mathematics, University of Bristol, U.K., 1991.
17) Mellish, C., "Abstract Interpretation of PROLOG Programs," in 2), Wiley, pp. 181-198, 1987.
18) Mendelzon, A., "Logic and Databases," *VIII Conference of the Chilean Computer Science Society*, Santiago, 1988.
19) Mitchell, T. M., Keller, R. M., and Kedar-Cabelli, S. T., "Explanation-Based Generalization: A Unifying View," in *Machine Learning, Vol I*, 1986.
20) Sterling, L. and Beer, R., "Incremental Flavor-Mixing of Meta-Interpreters for Expert System Construction," *Technical Report, TR 103-86*, Center for Automation and Intelligent Systems Research, Case Western Reserve University, 1986.
21) Pottosin, I. V., "Analysis of Program Optimization Possibilities and Further Development," *Theoretical Computer Science, 90*, Elsevier, pp. 17-36, 1991.
22) Takeuchi, A. and Furukawa, K., "Partial Evaluation of PROLOG Programs and Its Application to Meta-Programming," *Information Processing 86* (H. Kugler, ed.), *Proc. IFIP 86 Conference*, North-Holland, 1986.
23) van Harmelen, F. and Bundy, A., "Explanation-Based Generalization = Partial Evaluation," *Artificial Intelligence, 36*, pp. 401-412, 1988.

David A. Fuller, Ph.D.: He received the B.S. degree in Electrical Engineering from the Pontificia Universidad Católica de Chile in 1982, the M.S. degree in Computer Science from the University of California at Los Angeles in 1984, and the Ph.D. degree in Computer Science from Imperial College of Science and Technology in 1989. He worked as a research assistant at Imperial College in the HOPE project. Currently he is an Assistant Professor of Computer Science at the Pontificia Universidad Católica de Chile where he directs a Research Lab. on Computer Supported Cooperative Work. His research interests include symbolic computation and intelligent cooperative systems.



Sacha Bocic, M.S.: He received the B.S. and the M.S. degree in Computer Science from the Pontificia Universidad Católica de Chile in 1991 and 1992, resp. He is currently a Manager of the Core Technology Group at Oracle Chile. His current interests include symbolic computation, distributed data base systems and multimedia.

Leopoldo E. Bertossi, Dr.: He received a Doctor in Exact Sciences (Mathematics) degree from the Pontificia Universidad Católica de Chile in 1988. He received a DAAD Scholarship (University of Freiburg, Abteilung fuer mathematische Logic und Grundlagen der Mathematik, Germany), and has been visiting Assistant Professor at the Department of Computer Science, University of Toronto, and the Department of Electrical Engineering and Computer Science of the University of Wisconsin-Milwaukee. Since 1992 he is an Associate Professor at the Pontificia Universidad Católica de Chile. His current research interests are knowledge representation, deductive databases, logic programming and foundations of probability and statistics.